University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Bachelor's thesis

Reinforcement Learning for Optimizing Agent Strategies

Plzeň 2020 Seják Michal

1. **Study the field of reinforcement learning with focus on the application of neural networks and optimalization of game strategies.**
2. **Create a suitable custom environment for optimizing the agent's strategy.**
3. **Implement the mechanism of reinforcement learning using the Tensorflow framework.**
4. **Design a metric for evaluating the agent's performance and critically asess the achieved results.**

Declaration

I hereby declare that this bachelor's thesis is completely my own work and that I used only the cited sources.

Plzeň, 7th May 2020

Seják Michal

Acknowledgement

I would hereby like to express gratitude towards my supervisor Ing. Miloslav Konopík, Ph.D. for his guidance and valuable suggestions regarding my work. Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Abstract

Reinforcement learning agents are one of the best methods of general problem solving. The algorithm AlphaGo Zero (AZ) in particular achieved state-of-the-art results in solving multiple board games. However, it is suited only for solving adversary deterministic environments and finds few real-life applications, as finding complete information about real-life processes is next to impossible. In our work, we analyze how exactly does AZ function and how it can be adjusted for solving non-adversary stochastic environments, while introducing a redundancy checking technique to prune the state tree more effectively. Finally, we design a custom environment and examine how the simple DQN algorithm compares to the adjusted AZ both with and without redundancy checking, showing that the version utilizing the redundancy checking heuristic remarkably outperforms both the DQN and the unamplified AZ.

Abstrakt

Agenti zpětnovazebného učení v současnosti patří mezi nejlepší způsoby, jak řešit obecné úlohy. Konkrétně algoritmus AlphaGo Zero (AZ) se v hraní mnoha deskových her drží v současnosti na nejvyšších příčkách. Nicméně, hodí se pouze na práci s deterministickými adverzálními prostředími a jako takový nenachází ve skutečném světě mnohá uplatnění, jelikož obdržení veškeré informace o běžných procesech je takřka nemožné. V této práci analyzujeme způsob, jakým AZ dosahuje svých výsledků a jak lze tento algoritmus upravit tak, aby řešil obecné stochastické neadverzální problémy, přičemž zavádíme techniku kontroly redundance, pomocí níž lze efektivněji prořezávat stavový strom. Na závěr navrhneme vlastní prostředí a otestujeme, jakých výsledků dosahuje obyčený algoritmus DQN ve srovnání s upraveným AZ bez a s kontrolou redundance, kde ukážeme, že verze AZ využívající kontrolu redundance dosahuje mnohem kvalitnějších výsledků, než ostatní dva algoritmy.

Contents

1 Introduction **10**

2 Reinforcement learning primitives **11**

2.1 Environment 11
2.1.1 Definitions 11
2.1.2 Properties 12
2.1.3 Reward 13
2.1.4 Bounding states and the discount factor 14
2.2 Agent 15
2.2.1 Policy function 16

3 Neural Networks **17**

3.1 Tensors 17
3.1.1 Examples 19
3.1.2 Usage 19
3.2 Tensor operations 20
3.2.1 Broadcasting 20
3.2.2 Indexing 20
3.2.3 Slicing 21
3.2.4 Matrix-like multiplication 22
3.2.5 Convolution 22
3.2.6 Batch Normalization 22
3.2.7 Bias 23
3.2.8 Activation function 23
3.2.9 Dropout 24
3.3 Function approximation 24
3.3.1 Cost function 24
3.3.2 Optimization 25
3.4 Regularization 26

4 Existing agents **28**

4.1 Deep Q-learning network (DQN) 28
4.1.1 Q-value 28
4.1.2 The Q-learning algorithm 29
4.1.3 Amplification by neural networks 30
4.2 Expecti-max Monte Carlo Tree Search (MCTS) 31

4.2.1 Selection . . . 32
4.2.2 Expansion + Rollout . . . 33
4.2.3 Update . . . 33
4.2.4 Finalization . . . 34
4.3 AlphaGo Zero (AZ) . . . 34
4.3.1 Introduced MCTS adjustments . . . 35
4.3.2 The AZ algorithm . . . 36

5 Implementation analysis **39**

6 Custom environment design **40**
6.1 Basics . . . 40
6.1.1 Map . . . 40
6.1.2 Entities . . . 40
6.1.3 State transitions . . . 40
6.2 The Infected . . . 41
6.2.1 Appearance and position . . . 41
6.2.2 Movement . . . 42
6.2.3 Aggression . . . 44
6.3 The buildings . . . 49
6.3.1 Appearance and position . . . 49
6.3.2 Types . . . 49
6.4 Simulator . . . 51
6.5 Actions . . . 52
6.6 Control actions . . . 53
6.7 State output . . . 54
6.8 Data structures . . . 55
6.9 Communication . . . 57
6.9.1 Protocol . . . 57
6.10 Main loop, parameter discussion . . . 59

7 Agents **61**
7.1 DQN, AZ (EXPI) . . . 61
7.2 Redundancy augmentation . . . 61
7.2.1 Motivation . . . 62
7.2.2 Realization . . . 62

8 Experiments **65**
8.1 Network model . . . 65
8.2 Agent parameters . . . 65
8.2.1 DQN . . . 65

8.2.2 EXPI/RE 66
8.3 Training procedure 66

9 Conclusion **68**

10 Common abbreviations **69**

Bibliography **70**

1 Introduction

The field of artificial intelligence (AI) is a scientific branch building on foundations of mathematics, biology, linguistics and ultimately, computer science. Using AI, humans have created machines capable of *image recognition* [38], understanding *human language* in both textual [11, 12] and audial form [35], or swapping a human face in a video for another [21], with human observers unable to recognize the forgery. AI systems classified as *narrow* AI are applied to all of the above tasks; their key feature being the inability to adapt, to solve problems other than those they has been specifically designed for.

Reinforcement learning (RL) itself is merely a tiny fraction of the vast domain that is AI, making use of mathematical abstractions called *agents* and *environments* to create decision-making algorithms which can then be used as a general solution to any problem which fits the algorithm's respective set of constraints. These constraints are - thanks to the abstract nature of the environment - always very loose, allowing for one such algorithm to solve a broad range of real-world problems. RL algorithms can be applied to everything that includes decision-making, the most relevant being different branches of logistics, robotics and data processing. Although humans are not known to have created *general* AI yet, what we will explore in this work is a step towards such a construct.

The goal of our work is to explore the potential of RL by designing an agent capable of solving our custom stochastic strategy game environment. This agent will be provided only with the game's rules, without any pre-training or prior game knowledge. The reason we specifically focus on stochastic environments is their inherent complexity, which limits the range of possible agents applicable for the task.

2 Reinforcement learning primitives

At the core of reinforcement learning lies the environment-agent feedback loop. Its individual components cannot be truly separated, for one largely influences the other, nonetheless, for the purpose of this work, they shall be analyzed separately.

2.1 Environment

In the context of RL, an environment is a dynamic system model, called a Markov Decision Process, or MDP, and has these two essential properties:

- at any instant t, it is found in a *state* s_t
- its state can be influenced by outside *actions* a_t, inducing an inner transition $s_t \to s_{t+1}$

RL designers use it as an abstraction layer over any sort of compatible time-evolving process. Any conceivable entity that fulfills the rules above can be modeled as an RL environment. The purpose of RL, however, is to automatically design a strategy, that is, to construct an *oracle* which observes the environment's state and outputs an action which should be taken next. This target strategy is, in some sense, optimal: trading stocks in order to maximize profit, toggling traffic lights in order to minimze the total waiting time for all drivers, etc. To abstract this notion of optimality, the *reward* comes into play. One of the essential characteristics of an environment is that after transitioning to s_{t+1} as a result of an action a_t, it produces an observable *reward* value $r_t \in \mathbb{R}$. Then, we say that a strategy is **optimal** when the sum of individual rewards observed at all transitions **is at least as big** as when following any other strategy. [10]

2.1.1 Definitions

These properties give rise to a more formal mathematical definition, in which the environment is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where:

- $\mathcal{S}$ is a set of all possible states s,

- $\mathcal{A}$ is a set of all recognizable actions a,
- $\mathcal{T} : \mathcal{T}(s_t, a_t) \to \mathcal{S},\ s_t \in \mathcal{S},\ a_t \in A_t,\ A_t \subset \mathcal{A}$, and
- $\mathcal{R} : \mathcal{R}(s_t, a_t) \to \mathbb{R},\ s_t \in \mathcal{S},\ a_t \in A_t,\ A_t \subset \mathcal{A}$,

with $\mathcal{T}$ defining transitions between states with respect to incoming actions and $\mathcal{R}$ defining rewards for taking actions at states. Generally speaking, neither $\mathcal{T}$ nor $\mathcal{R}$ must be deterministic and can, instead of returning constant values with respect to constant parameters, map to random variables with their respective supports (as explained by [4]). By studying the domain of $\mathcal{T}$, one can infer two additional elements regarding the environment, these are however merely descriptive, not defining, therefore not part of the aforementioned tuple. Elements in concern are:

- $\alpha : \alpha(s) \to \mathcal{P}(\mathcal{A})$, such that $\forall a \in \alpha(s) : (s, a) \in \mathcal{D}(\mathcal{T})$, and
- $\mathcal{Z} \subset \mathcal{S},\ s \in \mathcal{Z} \wedge \forall a \in \mathcal{A} : (s, a) \notin \mathcal{D}(\mathcal{T})$,
  with $\mathcal{P}$ being a *powerset* function (set of all subsets of its argument), and $\mathcal{D}$ being the domain of a function,

making α the *valid action* function, specifying which actions can be taken at a state and $\mathcal{Z}$ the set of all terminal states. The starting state does not have to be constant, however, it always exists. A series of transitions from any starting state to any terminal state is considered an *episode* (see [10]), or a *roll-out*.

2.1.2 Properties

The most important feature of an environment is inferred from the shape of $\mathcal{T}$, particularly, whether its output is a deterministic or a random variable, which is what we use to separate *deterministic* and *stochastic* environments. When an agent cannot rely upon the fact that taking an action at a state results in one predictable state, the algorithms used to solve these environments grow in complexity.

States can either be *fully* or *partially observable*. The former indicates the ability of outside observers to gather all information about S_t there is, in the latter case, observers can only gain some of the information. A game of chess, for example, is fully observable, whereas a game of poker is only partially observable (no player can see other players' cards).

We recognize *static* environments, where states transition between each other strictly as a consequence of outside actions. *Dynamic* environments, on

the other hand, can change their state internally, without outside influence. It is very typical for static environments to represent turn-based games (a state changes only when a piece is moved by a player - the actor), as opposed to problems of traffic light control (cars move even if no lights are being switched) or real-time resource distribution.

The last but not least noteworthy characteristic of environments is the distinction between action space types, namely, whether $\mathcal{A}$ is *discrete* or *continuous*. One might use the former for modeling various board game problems, and the latter for optimizing stock exchange behavior. (Note the fact that computers cannot represent continuous variables, therefore continuous action spaces will only approximate the reality. The main difference is in the size of the resulting action space, which has a great impact on choosing an appropriate learning algorithm.) [37]

2.1.3 Reward

Before we move onto defining an agent, the notion of reward should be discussed. The reward somewhat resembles the *fitness function* in genetic algorithms and hence shares all the common pitfalls one might find themselves in when modeling an environment. Of course, the first question is, what does the author consider to be an optimal solution to a decision problem. In the simplest cases, the desired strategy is such that the survival time in the environment is maximized, resp. a goal is achieved in the least possible time: usually, one would then define the reward function to always return Δt, resp. $-\Delta t$, the time it takes to transition between respective states, regardless of arguments, for the sum of these rewards is equal to time taken, resp. negative time taken. When learning to play *Pong*, the reward will be given only at states where the actor scores a goal. However, there are cases where a more complex reward function is required, for example when discrete high-reward events are scattered across the episode and the agent must reach these events as soon as possible consuming his resources, and the reward function must be adjusted accordingly. If it is defined incorrectly, learning algorithms will most likely lead to *reward hacking* (see [18]), where the learned strategy exploits sources of secondary reward (usually in cycles) instead of solving the original problem.

Another problem, perhaps a more important one, occurs when the reward function is non-zero for very few state-action pairs throughout the episodes; in the most extreme case, it is non-zero only when the resulting state is terminal, *videlicet*, one might decide to reward the agent by some $\epsilon > 0$ when it succeeds at a particular task which consequently terminates the environ-

ment, and by $-\epsilon$ if it fails to do so (see [8, 34] for details). An environment such as this is said to be producing a *sparse reward*. This approach is very convenient for the designer. If one got creative, they might try to reward the agent if it succeeds to compose a news article, or a symphony, or if it defeats its opponent in a game such as go or chess. We will take robot navigation as our example: let there be a robot that applies torque to its joint motors (= actions) that needs to reach a goal, such as an allied military base, through a hazardous terrain. It must reach its goal, therefore it is only rewarded if it does, if it gets 'destroyed' along the way, it receives no or negative reward. As [8] implies however, basic RL algorithms, which depend largely on random exploration, will be receiving these zero rewards exclusively, unless they just so happen to arrive at a perfect solution, which is highly improbable. Authors of [34] hint at a partial solution to this problem, which is to manually add custom reward signals (called *reward shaping*), which are rewards received during such states that the designer themself **believes they should** be part of an optimal solution. Such approach does indeed shorten the necessary exploration times for said algorithms, but not without a cost; it is clearly flawed. Not only does it invite the aforementioned *reward hacking*, but it also places a cap on the optimality of the final solution. Let's say the designer shapes the reward like this: if the robot encounters a (sufficiently small) stream of water and uses such action as to jump over it, it gets rewarded for bypassing the obstacle. If there indeed is a better policy for the robot and the designer doesn't foresee it, the robot will be forced to jump over water streams for the entirety of its training, because it is advantageous for it in terms of reward maximization. However, if instead, entering the stream and letting itself be carried by it potentially towards the destination results in reaching it more often, the robot loses the potential to exploit this option, which results in it performing sub-optimally.

