ChatGPT

Reward Design and Tuning in DQN Tetris Agents

Introduction

Designing an effective reward function is critical for Deep Q-Network (DQN) agents in Tetris due to the game's sparse natural rewards and complex strategies. Developers often incorporate *reward shaping* – additional incentives or penalties – to guide the agent toward desirable in-game behaviors. Common shaped rewards encourage clearing lines and maintaining a flat, low stack, while penalties discourage creating "holes" or uneven surfaces that lead to failure. The magnitude of these rewards and penalties must be tuned carefully: values are often chosen via empirical experimentation to provide a clear learning signal without encouraging pathological behaviors. Techniques like curriculum learning (progressively changing the reward structure or task difficulty) are also used to fine-tune training. Below, we break down typical reward components, how their magnitudes are set, and methods (including helper functions and curricula) used to shape DQN agents for Tetris.

Rewarded vs. Penalized In-Game Behaviors

Positive (Rewarded) behaviors: DQN Tetris agents universally receive positive reward for clearing lines – the primary objective of the game. Many implementations use the *number of lines cleared* as a basis, often giving larger rewards for clearing multiple lines at once (e.g. a "Tetris" of 4 lines) [1]. For example, one open-source DQN uses +40 for a single line, +100 for a double, up to +1200 for four lines cleared at once [1], mirroring classic Tetris scoring. Some agents further amplify multi-line clears by squaring the count: *clearing 2 lines yields 4 points, 4 lines yields 16 points*, making big line combos disproportionately valuable [2]. In addition to line clears, agents are often rewarded simply for **survival or piece placement** – e.g. a small reward each turn the game continues. One implementation gives +1 for every piece placed ("Alive") [1], encouraging the agent to avoid losing and play longer. Another gives a constant +1 per time-step survived, which helps the agent learn to prolong the game even when line clears are infrequent [3].

Agents may also get shaped rewards for **filling gaps or using the board efficiently**. For instance, if a move covers a previously empty gap or "hole" in the stack, the agent can receive a bonus. In one Tetris environment, placing a piece that fills horizontal space or covers an empty cell adjacent to existing blocks is explicitly rewarded [4]. Some reward functions promote **wide board usage**: spreading pieces across all columns instead of stacking in one area. This can be done by rewarding a higher *horizontal dispersion*. For example, a custom metric computes how evenly blocks occupy columns (variance of column occupancy), yielding a higher value when pieces are spread out; the agent then gets a positive reward proportional to this spread [5] [6]. Simpler variants count the number of columns used and reward filling more columns [6]. The intent is to prevent the agent from "center stacking" or relying on a single tall column. Additionally, **creating a deep well on the side** (a strategy human experts use to set up Tetris line clears) can be positively reinforced in some shaping schemes. For example, if the agent's stack forms a tall ridge with one empty column at either far left or right (a classic Tetris well), a large bonus might be given [7]. And of course, **completing lines** remains a primary positive signal in all cases – often the largest single reward the agent can get in one move [2].

Negative (Penalized) behaviors: The most heavily penalized outcome is letting the stack reach the top, i.e. **losing the game**. A **game-over penalty** is nearly always applied, though its magnitude varies (discussed below). For example, a small penalty like –5 [1] or –1 [8] is sometimes used to slightly discourage losing, whereas other implementations use a very large negative reward (e.g. –200) to strongly bias the agent against dropping pieces in a way that ends the game [9]. Aside from outright failure, most shaped reward functions penalize board states that make future play difficult. The chief culprit is **holes** – empty cells that have at least one filled block above them in the same column (making them inaccessible until cleared) [10]. Every new hole created is typically given a negative reward. For instance, one reward function subtracts 1.0 (or more) for each hole in the board [11], and further penalizes if the hole count increases compared to the previous state [12]. Holes are singled out because they prevent clearing lines above them and often lead to eventual loss if they accumulate. Another related penalty is for **"blocks on top of holes"**, sometimes counted separately as "covered cells" or "gaps." Filling a hole (reducing the count of holes) can conversely give a small reward [13], reinforcing moves that eliminate buried gaps.

