MASTERING TETRIS
WITH REINFORCEMENT-LEARNING

SHENGDI CHEN[α]

Abstract. This thesis explores various techniques from Reinforcement-Learning for the purpose of solving the video-game Tetris. A modularized, modifiable and extendable implementation of the Tetris-game is written from scratch, complying with the essential elements of the game's standardized guideline. Furnished with feedback capability and support for both human-interactive and machine-direct inputs, the implemented engine is then forged to a trainable agent environment in accordance to the development of the Markov Decision Process formulation of the general Tetris game-play.

Four different modes with varying observation-mechanisms are deployed for training with three state-of-the-art algorithms of established implementations, one of which, the Deep-Q-Network, is implemented anew with injected adaptations to accommodate the specific evolution structure of the Tetris-game.

Two assessment criteria are utilized for the performance evaluation of the trained agents: in addition to heuristic-driven threshold-comparison of parameters logged during training, a static, hard-coded reference-agent is adapted to the environment. Whereas the direct usage of existing library methods produces lackluster results, the self-implemented Deep-Q-Network is found to yield a trained agent of marginally inferior capability in comparison to the reference performance in the majority of the testing scenarios given the selection of the suitable training-mode.

This project, conducted under the supervision of Prof. Dr. J. Buhmann, is executed within the framework of a Bachelor-Thesis under the catalogue identifier `401-3990-18L` at ETH Zürich.

RW/CSE, D-MATH, ETH Zürich: `shenchen␣AT␣ethz␣DOT␣ch | 17-943-358`.

Contents

Part 1. A Survey of the Status-Quo

1. Synopsis

This thesis-project explores the possibility of the deployment of methods of Reinforcement-Learning (RL) to the game-play of Tetris. An amalgamation of the name of its author and the underlying game to solve leads to its internal code-name of »Shetris«, as shown in the header.

The purpose of applying techniques from RL to the specific game-play of Tetris tangents upon two very specific domains, both boasting of a broad range of established researches. Notable examples are enumerated below.

1.1. Related Researches.

1.1.1. *On the game of Tetris.* Analysis of the game-play and rule-sets performed solely on the Tetris game are generally published by experts of the game, such as the individually maintained [kor02], as well as the community wiki [Dom15], whose relevance is further discussed in Section 3.1.

Purely theoretical researches such as [Dem+17]; [DHL02] have demonstrated the NP-completeness of optimally solving »the Tetris-problem«.

On the other hand, algorithms from the field of RL have been observed to perform reasonably well in learning the strategy for playing Tetris, as demonstrated in [Liu20]; [Mel18]; [Ste16]; [Lee18]; [nun21]; [Car05]; [Ngu18].

1.1.2. *On Reinforcement-Learning RL.* Reinforcement-Learning finds root in established literature from classical control-theory, with reference-literature [Ber95]; [Bel57].

The original idea of Q-Learning in [WD92] has recently been augmented with techniques of deep-learning, giving rise to the method of Deep Q-Networks (DQN) as originally published in [Mni+13]. The landmark publication [Mni+15] shows the potency of this fusion on the collection of Atari-games, achieving super-human performance on multiple occasions.

The DQN has been notably implemented by the original publisher [Dee15] and further tuned by [Kuz18] and utilized by projects such as [Kuz18]; [Ngu18]; [nun21].

General libraries such as [Hil+18] provide reference implementations of exemplary methods of RL beyond DQN, such as those proposed in the researches of [Sch+17]; [Mni+16]. The relevance of these algorithms to Shetris is discussed in later Section 33.

2. Technicalities

A summary of the technical details of this thesis is presented as follows.

2.1. Programming.

The main source-code package is further decomposed into two separate sub-packages: the first implements the logic of the Tetris-game, and the second performs analysis, training and testing.

2.1.1. *General usages.* The programming is performed with the `Python` language. Despite the dynamic-typing nature of the language, the source-code utilizes extensive type-hinting. Python versions no earlier than `Python 3.9` is required[1].

The source-code is consistently processed by the `black`-formatter [Wil+19] and thus implicitly conforms to the `PEP8` standard.

2.1.2. *Required packages.* The framework for the agent-interactive environment is provided by the `gym`-package developed by OpenAI[Bro+16].

The reference implementation of algorithms of RL, for example, the Deep-Q-Network (DQN) [Mni+15]; [Mni+13] and the Proximal-Policy-Optimization (PPO) algorithm [Sch+17], are provided by stable-baselines3[Hil+18], referred to as »`SB3`« in this thesis.

Note that both `gym` and `SB3` enforce project-wide type-hinting of source-code in `Python` as well.

2.2. Formatting and Typesetting.

This paper is compiled by the LuaTeX typesetting-engine featuring the open-source font-packages of `Libertinus` and `Source-Code-Pro`.

[1] Version requirement of `Python`'s `typing`-Module: https://docs.python.org/3/library/typing.html#module-contents

2.3. **Tracking.** The source-code of the Shetris-Project is distributed under the following repositories:

- The modules for the formulation of various agents, as well as those for training and testing are published under the repository: Link to GitHub;
- All other aspects of the Tetris-game, including the `engine` as described in Section 4, are found at: Link to GitHub;
- This thesis-document itself is accessible under: Link to GitHub.

Both source-code repositories deploy the »GNU Affero General Public License v3.0« license, else known as `GNU AGPLv3`.

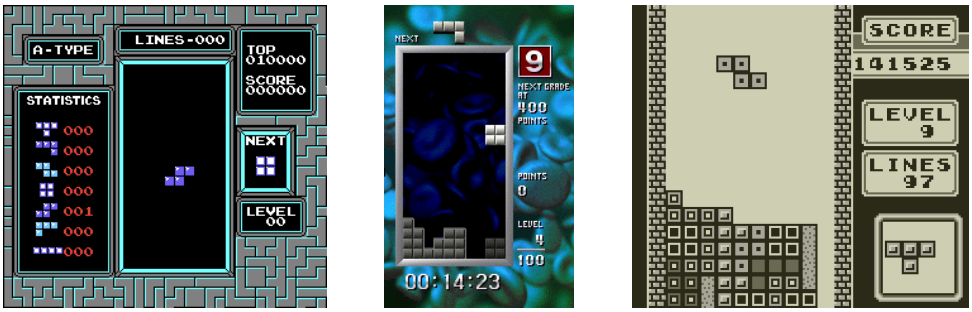


Figure 3.1. Several established implementations of the Tetris-game. From left to right: Tetris for NES, Nintendo (1989); Tetris The Grand Master (1998); Tetris for Game Boy (1989)

Part 2. The Birth of an Agent

3. General Notices

3.1. **Source of Information.** Despite its ubiquitousness, the term »Tetris« alone by no means offers a stringent set of definitions for the implementation and the execution of the game. Indeed, from the moment of its original conception in the 1980s by its inventor A. Pajitnov[2], the Tetris-game has since evolved into a myriad of variations across over 60 platforms. Figure 3.1 displays several historic implementations of the Tetris game.

The fact that every game release implements a separate set of derivative components from the visual effects to the specific rules not only leads to diversity of the game, but also to difficulty of standardization at the same time, complicating the analysis of the underlying Tetris-game itself.

In light of this, the publicly maintained `Tetris.wiki` [Dom15] is created to serve as a centralized collection of information on the Tetris-game, with the landmark article »Tetris-Guideline[3]« documenting the official set of recommended implementation for common components of a Tetris implementation, ranging from the size of the field (cf. Definition 4) to the algorithm of the piece-id generator (cf. Section 4.3). This page, from now on referred to as »The Guideline«, shall serve as the main reference for the Shetris-project, as well as this thesis. The Guideline is also quoted by established solution [Lee18].

Note 1. Citations from `Tetris.wiki`

Due to its vast topic coverage, citations therefrom are inserted as foot-note to the referenced sub-page instead of the traditional style of within-text-display. The reader shall note that the Creative Commons `CC BY-NC-SA 3.0` license is deployed site-wide, which in particular applies to the illustrations in Figure 3.1.

3.2. **Recapitulation of the game.** The general concept of the Tetris-game hardly requires introduction. This curt folk-lore more or less captures the general public's impression:

> At the very beginning, a board of empty boxes emerges.
>
> Then pieces begin falling, and one rotates and moves them, one at a time.
>
> If luck or skill shall so allow, some rows become complete, when the bottom is reached;
> and thus these rows are cleared, the height reduced and a new piece provided.
>
> Finally, the top of field no longer contains the fresh piece, and the game duly calls quit;
> unless the insatiated soul resets all, just to start again.

While intuitively explanatory, further details are required for a stringent formulation of the Tetris-game. In particular, there exist several frequent »moving-parts« of the game that must be clearly defined.

[2] In Russian: Алексе́й Па́житнов
[3] The Guideline: https://tetris.wiki/Tetris_Guideline

3.3. **Transferred Philosophy.** Concerning those variable components, the philosophy of Shetris is summarized as follows:

standard-compliant: deploy suggestions of The Guideline as the default whenever possible;
user-configurable: provide configuration possibilities to adapt to other existing implementations;
future-extendable: offer maximal API-extendability for future work beyond any existing implementation variants of these components.

Note that the compliance to The Guideline (or any standardized recommendation) is a delicate matter. In fact, as shall be demonstrated later in specification 40, deviations from The Guideline sometimes produce unexpected, yet helpful results, and might even provide insights from otherwise inaccessible angles. The reader shall thus pay attention to statements regarding adherence to the official standard, whenever explicit statements are made in this thesis.

4. Essential Implementations

The diversity of the Tetris-game as demonstrated in previous sections significantly influenced the programming of Shetris.

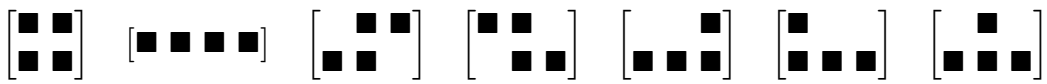
This section explains key implementation-details in Shetris by providing notes and justifications on the specific design decisions. This discussion also establishes the low-level foundation for the theoretical formulation of the control-process and the practical adaptation to the learning-environment in future chapters.

4.1. First touch with a piece.

4.1.1. *The identity of a piece.* The vertically falling shapes of four squares of the Tetris-game are instantly recognizable. Formally, they are defined as follows:

Definition 1. The `pid`: the identity of a piece[4]

A »*piece*« is defined as a collection of »*blocks*«. Commonly referred to as »Tetromino« and »Tetrimino«, seven of which are officially stipulated by The Guideline:



In Shetris, each of these seven pieces is uniquely identified with the variable `pid` of integer values from 0 to 6 in the exact order from left to right as shown above.

A brief justification of the ordering among the pieces is found in Remark 9.

Remark 1. Beyond the four blocks

The term »piece« is preferred over »Tetrimino« or »Tetromino« for generality potential to shapes beyond the aforementioned 4-block limitation. Indeed, researches such as [Dem+17] have studied exotic variations such as Trominoes (3 blocks per piece) and Pentominoes (5-blocks per piece); and [Car05]; [Mel18] demonstrate reinforcement-learning applied to the Tetris-game with other styles of modified pieces.

Though Shetris currently includes only the 7 official 4-block pieces as mentioned in Definition 1, extension to other constellations of pieces with arbitrary number of blocks is easily realizable under conformity to the API as described in Listing 2 and Listing 3.

4.1.2. *The rotation and SRS.* Given the `pid` of the range $\texttt{pid} \in \{0, 1, \cdots 6\}$, the specific *identity* of the piece is known. However, rotational inputs to the piece might still lead to changes of its »*shape*« (without modification to the `pid`). Though seemingly straight-forward, the execution of such rotations has seen numerous implementation variations. Instead of delving into the details, the author presents the rotation mechanism of Shetris and refers the interested reader to the corresponding Wiki-page[5] for in-depth investigations of other variants.

Definition 2. Excerpt of rotation system of Shetris

[4]Piece, or »Polyomino«: https://tetris.wiki/Piece
[5]Various rotation-systems: https://tetris.wiki/Category:Rotation_systems

The Super-Rotation-System (SRS)[6], as recommended by The Guideline, is deployed in SHETRIS. By no ways a summary of its features, the following list enumerates essential components of the SRS:

- Every piece can be rotated in the counter-clockwise and the clockwise sense;
- Every 4 rotational inputs in the same direction yields the original position;
- Every piece has one default rotational state.

Remark 2. Comparison to other projects

Most projects with self-implemented Tetris environments deviate from the recommended SRS [Ngu18]; [nun21]. Generally, the rotations are implemented in a »hand-wavy« fashion.

An exception is [Lee18] that explicitly states and deploys the SRS as well.

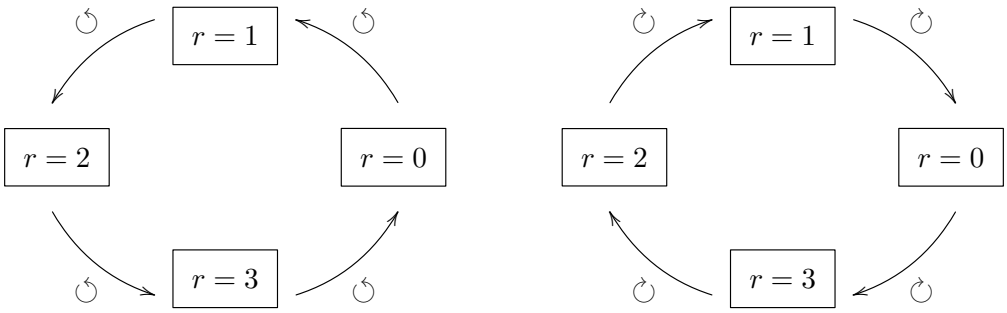
Note 2. Implementation of the SRS in SHETRIS

In SHETRIS, the rotational configuration of a piece, internally denoted as `rot`, is an integer value:

$$\texttt{rot} \in \texttt{range(4)} := \{0, 1, 2, 3\}$$

with 0 denoting the default rotational state.