2.1.4 Bounding states and the discount factor

The choice of starting states has a great impact on the required complexity of the optimal strategy. A deterministic, static environment with a fixed starting state allows a solution as a learned sequence of actions; not observing the states at all after the learning process completes. This can be advantageous, for a greater range of learning algorithms can be applied, and the solution can be found faster, however, if the starting state should ever change, the strategy loses its worth. If the starting state is not determined, a more general strategy must be found in order to solve the problem, limiting the available learning algorithms, and potentially drastically slow-

ing the optimization speed, but resulting in a more robust solution. Some environments have a determined starting state, such as the game of chess; usually however, when an environment designer is given the choice, they should always look to randomize the starting state.

The purpose of terminal states is simply to define a range of time instances for which to maximize the reward. Without a terminal state to which the states eventually converge, the episodes become infinitely large and determining the optimality of any strategy applied to the environment becomes rather unclear, for in order to compare any two policies, one must sum all the received rewards of any policy in order to evaluate it. And unless shown otherwise, such sums can be infinite and therefore non-comparable. An argument by [24] has been made that if an episode is infinite, one might introduce a *discount factor* $\gamma \in [0, 1)$ to the environment. A general way to compute the cumulative reward therefore becomes

$$Q = \sum_{i=0}^{\infty} \gamma^i r_i = r_0 + \gamma r_1 + \gamma^2 r_2 + ... \tag{2.1}$$

which bounds its value to real numbers even for infinitely long episodes. Since this work focuses only on finite episodes however, for all of our experiments, we set $\gamma = 1$. [24] also mentions that there are states which do not fit our definition of terminality at section 2.1, but entering it means that the total future undiscounted reward will be zero. This occurs when a state, or a group of states, is *absorbing* and unrewarded. For all intends and purposes, such states can be deemed terminal as well.

2.2 Agent

The product of RL is a *learned*, or *trained*, agent. Its core purpose is to produce a *policy function*

$$\pi : \pi(s) \to \alpha(s), s \in \mathcal{S}, \tag{2.2}$$

where $\mathcal{S}$ and $\alpha(s)$ is the set of all possible states and the valid action function of the corresponding environment, respectively. Improving the policy function in order to maximize the corresponding environment's cumulative reward sum is considered *training* the agent. A trained agent solves the respective RL problem.

A realized agent, as a software product, therefore consists of the following major components:

a) an implementation of the policy function itself, whose contract is π, and

b) an algorithm used to train the agent.

2.2.1 Policy function

As was stated, a policy function is any *mapping* from the state space $\mathcal{S}$ to a member of the action space $\mathcal{A}$. The mapping itself must clearly be parametric, for this is required by the agent's trainability characteristic, and tends to be very complex for non-trivial problems. For example: the most trivial applicable RL algorithm, the *Q-learning* [1, 14, 31] (which is explained in Chapter 4), represents this function as an *argmax* operation on a state/action table, with states as rows, actions as columns, and values as *Q-values*. These *Q-values* are equal to the expected cumulative reward the agent is going to receive if it takes an action (column) at a particular state (row). After the Q-learning algorithm creates such a table, when an episode is being rolled out and the agent is asked to find the optimal action at state s_t, the agent looks up the state's row and returns the index at which the largest *Q-value* lies. Such a function is virtually defined as an enumeration of input-output pairs, which is the most complex mapping representation possible. Using Q-learning on even a very simple environment whose state would be fully determined by a single real number would result in a Q-table that is potentially infinitely large. The aim of this work is to research applications of *neural networks* as policy function representations, which is a solution much lighter on memory resources [14].

However, before we start describing the existing agent algoritms that use neural networks, we must first understand what a neural network is, what it does and how it achieves this, which is thoroughly explained in the following chapter.

3 Neural Networks

Neural networks (NNs) - in our work's scope - are parametric non-linear function approximators. Commonly denoted as

$$f_n(x; \theta) = y, \tag{3.1}$$

where θ represents the NN parameters, these mathematical constructs are capable of modeling general functions $f(x) = y$, where x and y are *tensors* of potentially different *ranks* and *shapes*, as shown in [19, 23]. As the most influential AI frameworks, TensorFlow and PyTorch, both use tensors when working with neural networks, we find explaining NNs from the tensor perspective convenient, as no information is lost in contrast to the explanatory approach of [23], where the individual *neurons* as building blocks are used.

3.1 Tensors

A tensor is, from a computer-engineering point of view, a multi-dimensional array of elements $a \in X$, where X is a set; usually of natural or real numbers. Its dimension, or more commonly *rank*, is a non-negative whole number. It represents the number of indices a tensor query must contain in order to return a single entry, a, as shown below.

A tensor of rank 0 is simply a. Tensor of rank n is an array of tensors of rank $n-1$ of length l_{n-1}, where $l \in \mathbb{N}$. An array $L = \begin{pmatrix} l_{n-1} & l_{n-2} & \dots & l_0 \end{pmatrix}$ is called the *shape* of a tensor. The individual dimensions are usually called *axes* [26]. The reader should note that sizes of the last axes of the tensor reside at the first positions of its shape array ($L_0 = l_{n-1},\ L_1 = l_{n-2},$ etc.)

Thanks to its multidimensional-array-like nature, we recognize its individual parts, called *subtensors*. A subtensor of a tensor is any tensor contained within it that has smaller rank and is equal in shape with the original at their corresponding (remaining) dimensions. A vector's subtensors (shape (n)) are its scalar values (shape $()$), a matrix' subtensors (shape (m, n)) can be the individual scalars (shape $()$), its rows (shape (n)), or its columns (shape (m)), and so on. Subtensors of tensor of rank 3 are visualized at Figure 3.1.

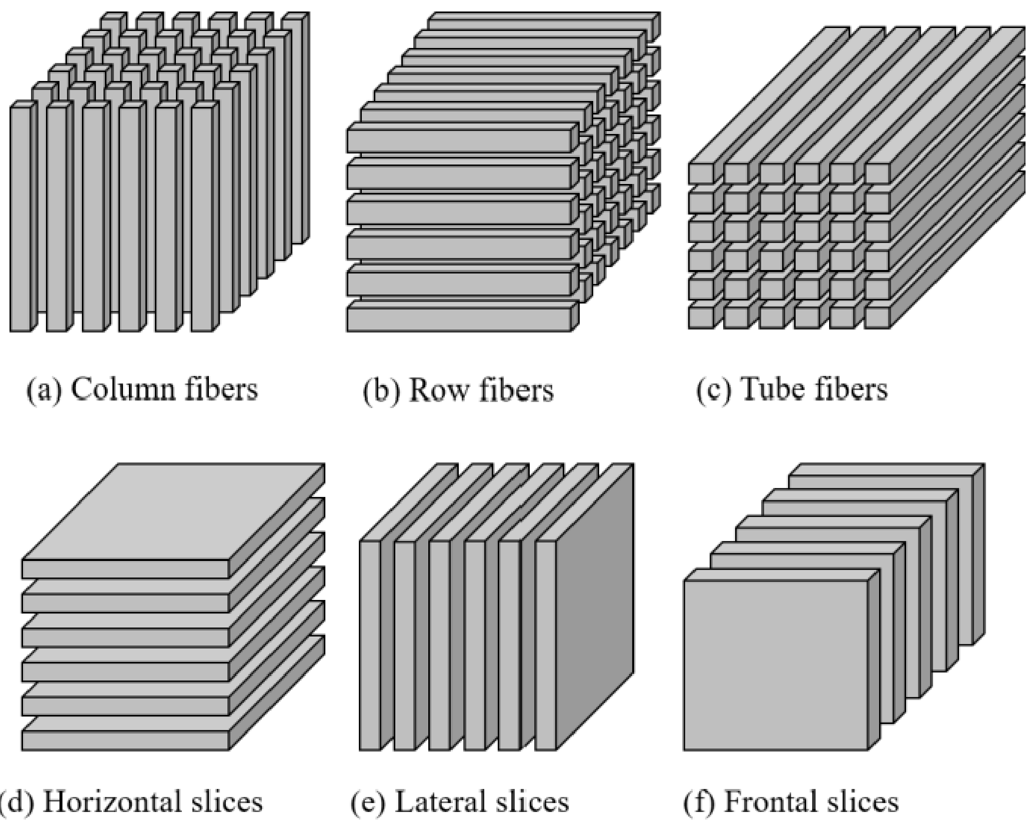


Figure 3.1: Rank 1 and 2 subtensors of rank 3 tensors. [39]

3.1.1 Examples

- A scalar a is a rank 0 tensor of empty shape.
- A vector $\begin{pmatrix} a & b & c \end{pmatrix}$ is a rank 1 tensor of shape (3). A single index from $[0, 2]$ is required to fetch a single value.
- A matrix $\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$ is a rank 2 tensor of shape $(2, 3)$. Two indices from $[0, 1]$ and $[0, 2]$ respectively are required to fetch a single value, or a single index from $[0, 1]$ is required to fetch a row of this matrix, which is its *subtensor* of rank 1 and shape (3). (see Figure 3.2)

One should note that a vector of size n and matrices of sizes $(1, n)$ and $(n, 1)$ are all distinct mathematical objects, despite their apparent similarity.

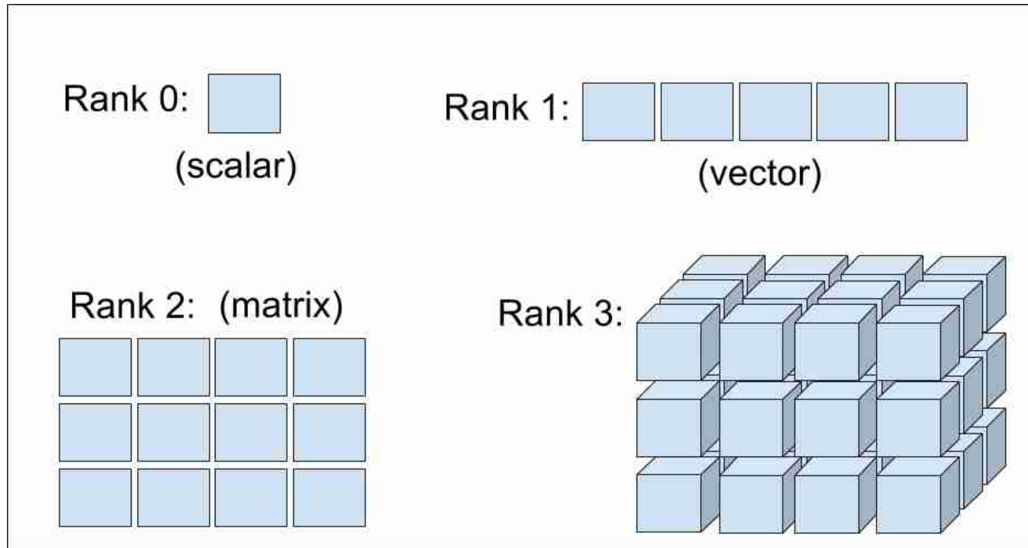


Figure 3.2: Visual representation of differently ranked tensors. [26]

3.1.2 Usage

We define tensors individually, instead of using the well-defined terms of *vector* and *matrix*, for even though that neural networks *can* operate on tensors of ranks 1 and 2, in practice, they are used for mapping from, to, and between tensors of higher ranks, as can be seen both in [13, 17, 27] and our work.

The reason neural networks are designed to work with tensors is their ability to accurately represent any well structured numerically describable object. Among many others, both the environment states, which can be

images or their series (sensory inputs) or various custom embeddings (board games), and actions (which are usually scalars), are representable by them which is the reason why using neural networks in RL algorithms is so popular [6, 20, 32].

3.2 Tensor operations

The neural network itself is a sequence of tensor operations applied to the input, resulting in an output tensor. The most common operations include, but are not limited to, multiplication by a *weight* tensor, convolution operation using a *kernel* tensor, element-wise increment by a *bias* tensor and an element-wise transformation, or so-called *activation*, by a non-linear *activation function* (further reading at [23, 30]). All of these will now be examined.