Towering height and unevenness of the stack is another common negative factor. A high stack means the game is closer to ending, so agents are penalized for **aggregate height** (the sum of heights of all columns) [11] or for the **maximum column height** if it increases. Keeping the board low is safer, so some rewards subtract a small amount for each row occupied or for any increase in height beyond a threshold [14] [15]. *Bumpiness* is a measure of how uneven the surface is – often defined as the sum of absolute differences between adjacent column heights [16]. A bumpy skyline usually indicates potential holes and difficult placement, so most Tetris agents incur a penalty proportional to bumpiness [11]. For example, a DQN implementation might subtract 0.5 or 1.0 for each unit of height difference between neighboring columns [11]. Similarly, **wells** – deep pits surrounded by higher columns on both sides – can be penalized since they can only be cleared with a long piece. A *well depth* metric (sum of the gaps where a column is lower than its immediate neighbors) is sometimes included, with deeper or more wells adding to the penalty [17] [18]. In summary, most agents learn that creating a flat, hole-free stack is good (low penalty), whereas leaving isolated holes, spiky "pillars" (tall isolated columns), or very uneven surfaces is bad (high penalty). Table 1 below summarizes these common features and their typical reward impacts:

Feature (Helper Metric)	Definition and Reward Impact
Lines cleared	Number of lines cleared by the last piece placement. Primary source of positive reward; often scaled nonlinearly to encourage multi-line clears (e.g. reward ∝ (lines)^2) [2].
Holes	Count of empty cells with at least one filled cell above in the same column [10]. Penalized heavily (each hole incurs a negative reward) to discourage moves that create buried gaps [11]. Reducing the number of holes (by filling them) may yield a small positive reward [13].
Aggregate height	Sum of heights of all columns (total stacked blocks) [19]. Serves as a proxy for how high the board is. Often a small penalty per unit height encourages keeping the pile low [11].
Max height	Height of the tallest column. Sometimes used to trigger penalties if above a certain level (indicating danger) [14]. Can be included in aggregate height or handled via a threshold-based penalty.

Feature (Helper Metric)	Definition and Reward Impact
Bumpiness	The unevenness of the surface: sum of absolute differences between adjacent column heights [16]. Penalized to promote a flatter top surface and avoid valleys/peaks that lead to holes [11].
Wells (Well depth)	Measures deep columns relative to neighbors. For each column, the well depth is how much lower it is than the adjacent columns [17]; summed over all columns. Typically penalized (deep wells can trap pieces) [18], *unless* intentionally used as a strategy (see "Side well" bonus below).
Columns used / Spread	Number of columns that have at least one block, or a dispersion metric of filled cells across the board's width. A higher value means pieces are spread out. Some reward functions add a bonus for using more columns or achieving a high spread variance [6], to prevent the agent from stacking everything in one area.
Side well configuration	A special pattern where one edge column is mostly empty while adjacent columns are high (setting up a Tetris). This can be *rewarded* in shaping to encourage the agent to build a well on the side for easier line clears [7]. If not using such shaping, wells are generally negative (as above).
Move count / Rotation	(Not a board feature, but an event-based metric) – Some implementations add a tiny penalty for each move or rotation of a piece [20]. This discourages the agent from stalling or "vibrating" pieces unnecessarily. It ensures smoother play by penalizing frivolous movements (e.g. wiggling a piece back and forth yields many move penalties).
Game Over	A flag for losing the game (stack overflow). Always incurs a negative reward. The penalty ranges from modest (e.g. –5 [1]) to very large (–100 or –200) in different implementations, as discussed below. A larger penalty makes the agent prioritize survival more strongly.

Note: Many of these metrics originate from known Tetris heuristic evaluations (e.g. the **Dellacherie** scoring method) and are used as *helper functions* in code to calculate rewards. For example, open-source projects often implement functions like `count_holes(board)`, `calculate_bumpiness(board)`, etc., to compute these features from a board state [21] [22]. The agent's reward function can then combine these features linearly (with weights) to produce a shaped reward signal each turn.