Conforming to conventions in mathematics, every counter-clockwise rotation increments the value of `rot` by $+1$, while every clockwise rotation translates to a decrement of -1. The modulus-nature of the rotational configuration is illustrated as follows:

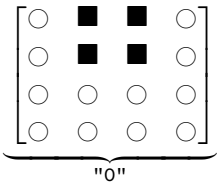


where r serves as the shorthand for `rot`.

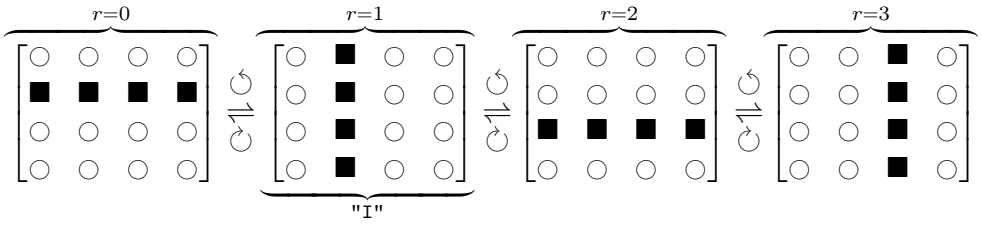
In the following, the full specification of every rotational configuration of all 7 pieces is given:

Definition 3. Rotation of all pieces

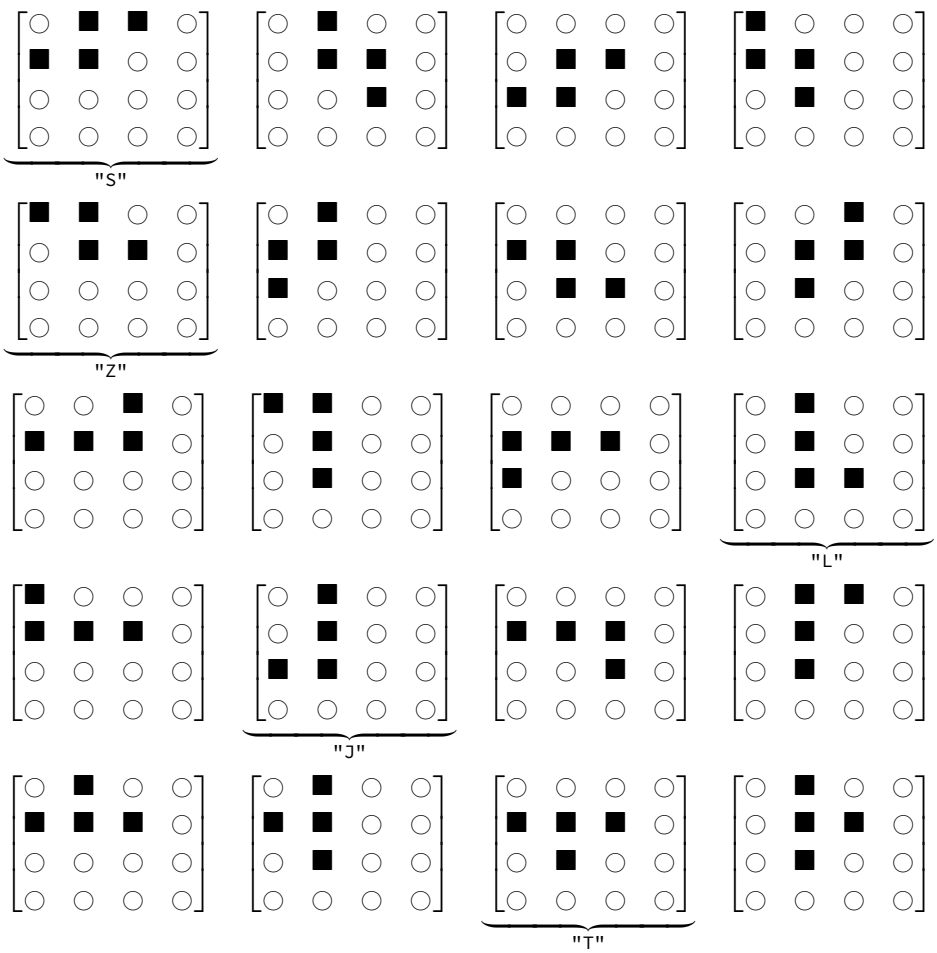
The O-piece has the same piece-shape for any of the rotational values $\texttt{rot} \in \{0, 1, 2, 3\}$:



Assuming no positional shifts in the vertical or the horizontal direction, every rotational value corresponds to one distinct configuration of the piece for all other 6 pieces. Their respective configurations at $\texttt{rot} =: r = 0$ to 4 are enlisted below:



[6]Super-Rotational-System: https://tetris.wiki/Super_Rotation_System



Note that this is by no means a misconception: the `"I"`-piece, for example, registers different configurations in the two horizontally ($\texttt{rot} \in \{0, 2\}$) and vertically ($\texttt{rot} \in \{1, 3\}$) »flat« placements.

4.2. **Indexing and Coordinates.** As soon as the piece's `pid` and `rot` is known, the »shape« of the current piece is uniquely identified from the enumeration in Definition 3. The final step of specifying a piece involves its vertical and horizontal positioning. To this end, the notion of the »board«, where every piece is actually placed, must first be formalized:

4.2.1. *The field.*

Definition 4. The field[7]

The »board« as mentioned in the introductory section 3.2, officially termed the »*field*«, is a two-dimensional, rectangular set of *blocks*, the same constituent of every *piece* as seen in Definition 1.

The Guideline stipulates the size of 20×10, i.e., the field shall encompass 20 rows and 10 columns of »blocks«.

At any time, each block is deterministically in one of the binary-states: occupied or unoccupied. Modeling the two states as `True` and `False` respectively, one might readily view the field as a (two-dimensional) matrix of boolean values:

$$\text{field} \in \{0, 1\}^{\#\text{rows} \times \#\text{columns}} \equiv \texttt{bool}^{\#\text{rows} \times \#\text{columns}}$$

which translates to $\text{field} \in \texttt{bool}^{20 \times 10}$ for the standard field.

Note 3. Implementation in Shetris

In Shetris, the field is represented by a two-dimensional array of boolean entries of the matrix-type `ndarray` by the (standard) array-library `numpy`. Referred to as the »field-array«, this matrix-like object, along with its size, is abstracted in the `Field` class as seen in 1.

```
1 class Field:
2     def __init__(self, field: np.ndarray):
```

[7] https://tetris.wiki/Playfield

```
        self._field = field
        self._size = field.shape
    ...
```

Listing 1. Constructor for `Field`

A brief enumeration of common operations of the field, for example, of finding fully occupied rows and of performing line-clears, is found in Listing 7.

Remark 3. Comparison with existing implementations

The reader shall note the argument of the constructor above: instead of accepting the size of the array to construct the field, as seen in [Ngu18]; [nun21]; [Lee18], the `Field`-class instead admits an object of `numpy.ndarray`.

This design creates an extra layer of separation: the `Field`-class is thus agnostic to the size of the underlying array, as well as its state at the time of creation, enabling initialization with arbitrary size and occupation status with external »factory«-style functions. In Shetris, The `FieldFactory` class of `util/fieldfac.py` provides such methods. In particular, it offers the utility to read the array from any external file, allowing initialization tailored to specific testing or debugging scenarios.

4.2.2. *Indexing.* A natural byproduct of choosing the underlying type of the field-array as `numpy.ndarray` is the indexing convention of the `numpy` library, dictating the axes and their directions when specifying the coordinates of the blocks of the field. This convention, as used throughout Shetris, is understood as follows:

Definition 5. Indexing convention of `numpy` and Shetris

Following the indexing rule of `numpy` (and various other numerical libraries), one observes:

- the positive direction of the first axis, axis-0, proceeds downwards vertically;
- the second axis, axis-1, points rightwards horizontally.

While this requires some mental adjustments compared to the traditional, two-dimensional Cartesian system, the abstraction is not only necessary for smooth deployment of `numpy.ndarray`, but also guarantees that the value first axis continuously increases as a piece »falls« downwards.

At this point, the coordinate of every block of the field is found with the following schematics:

$$\begin{bmatrix} (0,0) & (1,0) & \cdots & (9,0) \\ (1,0) & (1,1) & & \\ \vdots & & & \\ \vdots & & & \\ \vdots & & \ddots & \\ (19,0) & & & (19,9) \end{bmatrix}$$

4.2.3. *From relative to absolute coordinates.* With the notion of the coordinates of the blocks of the field as introduced above, it thus follows the natural extension to the coordinates of a specific piece. In preparation for this, the notion of relative-coordinates is first introduced:

Definition 6. The relative-coordinates of a piece

When viewing the schematic illustration in Definition 3, the reader might challenge the necessity of the extra padding by the unoccupied blocks around every piece and thus the necessity of surrounding all 7 pieces by the 4×4 »square-of-blocks«.

The justification of this stems from the notion of relative-coordinates, in the sense that given a piece and some rotational configuration, the »shape«, i.e., the relative coordinates of its blocks (4 of them for a Tetrimino or Tetromino) would always be constant.

By choosing the surrounding square of 4×4, one guarantees that all 7 standard pieces are contained in a uniform structure. The reader shall also note that this is the construction of the minimal size: indeed, decreasing its height or width leads to incapability to fully surround the `"I"`-piece in all rotational configurations.

Definition 7. Relative-coordinates in Shetris

With the upper-most and left-most block $\star$ as the reference-block, the coordinates of every other block in the 4×4 square-of-blocks *relative* to this reference-block is given with the following `[R-COORD]`-matrix:

$$\texttt{[R-COORD]} := \begin{bmatrix} \star & (+1,+0) & (+2,+0) & (+3,+0) \\ (+1,+0) & (+1,+1) & \cdots & \cdots \\ (+2,+0) & \vdots & \ddots & \\ (+3,+0) & \vdots & & (+3,+3) \end{bmatrix}$$

where the reference-block $\star$ is defined to have the relative-coordinates $(0,0)$.

Example 1. Some relative-coordinates

Some representative relative-coordinates are shown below:

$$\underbrace{\begin{bmatrix} \bigcirc & \blacksquare & \blacksquare & \bigcirc \\ \bigcirc & \blacksquare & \blacksquare & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc & \bigcirc \end{bmatrix}}_{\texttt{"O"}} := ((+0,+1),(+1,+1),(+0,+2),(+1,+2))$$

$$\underbrace{\begin{bmatrix} \bigcirc & \blacksquare & \bigcirc & \bigcirc \\ \bigcirc & \blacksquare & \bigcirc & \bigcirc \\ \bigcirc & \blacksquare & \bigcirc & \bigcirc \\ \bigcirc & \blacksquare & \bigcirc & \bigcirc \end{bmatrix}}_{\texttt{"I"}} := ((+0,+1),(+1,+1),(+2,+1),(+3,+1))$$

$$\underbrace{\begin{bmatrix} \bigcirc & \blacksquare & \blacksquare & \bigcirc \\ \blacksquare & \blacksquare & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc & \bigcirc \end{bmatrix}}_{\texttt{"S"}} := ((+1,+0),(+0,+1),(+1,+1),(+0,+2))$$

Remark 4. Choice of the reference-block $\star$

It shall be pointed out that the reference-block $\star$ can be chosen arbitrarily within the 4×4 square-of-blocks: one might as well have chosen some block situated more towards the center, or even the one in the lower-right corner.

The choice as made above for Shetris is based on the indexing-convention as described previously: in this way, all relative-coordinates in both index-axes within a the square-of-blocks are guaranteed to be always non-negative, and conforming to the standard 0-indexing, spanning the 4 integer-values from 0 to 3.

Note 4. Implementation in Shetris

In Shetris, the relative-coordinates of every `(piece, rotation)`-pair is stored as »class-members« of the class `RelCoord`, a snippet of which is shown below:

```
# shetris/game/src/engine/placement/srs/coord.py

class RelCoord:
   _rel_coords = np.array(
      (
         # O
         (
            ((+0, +1), (+1, +1), (+0, +2), (+1, +2)),
            ...
         ),
         # I
         (
            ((+1, +0), (+1, +1), (+1, +2), (+1, +3)),
            ((+0, +1), (+1, +1), (+2, +1), (+3, +1)),
```

```
            ((+2, +0), (+2, +1), (+2, +2), (+2, +3)),
            ((+0, +2), (+1, +2), (+2, +2), (+3, +2)),
         ),
         # S
         (
            ((+1, +0), (+0, +1), (+1, +1), (+0, +2)),
            ...
         ),
         ...
      )
   )
```

Listing 2. Data of relative-coordinates

The storage order of the relative-coordinates is defined as follows: the primary index is the `pid`, followed by the secondary index of the rotational configuration `rot`.

Finally, the method `get_red_coord()` handles the fetching of the relative-coordinates:

```
# shetris/game/src/engine/placement/srs/coord.py

class RelCoord:
   _rel_coords = ...

   @staticmethod
   def get_rel_coord(pid: int, rot: int) -> np.ndarray:
      return RelCoord._rel_coords[pid][rot]
```

Listing 3. Fetching relative coordinates

The `static` nature of this method and the class-wide setting of the variable `_rel_coords` frees the class `RelCoord` from the necessity of tracking any instance-variables: the caller only needs to pass in the arguments when invoking the method to find the relative-coordinates, without constructing instances or keeping track of internal values.