3.2.1 Broadcasting

Before we delve deeper into the realm of tensor operations, the common practice of *broadcasting* should be explained. For any operation whose argument is expected to be smaller in rank than the tensor that was actually supplied, if the supplied tensor contains subtensors of shape required by the operation, the operation is applied independently to all of them. A simple example of broadcasting can be observed when scaling a vector by a scalar. Multiplying a scalar by a scalar is trivial. Since the vector is however of larger rank, the operation is applied to all of its scalar subtensors. Figure 3.3 shows another example of broadcasting, where we subtract a tensor of shape (2) from a tensor of shape $(3, 2)$. Notice how the operation is applied to all subtensors of the minuend. If the subtrahend was instead of shape (3), the operation would be broadcast to the matrix' columns. Understanding of this process will be relied upon in the following sections.

3.2.2 Indexing

Let I be an ordered tuple of integers of size k, called an *index tuple* or *tuple of indices*. We say that an index tuple is valid with respect to a tensor T with shape array L if and only if $\forall j \in [0, k-1] : I_j \in [0, L_j - 1]$. One can then use such a tuple to query the tensor, which we denote as T_I, receiving a corresponding tensor of shape $n - k$ in return. Since each tensor of rank greater than zero is an array, the corresponding tensor is obtained by popping integers from the index tuple until emptied and using them to query the original tensor. By definition, a tensor of rank n contains tensors

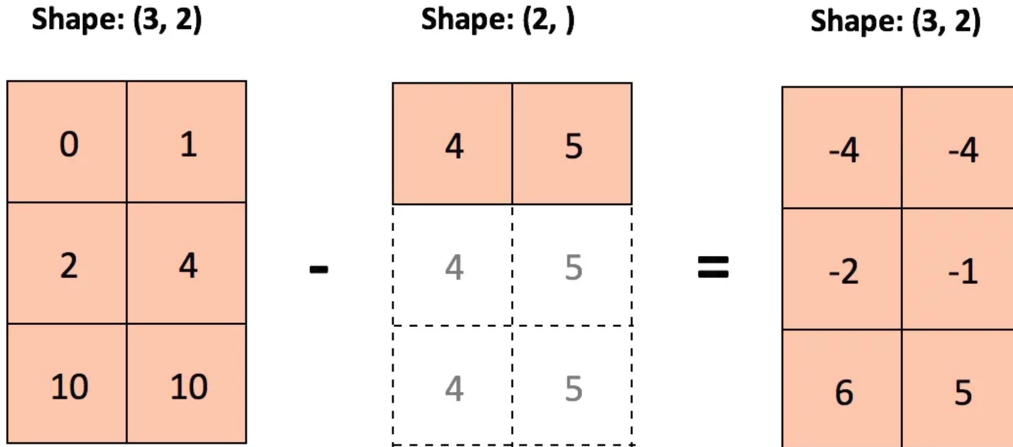


Figure 3.3: Broadcasting visualized on small tensors. (kmario23, StackOverflow, 2018)
.

of rank $n-1$, hence querying by a single integer and letting $T = T_i,\ i \in \mathbb{N}$ reduces the current tensor rank by 1. After the index tuple is depleted, the remaining tensor is returned.

3.2.3 Slicing

Let the sum of index tuples $I_1 + I_2$ be equal to their element-wise sum. For any index tuples L and U, where U is element-wise not less than L and has the same size k, let $L{:}U$ denote a *slice* of indices between L and U, that is, a sorted sequence of all index tuples S such that $\forall i \in [0, k-1] : S \in L{:}U \iff L_i \leq S_i \leq U_i$. All members of such a slice have length k. If these members are valid index tuples with respect to some tensor T, one can index a tensor by an entire slice, which is conveniently called *slicing* the tensor. Slicing a tensor results in a tensor of the same rank, but smaller shape (unless the slice encompasses the entire original tensor), and since we think of it as a sorted sequence, we can let O be the original, $L{:}U$ be the slice, $(L{:}U)_i$ a single tuple in the slice, T be the result and let

$$T_{(L:U)_i - L} = O_{(L:U)_i}\ \forall i \in [0, |L{:}U| - 1], \tag{3.2}$$

rigorously defining the slicing operation. Should k be smaller than the rank of the original tensor, the operation is subject to broadcasting.

3.2.4 Matrix-like multiplication

A tensor multiplication is an operation on tensors A, B of rank at least 2 such that their shape at the last two axes is (k, m) and (m, l), respectively. The product on the last two axes is a second rank tensor P of shape (k, l), such that

$$\forall i, j,\ i \in [0, k-1],\ j \in [0, l-1] : P_{ij} = \sum_{w=0}^{m-1} A_{iw} \cdot B_{wj}, \tag{3.3}$$

resembling the standard matrix multiplication, with the only difference being the potential option to broadcast the operation along possible excess axes of A.

3.2.5 Convolution

The convolution operation is, in layman's terms, a sliding dot product. Let A be a tensor of rank n and shape L. Let B be a tensor of rank $m \leq n$ and shape K such that $\forall i \in [0, m] : K_i \leq L_i$. Then, the result of a convolution of A with kernel B is a tensor P of rank n and shape S, where $\forall i \in [0, n-1]$:

$$S_i = \begin{cases} L_i, & i < n - m \\ L_i - K_i, & \text{otherwise,} \end{cases} \tag{3.4}$$

the result of their convolution being

$$P_T = A_{T:(T+K)} \cdot B \quad \forall\ T \in \mathbf{0}:(L-K), \tag{3.5}$$

where $\mathbf{0}$ is a zero index tuple and $\cdot$ a dot product operation (= sum of element-wise products between the operands). If m is indeed smaller than n, the operation is broadcast, which is why $S_i = L_i$ for these axes.

The case of 2D convolution (kernel rank = 2), illustrated by Figure 3.4, is the most common and the only convolution version used in our work.

3.2.6 Batch Normalization

This operation has been introduced by [15]. When the input of this operation represents an array of tensor values, the batch normalization operation normalizes these values so that their mean is close to 0 and variance close to 1. If $B = x_1, x_2, ..., x_n$ is the input tensor, the operation calculates the array's mean and variance:

$$\mu_B = \frac{1}{n} \sum_{i=1}^{n} x_i, \tag{3.6}$$

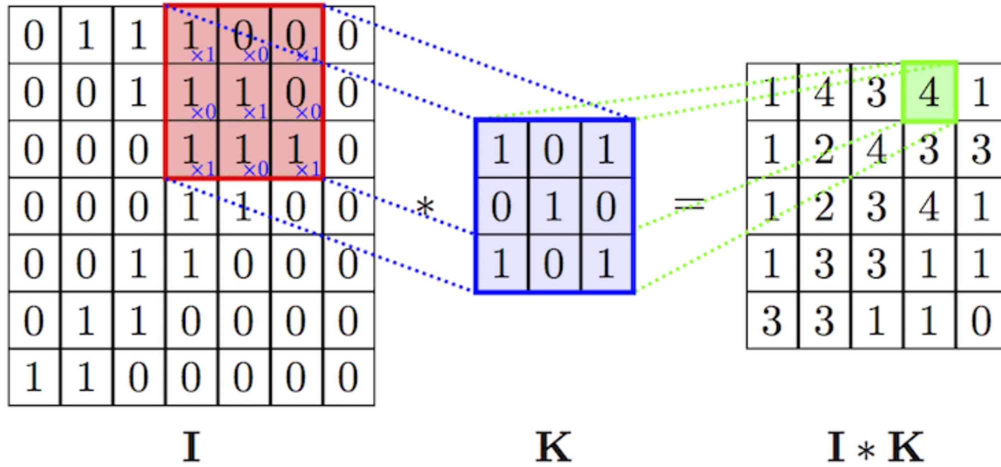


Figure 3.4: A convolution example visualized on rank 2 tensors. [29]

$$\sigma_B^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_B)^2, \tag{3.7}$$

and introduces parameters γ and β to output

$$f(x_i) = \gamma\frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta. \tag{3.8}$$

This operation is broadcast to the entire tensor B, applying the transformation to each member x_i, producing a tensor P of the same rank and shape as the original.

3.2.7 Bias

The bias operation is called the way it is to emphasize its influence on the network's output independently of the input, thus biasing it. The result of this operation is an element-wise sum of the inputs. It is - again - broadcast over the remaining axes of A if possible.

3.2.8 Activation function

Activation functions are unary non-parametric transformations. Its core purpose is to introduce non-linearity to the neural network model, in order to enable the option to approximate general functions. It can be shown that consecutive affine operations such as those described above introduce no non-linearity and therefore the network as a whole cannot approximate non-linear functions.

An activation of a tensor A is simply $f(A) = B$, with B being of identical rank and shape as A, for f expects a single element, so the operation is broadcast and applied element-wise.

3.2.9 Dropout

Dropout behaves similarly to the activation function. A constant $\alpha \in [0, 1]$ is supplied when introducing this transformation to the network, which is the probability that a scalar value in the tensor will be set to zero after this operation is applied. This operation is used as a *regularization* technique (see below), which helps the network find more general mappings between inputs and outputs. [9, 33]

3.3 Function approximation

The elements of this sequence of transformations are called the network's *layers*, for each of these layers accepts an input and returns an output independently of the other layers. The output of the current layer is the input of the next one. Most of the operations described above have two inputs: those are the parametric operations. The second input is stored in memory of the neural network model and it is that which defines to what outputs does the network map its inputs. For example, a weight tensor might be used to multiply with the input, a filter tensor is used in convolution as an array of kernels, each producing their own separate output, which are stacked in the end to produce the final convolution layer output, etc.

3.3.1 Cost function

Now, the primary usage of a neural network is making it approximate a function $X \rightarrow Y$, where the user possesses multiple (x, y) tuples and is keen on finding a mathematical relation between the sets X and Y such that can be sufficiently represented by the neural network's parametric transformations. An idea of a *cost function* quickly came into mind, for when provided with the input/output pairs, the problem is to find θ such that $\forall x \in \mathcal{D}(f) : dist(f_n(x, \theta), f(x)) = 0$, i.e., we are looking to minimize the difference between $f_n(x, \theta)$ and $f(x)$. There are many ways one might define the distance between two functions, which are all abstracted by the cost function. A common cost function for approximating real values is the *mean squared error* (MSE), where $d(y_0, y_1) = (y_0 - y_1)^2$. MSE, being our example, has a desirable property for RL applications, for if one lets the network train

on separate tuples $(x, y+e)$ and $(x, y-e)$ with e being any error tensor, the network's mapping will eventually converge to (x, y); an important feature, as we will see in the following chapters. The exact choice of the cost function is therefore task-dependent and it is up to the data scientist to use such that it has the appropriate properties.

3.3.2 Optimization

For a given set of (x, y) tuples, the cost C becomes a function of the neural network's parameters, θ. The problem is thereby transformed to a problem of finding a global/local minimum of the cost function, which can be solved using partial derivatives, if the cost function is properly differentiable. Unfortunately, for the amount of parameters the neural network usually has (ranges from 10^2 to 10^6), it is impossible to arrive at an analytical solution directly [22]. In order to solve this problem, one must apply a technique called *gradient descent* (GD). It is an iterative numerical method of finding a local minimum of a function. The main idea behind *gradient descent* is that if there is a solution θ_i yielding C_i, then by shifting the solution in the direction of in absolute value largest negative gradient at θ_i, resulting in θ_{i+1} and C_{i+1}, one can expect that $C_i > C_{i+1}$ and therefore the cost should thereby converge to a local minimum [23].

$$\theta_{i+1} = \theta_i + \eta \cdot -\nabla(\theta_i), \eta \in \mathbb{R}^+ \tag{3.9}$$

Equation 3.9 uses a parameter η common to all numerical methods, called the *learning rate*, with smaller values leading to slower but more stable convergence and higher values speeding the training up but risking divergent behavior. η scales the gradient descent step size linearly, so that it is calculated directly from the current gradient. Lowering η alone, however, does not guarantee convergence. After the gradient is calculated, only the direction in which the steepest slope is and its actual steepness is known, but the information about how large the slope is and how big of a step one should therefore make is simply not there. The GD can only guarantee convergence for an infinitesimal step size, which is impractical. Humans have, of course, developed a notable amount of gradient-descent-based parameterized algorithms that improve convergence speed and quality of local optima [28]. The most recent and popular method, *adaptive momentum*, allows for faster descent into better solutions, and can be considered good enough to ensure convergence to some local minimum, thus solving the problem of tuning the parameters and creating a trained neural network $f_n(x; \theta^*)$, for which the

difference between it and $f(x)$ is as small as possible with respect to the network's inner design [23].

In order to execute this procedure however, one must have an algorithm to compute the gradient itself:

$$\frac{\partial C}{\partial w_1}(\theta_i),\ \frac{\partial C}{\partial w_2}(\theta_i),\ \dots\ ,\ \frac{\partial C}{\partial w_n}(\theta_i), \tag{3.10}$$

where w_j is a single parameter of the NN. This is done by an algorithm called *backpropagation*, thoroughly explained in [16, 23]. The key takeaway is the following: as long as the individual layer operations and activation functions are differentiable, the gradient of the cost function with respect to any parameter in the network is efficiently computable using dynamic programming.

If the amount of (x, y) tuples is too large to fit into memory, gradient descent is approximated by *stochastic gradient descent* (SGD), where the initial data is randomly split into small enough groups of equal size and the gradient descent is then applied to each of them sequentially.