Choosing Reward Magnitudes and Tuning Parameters

The relative scale of rewards and penalties is crucial for stable learning. In practice, these magnitudes are often set through **empirical tuning**: developers adjust weights and observe the agent's behavior and training convergence. Several considerations guide these choices:

- **Maintaining balance between objectives:** Rewards for line clears must be high enough to encourage completing lines (the main goal), but if they are *too* high relative to penalties, the agent may adopt risky strategies to force line clears. For instance, one project found that **over-rewarding Tetris line clears** caused the agent to stack pieces into a very tall structure waiting for the long "I"

piece – it would sacrifice board health for a big 4-line reward, often resulting in a top-out [23]. To avoid this, the reward for clearing lines is balanced with penalties for height and holes, so that making a mess of the board negates the gained points. Some implementations even scale the line-clear reward by how "clean" the board is. In one shaping scheme, the reward for clearing lines is multiplied by a factor *inversely proportional to the current holes and bumpiness* – meaning a line clear on a flat, hole-free board yields the full reward, whereas clearing lines in a messy board gives less reward [24] [25]. This encourages the agent to *prepare* the board (no holes, flat surface) before scoring big line clears.

- **Preserving learning signal vs. saturation:** Extremely large positive or negative rewards can destabilize DQN training by causing huge Q-value updates or overshadowing other feedback. Thus, designers often keep the scale of shaped rewards moderate. For example, one open-source agent initially set the death penalty to only –5 [1] so that losing wasn't an overwhelmingly large negative compared to the +1200 maximum line clear reward. In contrast, another team found their agent was too indifferent to losing with a small penalty – it would sometimes top-out intentionally after a Tetris since the loss penalty (–2 in their original setup) was trivial next to the line clear reward. They experimented with a *much larger* game-over penalty (–200) and reported that the **heavier death penalty significantly improved performance**, making the agent far more careful about survival [9]. This illustrates tuning by trial: if the agent isn't valuing survival enough, increase the loss penalty; if it fixates on avoiding death but never scores, one might reduce that penalty or boost line rewards. Some implementations also **clamp or normalize** the reward values to avoid outliers. For instance, a progressive shaping function limits the total shaped reward per step between –150 and +600 [26] to prevent any single action from having an excessive impact on training stability.

- **Empirical reference points:** A common strategy is to start with known scoring systems or heuristic weights as a baseline. Many projects base their reward magnitudes on **classic Tetris scoring** or prior research. For example, the reward table with +40/+100/+300/+1200 for clears of 1/2/3/4 lines is directly from the NES Tetris score rules (scaled down by a factor) [1]. Similarly, feature weights like "–0.5 * height, –0.18 * bumpiness" came from a known good hand-crafted heuristic [27]. Using these as initial values, one can run test training runs and adjust: if learning is slow or the agent exhibits undesired behavior, tweak the weights. In academic work, authors sometimes justify weights to ensure the reward provides a **dense gradient**. For example, in a Stanford project, the team first tried using only game score (which is sparse and spiky) and found learning was very hard with an ε-greedy policy (the agent rarely cleared any lines by random play, so it got almost no reward to learn from) [28] [29]. To address this, they *scaled up* certain rewards and added small incremental penalties to make the feedback more frequent. They gave a small penalty for every piece movement (to stop the agent from dithering pieces endlessly) [20], and ensured every state transition had *some* reward by using a smoother heuristic function initially [30]. In effect, the magnitudes and presence of each term were tuned to preserve a clear learning signal (reward differences) at each step.

- **Avoiding conflicting gradients:** When combining multiple objectives (line clearing, low holes, low height, etc.), their weights should be set such that they generally align rather than conflict. If one part of the reward heavily encourages a behavior that undermines another part, the agent can get "confused." For example, if staying alive (survival) is rewarded too strongly per time step, the agent might learn to just stall pieces indefinitely without clearing lines. One repository noted that if the survival reward was large, the agent would avoid any risky line clears and just try not to die – which in Tetris is not viable long-term because the board fills up [31]. Thus, survival bonuses are usually

kept small – just enough to prefer not dying *when all else is equal*, but not so high that the agent ignores scoring. In one implementation the alive bonus was +1 per piece, whereas a single line was +40 [1] (so clearly, clearing a line is far more valuable than just dropping pieces slowly). This ratio was chosen to ensure the agent seeks line clears; the survival reward mainly helps at the very start when the agent can't clear lines at all, by differentiating dying immediately from lasting a bit longer. Similarly, penalties like bumpiness vs. holes might be tuned relative to each other: if hole creation is much more severely punished than bumpiness, the agent learns *never* to make a hole even if it could get a Tetris by doing so – it might instead place pieces awkwardly to avoid any hole, which could be suboptimal. Some researchers have observed that **focusing on holes yields better survival than focusing on line clears** in early training [31]. As a result, they weight hole penalties quite high initially to teach the agent "don't make holes," and only later emphasize line clearing. In summary, reward magnitudes are often adjusted through iterative testing to find a sweet spot where the agent gets consistent, sensible feedback. Table 2 highlights a few different implementations and how they set key reward terms:

Source	Positive Rewards	Negative Rewards	Notes
GitHub: DQN Tetris (Michiel) [32]	+1 per piece placed (survival); +40/+100/+300/+1200 for 1–4 line clears (Tetris scoring); no extra feature-based rewards.	–5 for game over (loss). No explicit penalties for holes or height in reward (gameplay implicitly handles via state features).	Used classic scoring to reward clears. Small death penalty to improve convergence without dominating the signal [32]. Relied on Q-learning to discourage moves that create future problems (the Q-network learns that creating a hole now leads to less long-term reward) [33].
CS231n Project (Yang et al.) [34] [35]	Reward = (lines cleared)^2 (so 4 lines = 16, 1 line = 1). Added tiny penalty for each move/rotation to discourage "vibrating" pieces [20]. During a pre-training phase, used a *dense heuristic reward*: `-0.51*height + 0.76*lines - 0.36*holes - 0.18*bumpiness` evaluated each move [27].	–∞ for loss (episode end). Included a game-over penalty in the score to help convergence [36]. The move penalty was small (10–50% of an average reward) so as not to overwhelm line rewards [20].	Faced extremely sparse rewards with game score alone; solved by *transfer learning*: train first on the smoother heuristic reward (which gives feedback almost every drop), then switch to the actual game score reward [30]. This helped the network avoid local optima and see enough positive examples.

Source	Positive Rewards	Negative Rewards	Notes
"BeatrIS" Q-learning (Agnihotri) [37] [9]	Initially used a score = (lines cleared)^2 * board_width (rewarding clears more if the board was wide). Later included **bumpiness** in the reward function (penalizing rough surfaces) in addition to lines^2 [37] [38].	Originally –2 for loss; later changed to –200 [9]. Also implicitly penalized holes and height via including bumpiness and limiting score to line clears^2 (which indirectly favors smoother play).	Found that adding *state features* like bumpiness to the reward function dramatically improved performance (agent learned to keep board smooth) [39]. A much larger death penalty was needed to properly discourage losing [9]. Normalized evaluation scores back to using just lines^2 so different reward functions could be compared fairly [40].
Progressive DQN (Open-source Curriculum) [41] [42]	Uses different shaping per stage. Final stage "Balanced": moderate line-clear reward (+80 per line, +120 extra for a Tetris) [42]; strong bonus for wide spread (+25 * spread metric, +6 * each column used) [43] [44]; side wells bonus; small survival bonus per step. Earlier stages gave smaller line rewards but higher focus on other terms.	Balanced stage penalties: –0.75 per hole, –0.05 per height, –0.5 * bumpiness, –0.10 * wells [45]. Early stages had even higher hole penalties (up to –1.0 or –1.5) and height penalties, to drill those skills [41] [46]. Death penalty is light (–5) [47] in all stages to avoid destabilizing training.	Emphasizes different aspects over the training timeline (see next section). Reward magnitudes were hand-tuned through testing to ensure the agent learns each skill in order. The final weights reflect a balance where no single term dominates – e.g. even a large spread bonus won't save the agent if it creates too many holes, and vice versa. The shaping reward is clipped to a range to prevent extremes [26].

Table 2: Comparison of reward function designs in various Tetris RL implementations. Weights and values are as reported in sources or approximate where not explicitly given. This highlights how some implementations stick close to game score, while others incorporate heavy-handed shaping terms. Each approach required tuning to align the agent's priorities with long-term success in Tetris.

Reward Shaping and Adaptive Fine-Tuning Techniques

Designing a reward function often involves an iterative *shaping* process. **Reward shaping** means adding additional reward terms (like the penalties for holes, bumpiness, etc. above) beyond the sparse game rewards, in order to guide the agent. The challenge is to shape rewards in a way that accelerates learning without fundamentally misguiding the agent about the true goal. A well-known theoretical result (Ng et al.