With the relative-coordinates defined, the calculation of absolute-coordinates follows naturally:

Definition 8. Construction of absolute-coordinates in Shetris

For the 4×4 square-of-blocks as formalized in Definition 6, the absolute-coordinates of all the 16 blocks within are found by »broadcast-adding« the absolute position, internally denoted by as `pos`, of the reference-block $\star$ to the `[R-COORD]`-matrix as first defined in 7:

$$\texttt{pos}\,(\star)\,\widetilde{+}\,\texttt{[R-COORD]} := \begin{bmatrix} (\star_0,\star_1) & (\star_0+1,\star_1+0) & (\star_0+2,\star_1+0) & (\star_0+3,\star_1+0) \\ (\star_0+1,\star_1+0) & (\star_0+1,\star_1+1) & \cdots & \cdots \\ (\star_0+2,\star_1+0) & \vdots & \ddots & \\ (\star_0+3,\star_1+0) & \vdots & & (\star_0+3,\star_1+3) \end{bmatrix}$$

For a specific *piece*, its absolute-coordinates are analogously found by »adding« the `pos`-value of its surrounding 4×4 square-of-blocks to its relative-coordinates retrievable via Listing 3.

Remark 5. Convenient broadcasting with `numpy`

Note that with the relative-coordinates stored in the format of `numpy.ndarray` as shown in Listing 2 and 3, the operation of »addition via broadcasting« described in the previous Definition 8 is trivially realizable with the generic addition syntax, saving the extra loop-calls seen in [Ngu18]; [nun21].

4.2.4. *Full specification of the piece.* At this point, the full-information of a piece could be defined and retrieved. In Shetris, this is implemented as follows:

Note 5. Implementation in Shetris: full specification of the piece

In Shetris, the coordinates of the reference-coordinate $\star$ is tracked with variable `pos` as seen in Definition 8. The `Config`-class encapsulates the rotational and the positional value `rot` and `pos`:

```
# shetris/game/src/engine/placement/piece.py

class Config:
   def __init__(self, pos: np.ndarray = np.array([0, 0]), rot: int = 0):
      self._pos = np.copy(pos)
      self._rot = rot % 4
```

Listing 4. Constructor of the `Config`-class in Shetris

The `Piece`-class builds upon the `Config`-class through membership and further includes the `pid` and `coord` to fully capture the information of a piece:

```
# shetris/game/src/engine/placement/piece.py

class Piece:
   def __init__(
      self,
      pid: Optional[int] = None,
      config: Optional[Config] = None,
      coord: Optional[np.ndarray] = None,
   ):
      self._pid = pid
      self._config = config
      self._coord = coord
```

Listing 5. »Disabled« Constructor of the `Piece`-class in Shetris

Instead of providing a »default« constructor, the `Piece`-class relies on `Python`'s language utility `classmethods` to initialize objects, as the creation process necessitates non-trivial operations such as the look-up of the relative-coordinates as mentioned in Definition 8.

4.3. **Generator.** The generator[8], produces the `pids`, i.e., the piece-ids, as hinted at in the introductory section 3.2. The Tetris-Guideline stipulates the usage of the shuffled »7-bag« generation algorithm, which is specified below:

Definition 9. Shuffled 7-bag generator

From the first generated `pid`, every 7 consecutive pieces is a (random) permutation of all the 7 `pid`-values:

$$\overbrace{\texttt{perm(7)}}^{1^{\text{st}}\text{ bag}} \quad | \quad \underbrace{\texttt{perm(7)}}_{2^{\text{nd}}\text{ bag}} \quad | \quad \cdots$$

where the pseudo-code `perm(n)` denotes any suitable interface returning a n-length, sequence-like object of the first n integers, i.e., $0, 1, \cdots, (n-1)$, in (pseudo-)random order.

Note 6. Implementation in Shetris

In Shetris, `numpy`'s library method `permutation()` is used instead of `python`'s built-in `random` package as seen in existing implementations [Ngu18]; [nun21]. This decision is based on two arguments:

On the one hand, `numpy`'s »in-house« random-number-generator (RNG) of `np.random.default_rng()` is cleanly default constructed without any extra argument specification. The advantage of this is seen when multiple iterations of the game are played: instead of burdening the caller with the responsibility to provide a

[8] not to be confused with the language facility of `Python`

different seed after every expired iteration, simply creating another generator object of this facility suffices as new randomness-initialization.

On the other hand, the aforementioned `permutation()` method returns a permuted sequence of type `numpy.ndarray`, which can be further processed naturally by the caller if more than one piece-id is required per call. This is by no means an exotic requirement: colloquially, it is known as the *previewing* of future pieces and is widely utilized in existing implementations. Considering the fact that the caller is likely an observing agent as shown in Example 4 and would likely require the observation to be of the format `numpy.ndarray` anyway, the usage of `numpy`'s library methods for the generator prevents extra conversions and renders the interface more type-coherent.

Note 7. Beyond the 7-bag algorithm in Shetris

The generator of Shetris reaches far beyond the standard, shuffled 7-bag system: in particular, the randomized bag-based algorithm accepts any input sequence to form the pool of `pid`-values to later draw from.

Also, Shetris provides the same flexible bag-based facility for non-shuffled `pid`-generation. In other words, the members of the input sequence-pool are simply repeated over in the exact order as specified at creation when exhausted.

The flexible interface for manipulating piece-generation is vastly exploited in the testing framework 40 shown later.

To the best of the author's knowledge, such facility of the generator does not seem to be standardized in existing implementations.

4.4. **Other Components.**

4.4.1. *Movement of pieces.* Apart from the convenient syntax for performing the broadcasting-addition as seen in Definition 8, the other, much more profound advantage of managing the coordinates using the piece-rotation stored in the format of `numpy.ndarray` is seen with the dynamic construction of the absolute-coordinates under movement inputs. This section provides a sneak-peak into the movement mechanism in Shetris, whereas detailed discussions are found in 16.

Definition 10. Displacement of pieces

The manipulation of a piece vertically and horizontally, otherwise referred to as »displacement« or »shifting«, is almost trivial to implement: as the relative-coordinates obtained by 3 is of `numpy.ndarray`, any such shifting operation translates to a simple addition or subtraction operation applied to the relative-coordinates, as explained in Definition 8.

On the other hand, inputs made to modify the rotational configuration `rot`, i.e., »rotation«, necessitates one extra re-reading from the relative-coordinates data with the routine in Listing 3.

The potential penalty of the re-read at every rotational change is heuristically mitigated, as rotations are expected to be considerably scarcer than shiftings. This is intuitively understood by considering the following example:

Example 2. Asymmetry between rotations and displacements

Consider the standard field with the height of 20 blocks:

A piece would have to shift vertically downwards, i.e., »fall«, for around 10 blocks if the field is occupied to half the vertical height, whereas the rotational inputs should never exceed 4 considering the modulus-by-4 property of rotation.

Intuitively, one thus concludes that displacements occur at substantially higher frequency than rotations.

Note 8. Implementation of displacements in Shetris

All displacements operations to the piece are handled in the `Mover`-class in the module:

```
1 game/src/engine/placement/mover.py
```

In particular, several *modes* of displacements are implemented with built-in checks for failure of the displacement: collision with already occupied blocks or exceeding of the boundaries of the field. Three of these modes, or types, are enlisted below, the source code of which is found in Listing 6:

atomics: the movement amount (of type rotation or shifting) is of exactly one unit, which corresponds to the traditional interactive game-play;
multi: the movement is performed as a succession of multiple atomics of the same movement-type;
maxout: the same movement-type is performed until failure, enabling, in particular, the »dropping«, or »falling« of a piece as specialization.

```
# game/src/engine/placement/mover.py

class Mover:
    ...

    def attempt_atomic(
        self, move_type: int, piece: Piece, pos_dir: bool
    ) -> Optional[Piece]: ...

    def attempt_multi(
        self, move_type: int, piece: Piece, delta: int
    ) -> Optional[Piece]: ...

    def attempt_maxout(
        self, move_type: int, piece: Piece, pos_dir: bool
    ) -> Piece: ...
```

Listing 6. Common movement operations of the `Mover`-class

Note that the methods of the `Mover`-class uniformly accepts as argument an object of the type `Piece` to represent the information of a piece in a non-failing, valid state, i.e., not colliding with already occupied blocks of the field and not exceeding the field boundaries. Also, the `None`-type of `Python` is returned to indicate movement failure, with the exception of the maxout-mode: if no unit of the requested movement type is executable, the state of the piece as provided is returned instead.

4.4.2. *Field operations.* Some of the essential operations of the `Field`-class that abstracts the field of the Tetris-game include locating fully-occupied rows, finding the indexes of currently occupied blocks, and ultimately, performing the line-clear operation. The signatures of these functionalities are provided in Listing 7.

```
# game/src/engine/placement/field.py

class Field:
    ...

    def _full_row_num(
        self, target_range: tuple[int, int]
    ) -> Optional[np.ndarray]: ...

    def idx_nonzero(
        self, higher_than: int
    ) -> tuple[np.ndarray, np.ndarray]: ...

    def lineclear(
        self, span_of_piece: Optional[np.ndarray] = None
    ) -> list[np.ndarray]: ...
```

Listing 7. Essential operations of `Field`

4.5. **Overview.** To sum up the above walk-through of the essential components of Shetris, one observes:

- The full information of a piece is abstracted in the `Piece`-class, which includes its `pid`, its coordinates of the four blocks `coord` and its configuration of the class `Config`, which, in turn, provides its vertical position `pos0`, its horizontal position `pos1` and its rotational value `rot`. The generator object spawns new `pid` values from any pool of potential candidates with optional randomness.
- The field is abstracted in the `Field`-class, providing operations such as checks against fully occupied rows and line-clears.
- The movement inputs are handled with the `Mover`-class, enabling multiple modes of movement for any component of the `Config`: the vertical position, the horizontal position and the rotation.

Note that all of the elements described above are categorized into the `engine`-package of Shetris, which is designed to serve as the entry point to the underlying game-logic of Tetris as a whole. Structurally, one observes:

```
shetris/game/
|-- src/
| |-- engine/
| | |-- generator/
| | | |-- baggen.py
| | |-- placement/
| | | |-- field.py
| | | |-- mover.py
| | | |-- piece.py
| |-- ...
|-- ...
```

The interested reader is referred to the source-code library for deeper immersion in the unabridged listings of the source-code.

5. Formulation of the Control-Process

The introduction to the `engine` of Shetris sets the foundation for subsequent formalization. In particular, the mechanism of the game-play shall be further specified within the framework of a control-process, as defined in established literature such as [Bel57]; [Ber95]. This chapter commences with the theoretical introduction to this formulation and proceeds to its application to the nuances of Shetris.

5.1. General Control-Process.

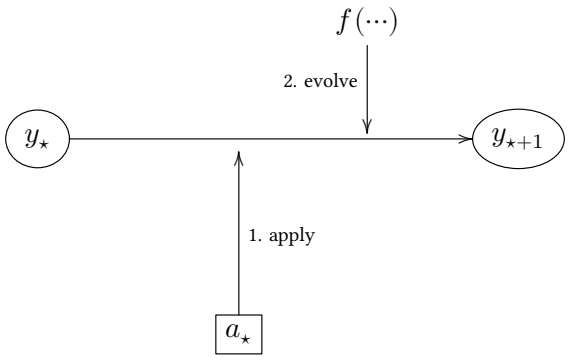
Definition 11. Formulation of a discrete Control-Process

Of interest is a system evolving with some time-stepping scheme $t \in \{t_0, t_1, t_2 \cdots\}$ represented by its internal *state* $y \in \{y_0, y_1, y_2 \cdots\}$ of some state-space $Y \ni y_{\forall t}$.

At any time-step $t_\star$, an *action*[9] $a_\star$ of some action-space $A \ni a_\star$ is applied to the system, causing the current state $y_\star$ to evolve with its internal evolution-dynamics f to the next state $y_{\star+1}$.

[9] cf. »control-input« u in control-theory

The following schematics depicts this process:



5.2. **Application in Shetris.** The game of Tetris could be readily expressed as a control-process described above. Among all possible formulations, the following is used in Shetris:

5.2.1. *The state y and time-stepping t_k.* A Tetris-game comprises of the state of the field y^F and that of the piece y^P. Thus:

Definition 12. State y of the control-process of Shetris
The state is the vectorized stacking of the field and the piece:

$$\vec{y} \equiv y := \begin{bmatrix} y^F \\ y^P \end{bmatrix}$$

where:

- the field-state y^F is a two-dimensional matrix of boolean-entries as specified in Definition 4;
- the piece-state y^P is abstracted with the type of the `Piece`-class, containing, as stated in Section 4.2.4, its piece-id `pid`, its rotational and positional configuration `config`, as well as the coordinates of its four blocks `coord`.

Before defining the action a and the evolution f, the concrete time-stepping scheme must be defined.

Definition 13. Discrete time-stepping of Shetris
For $\forall k \in \mathbb{N}$, the discrete time-step t_k advances to t_{k+1} with every call to the generator

```
pids = generator.get_pids()
```

to generate and retrieve the new piece-id(s) as described in Section 4.3.

Thus follow the definitions of the action a and the evolution f:

5.2.2. *Action and Evolution.*

Definition 14. Action a of Tetris-game
The action a is defined as any modification applied to the `config`-component of the piece-state y^P:

$$\vec{a} \equiv a := \begin{bmatrix} a^{\texttt{rot}} \\ a^{\texttt{pos0}} \\ a^{\texttt{pos1}} \end{bmatrix}$$

The action a of the general Tetris-game is thus the three-part aggregation of changes to the rotational (`rot`), vertical (`pos0`) and horizontal (`pos1`) configuration of the piece.

Definition 15. Evolution f of Tetris-game
At any time-step $\star$, the evolution of the state $y_\star$ is formulated in three parts:

- Once the action $a_\star$ has been applied to the current piece $a_\star \to y^P_\star$, it is then »displaced« vertically downwards in max-out mode to form the intermediate piece-state $\tilde{y}^P := f^{\text{fall}} \to a_\star \to y^P_\star$. The »fall«-evolution-component f^{fall} directly refers to Listing 6 in the Note 8 on displacement.