3.4 Regularization

Overfitting is a major problem in function approximation. It occurs when the model approximates the data tuples (x, y) too closely, without generalization, as seen in Figure 3.5. Approximation models that overfit are of no use, because they fail to produce correct outputs for inputs they have not been explicitly trained on [3]. Deep networks with many parameters are capable of fitting such sets of data tuples perfectly, so even if they are completely random and therefore no mathematical relation between them exists, gradient descent will lead to overfitting, as proven by [40].

Humans therefore engineered methods that prohibit the network from abusing its large amount of parameters effectively to overfit the data, such as introducing parameter norms to the total cost, or using special layers like the dropout [33], or batch normalization [15].

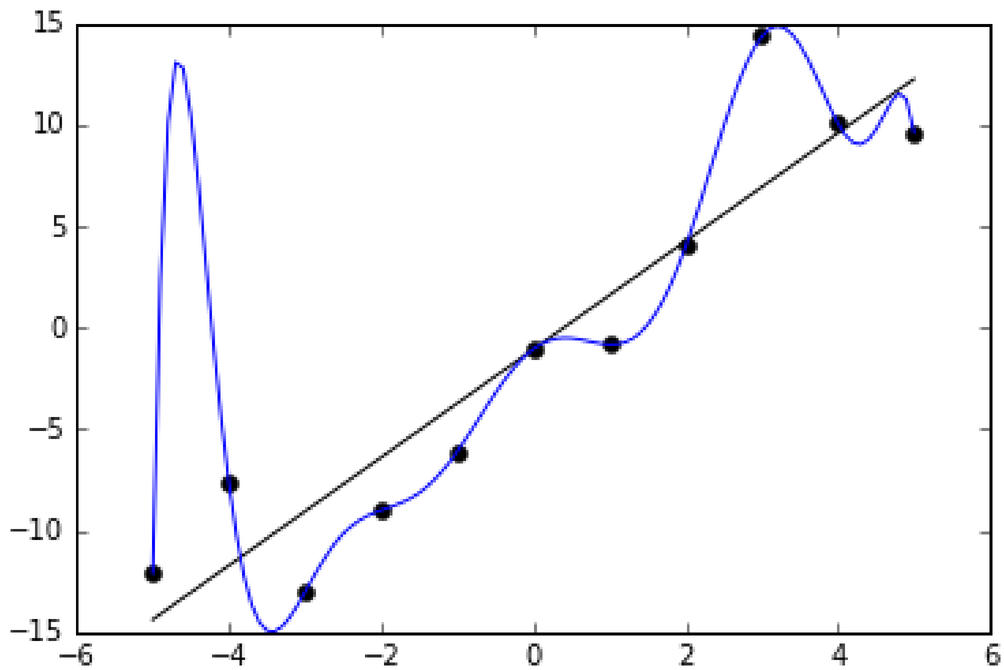


Figure 3.5: Visualization of an overfitted model. The data tuples (x, y) are shown as dots on the plot. The straight line is a good approximation: it somewhat captures the linear trend of the data. The curved line, despite there being zero distance between it and the data points, interpolating or extrapolating the data using the line will most likely contain very inaccurate results with respect to the original data source. By Ghiles - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=47471056

4 Existing agents

Section 3.4 marking the end of our neural network theory, we are now able to perceive the network from a higher level; combined with the algorithms associated with it, we think of it as a separate program capable of approximating any mapping between tensors. With that being said, we are now ready to proceed onto describing the existing agents that use neural networks and have been deployed to solve multitude of problems.

4.1 Deep Q-learning network (DQN)

When presented with an environment with a small, finite state space and discrete action space, then even though its *transition* and *reward functions* may be stochastic, such an environment is solvable using the simple *Q-learning* algorithm. The classic Q-learning requires only a way to hash the environment's states so as to create a hashmap from each state to an n-tuple of *Q-values*, and over time, this table will converge to a tool capable of representing the optimal policy.

4.1.1 Q-value

The entirety of Q-learning is based upon the idea of a *Q-function*, which is defined as

$$\mathcal{Q}(s_t, a_t) = \mathbb{E}(\mathcal{R}(s_t, a_t)) + \mathbb{E}(\sum_{i=t+1}^{term} (\mathcal{R}(s_i, a_i^*))), \tag{4.1}$$

where s_t is an arbitrary state at time t, a_t is an arbitrary valid action at this state, $term$ is the terminal time, $\mathcal{R}$ is the *reward function*, and the asterisk superscript signifies the associate's *optimality.* Essentially, this function returns the *expected cumulative reward* for taking an action a_t at state s_t, should the consecutive actions be taken in an optimal manner. This expected cumulative reward is called the *Q-value.* Should an agent possess such a function, it could follow optimal policy very simply:

$$s_t \in \mathcal{S} : a_t^* = \underset{a' \in \alpha(s_t)}{\operatorname{argmax}} \{a' | \mathcal{Q}(s_t, a')\} \tag{4.2}$$

Combining equations 4.1 and 4.2, we arrive at

$$\mathcal{Q}(s_t, a_t) = \mathbb{E}(\mathcal{R}(s_t, a_t)) + \max_{a' \in \alpha(s_{t+1})} \mathcal{Q}(s_{t+1}, a'), \tag{4.3}$$

which guides the Q-learning update step.

4.1.2 The Q-learning algorithm

The goal of the Q-learning algorithm is to produce a sufficiently good approximation of the Q-function associated with a given environment, and it does so as follows:

Algorithm 4.1 q_learning

```
1: input: R, T, initial state
2: i ← 0
3: Q_i ← empty hashmap
4: insert initial state into Q_i with a zeroed-out Q-value tuple
5: repeat
6:    t ← 0
7:    s_t ← initial state
8:    repeat
9:       a_t ← chosen action at s_t according to Q_i
10:      r_{t+1} ← R(s_t, a_t)
11:      s_{t+1} ← T(s_t, a_t)
12:      if s_{t+1} ∉ Q_i then
13:         insert s_{t+1} into Q_i with a zeroed-out Q-value tuple
14:      end if
15:      Q_{i+1} ← update(Q_i, s_t, a_t, r_{t+1}, s_{t+1})
16:      i ← i + 1
17:      t ← t + 1
18:   until s_{t+1} is terminal
19: until convergence
20: return Q_i
```

Lines 5-20 of Algorithm 4.1 represents a single episode rollout. Line 15 utilizes equation 4.3 to gradually converge the initially empty hashmap to an optimal Q-value table. In the background, the update step is actually a slight extension of 4.3:

$$\mathcal{Q}_{i+1}(s_t, a_t) = (1-\beta)\mathcal{Q}_i(s_t, a_t) + \beta(r_{t+1} + \max_{a' \in \alpha(s_{t+1})} \mathcal{Q}_i(s_{t+1}, a')), \quad (4.4)$$

introducing a hyperparameter $\beta \in (0, 1)$ called the *learning rate*; this is very common for all numerical methods, including SGD in neural networks. Without a learning rate, Q-values could never converge to expected cumulative rewards, for they would instead oscillate between the last seen cumulative rewards as the update would be reapplied in future episodes.

Line 9 does not always generate actions currently regarded as optimal, but instead follows what is called an *epsilon-greedy policy*. This policy lets

$\epsilon = f(i)$, where f is another hyperparameter: a decreasing function, such as $f(i) = 1 - ci, c \in \mathbb{R}^+$, and chooses a random valid action if $\epsilon > x \sim \mathcal{U}(0,1)$, otherwise the optimal valid action according to $\mathcal{Q}_i$. This lets the Q-learning agent *explore* the environment in the early episodes, and *exploit* its knowledge in the later episodes. Finding good balance between these two approaches of knowledge gaining is crucial: with early exploration transitioning into late exploitation, the agent gains thorough information about every stage of the episode.

Convergence may be tested using standard norms of update changes, claiming the algorithm is complete as soon as N consecutive updates change the $\mathcal{Q}$ representation by a sufficiently small normalized amount, or by continuous testing, where the agent is evaluated after each N training episodes, claiming the algorithm is complete as soon as the agent achieves a high enough reward to surpass some constant R in M consecutive evaluation episodes. An evaluation episode does not follow the *epsilon-greedy policy*; line 9 always produces actions optimal according to $\mathcal{Q}_i$.

4.1.3 Amplification by neural networks

As useful as Q-learning might sound, its inability to work with large state spaces is its major disadvantage. With a vast amount of possible states, one cannot hope for being able to manipulate all of them in any conceivably large amount of time, nor have them stored in memory on a machine. This is where the approximative abilities of neural networks come into play, as they can learn to map between states, actions and Q-values using significantly less memory than when using a table, and most importantly, return Q-values for any state/action pair, even those previously unseen.

This idea gave rise to Deep Q-learning, or DQN. In DQN, the hashmap is replaced by a neural network, whose input is an arbitrary state tensor and whose output is a rank 1 tensor, with a real number value for each action in the environment's action space. The network's parameters are initially randomized, so that previously unseen states can produce non-zero Q-values for any action, in comparison with the hashmap, where all Q-values of unseen states were equal to zero. If we were to fully replace the Q-function representation with no changes, terminal states would not be recognized. Therefore, in DQN,

$$\mathcal{Q}(s_t, a_t) = \begin{cases} 0 & \text{if } s_t \text{ is terminal} \\ \mathcal{Q}_\theta(s_t, a_t) & \textit{otherwise.} \end{cases} \tag{4.5}$$

The symbol θ represents the fact that $\mathcal{Q}$ is determined by the network's

parameters. Since the network only receives one parameter as the input, $\mathcal{Q}_\theta(s, a)$ is to be understood as returning the Q-value corresponding to action a in the output tensor.

The DQN utilizes a distinct *target* and *candidate* network, where both are initialized randomly, *candidate* is being trained using values from *target*, and *target's* parameters are set to equal the *candidate's* only every N episodes. This helps break the influence of the current network parameters on incoming data samples, as that introduces a feedback loop to the system which can easily lead to divergence [20]. Additionally, the training data required for line 9 (the update step), s_t, a_t, r_{t+1}, s_{t+1}, are stored in a *circular buffer* (semi-ordered array-like structure with constant time *add* and limited size, automatically removing the oldest inserted item as soon as maximum size is achieved), and randomly sampled during the update step for training. This technique is called *experience replay*, and it allows us to move the update step out of the episode loop and insert it between lines 20/21, updating after each episode is finished. Using experience replay introduces multiple adjustments to the learning algorithm, such as randomizing the input data and thus disrupting correlations between data samples in one batch, which in turn reduces the variance between consequent data batches, or providing the same datum for multiple training steps, which improves data efficiency [20].

This model will serve as our baseline, since there are arguably no simpler neural network agents than DQN. Although [20] shows their success in deploying DQN to solve many Atari games, surpassing human experts, there are games like Montezuma's Revenge, where DQN struggled to achieve any meaningful reward. That being said, as we plan our environment to be even more complex than Montezuma's Revenge, we do not expect DQN to outperform other agents.

4.2 Expecti-max Monte Carlo Tree Search (MCTS)

The Monte-Carlo Tree Search by itself is a method of guessing the optimal action a_t^* to take at any state s_t in the environment, or more precisely, estimating the probability of any valid action at s_t being optimal. It progressively generates an approximation of the *state tree*, which is a tree with environment states and actions as nodes connected by edges representing the corresponding relations between them. The expecti-max version in particular, which was devised by [36] and which we will describe here is used

to solve stochastic environments, where one state-action pair can result in multiple different states based on some underlying probability distribution of $\mathcal{T}$ and $\mathcal{R}$.

State-reward pair nodes are called *decision nodes* with each having a child node for each action that can be taken at that particular state. The nodes corresponding to actions are called *chance nodes*, with each having a child node for each state-reward pair that is generated by taking that action at the state of the parent decision node. It is possible to receive identical states and different rewards, since both $\mathcal{T}$ and $\mathcal{R}$ are stochastic, hence the need for decision nodes to correspond to each individual state-reward pair. Clearly, the naming introduced by [36] is convenient, as the agent can make decisions at decision nodes, and has a certain chance of ending up at different states, and obtaining different rewards, when performing any action.

This partial tree is built by exploring the environment, adding new nodes to the graph for each explored state. Each node is assigned a value and a visit counter, according to which - after the MCTS completes by running out of time - the best action to take from the root is selected. The manner which the tree is built in and the values are assigned in will now be demonstrated.

After initializing the tree root, which is a decision node corresponding to s_t, the entirety of the MCTS algorithm can be broken down into three distinct procedures: *selection*, *expansion + rollout*, and *update*. These procedures are repeated in order N times, and the final tree produced by them is deemed a good enough approximation of the full state tree.

4.2.1 Selection

Gradually expanding the tree breadth-first or depth-first are both possible, but defective approaches. Trying to expand a tree with a big branching factor breadth-first will result in a shallow tree with insufficient *think-ahead*, resulting in poorly optimized decisions in highly strategic scenarios where past behavior has a huge impact on future states; on the contrary, depth-first search eliminates this problem, but unnecessarily expanding branches, which are already known to be lacking reward, results in time wasting and proneness to missing out potentially advantageous branches completely. The *selection* subprocedure of MCTS identifies chance node leaves of the current tree, whose expansion will most likely result in a solid final approximation, based on *upper confidence bounds applied to trees*, or simply UCT.