1999) states that adding a *potential-based shaping function* (i.e. as the difference of some potential $Φ(state)$ between successive states) won't change the optimal policy, but many practical implementations use simpler ad-hoc shaping. In Tetris DQN projects, the following techniques are common:

- **Direct feature-based shaping:** From the start of training, define the reward $R = R_{\text{game}} + R_{\text{shaping}}$, where R_{game} might be something like lines cleared, and R_{shaping} is a weighted sum of features (holes, height, etc.). This provides immediate feedback for *subgoals* (e.g. "don't create holes now") even if the final outcome (clearing a line or losing) is far in the future. For example, in one implementation the shaping reward added `-1.25*holes -0.6*bumpiness -0.03*aggregate_height` each step [48] on top of the environment's base reward. The agent thus continuously gets negative points for messy structure, which steers it to prefer moves that keep the board clean. Direct shaping is powerful but must be monitored: the agent might become too dependent on the shaped terms and neglect the actual objective (clearing lines). In practice, some projects plan to **phase out** or reduce shaping over time (see curriculum learning below) so that the agent ultimately optimizes the real game score.

- **Adaptive adjustment of weights:** An advanced strategy is to adjust the reward weights during training based on the agent's progress. This can be thought of as *adaptive reward shaping*. For instance, if the agent is still creating many holes after 100 episodes, one could dynamically increase the hole penalty to push it harder to stop making holes. Conversely, if the agent has mastered avoiding holes but isn't clearing enough lines, one might boost the line-clear reward or height penalties to focus on scoring. In the "Progressive Reward Shaper" example, the code tracks running averages of holes, height, and column usage [49] [50]. While their default stage progression is tied to episode count, this infrastructure could allow the curriculum to advance only when certain performance criteria are met (e.g. *if average holes < 2 for the last N games, move to next stage*). Such performance-based adaptation ensures the agent fully learns one aspect before another is emphasized. Even without explicit performance gating, many curricula inherently adjust weights as epochs go by (which is effectively time-based adaptation). Another adaptive approach reported in literature is to start with a dense shaping reward and then **"wean" the agent off it**. The CS231n project mentioned earlier is a case in point: they first trained the network on a smooth feature-based reward, then switched to the sparse true reward entirely [30]. This two-phase training is a form of adaptive strategy: the reward function the agent sees is changed once it reaches a certain competency.

- **Reward normalization and scaling:** To fine-tune learning dynamics, some implementations normalize the shaped rewards (e.g., dividing by a running standard deviation or setting a cap as noted). This doesn't change the policy goal, but helps keep learning stable. Clipping extreme rewards, as in the progressive shaper (clamping between -150 and +600) [26], is one such technique. Another is periodically scaling all rewards to have a certain variance. While not unique to Tetris, these methods ensure no single experience blows up the Q-network updates.

- **Heuristic assistance during exploration:** Although not a reward shaping per se, a related idea is to use a heuristic agent to assist exploration. The CS231n team did this by occasionally letting a hand-crafted agent (using the Dellacherie fitness function) choose the action with some probability during training [28]. The heuristic agent naturally plays in a manner that clears lines and avoids holes, thus generating more *rewarding experiences* for the replay buffer. This can be seen as *indirect shaping*: the reward function didn't change, but the distribution of experiences was biased towards more

successful (reward-rich) states, helping the DQN learn faster. This technique addresses sparse rewards by ensuring the agent sees what "good" states look like early on.

In summary, reward shaping in Tetris DQN is often a **progressive journey**. Designers start with an initial guess (maybe weighted features plus line clear scores), observe training; if the agent gets stuck or learns a degenerate strategy, they tweak weights or add new terms. Over time, they might reduce dependence on shaped terms to let the agent optimize actual game score. The ultimate goal is to provide *just enough guidance* so the agent can discover a high-performing strategy (which usually aligns with human Tetris strategy: stack low, no holes, clear lines when possible), without spoon-feeding it so much that it never explores creative tactics.