- This intermediate piece-state $\tilde{y}^P$ is then applied to the current field-state $y^F_\star$: fully occupied rows, if existent, are cleared with the aforementioned line-clear operation $f^{\text{line-clear}}$ in Listing 6 to produce the next field-state $y^F_{\star+1}$.
- The generator is called upon to produce `pid` of the next piece with f^{gen}, which is then initialized to evolve to the next piece-state $y^P_{\star+1}$, as described in Section 4.3.

In short, the following Control-Process describes the Tetris-game:

$$\begin{bmatrix} y^P_{\star+1} \\ y^F_\star \\ \tilde{y}^P \end{bmatrix} := \begin{bmatrix} f^{\text{gen}} \to y^P_\star \\ f^{\text{line-clear}} \to \tilde{y}^P \\ f^{\text{fall}} \to a_\star \to y^P_\star \end{bmatrix}$$

Definition 16. Action a of SHETRIS

In the current version of SHETRIS, as seen in other implementations of Tetris [Lee18]; [nun21]; [Ngu18], deploys a reduced action-space:

$$a := \begin{bmatrix} a^{\texttt{rot}} \\ a^{\texttt{pos0}} \\ a^{\texttt{pos1}} \end{bmatrix} \in \begin{bmatrix} A^{\texttt{rot}} & := 4 \\ A^{\texttt{pos0}} & := \emptyset \\ A^{\texttt{pos1}} & := 10 \end{bmatrix} \tag{5.1}$$

Specifically, the action is specified as a two-element pair of rotational and horizontal displacement:

$$a := (a^{\texttt{rot}}, a^{\texttt{pos1}})$$

In other words, the vertical displacement (in `pos0`) of the piece is completely governed by the evolution f^{fall} as specified in Definition 15.

Remark 6. Generality of the action-space

Note that the action-space defined in Equation 5.1:

$$A := \begin{bmatrix} A^{\texttt{rot}} & := 4 \\ A^{\texttt{pos0}} & := \emptyset \\ A^{\texttt{pos1}} & := 10 \end{bmatrix}$$

offers general accommodation for all possible `(pid, rot)`-pairs. However, the perceptive reader might object the implicit wastefulness of this approach. This specific intricacy is further elaborated upon in discussion 37, as well as 15.

Part 3. The Growing of its Consciousness

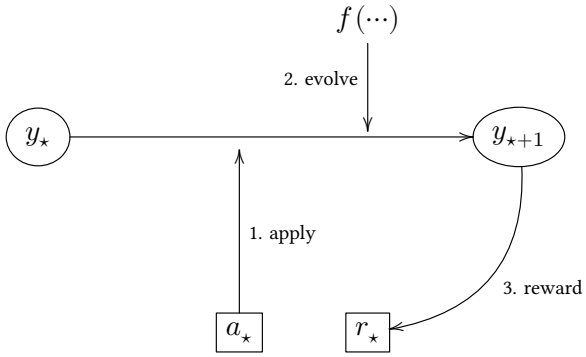
6. Formulation of the MDP

A critical framework for the analysis and quantification of agent performance, the Markov Decision Process (MDP), as described in standard literature such as [Ber95]; [Bel57], builds upon the discrete control-process as formulated previously in Section 5 by enriching it with the notion of step-wise *rewards*. With the foundation of the formulation of the control-process for the Tetris-game, this section describes the reward structure as used in Shetris.

6.1. Definition of reward.

Definition 17. The reward $r_\star$

At any time-step $\star$, the reward $r_\star$ is produced as the system evolves from state $y_\star$ to $y_{\star+1}$. This leads to the following augmented illustration based on its first rendition in Definition 11:



Note that no theoretical range for reward r is imposed, i.e., $r \in \mathbb{R}^1$.

The following section describes the construction of the reward mechanism for Shetris.

6.2. Reward-Engineering for Tetris.

6.2.1. *Episode Termination.* Following standard practices in reward formulation, the component of the reward by checking if the episode has elapsed r^{GO} is defined as:

$$r^{\text{GO}} := \begin{cases} +1 & \text{if episode not terminated} \\ -2 & \text{otherwise} \end{cases}$$

Similar reward components based on episode status are also utilized in [Ngu18]; [nun21].

The seemingly innocuous checking against episode-termination, or colloquially against the state of »game-over«, is non-trivial, as multiple definitions exist across various implementations of the game. Again, the reader is encouraged to visit the wiki-page[10] for more details. The mechanism adopted by Shetris is described below:

Definition 18. Criteria of episode-termination in Shetris

An episode is finished if and only if *any* block of the current piece overlaps with an occupied block of the field before *all* blocks of the piece are visible.

Note that this mechanism is inspired by the »block-out« variation described in the page nominated above.

Apart from the classical reward criteria of episode-termination of general control-problems, another candidate specific to the Tetris-game is described in the following section.

6.2.2. *Line-Clears.* Resulting in scores in most Tetris implementations, the line-clears constitute another apparent reward candidate r^{LC}. The exact mechanism of calculation is provided below:

Definition 19. Reward by Line-Clears r^{LC}

[10] Various Episode-Termination conditions: https://tetris.wiki/Top_out

The reward induced by a line-clear of n (full-)lines is distributed as follows:

$$\begin{aligned} r^{\mathrm{LC}}(n) &\propto k(n) \\ &:= c \cdot k(n) \end{aligned} \tag{6.1}$$

where the function $k(n)$ for the standard, 4-block pieces only game-play[11] is defined by The Guideline[12] as:

$$k(n \in \{0,1,2,3,4\}) := \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ 3 & n = 2 \\ 5 & n = 3 \\ 8 & n = 4 \end{cases}$$

Implicitly, higher »per-cleared-line« score is awarded with more lines cleared at the same time. Note that at the maximal value of $n = 4$, oftentimes referred to as a »Tetris-clear«, the per-line score rewarded is doubled compared to the case of $n = 1$, which is also known as a »single-clear«.

Note 9. Proportionality-factor in Shetris

In Shetris, the proportionality factor c of Equation 6.1 is chosen as:

$$c := 10 \cdot (\text{width-of-field})$$

translating to $c \equiv 100$ for the standard field of 10 columns.

The addition of the extra multiplicative factor 10, in contrast to other implementations such as [nun21]; [Ngu18], allows immediate identification of any line-clearing when manually monitoring the agent: with the reward-component from observing episode termination r^{GO} valued at 1 per non-terminating step, the sum from simply taking 10 steps would otherwise be indistinguishable from achieving a »single-clear« as defined previously. Setting the line-clear rewards r^{LC} to multiples of 100 thus eliminates such ambiguity.

6.2.3. *Reward in Entirety.* With the two components r^{GO} and r^{LC} defined above, one concludes the final reward function of Shetris:

Definition 20. Reward $r_\star$ of Shetris

At any time-step $t_\star$, the reward induced by undertaking action $a_\star$ is defined as the sum of the reward from checking episode termination $r_\star^{\mathrm{GO}}$ and calculating the line-clear induced score $r_\star^{\mathrm{LC}}$:

$$r_\star := r_\star^{\mathrm{GO}} + r_\star^{\mathrm{LC}}$$

Note 10. Implementation in Shetris

In general, the reward-mechanism of any control-problem is prototyped with the base-class below:

```python
# env/reporter/reward/reward.py

class RewardFactory:
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def get_reward(self, **kwargs) -> float:
        pass

    def get_reward_gameover(self, **kwargs) -> float:
        pass
```

Listing 8. Abstraction for General Reward-Functions

[11] In other words, every piece is a »Tetromino« or »Tetrimino«.

[12] Scoring Guideline: https://tetris.wiki/Scoring

In the case of Shetris, the inherited class `RewardFactory` is initialized with two components: one for the line-clear r^{LC} and one for the episode-termination r^{GO}:

```
# env/reporter/reward/reward.py

class RewardStandard(RewardFactory):
    def __init__(self, engine: Engine):
        self._engine = engine

        self._lineclear = RewardLineClear(self._engine)
        self._gameover = RewardGameover(self._engine)

        super().__init__()
```

Listing 9. Specialization of Reward-Function in Shetris

Note that each individual reward-component subsequently follows the abstraction:

```
# env/reporter/reward/component.py

class _RewardComponent:
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def get_reward(self, **kwargs) -> Any:
        pass
```

Listing 10. Abstraction for General Reward-Components

The reader is referred to the source-code to inspect the details of the implementation of the reward mechanism.

6.3. **Summary of MDP-Formulation.** With the formula for reward calculation, the MDP-formulation of Shetris is complete at this point. The following definition recapitulates all previously mentioned components:

Definition 21. MDP-formulation of Shetris

The 4-tuple:

$$\Big((y, Y), (a, A), f, r \Big)$$

of the MDP-formulation for Shetris is defined as follows:

As seen in Definition 12, the state y is composed of the field y^F and the piece y^P. The action a, as specified in Definition 16, consists of one input to the rotation of the piece $a^{\texttt{rot}}$ and another to the horizontal position $a^{\texttt{pos1}}$.

With the transitional evolution f is captured by Definition 16, the final reward mechanism r is provided by 20.

Remark 7. Source of randomness

For the game of Tetris as a whole, the only possible source of randomness is the generator, as the evolution mechanism of the field is otherwise completely deterministic.

In Shetris, this source of stochasticity could also be removed by specifying a non-random generator as described in Section 4.3.

With the availability of the MDP-formulation of Shetris, the upcoming section introduces the final adaptations required for creating a machine-compatible interactive environment for training.

7. Observation Extraction

7.1. **Context.** The `gym` library [Bro+16] as developed by OpenAI stipulates a standardized framework for the description of a dynamic system accessible via interaction with a decision-making agent $\mathcal{A}$. With the

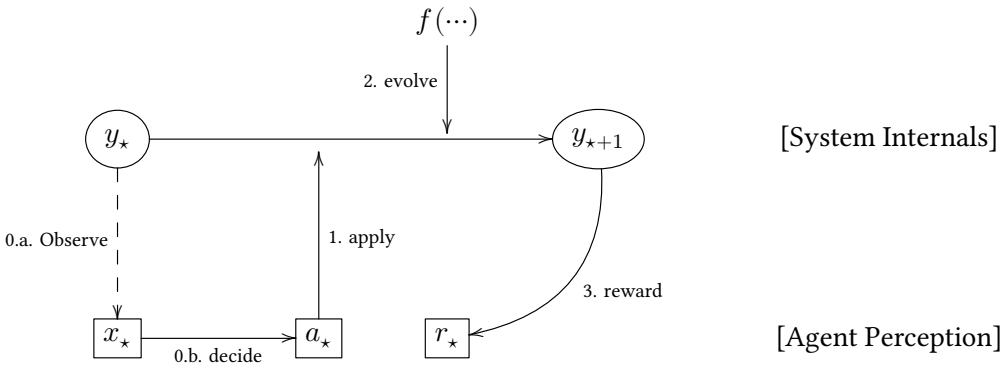
MDP-formulation in the previous chapter, the only missing piece towards the description of an agent conforming to the interface of `gym` is the notion of partial observation. This is defined as follows:

7.1.1. *Partial Observations.*

Definition 22. Partial Observations

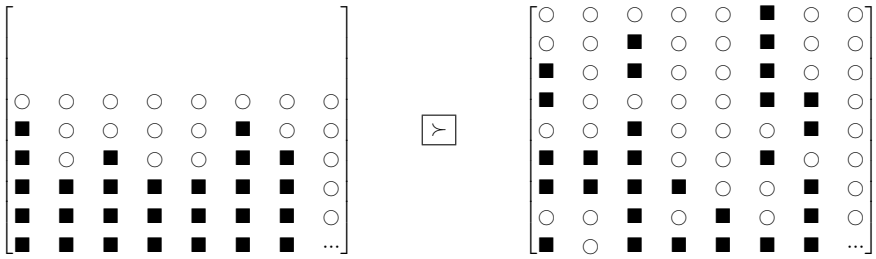
At any time-step $\star$, instead of the (full) state $y_\star$ of the system, the agent $\mathcal{A}$ perceives the system by the *observation* $x_\star$ of some observation-space $X \ni x_\star$, upon which the agent then decides the action $a_\star$ to apply to the system. The subsequent evolution-dynamics $f(\cdots)$ and the induced reward value r maintain the same definition as seen previously for the MDP-formulation.

Microscopically, the now fully augmented illustration below captures the evolution of the system from both the perspective of the internal system $y_\star$ and that of the agent $\mathcal{A}$:



Given the aforementioned MDP-formulation, it thus remains to specify the process of extracting the observation x of the agent $\mathcal{A}$ from the system state y. The following sections provide insights into the implementation of SHETRIS.

7.1.2. *Intuitions.* It does not take a veteran of the Tetris-game to conclude that the field on the left is in considerably better shape than that on the right as depicted in the comparative illustration 7.1 below: the uneven columns and the numerous holes of the field both constitute undesirable traits, leading to the rejection of the field on the right-hand-side to be rejected as decent game-play.



(7.1)

This section formalizes such intuitions to provide mechanisms for extracting various components of observation from the existing state of the game-play y as defined previously.

7.2. **Field-based Observations.** The fact that the introductory paragraph uses solely the field to gauge performance of the game-play conveys the message that the layout of the field alone encapsulates a myriad of performance indicators. Three field-based observations are currently deployed in SHETRIS to measure the performance of an agent, which align with the intuitions delineated above: the height x^H, the elevation x^E and the number of holes x^N. The following subsections introduce them separately:

7.2.1. *Height x^H.* To begin the investigation of the field, the height-based observation x^H is introduced:

Definition 23. Height $x^H(c)$ of a column c

The height x^H is intuitively defined for one single column c as the maximal vertical coordinate of an occupied box. Thus, the range of x^H is:

$$x^H(\forall c) \in [0; \text{field-height}]$$

translating to:

$$x^H \in \texttt{range(21)} \triangleq [0; 20]$$

for the standard field. The following corner cases shall be noted:

- $x^H \equiv 0$ equates to a completely free column;
- $x^H \equiv 20$ indicates occupation of the highest box.