The UCT is a mapping from MCT chance nodes to real numbers, which signify the beneficialness of expanding these nodes. Selection then simply descends the tree from root to leaves, while choosing between different path

options by maximizing UCT. UCT does not exist for decision nodes, for the results of actions are determined by the *transition function* of the environment and cannot be chosen by the agent.

We calculate UCT for each child c of decision node d as follows:

$$UCT(c) = \frac{V(c) - \min_{c' \in C} V(c')}{\max_{c' \in C} V(c') - \min_{c' \in C} V(c')} + E\frac{\sqrt{T(d)}}{T(c)+1}, \tag{4.6}$$

where C is the set of d's child nodes, V is the value function, T is the visit counter function, and E is a constant hyperparameter, which controls the ratio of exploration/exploitation.

Since by visiting any chance node, d must be visited previously, so the last term is gradually increasing for each unexplored action, maximizing UCT for scarcely explored chance nodes. However if that's the case, then as $T(d)$ approaches infinity, $T(c)$ will too, and so $\frac{\sqrt{T(d)}}{T(c)+1}$ will approach zero. Gradually removing the term from the equation completely, selection at d gives way to the *relative value* instead, exploiting the knowledge about expected cumulative rewards. Both the relative value and $\frac{\sqrt{T(d)}}{T(c)+1}$ are numbers between 0 and 1, which lets us compare them with ease and even weigh their importance, as expressed by including the parameter E.

After a chance node is selected, its corresponding action is executed; if the resulting state-reward pair exists among this chance node's children, selection continues from the related decision node, otherwise *expansion* begins.

4.2.2 Expansion + Rollout

With a new state, reward, or both being discovered, a new decision node is created and linked to the previously expanded leaf. For each valid action in this newly explored state, a chance node is created and linked to it, creating new leaves, and finally, the decision node is assigned an expected cumulative reward estimate. This is achieved by rolling the episode out from this state, taking random valid actions, and observing incoming rewards. The final reward one achieves using this random policy is assigned to the newly created node and represented by V in the Selection section.

4.2.3 Update

Now that a new decision node is added to the tree, its ancestral nodes are updated from leaf to root: their values are adjusted so as to keep representing the expected cumulative reward estimate with respect to the newly created

decision node:

$$V = \frac{1}{T+1}(r + TV), \tag{4.7}$$

where V is the value estimate, T the visit count, and r the value estimate of its updated or newly created child. Their visit counters are incremented afterwards.

4.2.4 Finalization

After the loop is complete, we claim that the probability of any valid action a being optimal at s_t is proportional to the visit count of its corresponding chance node, so we take the visit counts of the root's children, and divide them by their sum to get a probability distribution, which is returned by the MCTS algorithm. Although MCTS can be used as a standalone agent, it is neural-network-less and as such out of our work's scope, but it has been included as it is a crucial part of the next algorithm.

4.3 AlphaGo Zero (AZ)

DQN falls under a category of agents who utilize *value iteration*, where the agent tries to estimate the actual value function $\mathcal{Q}$ and extracts the optimal policy from this function, while *policy iteration* agents represent the policy directly and try to improve it to the point of convergence and optimality. These agents' neural networks accept an arbitrary state as an input, similarly to DQN, but instead of outputting the expected cumulative reward received by taking an action a, they output the probability of action a being part of the optimal policy. Neural networks can also branch and output multiple tensors. AlphaGo Zero, [32], is one such example: apart from the policy function, the model outputs a single real number, representing the input's expected Q-value, which is used during its own training procedure.

The authors of AZ utilize a **mini-max** Monte Carlo Tree Search technique to gradually improve the network's policy, while using the network to guide the MCTS. As such, AZ achieves state-of-the-art results in games of go, shogi, and chess, despite the sparse reward problem (the agent is rewarded only for winning a game).

However, unlike go, shogi, or chess, which are all **adversary** (= featuring two or more opponents instead of a single actor) and **deterministic** board games, where the agents strive to defeat their opponents, we plan to create a **single-agent stochastic** system based around survival optimization, and the reward will be based upon the amount of simulation seconds after

reaching a terminal state. As such, AZ will be unfit for solving it, for it will not fulfill AZ's contract. Fortunately, the cause of this limitation is merely using the mini-max version of MCTS in contrast to the expecti-max version described above. The mini-max MCTS does work in a very similar manner, except it has no need for *chance nodes*, as there is no such thing as chance in deterministic environments.

4.3.1 Introduced MCTS adjustments

As the removal of chance nodes alone introduces no non-trivial changes to the standalone MCTS itself (when the currently selected node is a *chance node*, its action is immediately executed and the algoritm proceeds to the next *decision node*), in this section, we will focus exclusively on the tweaks AZ introduces to MCTS in the Selection and Rollout procedures.

Rollout

While the standalone MCTS utilizes a random policy to receive an estimate about the value of an expanded state node, since AZ uses a neural network which predicts state values, we can skip the rollout step and instead assign the network's predicted value of the new state to the new node. This approach does not only save time, but also yields better results as the network improves.

Moreover, as the network also outputs the likelihood of each action leading from the new state being optimal, AZ utilizes this information to bias the selection step towards such nodes, as it is a waste of time to explore suboptimal nodes. When the new state node is created, apart from the predicted value, it also receives a *prior probability* distribution P from the network, which corresponds to the policy probabilities of each individual action one can take from this state.

UCT

This was the idea behind claiming that AZ uses its neural network to *guide* the MCTS. When the adjusted MCTS selects nodes for expansion, it too does so based on a UCT function, but a new term is added to the equation, the prior probability.

$$UCT(c) = \frac{V(c) - \min_{c' \in C} V(c')}{\max_{c' \in C} V(c') - \min_{c' \in C} V(c')} + P(c) \cdot E \frac{\sqrt{T(d)}}{T(c) + 1}, \qquad (4.8)$$

Finalization

As the MCTS returns the policy probabilities at s_t proportional visit counts, [32] uses this strategy only for $t \in [0, 30)$; for larger t, the authors let the returned probability distribution have probability 1 for the most likely optimal action and 0 for other actions, but research conducted by [25] shows that this may have negative impact on performance, so we have decided not to utilize this approach.

4.3.2 The AZ algorithm

As we now obtain means to get a rough estimate of the optimal policy, we can now describe the full AlphaGo Zero training algorithm in Algorithm 4.2.

Lines 7-26 show a single training episode. This is where special *save* and *restore* operations come into play; after the current state changes in the environment simulator, it is repeatedly restored before the selection phase to properly enable tracing the state tree from the current state. Further discussion regarding these operations is found in Section 6.6.

Line 9 initializes the Monte Carlo tree, whose root - after expansion - receives a bias for its *prime probabilities* drawn from a Dirichlet distribution at line 11. This is encouraged by [32], as it helps exploration, all the while being a *noise*, which the network can learn to filter.

Lines 12-19 show the MCTS algorithm, employing the procedures described in the previous section. Afterwards, p_t is generated, a_t^* identified and executed, producing s_{t+1} and r_{t+1}. The original state is stored in a tuple along with the estimated optimal policy probability and the true received reward. After the episode is finished, lines 27-29 use the immediate reward values to assign cumulative reward values to each state, as the stored tuples are used for training. The training itself is executed only after N successive episodes.

After each episode, if the newly generated state, s_{t+1}, exists in the current MCT, the corresponding decision node is set to be the new root and the Dirichlet noise is added to its *prior probabilities.* Since the network never changes in-between episode steps, the new partial MCT would be generated in roughly the same way (not exactly if $\mathcal{T}$ and $\mathcal{R}$ are stochastic, but the data is valid nonetheless), so it is preserved and time is saved as the next MCTS procedure will only need to generate $M - T(new_root)$ nodes.

As the network is being trained, its *value head* (= output of the expected state cumulative reward) is trained on true values sampled from the environment, which lets it converge very quickly, compared to the *policy head* (=

Algorithm 4.2 alphago_zero_algorithm

1: input: $\mathcal{R}, \mathcal{T}$, initial state
2: initialize $\mathcal{P}_\theta, \mathcal{V}_\theta$ - single neural network with two outputs
3: training_data $\leftarrow$ empty arraylist
4: **repeat**
5: $\quad t \leftarrow 0$
6: $\quad s_t \leftarrow$ initial state
7: $\quad$ **repeat**
8: $\quad\quad$ $save(s_t)$
9: $\quad\quad$ $root \leftarrow (s_t, 0)$
10: $\quad\quad$ $expand(root)$
11: $\quad\quad$ $P(root) \leftarrow (1-\epsilon) \cdot P(root) + \epsilon \cdot Dir(\gamma)$
12: $\quad\quad$ **for** $i = 0$ to M **do**
13: $\quad\quad\quad$ $restore(s_t)$
14: $\quad\quad\quad$ $leaf \leftarrow select(root)$
15: $\quad\quad\quad$ $expand(leaf)$
16: $\quad\quad\quad$ **for all** $node \in path(leaf, root)$ **do**
17: $\quad\quad\quad\quad$ $update(node)$
18: $\quad\quad\quad$ **end for**
19: $\quad\quad$ **end for**
20: $\quad\quad$ $p_t \propto T(c)\ \forall c \in children(root)$
21: $\quad\quad$ $a_t^* \leftarrow \text{argmax}_{a' \in \alpha(s_t)}\{a' | p_t(a')\}$
22: $\quad\quad$ $s_{t+1} \leftarrow \mathcal{T}(s_t, a_t^*)$
23: $\quad\quad$ $r_{t+1} \leftarrow \mathcal{R}(s_t, a_t^*)$
24: $\quad\quad$ training_data $\leftarrow$ training_data + (s_t, p_t, r_{t+1})
25: $\quad\quad$ $t \leftarrow t + 1$
26: $\quad$ **until** s_{t+1} is terminal
27: $\quad$ **for all** $(s_i, p_i, r_{i+1}) \in$ training_data **do**
28: $\quad\quad$ $(s_i, p_i, r_{i+1}) \leftarrow (s, p, \sum_{j=0}^{t-i}(r_{j+1}))$
29: $\quad$ **end for**
30: $\quad$ **if** N-th episode **then**
31: $\quad\quad$ train $\mathcal{P}_\theta$ on (s, p)
32: $\quad\quad$ train $\mathcal{V}_\theta$ on (s, v)
33: $\quad\quad$ training_data $\leftarrow$ empty arraylist
34: $\quad$ **end if**
35: **until** convergence

output of the likelihood that each action is part of the optimal policy), which is trained on MCTS-generated data, since the probabilities as MCTS outputs

are heavily biased by the network, unlike the value. Looking back at equation 4.8, without the influence of relative value - the first term - the chance nodes would be selected more-or-less according to the prior probabilities, and so the MCTS' output would differ insignificantly from the network's original predicted policy at s_t. But as the network learns to evaluate each state, it consequently gains the ability to improve its policy.

5 Implementation analysis

The two following chapters will focus on the implemented systems we used for all of our experiments. As was suggested in the introduction, we have created a standalone environment and custom agents, which we compare in terms of performance.

The environment being composed in order to remotely resemble the computer game They Are Billions[1], we have conveniently named it They Are Trillions (or TAT, for short), in honor of the inspiration. The simulation takes place on a two-dimensional surface, where the agent constructs different types of buildings in order to protect its main building, which is the target of an ever-increasing horde of infected. The goal of the agent is to find such policy that maximizes survival time of the main building, which classifies the environment as a *tower defense* [5]. We designed it to be both stochastic and partially observable, setting the bar high enough to be unsolvable by simpler agents, and implemented it in C for maximum performance.

As all of our agents will need neural networks to function, we have decided to implement the agents in Python, partially using TensorFlow and Keras, which are AI frameworks.

Our implementation will use TCP sockets as a means of transporting TAT-relevant information to agent processes. It is reliable, fast over short distances, but allowing for distributed computing as well, and frees agents from the necessity of being implemented in the same language. As per our assignment, the agents are to be implemented using TensorFlow, which is primarily used in Python. Although TF API exists for multiple different languages outside of Python, including C, the API is limited and trying to use it would be an unnecessary hindrance. UDP datagrams, although fast, are unreliable, and using pipes provides no significant advantages over TCP sockets while losing the possibility of distributed computing.

Since the environment is used for querying states, it makes sense to think of it as a stateful server, which would make the agents the clients. As such, we intend to run the TAT process from the agent processes. The final product of our work will therefore consist of two separate programs; the Python agents (the exact type selectable by an input argument), and the TAT server.

[1] http://www.numantiangames.com/theyarebillions/

6 Custom environment design

In this chapter, we will describe all principles and interactions in TAT, while introducing a considerable amount of tunable constants, which can be changed directly in our code to adjust TAT's behavior at will. Their default values are included in our text and their impact on TAT will be briefly discussed at the end of this chapter.