Curriculum Learning for Progressive Training

Curriculum learning is a specific form of training strategy where the agent is not presented with the full difficulty of the task or the full set of objectives all at once. Instead, it progresses through *stages*, each with its own focus or simplified conditions, so that the agent masters simpler subtasks before moving on. In Tetris, curriculum learning is used to gradually shape the agent's behavior – effectively an extension of reward shaping over time. A common approach is to divide training into phases, each with a different reward emphasis:

- **Stage 1 – Basic Placement (Avoiding Holes):** The first phase teaches the agent fundamental habits like not creating holes. The reward function here heavily penalizes holes (e.g. a high negative weight on each hole, possibly –1.5 or –2 per hole) and may also moderately penalize bumpiness and height [41] [51]. Line clears might be given only a small reward or even ignored in this stage. The idea is to make the agent *first learn to stack cleanly*. For example, a curriculum might start with: *Reward = -2×holes -0.5×bumpiness -0.05×height + a small survival bonus*. This means an agent that places pieces without leaving gaps will get a higher reward (less penalty) than one that drops pieces carelessly, regardless of lines cleared. By the end of Stage 1 (after X episodes), the agent ideally has learned to avoid most holes and keep the surface relatively even.

- **Stage 2 – Height Management:** Once the agent can stack without making too many holes, the curriculum introduces a stronger incentive to keep the stack low. In this phase, the **height penalty is increased** [41] [52] and remains alongside the hole penalty. The agent must now learn not only to place pieces cleanly but also to disperse them to prevent any column from getting too tall. A small reward for using more columns might kick in here to nudge the agent toward spreading out. For instance, Stage 2 might use: *Reward = -1.2×holes -0.4×bumpiness -0.1×aggregate_height + 8×spread_bonus + survival reward*. The hole penalty might be slightly reduced compared to Stage 1 (assuming the agent already learned that) [53], while the height/spread terms are amplified. By focusing on height control, the agent learns to avoid building a single towering pillar ("pillar" is basically the highest column) which was identified as another key factor in survival [31].

- **Stage 3 – Board Spreading and Advanced Patterns:** In a later stage, the curriculum can encourage more advanced strategic behavior, like **spreading across all columns** and setting up wells for tetrises. Here the reward function gives strong positive feedback for using the full board width. For example, one curriculum's Stage 3 added a *large bonus* for the horizontal distribution metric – e.g. +25×spread and +6×per column used [54] – and also explicitly punished leaving the far left/right columns empty (to break any habit of center-stacking) [55]. The hole penalty might be further

reduced now (since the agent rarely makes holes at this point) to allow occasional holes if it leads to better spread or line clears [56]. This phase might also include a special reward for creating a deep side well (as mentioned earlier, a common human strategy for scoring tetrises). In the "side-well" shaping example, if the agent's board has, say, columns 1–3 much higher than column 0 (left edge) by a certain amount, a big bonus is given [7] – effectively teaching the agent the concept of building a Tetris well on one side. Stage 3 thus pushes the agent from merely "not dying" toward actively setting up high-reward situations (like preparing for 4-line clears) in a safe way.

- **Stage 4 – Balanced Full Game Play:** In the final curriculum stage, the reward function is "complete" or balanced, incorporating all pieces: line clear rewards are fully weighted, survival bonus is minimal, and moderate penalties remain for holes/height to prevent regression. By now the agent should handle all aspects, so the shaping is more gentle. For instance, the final reward might look like: *Reward = +80×lines_cleared (+120 extra if 4 lines) - 0.75×holes - 0.5×bumpiness - 0.05×height + moderate spread bonus + small death penalty* [42]. This is a mix of scoring points and maintaining a good board. The agent trained through the earlier stages will hopefully play in a disciplined way (few holes, low stack), and now it is encouraged to maximize score with those skills in place. If this stage were the starting point from scratch, the agent might have struggled, but thanks to the curriculum it enters Stage 4 with good habits already formed.

The progression above is exactly implemented in one open-source project, which defined a **4-stage curriculum** with: *Stage 1 (basic) focus on avoiding holes, Stage 2 (height) add height management, Stage 3 (spreading) encourage all columns, Stage 4 (balanced) final combined rewards* [57]. By default, they advanced the stage every 200 episodes of training [41], but as mentioned, one could also use performance triggers. Indeed, the code logs the average holes and columns used at each stage transition for monitoring [58]. The curriculum advancement can be visualized as the agent first learning *not to shoot itself in the foot* (no holes), then learning *how to stack efficiently* (flat and low), and finally *how to score* (clearing lines, making tetrises) once the foundational skills are in place.