The height of the column could be readily generalized to the corresponding observation of the field-state y^F:

Definition 24. Extracting x^H from the field-state y^F

Intuitively, the height of a field is the vectorized composition of the height of every column:

$$x^H\left(y^F\right) := \begin{bmatrix} x^H\left(c_0\right) \\ x^H\left(c_1\right) \\ \\ \\ \end{bmatrix} \in \left[\texttt{range}\left(\text{n-rows} + 1\right)\right]^{\text{n-columns}}$$

where c_i denotes the i^{th} column from the left under the indexing convention as described in Definition 5. For the standard field of 10 columns, the calculation translates to:

$$x^H\left(y^F\right) := \begin{bmatrix} x^H\left(c_0\right) \\ x^H\left(c_1\right) \\ \\ x^H\left(c_9\right) \end{bmatrix} \in \left[\texttt{range}\left(21\right)\right]^{10}$$

On the other hand, the »full« observation-vector x^H above could be further compressed into a scalar-value via summation. For the standard-field, one observes the following »compact« field-height observation:

$$x^{H,C}\left(y^F\right) := \sum_{i \in [0;9]} x^H\left(c_i\right)$$

Note the range of this compact variant of the field-height must be calculated separately. For the standard field, one concludes:

$$x^{H,C}\left(y^F\right) \in \texttt{range}\left(201\right) \triangleq [0; 200]$$

Note 11. Desirable height observations

As the criteria for deciding if an episode is over correlates directly to the »height« of the field, an intuitive metric concerns the maximal height. Intuitively, higher values of $x^{H,C}$ and x^H are more susceptible to termination of an episode and are thus undesirable.

Example 3. Demonstration of x^H and $x^{H,C}$

The following excerpt of the field illustrate the x^H applied to the individual columns:

○	○	○	○	○	○	○	○	○	○
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
○	○	○	○	○	○	○	○	○	○
○	○	○	○	○	○	○	○	■	○
■	○	○	○	○	○	○	■	○	○
■	○	■	○	○	○	○	■	■	○
■	■	■	○	○	■	○	■	○	○
■	■	■	○	○	■	■	○	■	○
■	■	■	■	○	○	■	■	■	○
5	3	4	1	0	3	2	5	6	0

And thus yields the compact counterpart $x^{H,C}$:

$$x^{H,C}\left(y^F\right) := \sum_{i \in [0;9]} x^H\left(c_i\right) = 29$$

Observations	numbers of	discrete-range	gym-translation
Height x^H	10	$[0;20]$	`range(21)`^{10}
Elevation x^E	9	$[0;20]$	`range(21)`^{9}
Holes x^N	10	$[0;19]$	`range(20)`^{10}
Full Space	`MultiDiscrete([21...21 | 21...21 | 20...20])`		

TABLE 7.2. Full field-based Observations

7.2.2. *Elevation x^E.* Apart from the absolute height of the individual columns, the *relative* relationship *among* the columns are studied:

Definition 25. Elevation x^E of a column c

The elevation x^E is defined as the difference in height between two adjacent columns.

Without loss of generality, SHETRIS calculates the elevation by subtracting the height of the right-hand-side column from that of the left-hand-side one:

$$\begin{aligned} x^E\left(c_i, c_{i+1}\right) &:= x^H\left(c_i\right) - x^H\left(c_{i+1}\right) \\ &\in \texttt{range(21)} \triangleq [0;20] \end{aligned}$$

In other words, a standard field of 10 columns each of monotonously decreasing height from left to right has 9 non-negative elevation-values.

Similar to the observation on the height x^H, the elevation-observation x^E could be extended to the entire field:

Definition 26. Obtaining x^E from the field-state y^F

The »full« elevation-observation x^E of a field is naturally defined as the elevation of every column of the field:

$$\begin{aligned} x^E\left(y^F\right) &:= \begin{bmatrix} x^E\left(c_0, c_1\right) \\ \vdots \\ x^E\left(c_8, c_9\right) \end{bmatrix} \\ &\in \left[\texttt{range}\left(21\right)\right]^9 \end{aligned}$$

The »compact« alternative $x^{E,C}$ is constructed as the summation of the elevation of every column, analogous to the definition of the compact height-observation in Definition 5:

$$\begin{aligned} x^{E,C}\left(y^F\right) &:= \sum_{i \in [0;8]} x^E\left(c_i, c_{i+1}\right) \\ &\in \texttt{range}\left(181\right) \triangleq [0;180] \end{aligned}$$

7.2.3. *Holes.* Probably the most dreaded element for the game-play of Tetris are »holes« in the field. This concept is formalized as follows:

Definition 27. Holes of a column $x^N\left(c\right)$

Given the definition of the holes of a column c as every unoccupied block lower than the highest occupied block, the hole-observation $x^N\left(c\right)$ is naturally defined as the number of such holes.

For the standard-field, the value range is given as follows:

$$x^N\left(c\right) \in \texttt{range(20)} \triangleq [0;19]$$

The extension to the field $x^N\left(y^F\right)$ and the formulation of the compact variant follow similarly.

7.2.4. *Two levels of abstraction.* Summing up the three field-based measures enlisted above, one concludes the following two different levels of details when observing the field, inducing the trade-off between the number and the range of vectors to track in the observation:

Observations	numbers of	discrete-range	gym-translation
Height $x^{H,C}$	1	$[0;200]$	`range(201)`
Elevation $x^{E,C}$	1	$[0;180]$	`range(181)`
Holes $x^{N,C}$	1	$[0;190]$	`range(191)`
Full Space	`MultiDiscrete([201, 181, 191])`		

Table 7.4. Compact field-based Observations

Definition 28. Two modes of field-based observations

»compact«: the results of the same measures are summed over to produce one scalar each, leading to less vectors, each of a wider range. This is demonstrated in Table 7.4

»full«: the three measures of height, elevation and hole are applied to the column level and reported as simple stacking. This produces more vectors, each of a narrower range, as seen in Table 7.2;

The juxtaposition of the two modes of observing the field is further explored in 12 and 13.

7.3. **Beyond the field.** Several other observation components are conceivable beyond the three field-based ones described previously. Two of them are enlisted below:

7.3.1. *Line-Clears.*

Definition 29. Observation of line-clears x^{LC}

Similar to the formulation of Definition 19, the observation of the line-clears is defined as the number of fully occupied lines subjected to clearing. The number of line-clears after one move is always within the range of 0 and 4, including both boundaries.

Observations	numbers of	discrete-range	gym-translation
Lines-cleared x^{LC}	1	$[0;4]$	`range(5)`

7.3.2. *The piece-id.* Finally, the piece-id also constitutes a viable candidate for observation:

Definition 30. Observation of piece-id $x^{\mathtt{pid}}$

Observations	numbers of	discrete-range	gym-translation
`pid` $x^{\mathtt{pid}}$	1	$[0;6]$	`range(7)`

This intuitive component seems to nevertheless induce disagreement on its relevance as observation. In particular, [Ngu18] excludes $x^{\mathtt{pid}}$ in the observation completely. The impact of the inclusion of this observation-component is further elaborated upon in 12 and 13.

7.4. **Summary of the environment.**

7.4.1. *Four types of observations.*

Definition 31. The four modes of training

With the consideration of varying components of the observation, one observes the following possibilities of varying the observation structure:

- for the three field-based observations (the height x^H, the elevation x^E and the number of holes x^N): two modes of detailing, »compact« and »full«, are possible, as enlisted in 28;
- for the `pid` $x^{\mathtt{pid}}$: this component could be included or discarded in the observation;
- for the number of line-clears x^{LC}: this is always included.

Variation-Code	Obs: field	Obs: `pid`	rel. save-load location
Default	Compact	Included	`./shetris/`
Variation-1	Compact	Omitted	`./shetris01/`
Variation-2	Full	Included	`./shetris02/`
Variation-3	Full	Omitted	`./shetris03/`

Table 7.8. Encoding of Variations of Training-Modes

Thus, the impact of the various formation of the observation must be investigated. Specifically, the four variants as captured in Table 7.8 are to be studied.

As an example, the specification of Variation-2, along with the pseudo-code for its corresponding `gym`-space, is provided:

Example 4. Specification of Variation-2 in `gym`

Observations	numbers of	discrete-range	gym-translation
Height x^H	10	$[0; 20]$	`range(21)`^{10}
Elevation x^E	9	$[0; 20]$	`range(21)`^{9}
Holes x^N	10	$[0; 19]$	`range(20)`^{10}
Lines-cleared x^{LC}	1	$[0; 4]$	`range(5)`
`pid` $x^{\texttt{pid}}$	1	$[0; 6]$	`range(7)`
Full Space	`MultiDiscrete([21...21 | 21...21 | 20...20 | 5, 7])`		

7.4.2. *Final Remark.* At this point, the full environment of Shetris conforming to the framework of the interface of `gym` is fully specified. This environment class, internally named `ShetrisEnv`, is implemented in the module `src/rl/shetris/env/shenv.py` of the source-code repository.

8. Solution with `sb3`

8.1. **The three Algorithms.** Due to the specific environment properties of `ShetrisEnv`, not all algorithms of the `SB3`-library are applicable for training. In particular, since both the action and the observation-space are of the `MultiDiscrete` type of `gym`, as shown in Definition 16 and Definition 31, the following two algorithms are viable by default:

Definition 32. Directly applicable algorithms of `SB3` for `ShetrisEnv`

- the Asynchronous-Method [Mni+16], also known as »A2C« and implemented by the class:

```
stable_baselines3.a2c.A2C
```

- the Proximal-Policy-Optimization algorithm, otherwise referred to as »PPO«, as originally introduced in [Sch+17] and realized under:

```
stable_baselines3.ppo.PPO
```

In addition, for the sake comparative completeness as shown in later sections, extra adaptations are made to allow usage of the Deep-Q-Network (DQN) algorithm [Mni+13]; [Mni+15] of `SB3`:

Solution 1. Adaptation of `ShetrisEnv` for `DQN` by `SB3`

As only scalar-valued actions as acceptable for `SB3`'s implementation of DQN, the two-member, vector-input action a of `ShetrisEnv` as seen in Definition 16:

$$a := (a^{\texttt{rot}}, a^{\texttt{posl}})$$

must be inter-convertible with a »flattened«, scalar-value alternative action produced by the `DQN` algorithm.

In Shetris, this is achieved with the following method within the class `ShetrisEnv` itself:

```
# src/rl/shetris/env/shenv.py

class ShetrisEnv:
   ...

   def _action_int_to_np(self, action: int) -> np.ndarray:
       width = self.engine.field.size[1]
       return np.array((action // width, action % width))
```

Listing 11. Reconstruction of vector-action from `SB3-DQN`'s scalar representation

To summarize, training with `SB3` is performed as follows:

Definition 33. Training with `SB3`

The following three algorithms as implemented by `SB3` are used for training: PPO, A2C and DQN. For each of the three algorithms, all four observation modes of Shetris in Definition 31 are further used.

In other words, a total of 12 training modes are deployed.

9. Deep Q-Network

The algorithm Deep-Q-Network (DQN), as one of the three algorithms to deploy by `SB3` seen in Definition 33, is programmed separately to allow more granular control and finer tuning. This self-implemented DQN is then deployed to train the agents $\mathcal{A}^{\text{DQN}}$ for the `ShetrisEnv` environment under the same four observation modes for direct comparison with the `SB3`'s training results. This section documents the essential theoretical and practical aspects during the implementation of DQN.

9.1. Conceptual overview.

A theoretical summary to the technique is provided below. The interested reader is referred to publications such as [WD92]; [Mni+13]; [Mni+15] for more details.

9.1.1. *Decision Policy.*

9.1.2. *Q-Values.*

The underlying mathematical formulation of Q-Learning targets an optimization problem much similar to that of the cost-to-go minimization from classical control-theory[Ber95]. This concept is formalized in this section.

Definition 34. Reward-to-go

At any time-step t_k, the »reward-to-go[13]«, also known as the *return*, is defined as the summation of all future rewards until the ending of the system evolution at t_T:

$$R_{t_k:t_T} := \sum_{t\in[t_k;t_T]} \gamma^{t-t_k} r_t$$

Definition 35. Q-Values

Given the decision-making policy π of the agent $'\mathcal{A}$:

$$x_\star \xrightarrow{\pi} a_\star$$

[13]This term does not seem to be utilized in standard literature, but the correspondence to the »cost-to-go« of control-theory begs its usage.

the Q-value is then defined in [WD92]; [Mni+13]; [Mni+15] as the expected reward-to-go at time t_k of performing the immediate action a_{t_k} given the current observation x_{t_k}, followed by the deployment of the policy π for all future actions until end of evolution $a_{t_{k+1}:t_T}$:

$$Q_{t_k:t_T} := \mathbb{E}\left[R_{t_k:t_T} | x_{t_k}, a_{t_k}, \pi\right]$$

Definition 36. Optimal Q-Values

The optimal Q-Value Q^*, defined as:

$$Q^* := \max_{\pi} \left\{ Q_{t_k:t_T} \right\}$$

is achieved with the optimal, maximizing policy π^*:

$$\pi^* := \arg\max_{\pi} \left\{ Q_{t_k:t_T} \right\}$$

which is subjected to the Bellman's optimality-principle and the (discrete) Bellman-Equation, as shown in [WD92]; [Mni+13].

9.1.3. *Deep Q-Learning.* Deep Q-Networks (`DQN`) formalizes the notion of the combination of deep-learning with Q-learning[WD92]. Specifically, the (potentially) complex mapping of the optimal action-value function Q^* is approximated with a function Q learnable by a deep-network parameterized on the weights θ:

$$Q\left(x, a; \theta\right) :\approx Q^*$$

This framework enables the utilization of the toolkit from deep-learning to find the approximating Q. As the original conceivers pointed out in [WD92]; [Mni+15], and as existing implementations demonstrated [Ste16]; [Ngu18]; [nun21]; [Hil+18], the familiar optimization-techniques and loss-functions from evaluating deep networks are available for Deep Q-Learning.