6.1 Basics

6.1.1 Map

The *map* in TAT is a 2D grid consisting of 13x13 squares, or *fields*. This particular size has been chosen in order to keep the map shape a square, allow for true central coordinates (odd size) while limiting the total amount of possible states. We call the length of field's side a *unit*. Each field has discrete integer coordinates.

6.1.2 Entities

Two kinds of entities reside on the map, the *buildings* and the *infected*. The infected roam freely, while the buildings are immovable and locked to the field they have been built on. Each field can contain one building.

Combat

Both types will be thorougly described later, however, their key common feature is posession of a certain amount of *health* $\in \mathbb{R}^+$. The buildings interact with the infected via the process of *combat*, where one can attack the other and vice versa, inflicting damage and reducing health. If an entity's health depletes to zero, it is destroyed/killed. Killing an infected has no exceptional influence on the simulation, however, destroying a building by infected results in all the building's imaginary non-infected occupants to become new infected, *spawning* at the center of the building's field.

6.1.3 State transitions

The simulation contains a timer, t, which measures the amount of seconds since the initial state. It is technically real-time, however, since the action

space includes a 'do nothing' action, or an *idle* action, it can be viewed as turn-based. For a human, the environment behaves like a *dynamic* one: after each tick, if you do not specify an action, since the *idle* action is always valid, it is automatically taken for you; in contrast, the artificial agent is queried about which action to take after every tick of the simulation, pausing it in the process. By definition, this makes the environment *static*, since there is always some action between two different states, however, the agent has the option to not influence the environment's state by taking the *idle* action, so the environment's state changes purely by its internal transition procedures. A theoretical alternative would have been to let the agent choose an action based on a state s_t while updating to a state s_{t+n} until the agent returns an action and then updating to $s_{t+n+1} = \mathcal{T}(s_{t+n}, a_t)$, but we have ultimately decided against it, with the reasoning being the following: since the returned action would have been based on an outdated state and thus outdated set of valid actions; this would force the simulator to use the idle action as a sink for all the possible invalid outputs the agent would inevitably produce. This n would be a random variable dependent on the machine's conditions, particularly its computational power; if we were to train such an agent, deploying it on different machines would yield different policies, rendering the approach unusable. Even if the technology would eventually progress so far that for any conceivable agent n would be always zero, one could always find a smaller $\Delta t > 0$ tick time for which some agent implemented using the technology will be unable to map current states to appropriate actions in time.

6.2 The Infected

6.2.1 Appearance and position

The Infected are the antagonistic force in TAT and their influence will ultimately lead to terminal states. From the perspective of the environment, each infected is an infinitesimally small *point* on the map, its position being decimal (*e.g.* $[1.0, 1.0], [5.67, 12.83], [0.5, 0.7]$, etc.), represented by IEEE 32-bit floating point numbers, which makes them continuous, just like in the original game. Since the size of the map is 13x13, both its coordinates must always be in range $[0, 13)$. If such a coincidence occurs, multiple infected can occupy the same exact coordinates.

At the initial state, no infected exist on the map, and their number is increased through the process of *spawning*. To *spawn* is synonymous to 'create' at some location $[X, Y]$, spawn being the contextually more appro-

priate word. The infected can be spawned through two possible means, either through a *horde spawner*, which is a special field on the map at $[6, 0]$, spawning t infected every 5 seconds, or by successfully destroying a building. Both the spawner's coordinates and its spawn rate are adjustable. The reason behind choosing these default values was to give the agents enough time to setup defences and prepare for the invasion, which allows for far greater complexity of resulting strategies.

6.2.2 Movement

Basics

Likely the most obvious difference between the infected and the buildings is the former's ability to move around the map. If an infected is moving, it always has a concrete floating point heading $[X, Y]$, to which it approaches at velocity *1 unit per second*, which is constant for all moving infected. Not unlike as with the spawner rate, this exact value became the infected velocity mainly because it balances episode time.

Non-moving infected have velocity 0.

Movement bias

Every infected also has its own *movement bias* distribution $\mathcal{B}$, which is a discrete distribution over the set $\{0, \frac{2\pi}{p}, \frac{4\pi}{p}, ..., \frac{2(p-1)\pi}{p}\}$, where $p = 2k, k \in \mathbb{N}$. The purpose of this distribution is to bias the random movement of the infected (described below) towards the main building's field. The parameter p specifies the amount of distinct angles in $\mathcal{B}$. Infected spawned by the horde spawner always have their movement biased towards the main building, since they are part of the original attacking horde, as opposed to infected spawned from destroyed buildings, for which $\mathcal{B}$ is uniform (no bias), since these infected are former inhabitants and as such are excluded from the original horde. The exact shape of $\mathcal{B}$ is determined by Algorithm 6.1.

The final shape of $\mathcal{B}$, if wrapped around a point and represented in polar coordinates, resembles the shape of a discrete ellipse, as shown in Figure 6.1. The point $[0, 0]$ in these polar coordinates is one of the ellipse's focal points. The parameter p determines its ruggedness and e its eccentricity. Its maximal radius, which would normally be a, is non-parametric, since as $\mathcal{B}(i)$ is a probability distribution, it is required for $\sum_{i=0}^{p} \mathcal{B}(i)$ to equal 1. Realizing this, one could calculate a at line 2 immediately from e and p, but such calculation can be easily avoided: instead, we ensure that $\mathcal{B}(i)$ has the properties of a probability distribution function by line 16. Setting a to any

Algorithm 6.1 MB_focus

```
1: input: target angle α_f ∈ [0, 2π), eccentricity e ∈ [0, 1), precision p = 2k
   for k ∈ ℕ
2: a ← 1
3: focus_index ← ⌊pα_f / 2π⌋
4: opposite_index ← (focus_index + p/2) mod p
5: for θ = 0, i = focus_index to opposite_index do
6:    B(i) ← a(1−e²) / (1−e cos(θ))
7:    θ ← θ + π/(p/2)
8:    i ← (i + 1) mod p {first half clockwise}
9: end for
10: for θ = 0, i = focus_index to opposite_index do
11:    B(i) ← a(1−e²) / (1−e cos(θ))
12:    θ ← θ + π/(p/2)
13:    i ← (i − 1) mod p {second half counter-clockwise}
14: end for
15: ∀i ∈ ℤ ∩ [0, p) : B′(i) ← B(i) / Σ_{j=0}^{p} B(j)
16: return B′
```

positive number has no influence on the results (although numbers close to 0 can introduce rounding errors).

α_f is simply the angle between $[1, 0]$, the reference vector, and $[X_{MB} - X_{infected}, Y_{MB} - Y_{infected}]$. For p, we use 256 for ease of computation, since using unsigned 8-bit integers removes the need for modulation by 256, while allowing the infected to move in a great range of distinct angles. e has been empirically set to 0.8, so that the infected largely prefer to move towards the main building as a horde, but preserve some notion of individuality, as they are completely unorganized. Although unlikely, Figure 6.1 shows that it is still quite probable for an infected to deviate from the best possible angle, while moving away from the main building happens very scarcely.

Heading selection, random_roam

By default, the infected are *randomly roaming.* The random roam procedure begins immediately after spawning and is described by Algorithm 6.2:

This means that the infected always choose a random heading, approach it at velocity 1 unit per second, and wait for a second until choosing another heading again, imitating the intended shamble-like movement of the infected. By setting p to 256, we allow the infected to move in 256 different angles with respect to the vector $[1, 0]$, which we claim to be a negligible constraint

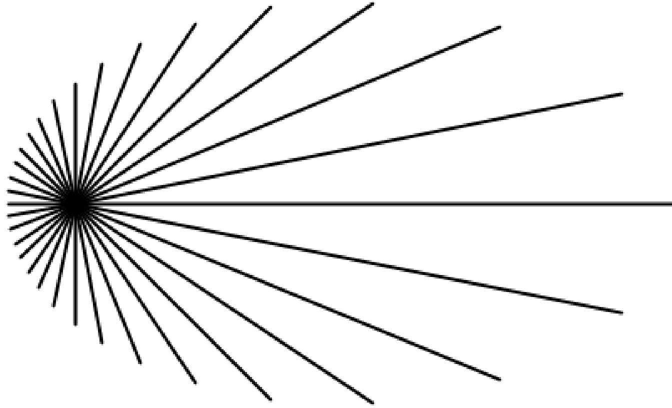


Figure 6.1: $\mathcal{B}(i)$ when visualized in radial coordinates. The longest line points towards the main building. This distribution has e set to 0.8, just like in the current TAT version, however p has been reduced to 16 in order to emphasize the ruggedness.

on their movement.

6.2.3 Aggression

Vision range

From now on, we will commonly use $L_2(p_0, p_1)$ as the Euclidean distance between inputs p_0 and p_1. Unless specified otherwise, a field has integer coordinates (by definition).

Each infected has what is called a *vision range*. Let $\mathcal{M}$ be the map, a set of all fields, and F be the field the infected is currently standing on (it has coordinates $[\lfloor x \rfloor, \lfloor y \rfloor]$ if we let $[x, y]$ be the infected's true coordinates). Then, the vision range is a set of fields $R_{infected} = \{f \in \mathcal{M} : L_2(F, f) \leq \sqrt{29}\}$, giving it a shape of a rasterized circle, as seen in Figure 6.2. The vision range's radius, $\sqrt{(29)}$, has been set to this value in order for the rasterized circle to contain no sudden discrete spikes (as shown in Figure 6.2, we can see that it is quite smooth), and for optimalization reasons, the implementation does not allow its easy customization (as this vision range is used for initializing the map's underlying structures, the points are enumerated manually).

Charging the buildings

The infected will immediately charge and attack any building built on a field within their vision range, with disregard for its type or any other status. Figure 6.3 shows how if we let A be the position of the infected and B be the

Algorithm 6.2 random_roam

```
1: input: position of infected x, y, movement bias of infected B
2: repeat
3:    r ~ U(2, 3) {a uniform distribution}
4:    α ~ B
5:    Δx ← r cos(α)
6:    Δy ← r sin(α)
7:    for all coord in x, y do
8:       if coord + Δcoord ∉ [0, 13) {if coordinate is out of map bounds}
         then
9:          Δcoord ← −Δcoord
10:      end if
11:   end for
12:   infected heading ← [x + Δx, y + Δy]
13:   walk to the heading destination
14:   wait 1s
15: until disturbed
```

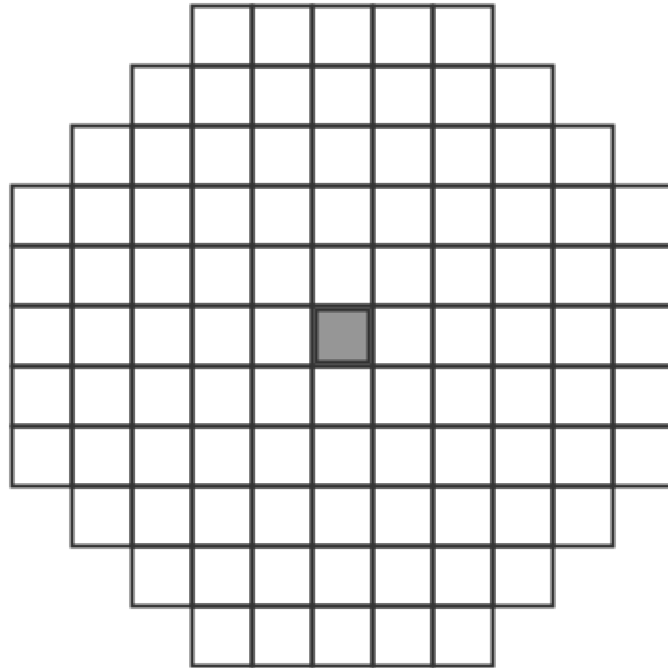


Figure 6.2: The unclipped vision range of infected standing at the center field. Standing too close to the map borders clips the range.

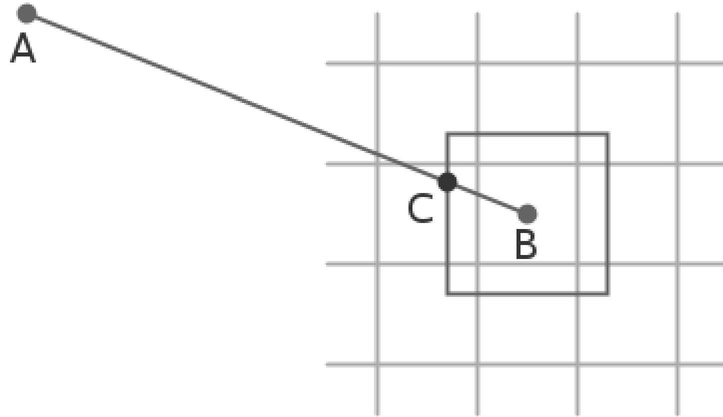


Figure 6.3: Shows the heading of a charging infected (C). The square's inscribed circle radius is not to scale.

charged building's field center, charging a building simply sets the infected's heading to a point on the intersection of line AB and a square centered at B, parallel to the coordinate axes, and with inscribed circle radius $= 0.55$. This movement is uninterrupted by pauses and can be performed under any real angle, unlike when randomly roaming.