Apart from reward-schedule curricula, one could also try an **environment curriculum** (though less common in Tetris): for example, start the agent on a smaller board or slower drop speed and then gradually move to the full 10×20 board at normal speed. This wasn't explicitly noted in the sources we surveyed – most curricula focused on reward shaping rather than changing the environment rules. The consistent finding, however, is that *staging the learning goals* greatly helps in Tetris. One PhD student who worked on Tetris RL commented that emphasizing holes (a Stage-1-type objective) and "pillar" height control led to longer survival and ultimately more lines cleared than directly rewarding lines cleared at the start [31]. This validates the curriculum approach: an agent that first masters not making holes will naturally clear more lines in the long run than one that greedily chases line clears without understanding board management.

Another example of curriculum-like training is the **transfer learning approach** used by the CS231n project: they effectively had a two-stage curriculum where Stage 1 used a **smooth heuristic reward** (dense feedback every move for height/holes/lines) and Stage 2 switched to the **sparse actual game reward** [30]. During Stage 1 (heuristic phase), the network learns basic principles (since it gets a shaped reward signal for every move, e.g. stacking flatter increases the heuristic value). Once it's somewhat proficient, Stage 2 forces it to optimize the true objective (clearing lines for score). They reported this switch was essential for the network to eventually play Tetris well, as training directly on the sparse reward was too difficult initially [30].

In summary, curriculum learning for Tetris DQN agents typically structures the training in phases such as:

- *Stage 1:* focus on avoiding catastrophic mistakes (holes) and learning to survive a decent number of moves.
- *Stage 2:* maintain a low, even stack (introduce height and minor line rewards).
- *Stage 3:* encourage optimal use of the board (spread out, set up tetrises) with reduced penalties for the now-mastered basics.
- *Stage 4:* full game objectives (maximize scoring while keeping the board safe), i.e. all shaping combined in balance.

This progressive shaping ensures the agent isn't overwhelmed by the complexity of Tetris from the get-go. Each stage builds on the last, and by the end, the agent ideally exhibits sophisticated play: it knows *not* to create holes, it keeps the stack low, uses the whole board, and times its line clears effectively. Curriculum learning has thus emerged as a key technique in training Tetris agents, addressing the sparse reward challenge and guiding the DQN to learn strategies that took humans decades to develop.

Sources: The above insights are synthesized from open-source implementations and academic reports on Tetris RL. For instance, Michiel's DQN Tetris repo provided an example of basic reward shaping with survival and line-clear points [1]. The CS231n student report detailed the use of a heuristic-based interim reward and specific feature weights [59]. Agnihotri's *"Beatris"* article discussed the impact of adding bumpiness and increasing loss penalties [37] [9]. A Reddit discussion by an RL practitioner emphasized the importance of hole and pillar penalties over naive line rewards [31]. Finally, our own project's code contributed a clear curriculum blueprint [57] and helper functions for computing features like holes, wells, etc. [10] [17], which we have cited throughout. Together, these sources illustrate how reward functions in Tetris DQNs are engineered and tuned to produce skilled agents that can *clear lines endlessly* (in the best cases) by learning the same principles a human Tetris master would, albeit guided by carefully shaped rewards along the way.

---

[1] [3] [32] [33] GitHub - michiel-cox/Tetris-DQN: Tetris with a Deep Q Network.
https://github.com/michiel-cox/Tetris-DQN

[2] [20] [27] [28] [29] [30] [34] [35] [36] [59] cs231n.stanford.edu
https://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf

[4] [8] [12] [13] [14] Thesis Template 2023
https://www.theseus.fi/bitstream/10024/863905/2/Popek_Dawid.pdf

[5] [6] [7] [10] [11] [15] [16] [17] [19] [21] [22] [24] [25] [48] reward_shaping.py
https://github.com/JCA374/Tetris-Gym2/blob/f36930e052887ec7995df83997c3f15ca2dcd42d/src/reward_shaping.py

[9] [37] [38] [39] [40] Beatris: an Evil Tetris AI. A data science guide to pitting a... | by Amogh Agnihotri | Medium
https://medium.com/@amoghhgoma/beatris-an-evil-tetris-ai-88fee6b068

[18] [26] [41] [42] [43] [44] [45] [46] [47] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] progressive_reward.py
https://github.com/JCA374/Tetris-Gym2/blob/f36930e052887ec7995df83997c3f15ca2dcd42d/src/progressive_reward.py

[23] [31] learning tetris through reinforcement learning : r/reinforcementlearning
https://www.reddit.com/r/reinforcementlearning/comments/1j8ek56/learning_tetris_through_reinforcement_learning/