9.1.4. *Replay-buffer.* Apart from introducing the fusion of Q-learning with deep-networks, Deep Q-Learning also deploys a replay-memory of length N chosen *a priori* to hold the »experience« of the agent of every evolution-step for later random mini-batch sampling.

The usage of such experience replay distinguishes Deep Q-Learning from the original Q-learning algorithm described above and the Temporal Difference (TD) algorithm[Tes95]. In particular, [Mni+13] cites higher sample efficiency and increasing potential for variance explanation due to randomness when sampling from the experience replay. Indeed, in the seminal publication of [Mni+13], outstanding results were achieved with such constructions.

Note 12. Implementation details in Shetris

The fixed size limitation of the replay-buffer, which in itself is intentional as explained in [Mni+13], requires modifications at both ends: insertion at the end and deletion at the beginning. The excellent choice for such operations is provided by `Python`'s standard library `collections.deque`.

The choice of this particular data-structure for abstracting the replay-buffer is also seen in [nun21]; [Ngu18].

9.2. **Implementation details.** Based on the algorithm as delineated in [Mni+13], the accompanying official implementation[14] as published in [Dee15] and user tuned solutions in [Ngu18]; [nun21]; [Kuz18], a Deep Q-Network with `pytorch` is separately implemented under the package `shetris/train/rl/util/dqn/`, containing, in particular, the `network.py` module defining the deep network, as well as the `model.py` module offering the definition and training routine of the Deep Q-Learning algorithm.

9.2.1. *Hyper-parameters.* Without delving further into the intricacies of the Deep Q-Networks, the thesis simply enlists several of the hyper-parameters essential to the algorithm. These are generally surveyed from reference implementations such as [Hil+18]; [Kuz18]; [Dee15].

Solution 2. The network

A network with 1 hidden layer with input and output feature-size of 64 is used. In addition, the output-layer has the apparent final size of 1 for returning the Q-value.

[14] The source-code is phrased in the `Lua` programming-language

The feature-size for the input-layer directly depends on the specific observation-variant, as indicated by the following table:

Variation	First layer size
Default	$4 + 1 = 5$
Variation-1	4
Variation-2	$10 + 9 + 10 + 1 + 1 = 31$
Variation-3	$10 + 9 + 10 + 1 = 30$

As recommended by [And+20] and also seen in [Kuz18]; [Dee15]; [Ngu18]; [nun21], the Adam's Optimizer and the MSE-Loss is used.

It shall be note that `SB3` deploys Huber's-Loss within its implementation of DQN instead. This difference is further explored in Examination 31.

Solution 3. Essential hyper-parameters of DQN in Shetris

ϵ-greedy-algorithm: the initial ϵ-value is to $\epsilon_0 \equiv 1$, decaying linearly to $\epsilon_{\text{final}} = 10^{-3}$ after $2'000$ episodes, as seen in [Ngu18];

discount-factor: $\gamma = 0.99$, conforming to the recommendation of [And+20];

replay-memory:: total step-capacity set to $N = 30'000$; pre-record to one-tenth of the capacity, i.e., $3'000$ steps, at the beginning of training, which translates to approximately 150 episodes based on empirical evidence as described later in Section 11.3.2.

Remark 8. Tuning of the discount-factor γ

The value of $\gamma = 0.99$ as agreed upon under general consensus in [And+20]; [Hil+18]; [Ngu18] is directly used without further modifications. Later Section 16.3 elaborates on its potential tuning.

9.2.2. *Environment Adaptations.*

Definition 37. Specialized pairing of action-observation

To fully adapt the DQN-algorithm to the environment-property of `ShetrisEnv`, the generation of action-observation pairs is specialized.

In particular, given the current action-space and evolution-mechanism as seen in Definition 16 and 15, both the rotational ($A^{\texttt{rot}}$) and the horizontal ($A^{\texttt{pos1}}$) component of the action-space could generally be further reduced: as an example, for the `"O"`-piece, the rotational action-space $A^{\texttt{rot}}$ could be set to 1 instead of the generic default value of 4, which was used to accommodate for other pieces such as the `"T"`-piece. In the particular case of the `"O"`-piece, this effectively shrinks the size of the action-space of $A^{\texttt{rot}}$ to $1/4$, drastically eliminating the redundant »noise« in the action-space during the training-process.

In the self-implemented DQN, similar specialization for the specific piece and rotational configuration is performed for both $A^{\texttt{rot}}$ and $A^{\texttt{pos1}}$ for the exact tailoring of the action-space to the current piece. The reader is encouraged to inspect the exact implementation in the source code.

The impact of the action-observation tuning is investigated in 15.

Remark 9. Increasing size of $A^{\texttt{rot}}$ with increasing `pid`

The reader might note that the rotational action-space $A^{\texttt{rot}}$ monotonously increases when inspecting the seven pieces from `pid`-value of 0 to 6 as enlisted in 1. The is, in fact, not a coincidence, as the number of rotational variations is one of the initial sorting criteria when the `pid`s are defined in Shetris to begin with.

9.2.3. *Training setups.* Each of the four observation-variants as listed previously is trained for the DQN as described above. However, due to the direct interaction with the `pytorch`-library, the observation must be output into its array-type `torch.Tensor`. Passing `False` to the keyword-argument `use_np` when constructing the Observation-Factory allows this transition. The following constructing syntax performs the type conversion:

```
# train/src/rl/shetris/env/reporter/reporter.py

class Reporter:
   def __init__(self, engine: Engine):
      ...
      self.obs_factory = ObsStandard(
         self.engine,
         use_compact_field=<set_as_needed>,
         use_pid=<set_as_needed>,
         # set to False to output torch.Tensor
         use_np=True,
      )
      ...
```

Listing 12. Setup for using the self-implemented DQN

10. Hard-coded Evolution logic

To conclude this chapter on the investigation of various agents, this section provides an alternative solution that requires no training. Originally studied and published in [Lee18], this agent chooses the optimizer-action a^* that maximizes a hard-coded reward-function at every step. The specifics are given below:

Definition 38. The reward-function r

The reward function is calculated with:

$$r := \begin{bmatrix} -0.51 \\ -0.18 \\ -0.35 \\ +0.76 \end{bmatrix} \circ \begin{bmatrix} x^H \\ x^E \\ x^N \\ x^L \end{bmatrix}$$

where the observation components are defined as previously.

Thus follows the agent-definition:

Definition 39. Reference-Agent with Hard-coded logic

At every step, the Agent applies the optimal action a^* in the sense of maximizing the reward r as defined above.

Remark 10. Adaptation details for usage with `ShetrisEnv`

The original work in [Lee18] operates with 1-length preview of the piece-id. Shetris currently disallows such previewing, which, in turn, requires slight adaptations to the search process for the optimal action a^*. Note that this agent is also incorporated as reference in [Ste16].

Also, the observation must be configured in Variation1, i.e., with the compact field-observation, excluding the `pid`. Finally, the observation must output the type `numpy.ndarray`.

The hard-coded nature of the reward-to-action process eliminates the process of training entirely, rendering this agent suitable as a »real-time« comparative criteria for evaluating all trained agents as described previously, as demonstrated later in Section 14.

Part 4. The Presentation of the Results

11. Analysis Framework

11.1. **Re-producing the environment.** All training is performed on the Euler scientific compute cluster with the common access-clearance of ETH-Zürich members. Essential setups required to reproduce the training environment are enumerated in this section.

Solution 4. Loading the Python interpreter

Given the version requirement for `Python` of `3.10.*`, the newer generation of software-stack[15] of Euler must be explicitly loaded[16], followed by manually calling the module-loading commands.

The excerpt of shell-commands in Listing 13 accomplishes the module-setup such that the version requirement of `Python` is fulfilled.

```
$ env2lmod
$ module load gcc/8.2.0; module load python_gpu/3.10.4

# confirm python version
$ python --version
Python 3.10.4
```

Listing 13. Loading essential modules on Euler

With the required python interpreter, the extra packages as listed in the introductory chapter of the thesis can now be installed. With `pipenv`, the process of generating a testing virtual-environment along with dependency installation is fully automated:

Solution 5. Virtual-Environment setup with `pipenv`

```
$ pipenv run
$ pipenv install
```

With the virtual-environment and necessary packages available, the setup is now complete. The virtual-environment is accessible either by acquiring a sub-shell or running the desired command in an one-off fashion.

```
# invoke a sub-shell
$ pipenv shell

# execute a specific command (in the sub-shell)
$ pipenv run <command_to_run>
```

Note 13. Direct execution

Note that some modules contain the following block of code at the end of the file:

```
if __name__ == "__main__":
    ...
```

indicating the possibility of script-style execution.

11.2. **Analysis Metrics.**

[15] Description of the new software stack: https://scicomp.ethz.ch/wiki/New_SPACK_software_stack_on_Euler

[16] In the older software-stack, `$ python` is linked to `python2.7`, whereas `python3` executes `python3.6`.

11.2.1. *Training setup.* Despite the capability of `ShetrisEnv` for modifications and extensions, the behavior of agents acting under mostly guideline-conforming conditions is studied. In particular, the following parameters are deployed for training:

Solution 6. Training setup

Whereas the hard-coded nature of spares the process of training, the following setup is used for training the `SB3` and `DQN` agents:

- the field is always chosen to be of the standard-size, i.e., 20 blocks in height and 10 blocks in width;
- the piece-id generator is set to the guideline-compliant, shuffled 7-bag algorithm.

Crucial to the specification of the training-process, the termination-criteria is given below:

Solution 7. Management of training resources

The training-process is terminated at the time of occurrence of either of the following two events, whichever happens first:

- 10 million evolution-steps are taken by calling the method `ShetrisEnv.step()`;
- 40 hours of wall-time have elapsed on the EULER-cluster.

11.2.2. *Testing setup.* The following framework is used for the testing of an agent $\mathcal{A}$

Definition 40. Testing framework for agent $\mathcal{A}$

Due to the deep integration of the field shape within the environment itself at creation time as seen in Table 7.4 and Table 7.2, the field shall maintain the standard size 20×10.

Given the flexible setup capability of the piece-id generator established previously in 7, it is now varied to introduce various training scenarios to the testing facility: On the one hand, non-random generators are deployed:

$$\texttt{pid}\text{-Bag}\,(\text{non-random}) \in \begin{cases} \texttt{"O"} \\ \texttt{"I"} \\ \texttt{\{"O", "I"\}} \\ \texttt{\{"O", "I", "T"\}} \\ \text{7-bag} \end{cases}$$

And for generators with randomness, i.e., with shuffling, the following bags are used:

$$\texttt{pid}\text{-Bag}\,(\text{random}) \in \begin{cases} \texttt{\{"O", "I"\}} \\ \texttt{\{"O", "I", "T"\}} \\ \text{7-bag} \end{cases}$$

Remark 11. Justification of the choice of generator

Bags not including all 7 pieces include only the three axis-symmetrical pieces (`"O"`, `"I"` and `"T"`), where non-random sequences are intuitively solvable by human players. It is thus expected that a well-behaved agent shall demonstrate decent performance on all such testing scenarios.

In addition to the standard, shuffled 7-bag, other random generation mechanisms are included for comparison with performances achieved with their non-random counterparts.

The non-random 7-bag is included as a portably re-producible testing framework that is simple to setup and non-trivial to solve.

11.3. **Measures and Thresholds.**

11.3.1. *Two Measures.* For the Tetris-game, one observes the length of an episode as the intuitive measure for evaluating the performance of an agent, i.e., the number of calls to the stepping function as previously described, before the episode is terminated with the »game-over« signal. Given the Definition 13 of the time-stepping scheme, the length of an episode translates directly to the number of new pieces n_P as returned by the generator.

In similar fashion, the positively correlated number-of-lines-cleared of an episode n_L further constitutes another performance indicator. The two measures n_P and n_L are also utilized in [Lee18]; [nun21]; [Ngu18].

n_P	Training Impact	General Assessment
< 20	Negative	Insufficient
$[20; 50]$	Positive	
$[50; 100]$		
> 100		Successful

TABLE 11.1. Guidelines of Agent-Assessment based on n_P-values

11.3.2. *Three Thresholds.* Of more practical utility are the following two critical thresholds for the metric n_P based on heuristic reasoning:

Conjecture. *Thresholds for n_P for the standard field*

Given the standard field-size of 20×10, the following two rules-of-thumb provide rapid insight into the performance of an agent at the Tetris-game:

Random-Performance-Threshold: *based on empirical evidence, the purely random-acting agent achieves the performance of number of pieces per episode n_P of approximately 20;*

One-Board-Threshold: *fundamental stacking behavior is acquired when the n_P value approaches 50, the theoretical number of 4-block-pieces*[17] *required to fill the board of $20 \times 10 \equiv 200$ blocks* once *in its entirety;*

Two-Board-Performance: *with the n_P value of 100, the board would theoretically have been filled to its capacity* twice*, requiring non-coincidental ability to produce full-lines.*

The four regions induced by the segregation through the three thresholds lead to the following guidelines utilized in SHETRIS to evaluate the overall performance of an acting agent:

In essence, the n_P-value offers immediate feedback to the assessment of an agent $\mathcal{A}$:

Definition 41. Heuristic criteria for agent success in SHETRIS

The agent $\mathcal{A}$ is defined as decently performing with three-digit n_P-values, i.e.:

$$n_P > 100$$
$$\iff \text{Acceptable Performance of } \mathcal{A}$$

With the preparation for defining the framework of analysis, the performance of the individual agents is studied in the following sections.