Target selection

The building content of an infected's vision range can change as a result of either creation/destruction of buildings inside its current vision range, or by changing the vision range directly by the infected moving to a different field. Each is handled in their own way, as defined by Algorithm 6.3.

Lines 4, 6, 11 and 18 are of particular interest to us, as they determine the infected's target in non-trivial situations. Lines 11 and 18 select the target building according to L_2 **distance** between its field's center and the infected, however, step 6 can select **any** newly seen building, regardless of true distance to the infected. Moreover, at step 4, we claim that no closer building than B exists. The motivation behind such different approaches is computational optimization.

The original aim is for the infected to always charge the closest building, which is what lines 11 and 18 achieve. The edge case occurs when the vision range content changes as a consequence of the infected moving around and changing its field in contrary to a building being created or removed directly inside its vision range. This is visualized by Figure 6.4.

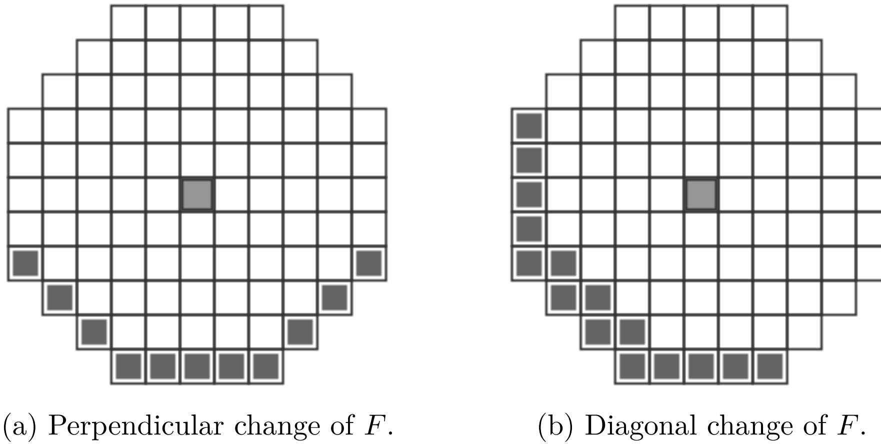
Notice the small variance between distances from fields in the *zone of immediate entry* to the central field. The average difference between relat-

Algorithm 6.3 handle_vision_range_content_change

```
1: input: R_infected at previous field R_prev, R_infected at current field R, currently charged building B, current field F
2: if F of infected changes then
3:     if B exists then
4:         do nothing
5:     else
6:         charge the building at first/any field ∈ R \ R_prev
7:     end if
8: else if a building b is created on a field within R then
9:     if B exists then
10:        if center of b's field is closer to the infected compared to B with respect to L_2 then
11:            charge b
12:        end if
13:    else
14:        charge b
15:    end if
16: else if B is destroyed then
17:    if R contains other buildings then
18:        charge the building of the field within R whose center is the closest to the infected with respect to L_2
19:    else
20:        proceed with random roaming
21:    end if
22: else
23:    do nothing
24: end if
```

ive closeness of any newly seen buildings and the truly closest newly seen building to the central field is indeed so small that the trade-off for increased performance has been deemed insignificant: instead of comparing potentially all 11 buildings in case of perpendicular central field change, or 15 buildings in case of diagonal central field change, Algorithm 6.3, line 6 claims that any such building is the closest and therefore saves time at negligible accuracy cost.

Since the infected begins charging the building directly after noticing it, no building in any of the following immediate entry zones can be closer to the original target, which is exploited by line 4.



(a) Perpendicular change of F. (b) Diagonal change of F.

Figure 6.4: If a vision range moves and fields with new buildings appear in it as a consequence, the dark gray area shows all such possible fields for horizontal, vertical or diagonal movement. We call sets of such fields the *zone of intermediate entry.*

Combat

When an infected finally reaches its heading at the end of the charge, it starts attacking the building in an attempt to destroy it. Each infected reduces their target building's health by 20 per second. After the building is destroyed, the infected finds a new target (Algorithm 6.3, line 18), or continues roaming randomly (line 20). This is true for all infected participating in destruction of the same building or those who are still charging it.

An important consequence of building destruction is the spawning of additional infected. After a building is destroyed, $s \sim \mathcal{G}(c_b)$ infected are spawned at the center of the destroyed building's field, where $\mathcal{G}$ is the geometric distribution and c_b is their expected amount. c_b is constant among the same type of building, but varying across different building types, as shown by Table 6.1.

All infected have health = 100, which is modifiable, much like their *damage per second*, both being intertwined with the values in Table 6.1. These values are comparatively small, which makes the infected very weak individually, but dangerous in large amount, as was intended.

6.3 The buildings

6.3.1 Appearance and position

The other fundamental entity in TAT are buildings, which provide resources and protection for the agent's colony. The agent must construct buildings so as to delay the inevitable destruction of its main building at field $[6, 11]$ - by the ever-increasing amounts of attacking infected - as long as possible. The location is purposely designed to be far away from the horde spawner so as to give the agents enough time to prepare their defence.

There are 5 different types of buildings (4 available for construction), all having distinct properties. Each field on the map contains at most one building.

Construction of a building takes infinitesimal time and costs *gold*. *Gold* $g \in \mathbb{R}_0^+$ is a special variable in TAT, acquired over time and spent purely on building creation. The agent uses a built-in field selector called the *target location* for this purpose. At s_0, the target location is $[6, 6]$ and $g = 30$. Through an action, the agent can either perpendicularly move the target location, build one of the buildings directly at the target location, or remove a building at the target location (which has no influence on and is not influenced by the current amount of gold). Individual building costs are shown in Table 6.1. The initial target location is set to the map's center in order to spare the agent a few actions should it decide to start building at any point on the map, and gold is set to 30 in correspondence to the building costs. Non-zero initial gold further increases the amount of available strategies.

6.3.2 Types

Main building

Works as a government center and a power plant for the colony. It is unique, unbuildable, irremovable and all states in which it is not existing are terminal.

Not unlike the infected's vision range, the main building has what is known as a *powering range*. The definition is the exact same: a set of fields on the map which are as close or closer to the main building's field than r units; the radius stays the same, $\sqrt{29}$. All buildings built on fields withing this area are considered *powered*. Some building types require electrical power in order to function properly, and the main building is its only source. The powering range's radius is adjustable and set to this specific value for

the exact same visual motivation as in with the infected's vision range.

The main building also generates gold at a continuous rate of 3 per second, which is tunable.

House

A fragile building, generates gold continuously at a rate 2 per second, also tunable. Gold generation stacks additively with both other houses and the main building. We went for these exact values particularly in order to motivate high risk/reward behavior in the agents. With houses generating nearly as much gold as the main building, we theorize that the agent should prefer to build them a lot, which forces it to properly defend the increasing area of low-health buildings.

Turret

A defensive building and the only one that is capable of inflicting damage upon the infected. It has its own *attack range*, similar to the other ranges, but with radius $= \sqrt{10}$ units. This particular value posesses the same visual property as $\sqrt{(}29)$, but is smaller to prohibit turrets from killing the infected from an unsatisfactorily safe distance. A turret's attack algorithm is much simpler that that of infected: it focuses on a single infected inside its range until said infected dies or leaves the range, and always attacks if there are any infected in range, the exact target being chosen 'randomly' (first in the set).

Turrets need power in order to attack. An unpowered turret simply idles and starts attacking immediately after power is supplied, and a powered turret stop attacking after power is lost.

Tesla tower

Spreads electrical power across the map. The Tesla towers (teslas for short) have the same *powering range* as the main building, however, they do not generate electricity themselves; they work instead as a 'repeater', extending the range of power distribution. Practically speaking, if a tesla stands on a powered field, it powers all fields within its powering range; if it stands on an unpowered field, it doesn't do anything and stands idle, waiting to get powered up.

Let the set {tesla, main building} be called *nodes*. Since the ranges of all nodes are the same, it follows that $A, B \in nodes : field(A) \in range(B) \iff field(B) \in range(A)$, making A and B a *connected* pair

of nodes. Consequently, $A, B, C \in nodes : conn(A, B) \wedge conn(B, C) \implies conn(A, C)$ by definition, and clearly $field(A) \in range(A)$ for any $A \in nodes$. This gives rise to an equivalence relation R between all nodes existing on the map at any point, and if we let V be the set of such nodes and $E = \{(A, B) : conn(A, B)\}$, we create an undirected *powering graph* $\mathcal{G}_p = (V, E)$ with multiple possible connected components, which correspond to the equivalence groups of R.

Whenever a tesla is added to or removed from the map, the graph is reestablished and the connected components recomputed. After that, the component containing the main building, C, is claimed to be powered, which in turn powers on all fields $f \in \bigcup_{N \in nodes(C)} range(N)$.

Wall

A wall is a cheap defensive building with a solid amount of health and a small amount of expected spawned infected at destruction. That being said, its only purpose is to provide a target for the infected while other buildings do their respective jobs.

Type	**Cost**	**Health**	**DPS**[1]	c_b
Main building	-	500	-	-
House	10	100	-	4
Turret	40	500	50	4
Tesla tower	50	100	-	0.5
Wall	20	500	-	1.5

Table 6.1: Building stats

6.4 Simulator

A TAT episode either unrolls linearly (infected movement, combat damage), or processes an event which interrupts its linear behavior, such as an infected being spawned, a building being created, infected entering turret ranges and getting attacked, arriving at headings, etc. By identifying and properly handling each linearity breaking (LB) event, all of which are given by sections 6.2, 6.3, we arrive at a procedure which allows us to increment the state timer t by any positive Δt (Algorithm 6.4).

[1]Damage per second

Algorithm 6.4 integrate

1: input: $\Delta t > 0$
2: $t' \leftarrow t + \Delta t$
3: **while** LB event will occur between t and t' **do**
4: $\quad t_{ev} \leftarrow$ time at which the soonest LB event occurs
5: $\quad simulate_until(t_{ev})$
6: $\quad handle(ev)$
7: **end while**
8: $simulate_until(t')$

Algorithm 6.5 simulate_until

1: input: final time t'
2: **for all** moving infected **do**
3: $\quad infected_pos \leftarrow infected_pos + (t' - t) \cdot infected_velocity$
4: **end for**
5: **for all** ongoing combats **do**
6: $\quad target_hp \leftarrow target_hp - (t' - t) \cdot attacker_dps$
7: **end for**
8: $h \leftarrow$ amount of houses present
9: $g \leftarrow g + (3 + 2h)(t' - t)$ $\{g =$ amount of gold$\}$
10: $t \leftarrow t'$

We have opted for such method of state incrementation particularly because transforming the state s_t (time t) to state s_{t+r} (time $t+r$) by Algorithm 6.4 produces constant, predictable results for any choice of Δt. The state increments are integrated together and provide stable outcomes. The ability to adjust Δt is utilized during training, evaluation, human testing and visualization.

6.5 Actions

All possible actions have now been mentioned at least once in previous sections, but let us recapitulate them and specify conditions under which they are valid.

We have omitted the fact that the main building must exist in order for any action to be valid in Table 6.2, since it is true for all actions and thus impractical to include in the table.

id	Short description	Valid
0	Idle (pass, do nothing)	Always
1	Subtract 1 from the target location's first coordinate c_1	$c_1 > 0$
2	Add 1 to the target location's first coordinate c_1	$c_1 < 12$
3	Subtract 1 from the target location's second coordinate c_2	$c_2 > 0$
4	Add 1 to the target location's second coordinate c_2	$c_2 < 12$
5	Build a house at the target location	$g \geq 10$, no buildings, infected or spawners occupy target location
6	Build a turret at the target location	$g \geq 40$, no buildings, infected or spawners occupy target location
7	Build a tesla at the target location	$g \geq 50$, no buildings, infected or spawners occupy target location
8	Build a wall at the target location	$g \geq 20$, no buildings, infected or spawners occupy target location
9	Remove building at the target location	such building exists, is not of type 'main building' and is not charged by or in combat with any infected

Table 6.2: TAT actions with validity conditions

6.6 Control actions

Section 6.5 enumerates all the possible actions an agent can take in TAT, however, in order to enable execution of various training procedures over the environment, three more distinct control actions are provided for the agent systems to administer the environment's state.

The first is the ability to *reset* the environment, which automatically sets the current environment state to the initial state, allowing the agent to apply reinforcement, search the state space and train itself after the episode ends up in a terminal state.

The second and third being options to *save* and *restore* the state, which is essential for certain agent types' training procedures, including those discussed in our work. Manual saving and restoring of states is attainable on the agent's side: store the *action history* that led to s_t (= sequence of actions applied after *reset*, resulting in s_t) to save the state, and reset the

environment before applying actions in the stored action history in the order they were taken to restore the state. This is a perfectly valid (albeit still sub-optimal) approach in deterministic environments, however in stochastic environments, such as TAT, a certain probability $P(s_0 \xrightarrow{history} s_t) \in [0, 1]$ exists, invalidating the idea of following a set action history to end up at a desired state. Moreover, since

$$P(s_0 \xrightarrow{history} s_t) = \prod_{i=0}^{t} P(s_i \xrightarrow{a_i} s_{i+1}), \tag{6.1}$$

the probability becomes exponentially small as t approaches infinity, so if one decided to repeatedly follow a stored action history until s_t is finally obtained, resetting in case of failure, the expected complexity of such an approach would lie in $\mathcal{O}(\mathbb{E}|\mathcal{T}(s,a)|^t)$. This would still be a valid method of state restoration for environments with short episodes, fast *reset* and *act* and slow or impossible *save* and *restore*, but since TAT episodes are comparatively long and its state transitions are more complex than its representation, it is very recommendable for the actual TAT realization to provide *save/restore* control actions itself.