12. AGENT BY `sb3`

12.1. **General setup.** General setups for training models with `SB3` are described as follows:

Solution 8. Recapitulation of training setup

As explained in the description 33, three algorithms implemented by the `SB3` library are used: PPO, A2C and DQN. Each algorithm is trained for all four variations of observations.

To monitor the training progress, periodic saving is performed after every $10'000$ steps. Specifically, apart from saving the parameters of the model at the particular time, extra logging is collected with `sb3`'s internal logging mechanism to `TensorBoard`, including the episode-length and the episode-reward that are used for the purpose of analysis in this section, which indicate the number of pieces n_P and the number of lines n_L due to the correlation as established in the opening Section 11.3.1.

Remark 12. General access of the training results

To further facilitate public access to the results presented in this thesis, logs created by `TensorBoard` have been uploaded to `TensorBoard.dev` and are publicly accessible without a Google-account. The interested reader is encouraged to visit the site under the provided links for interactive inspection of the results.

[17] Otherwise known as »Tetriminos« or »Tetrominos«

Variation		n_P	r	Training-time	Tensorboard
Default		42	350	18h 49m (9h 10m of interruption)	Link
Variation-1	↯	[26; 27]	[40; 55]	4h 21m (5Mil steps)	Link
Variation-2		52	760	20h 13m (8h 29m of interruption)	Link
Variation-3	↯	[29; 31]	50; 65	4h 9m (5Mil steps)	Link

TABLE 12.1. Result Statements of SB3-PPO, first run

Variation		n_P	r	Training-time	Tensorboard
Default		42	440	8h 24m	Link
Variation-1	↯	[27; 28]	45; 55	8h 49m	Link
Variation-2		55	770	10h 29m	Link
Variation-3	↯	[30; 32]	75; 80	8h 12m	Link

TABLE 12.2. Result Statements of SB3-PPO, second run

12.2. **PPO.**

12.2.1. *First run.* Two separate runs with PPO are conducted, with the first run composed of 5 million steps for Variation-1 and Variation-3, instead of the general threshold of 10 million steps. Note also that training for the Default-mode was interrupted for 9 hours and 10 minutes, and 8 hours 29 minutes for Variation-3.

The results are provided in Table 12.1, where the best performing variation is highlighted with a square surrounding box. Also, the symbol ↯ shall denote strong oscillation of performance at the point of training termination, and a range is provided instead of a scalar to indicate the value neighborhood.

12.2.2. *Second run.* In the second run, the usual framework of 10 million training steps is deployed. The training progress is documented in Table 12.2.

12.2.3. *Brief analysis.*

Solution 9. Result statement of SB3-PPO

Both variants without observing the pid, i.e., the Variation-1 and Variation-3, display oscillations at training termination. Whereas both variations without observing the demonstrate oscillation throughout the training process without convergence to unique values for the measures n_P and n_L; in contrast, the Default and Variation-3 mode both demonstrate a smoother training process with convergence to fixed values for n_P and n_L

The best performing agent is achieved with Variation-2 of the second run: the value for average episode length n_P terminates at approximately 55, scoring the total reward value of $r = 770$, approximately doubling the n_P-value of the worst performing agent of Variation-1.

12.3. **A2C.**

Solution 10. Result statement of SB3-A2C

At a glance of the Table 12.3, all agents trained by A2C display higher levels of oscillation compared to those by PPO in the previous section. In particular, a noticeable jump is displayed with the TensorBoard-log after approximately 7.7 steps or 9 hours of training for the Default variation algorithm.

Again, the best performance is achieved at Variation-2, whereas Variation-1 again finishes with the poorest performance.

12.4. **DQN.**

Solution 11. Result statement of SB3-DQN

Variation		n_P	r	Training-time	Tensorboard
Default		37	330	10h 22m	Link
Variation-1	↯	[26; 28]	[35; 45]	10h 44m	Link
Variation-2		46	450	12h 39m	Link
Variation-3	↯	[33; 35]	[75; 95]	10h 52m	Link

Table 12.3. Result Statements of SB3-A2C

Variation		n_P	r	Training-time	Tensorboard
Default		27	70	8h:54m	Link
Variation-1	↯	[21; 23]	[20; 30]	8h 38m	Link
Variation-2		45	370	11h 37m	Link
Variation-3		43	210	9h 43m	Link

Table 12.4. Result Statements of SB3-DQN

Variation	PPO	A2C	DQN	Best Alg.
Default	42	37	27	PPO
Variation-1	27	27	22	PPO
Variation-2	55	46	40	PPO
Variation-3	31	34	43	DQN
Best Var.	Var-2	Var-2	Var-3	

Table 12.5. Result Statements for n_P of all SB3-agents. Note: PPO at Variation-2 performs best.

Table 12.4 demonstrates the results of the four agents trained with DQN by SB3. Although Variation-1 still demonstrates oscillation, Variation-3 concludes surprisingly smoothly in comparison to the previous two algorithms.

Similar to PPO and A2C, Variation-2 performs the best, whereas the lowest n_P and r-values are obtained in training mode Variation-1.

12.5. **Summary.** Several closing statements on the SB3-trained agents are presented below:

12.5.1. *On training resources.* Every training variation finishes with plenty of time to spare. In fact, no training iteration spent more than 13 hours to finish executing 10 million steps. In this regard, A2C appears to require more wall-time for training than PPO and DQN under similar circumstances.

On the other hand, Variation-2 takes longer to finish than the other three training modes for all algorithms. This does not seem surprising, as this variant utilizes the vastest observation-space, as defined previously in Example 4. Interestingly, substantial reduction of training time is observed with Variation-3, which, in comparison, only removes the pid component.

On the subject of training time, training time, the interested reader shall further study the »FPS« section in the Tensorboard data accessible via the aforementioned links.

12.5.2. *General Results.* As the final summary to conclude the training results with SB3, one first observes the general number of pieces n_P values in Table 12.5 and concludes:

Solution 12. Result statement: all SB3-agents

By inspecting each row individually, one concludes that, with the exception of Variation-3, PPO is the best performing algorithm, which corresponds to the statement in [And+20].

Variation	Num-Steps ($\times 10^6$)	Num-Episodes ($\times 10^3$)
Default	5.311	16.88
Variation-1	5.294	10.04
Variation-2	6.701	45.64
Variation-3	6.801	40.01

Table 13.1. Number of steps and episode executed at training-termination

A vertical scan of the table, on the other hand, reveals that the training mode Variation-2 yields the highest scores, whereas the lowest values are found at Variation-1. The difference is especially pronounced for PPO. One shall note from Table 7.8 that Variation-2 translates to the highest length of the observation-space, whereas Variation-1 utilizes the lowest amount of components for the observation.

However, the striking message is the uniformly unsatisfactory performance of all agents trained with `SB3`: Even the best-performing agent, PPO trained on Variation-2, as marked in Table 12.5, barely surpasses the one-board-threshold as established in the heuristic argument 11.3.2. Instead of performing extra testing, one concludes that the agents as presented above fail to qualify as being successfully trained.

13. Agent by Self-Implemented DQN

13.1. **General Remarks.** Similar to the presentation of the results by `SB3` previously, parameters logged by `TensorBoard` are uploaded for public access:

https://tensorboard.dev/experiment/eA6s6uyXRFmdGCM6p6JcWg/

13.2. **Training Results.**

13.2.1. *Progress at Termination.* Unlike solutions by `SB3`, the 40 hours restriction expires before the threshold of 10 million steps is reached. The following table enlists the number of steps and episodes executed until training termination:

One observes that both training-modes using the compact field-observation (Default mode and Variation-1) terminated with considerably fewer number of steps and episodes, indicating longer durations on average in comparison to their counterparts with full field-observations (Variation-2 and Variation-3). The intuitive conclusion is thus better performance with compact field-observation.

13.2.2. *Log Excerpts.* Results for the two training modes using the compact field-observation (Default-mode and Variation-1) are demonstrated in Illustrations 13.1, whereas Illustrations 13.2 show that of Variation-2 and Variation-3, both using full field-observation.

The highly turbulent training process is obvious: each agent has achieved high n_P and n_L values at some point during the training process. To shed light on the oscillatory training process uniformly observed above, the average number of pieces n_P and the average number of lines cleared n_L are calculated with the initial $3'000$ »burn-in« episodes trimmed:

13.2.3. *Average n_P and n_L values.*

Solution 13. Result statement on self-implemented DQN

By observing Table 13.2, one draws the conclusion that aligns with the initial conjecture in Section 13.2.1: The Default-mode and Variation-1 out-perform the two other variants using full-observation.

Also, in this particular sense of average performance throughout training, the training-modes leading to the best and worst performing agent are exactly the opposite in comparison to the analysis of `SB3` in 12: here, the observation-mode Variation-1 with the fewest amount of components lead to the best-performing agent, more than tripling the n_P-value of Variation-2 with the highest observation component count.

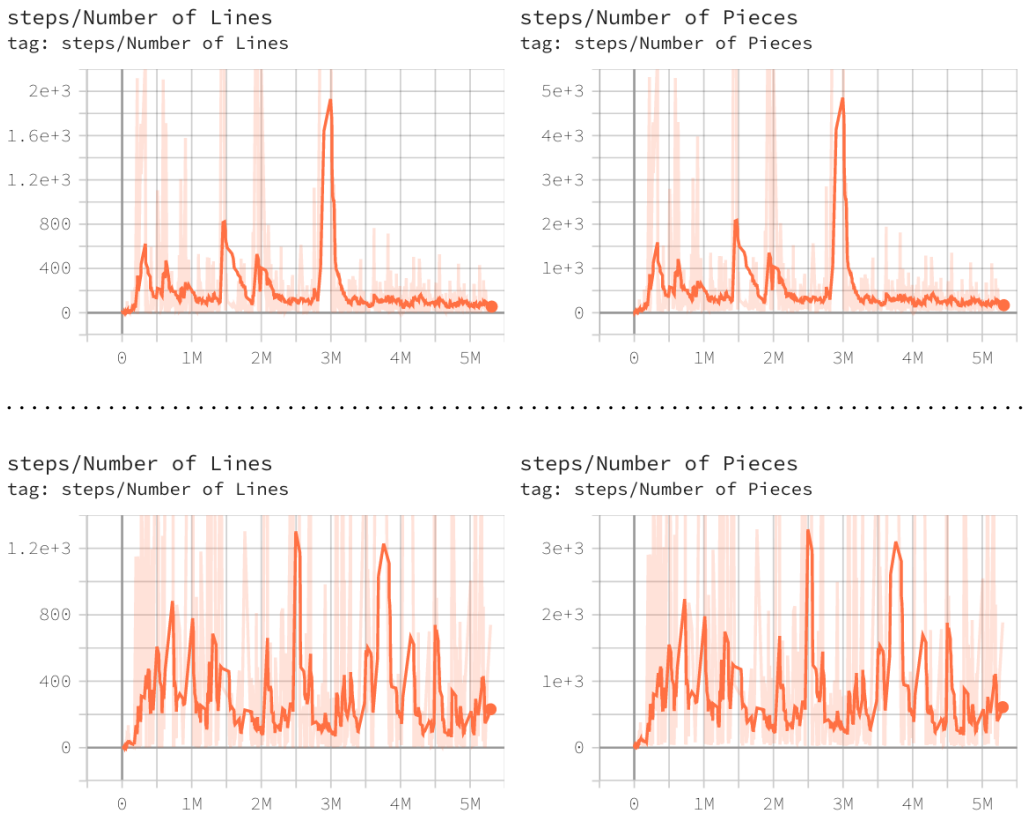


FIGURE 13.1. `Tensorboard` logs: Default-mode (upper) and Variation-1 (lower) [compact field-observation]

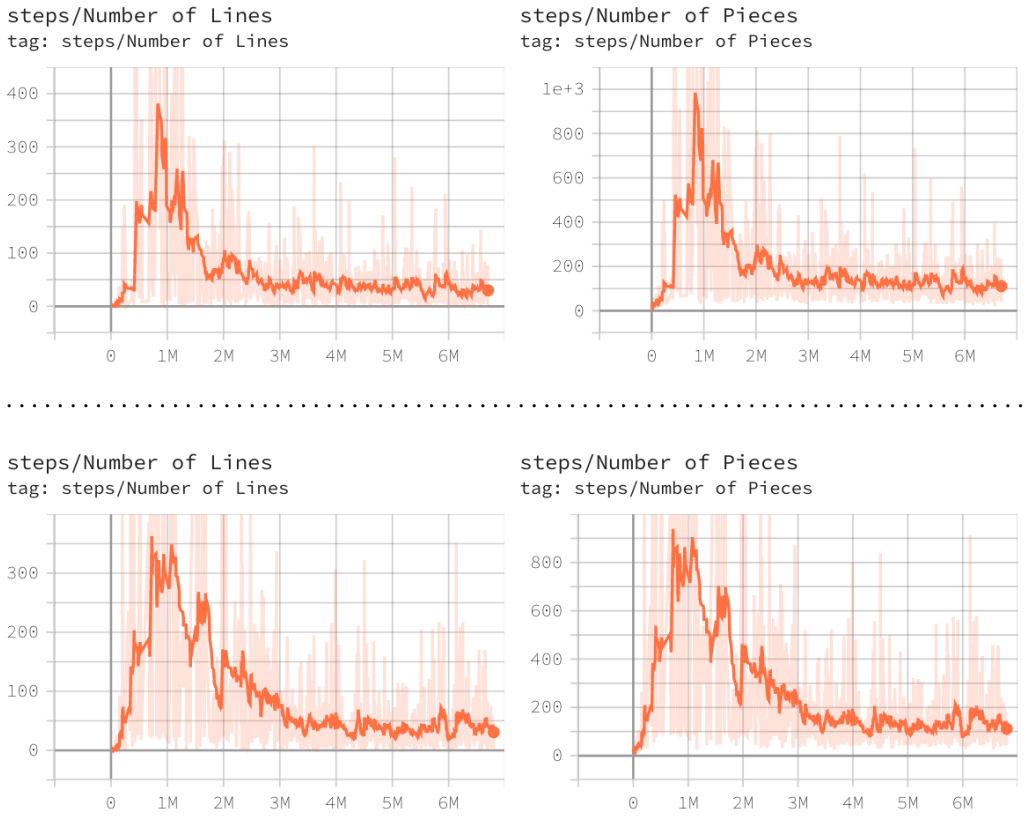


FIGURE 13.2. `Tensorboard` log for Variation-2 (upper) and Variation-3 (lower) [full field-observation]

Variation	n_P	n_L
Default	367.94	134.78
Variation-1	669.04	255.43
Variation-2	154.05	48.03
Variation-3	180.14	58.60

TABLE 13.2. Average n_P and n_L from the $3'000$ training-episode onward

`pid`-Bag	n_P
`"O"`	17
`"I"`	> 500
`"O", "I"`	> 500
`"O", "I", "T"`	307
Full-7	97

TABLE 13.3. DQN: n_P for various non-random generators [Variant-1]

`pid`-Bag	Average	Max	Min
`"O", "I"`	> 500	> 500	> 500
`"O", "I", "T"`	> 500	> 500	143
Full-7	215.3	> 500	41

TABLE 13.4. DQN: n_P for various random generators, averaged over 20 runs [Variant-1]

13.3. **Testing Results.** With testing framework established previously, one observes the testing results in Table 13.3 for non-random `pid`-sequences and Table 13.4 for randomly generated `pid`-values using the best-performing agent, i.e., the one trained on Variation-1. One thus concludes:

Solution 14. Training performance of self-implemented DQN:

The agent delivers satisfactory results in 3 of the 5 non-random testing scenario, only failing at the constant, unique `"O"`-sequence.

On the other hand, the agent shows excellent performance when averaged over 20 test episodes. Of worry is the substantial fluctuation of the agent facing the standard, shuffled 7-bag generator, with disappointing worst-case performance. However, given the average n_P-value vastly exceeding the »Two-Board-Performance« Threshold, as established in Section 11.3.2, one concludes that the agent trained on Variant-1 with the self-implemented DQN demonstrates successful training on the environment `ShetrisEnv`.

13.4. **Analysis.** To understand the performance improvement of the self-implemented DQN Agent in comparison to the DQN Agent by `SB3`, the following arguments are concluded:

Solution 15. Sources of Improvements of self-implemented DQN over DQN by `SB3`

On other one hand, the Loss of the self-implemented DQN follows the recommendation of empirical studies conducted by [And+20] by choosing the MSE-loss instead of the Huber's-Loss, which is explicitly rejected by the same research publication.

On the other hand, per the design property of self-implemented DQN as explained in 37, the specialized, »noiseless« action-space with the corresponding observation-space is utilized for the self-implemented DQN. In comparison, training with `SB3` utilizes the generic action-space with redundant adaptability as specified in Definition 16.

Bag	n_P
`"O"`	> 500
`"I"`	> 500
`"O", "I"`	> 500
`"O", "I", "T"`	416
Full	> 500

Table 14.1. Hard-coded: n_P for various non-random generators

Bag	Shuffled	Average	Max	Min
`"O", "I"`	Yes	> 500	> 500	> 500
`"O", "I", "T"`	Yes	> 500	> 500	> 500
Full	Yes	> 500	> 500	311

Table 14.2. Hard-coded: n_P for various random generators, averaged over 20 runs

This improved accuracy of the action-observation pair renders the agent trained by the self-implemented DQN more tuned to the intricacies of the underlying environment `ShetrisEnv`, possibly contributing the vastly superior result compared to the agents trained by `SB3`.

14. Agent by Hard-coded Evolution

As a final note, the excellent performance of the reference-agent specified in Definition 39 is readily visible in Table 14.1 and Table 14.2.

One shall note that under various testing setups, the performance of the agent trained on the self-implemented DQN delivers similarly pleasing results. However, the room for improvement is on immediate display under testing sequences such as the static `"O"`-sequence and the full 7-bag generator, both non-random and shuffled.

Activities	Duration (×week)
Theoretical formulation; Structuring of project	2
Survey, testing and inspection of `gym`, `SB3`	1.5
Implementation of Shetris's core modules	approx. 2.5
Adaptation to `gym`; initial training with `sb3`	approx. 1.5
Otherwise engaged: break from project	4
Engineer further Observation/Reward pairs	1
Implementation of Evolution-Algorithm; DQN	2
Collection of results; compose thesis	1.5

Table 15.2. Time-stamps of key activities

Part 5. The Outlook of this Project

15. Executive Summary

15.1. **The gist.** At this point, the Shetris-Project, at the time of writing, is summarized as follows:

The uneven quality of various online implementations of Tetris used for machine-based learning, coupled with the disappointing training results from direct usage of raw game files binaries as noted in [Liu20] led to the realization of the necessity of programming the Tetris-game from scratch. The consideration of the diversity of existing renditions of the game and the goal for adaptation to the maximal amount of rule-sets drive the implementation process of the core components of Shetris, including, in particular, the `engine` that abstracts the main logic of the game.

With the theoretical MDP-formulation of the Tetris, an environment conforming to the `gym`'s interface for the description of an decision-making agent is created. Three state-of-the-art algorithms of Reinforcement-Learning as implemented by the standard library Stable-Baselines3 (`SB3`) are utilized in four training-modes of the environment. In addition, a Deep-Q-Network tuned for the action and observation-structure specific to Tetris is implemented separately, and trained under the same conditions as for direct comparison.

Performance measures are established with both heuristic reasoning and an established hard-coded, purely reward-driven reference agent. While the raw usage of algorithms from the `SB3` library yields unsatisfactory results, the tailored DQN solution achieves marginally inferior performance in comparison to the reference agent under most testing scenarios. However, visible gaps are seen in other evaluation setups, leading to room of followup research and further improvement.

15.2. **Timeline.** The administrative details of this thesis can be retrieved under the course-catalogue of ETH Zürich under the registration number `401-3990-18L`.

The total execution time is estimated at approximately 3 months of full-time work, including the drafting of the thesis and the composing of the source-code. Specifics are found in Table 15.2.

16. Epilogue

16.1. **Original conception.** TheShetris-project was initially conceived when the author was battling for the top spot on the Haunted-mode of Tetris-Ultimate[18] on PlayStation, where pieces become invisible after initial appearance at the top of the game-field.

The immediate reaction after conducting a general survey of the status-quo of current researches on applying Reinforcement-Learning to the Tetris-game was mixed: on the one hand, several exciting projects promise the possibility of successful adaptation of the algorithms of the field to the game. On the other hand, such

[18] https://tetris.wiki/Tetris_Ultimate

project-repositories generally sport sloppy implementations of Tetris and display deficiencies on the matter of fundamental principles fundamental software engineering.

The decision was then made to first embark on the programming of the game as a stand-alone component, before transitioning to the training process within the reinforcement-learning framework, along with the theoretical scrutiny of Tetris as a whole towards the mathematical formulation of the mechanism of the game-play.

16.2. **Mixed sentiments.** Initially, this thesis-project envisioned the tackling of a variety of Tetris rule-sets with near expert-level finesse. However, as the project progressed, the lofty ambition soon subsided: the time-restriction embedded in the framework of a Bachelor-Thesis dictates down-scaling of the span of the goals and of the methodologies. Furthermore, the agent trained on the tuned algorithm still left a lot to be desired in some basic testing scenarios and failed to out-perform existing implementations.

However, the gulf between the original envisioning and the current progress also initiates room for improvement. Given its fundamental design concept of extendability and modularity, the fundamental structure of the source-code of Shetris welcomes modifications and additions of functionality. With this, various directions of future work are conceivable:

16.3. **Future work.** The currently used, trimmed action-space as specified in Definition 16 shall be enriched to accommodate the full action-space of the Tetris-game. The effect of piece-previewing, as mentioned in Section 4.3 shall be studied and compared to the current version without such look-ahead.

For the environment itself, reward mechanisms beyond the current formula as defined in Definition 20 shall be tested and analyzed, as suggested in general RL literature such as [Dew14]. In fact, some two-state rewards, i.e., rewards calculated based on two different states were originally explored in Shetris, but were neither further pursued nor documented in this thesis due to unsatisfactory results.

Furthermore, it shall be envisioned to allow side-by-side game-plays among multiple agents, in particular, the split-screen setup between human-player and software-agents shall be considered, as well as the potential for deploying multi-player[19][20] game-modes for dynamic intra-agent training and testing.

Finally, the training process itself shall be further stream-lined for the purpose of more detailed logging, more accurate parameter tuning and the possibility of effective usage of vectorized training on high-performance computing clusters. Also, more intensive tuning of hyper-parameters, as advocated by [And+20], shall be performed if freedom of resource and and time shall so permit.

The enumeration is by no means exhaustive. The final, and probably the most resounding message of the Shetris-Project thus lies within the breadth of the spanning coverage of continual research and the depth of the potential promise for future exploration.

17. Acknowledgements

To the supervisor of this thesis, Prof. Dr. J. Buhmann: my deepest appreciation for the acceptance of the original idea of this thesis in its earliest infancy, for the inspiring and philosophical approach to the field of machine-learning as a whole, and, of course, for granting the much needed extension for completion;

To my advisors, Ivan, Ami and Eugene: my heart-felt thanks to the extended sessions of talks, the continuous support and the constructive suggestions;

To my family and my friends: my sincere gratitude for the unyielding support and care;

Finally, to J. Neubauer: all my respect for his natural mastery of the game and brilliant deliveries in the most competitive environments. Rest in piece, *maestro*.

[19] Discussion of multi-player game-play https://tetris.wiki/Multiplayer_techniques
[20] Scoring-mechanism of multi-player mode https://tetris.wiki/Garbage

References

1. Andrychowicz, M. *et al. What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study* 2020. https://arxiv.org/abs/2006.05990.
2. Bellman, R. *Dynamic Programming* 1st ed. (Princeton University Press, Princeton, NJ, USA, 1957).
3. Bertsekas, D. P. *Dynamic programming and optimal control* (1995).
4. Brockman, G. *et al. OpenAI Gym* eprint: arXiv:1606.01540.
5. Brzustowski, J. *Can you win at TETRIS?* (University of British Columbia, 1992). https://open.library.ubc.ca/collections/ubctheses/831/items/1.0079748.
6. Carr, D. *Applying reinforcement learning to Tetris* in (2005).
7. Deepmind, G. *Lua/Torch implementation of DQN* https://github.com/deepmind/dqn. 2015.
8. Demaine, E. D., Hohenberger, S. & Liben-Nowell, D. Tetris is Hard, Even to Approximate. *CoRR* **cs.CC/0210020.** https://arxiv.org/abs/cs/0210020 (2002).
9. Demaine, E. D. *et al.* Total Tetris: Tetris with Monominoes, Dominoes, Trominoes, Pentominoes,... *Journal of Information Processing* **25,** 515–527 (2017).
10. Dewey, D. *Reinforcement Learning and the Reward Engineering Principle* in *AAAI Spring Symposia* (2014).
11. Domain, P. *Tetris.wiki* https://tetris.wiki. 2015.
12. Dulac-Arnold, G., Mankowitz, D. & Hester, T. *Challenges of Real-World Reinforcement Learning* 2019. https://arxiv.org/abs/1904.12901.
13. Effort, C. M. *Official Wiki on the Euler-cluster*
14. Ghosh, D. *et al. Why Generalization in RL is Difficult: Epistemic POMDPs and Implicit Partial Observability* 2021. https://arxiv.org/abs/2107.06277.
15. Hill, A. *et al. Stable Baselines*
16. koryan. *Koryan's Tetris Wiki* http://www.din.or.jp/~koryan/tetris/d-tst0.htm. 2002.
17. Kuzovkin, I. *Tweaked Lua/Torch implementation of DQN* https://github.com/kuz/DeepMind-Atari-Deep-Q-Learner. 2018.
18. Lee, Y. *A Tetris AI written in Javascript* https://github.com/LeeYiyuan/tetrisai. 2018.
19. Levine, S., Kumar, A., Tucker, G. & Fu, J. *Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems* 2020. https://arxiv.org/abs/2005.01643.
20. Liu, H. *Learn to Play Tetris with Deep Reinforcement Learning* in (2020).
21. Melax, S. *Reinforcement Learning Tetris Example* https://melax.github.io/tetris/tetris.html. 2018.
22. Mnih, V. *et al.* Asynchronous Methods for Deep Reinforcement Learning. https://arxiv.org/abs/1602.01783 (2016).
23. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518,** 529–533 (2015).
24. Mnih, V. *et al. Playing Atari with Deep Reinforcement Learning* 2013. https://arxiv.org/abs/1312.5602.
25. Nguyen, V. *Deep Q-learning for playing tetris game* https://github.com/uvipen/Tetris-deep-Q-learning-pytorch. 2018.
26. nuno-faria. *A deep reinforcement learning bot that plays tetris* https://github.com/nuno-faria/tetris-ai. 2021.
27. OpenAI. *OpenAI Spinning Up in Deep RL* eprint: arXiv:1606.01540.
28. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. *Proximal Policy Optimization Algorithms* 2017. https://arxiv.org/abs/1707.06347.
29. Sobrecueva, L. *Tetris OpenAI environment* https://github.com/lusob/gym-tetris. 2018.
30. Stevens, M. *Playing Tetris with Deep Reinforcement Learning* in (2016).
31. Tesauro, G. Temporal Difference Learning and TD-Gammon. *J. Int. Comput. Games Assoc.* **18,** 88 (1995).
32. Watkins, C. J. C. H. & Dayan, P. *Q-learning* 1992. https://doi.org/10.1007/BF00992698.
33. Willing, C. *et al. The uncompromising Python code formatter*