This breaks the original assertion that one agent should solve multiple diffenent problems, because in order for any state/restore-dependent agent to solve multiple problems, all of the representing environments must implement the functionality. However, as there is currently no trivial way of transforming a real world problem into an environment implementation, the RL designer will be forced to create the environment himself and thus can include the save/restore functionality while being able to reuse the depending agents.

A fourth control action, *terminate*, is included for convenience, and signals the environment that no more actions will be input by the agent system, allowing for clean termination.

6.7 State output

TAT provides rough map information to the outside observer. The state encoding informs agents about the conditions of each individual field, the current *target location* and the amount of gold available. If a building occupies a field, the agent is given information about its type and the amount of *sixteenths* of its maximum health: if it was 500, then 16 for health in [500, 468.75), 15 for health in [468.75, 437.5), etc.; if the infected occupy a field, only their number at this particular field clipped to 127 is given out to the

agent. The target location and gold is always disclosed precisely. The reasoning behind the specific building health and infected count outputs is the ability to fit the full field description into a single byte during the encoding phase, which is discussed in Section 6.9.

6.8 Data structures

As we have claimed in the Analysis chapter, the TAT environment has been implemented in C with our primary focus being computational performance. With no regard for ordering preservation, we utilized the trademark ability of *hashsets* and *hashmaps* to adjust their size according to needs with their original size having no influence on manipulation time. Sets have been created for each distinct type of entity, grouping them in terms of interaction with each other (such as *buildings, infected, moving infected, ongoing combats,* etc.), and Algorithm 6.4 used for state transitions. For querying the first occurring LB event, we deployed a *priority heap*, which is being filled with identified LB events over the course of simulations.

The priority heap sorts LB events according to time of occurrence and allows us to update this time should the need arise; an example being the following: say $t = 10$ and an infected A finishes charging building Z, arriving when Z's health is 400, and as a consequence, a *building_destroyed* event E at $10+400/20 = 30s$ is added to the heap. If an infected B charges the same building, arrives and starts attacking at $t = 12$, Z's health will already be at 360, and so E's time is advanced to $12 + 360/(20 + 20) = 21s$. B might become a target of an attacking turret, and if it dies at $t = 18$, Z's health will be 120, and E's time will be delayed to $18+120/20 = 24s$. The priority heap has been designed with this necessity in mind (see Table 6.3), so that it is capable of shifting the events around based on their changing priority (time of occurrence).

Saving and restoring of states is achieved by directly cloning the underlying data structures. Such approach is efficient and only requires correct pointer rewriting. An alternative would have been to use the application checkpoint utility called CRIU, which dumps the entire process' state to disk, but among other communication related difficulties this method would introduce, CRIU would backup the state of the pseudorandom number generator (PRNG) Xoroshiro +128 [7] we use, which would in turn determinize resulting states given an action history on restoration. The PRNG's seed would have to be manually reset to a random number after each restoration, but that would require another PRNG in the first place. The seed would

Name	Contract	T(n) of implementation
Hashset	Unordered set of unique items represented by a *hashcode* function, requires *add(item)*, *remove(item)* and *foreach(action)*	add $\in \mathcal{O}(1)$ remove $\in \mathcal{O}(1)$ foreach $\in \mathcal{O}(n)$
Hashmap	A general mapping between sets of items X, Y. Requires *add(x, y)*, *remove(x)*, *get(x)* and *foreach_pair(action)*	add $\in \mathcal{O}(1)$ remove $\in \mathcal{O}(1)$ get $\in \mathcal{O}(1)$ foreach $\in \mathcal{O}(n)$
Priority heap	A self-ordering data structure sorting elements by *key(item)* function. Requires *insert(item)*, *peek_first*, *pop_first* and *update(item)*	insert $\in \mathcal{O}(log(n))$ peek_first $\in \mathcal{O}(1)$ pop_first $\in \mathcal{O}(log(n))$ update $\in \mathcal{O}(log(n))$
Arraylist	An ordered set of items, where removal is unnecessary. Requires *add(item)* and *foreach(action)*	add $\in \mathcal{O}(1)$ foreach $\in \mathcal{O}(n)$

Table 6.3: The most important data structures used in TAT implementation. T(n) is *amortized.*

have to be generated and supplied from outside the TAT process, which we consider inapt.

6.9 Communication

6.9.1 Protocol

All incoming and outgoing information must therefore be serialized into a byte stream in a reconstructible manner. For this purpose, we have developed a custom communication protocol, which we describe in this section.

TAT first output

As soon as the agent's socket is connected, TAT sends out essential information that stays constant throughout all episodes: a **23 byte** message containing the following in order: **map edge size** (**1 byte**, theoretically up to 256 (as the coordinates are represented using 8-bit integers), but is 13 during all of our simulations and testing), **total amount of possible actions** (**1 byte**, theoretically up to 32, as the actions are transmitted using 5 bits [see TAT input], but the current version includes only the 10 actions from Section 6.5), **the amount of building types** (**1 byte**, theoretically up to 8, as the building types are transmitted using 3 bits [see TAT regular output], the current version includes 5), and **the amount of maximum health for each building type** (**4 bytes**, IEEE float 32-bit, in order 'main building, house, turret, tesla, wall').

TAT input

As the agent only needs to specify action or control action id, it needs a **single byte** to do so. The **3 most significant bits** represent the control action id, 1 for reset, 2 for save, 3 for restore and 4 for terminate. If they are zero, the **5 least significant bits** are interpreted as the desired action id, and the respective action is taken.

TAT regular output

For each incoming action, the TAT environment must send out four variables: the *encoded state*, the *reward* for taking this action, which following *actions* are *valid*, and whether the current state is *terminal*.

The first **13 bytes** of each message are fixed. These contain the reward value (**4 bytes**, IEEE float 32-bit), the gold value (**4 bytes**, IEEE float 32-bit), the target location (**2 bytes**, 1 byte for each integer coordinate), the valid actions (**2 bytes**, where the i-th most significant bit reflects the validity of the $(i-6)$-th action; 1 if valid and 0 if invalid, Table 6.4), and the *flags* (**1 byte**, the least significant being 1 if the state is terminal).

VA bits	0	0	0	0	0	0	.	.	.	.	.	.	.	.	.	.
Significance	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Action	-	-	-	-	-	-	0	1	2	3	4	15	6	7	8	9

Table 6.4: Valid action encoding.

The remaining bytes of each message are used to communicate the rest of the encoded state, which is the positional information about field states. The state of a single field is encoded in **8 bits** the following way: the **most significant bit** is 1 if the field contains a building, and 0 otherwise. If the field does indeed not contain a building, the **next 7 bits** represent the amount of infected at this field, [0 - 126] for the respective amount or 127 for 127 or more infected. If however the field does contain a building, the **next 3 bits** encode its type, [0 - 4] for main building, house, turret, tesla and wall respectively, leaving 5, 6, and 7 unused, and the **last 4 bits** encode the amount of sixteenths of maximum health the building currently has, reduced by 1: [0 - 15], 15 for health in $(\frac{15}{16}, \frac{16}{16}]$, 14 for health in $(\frac{14}{16}, \frac{15}{16}]$, etc. A zero health building is never encoded, since it would be destroyed. The agent knows the maximum health values of each building type prior to this communication, so it can calculate the approximate real amount of health each building has.

Fields however carry positional information, which has to be communicated as well, as the map itself is a matrix (rank 2 tensor). Two different methods of matrix encoding exist: the *sparse* encoding and the *dense* encoding; the former explicitly communicating the coordinates of each encoding-worthy field before the actual field's encoding, ignoring empty fields, and the latter making use of perfect ordering of elements in a matrix, encoding each field in order and thus implicitly communicating each field's coordinates, but having to encode empty fields, which carry no useful information. Since both field coordinates are in the range $[0, 255]$, a *sparsely* encoded field takes up **3 bytes** of space while *densely* encoded fields only take up **1 byte**. Therefore, it is optimal to encode the map *sparsely* if and only if the amount of non-empty fields is less or equal to $\lfloor \frac{13^2}{3} \rfloor$, so while building the infected field encodings, we calculate their amount and add the amount of buildings; only then deciding whether to send each individual map state in a sparse or dense manner. If the dense encoding is currently optimal, the remaining 13^2 = **169 bytes** of the message contain field encodings, where the field at $[x, y]$ is encoded by the $(x + 13y)$-th byte, and if the sparse encoding is currently optimal, the second least significant bit in *flags* is set to 1 (indicating sparse encoding), the next **2 bytes** represent the required sparse encoding length,

and **each following triplet of bytes** contains the field's encoding, its x and its y coordinate respectively. Using sparse encoding, infected are included before buildings in the message, but this has no impact on the result. Their order is undetermined, as they are enumerated by their hashsets/maps.

The final length of each message is then either **182 bytes**, or **15 + 3n bytes**, specified inside the message in the latter case.

6.10 Main loop, parameter discussion

Concluding the entire chapter, we arrive at the top layer of our TAT system, which provides an interface for the agents to operate and navigate the environment.

Algorithm 6.6 main_loop

```
Require: free port
 1: input: Δt
 2: listen(port)
 3: client ← accept()
 4: message ← get_first_output()
 5: send(client, message)
 6: while not terminate do
 7:    repeat
 8:       input ← get_input(client)
 9:    until input exists
10:    parse_input(input)
11:    integrate(Δt)
12:    message ← get_regular_output()
13:    send(client, message)
14: end while
```

All of the default values of adjustable constants have now been explained, which leads us to describing what kind of alteration would changing them introduce.

Tampering with the map size, initial target location and gold would work as a difficulty setting, as it directly influences the state complexity and episode length. Changing the infected movement constants would probably not be desirable, as they could hardly be thought of as a shambling horde anymore. Vision, targeting and powering range adjustments should not be easy to roll out either, since their current shape specifically minimizes the influence of paradoxes raising from rasterizing a circle, such as the infected

suddenly changing their direction by large angles as a result of a building appearing in an outlier field in their vision range. Using the Bresenham midpoint circle algorithm we have implemented minimizes this problem and as such, when adjusting map size, these ranges can be tuned correspondingly. Finally, we strongly encourage experimenting with building costs, health, dps and c_b values, as that will directly influence which building types should be built as a part of the optimal policy. As the utility-cost ratio of each building is context dependent (some building types work well when clustered, some are dependent on each other for powering/protection, etc.,) finding the right balance is a non-trivial task and as such, we leave it as a part of future work.

7 Agents

As the environment is now defined and implementation decisions associated with it explained, we are now ready to delve straight into describing the individual agents we have attempted to solve TAT with.

7.1 DQN, AZ (EXPI)

As we have claimed in Chapter 4, we will use DQN as our baseline agent, introducing no significant adjustments to the algorithm described in [20].

The second tested agent will be the AZ with an expecti-max MCTS instead of the original mini-max version. Although the introduced change is negligible, it would be incorrect to call this agent "AlphaGo Zero", so although it is still heavily inspired and based upon [32], we have decided to dub it "Expecti-max Policy Iteration" (or EXPI for short).

7.2 Redundancy augmentation

In Algorithm 4.2, apart from ϵ and γ, which merely amplify exploration, we recognize two other hyperparameters, M, the amount of MCTS decision nodes the tree approximation will have before returning probability estimates, and N, the amount of episodes for which training data is generated before updating network parameters. We can see that by increasing them, we gain some training improvement for the cost of extending the total training time.

The larger N is, the smoother and more accurate the training, since more data is provided to the network, but as collecting the data takes N episodes, the network will be updated less times in the same training time. One must find the right balance between these in order to achieve maximum performance.

However, we find M to be even more impactful than N, as changing it directly influences the accuracy of p_t and a_t^* produced by the MCTS algorithm. It is crucial to generate these as accurately as possible, because reinforcing the network with imprecise policies is undesirable. If M is too small, the policy improvement becomes negligible, if too large, the episodes get drawn out and rolling one out takes too much time. In this section, we will focus on a technique we call *redundancy checking*, which adjusts UCT

in order to get the most out of the M decision nodes available.

7.2.1 Motivation

In many different simulations, including most of those aforementioned, traffic light control, robot navigation, with board games and TAT alike, an arbitrary state s_t does not determine the action history required to get from s_0 to s_t. As with chess for example, 1. e4 e5 2. d4 d5 leads to the same game state as 1. d4 d5 2. e4 e5; in TAT, moving the target location up and then right results in the exact same state distribution as moving it right first and then upwards, and even though these states or state distributions are equivalent, their action histories are not, and therefore, during MCTS, they will each be represented by their own set of decision nodes. As the MCTS is incapable of recognizing the similarity of their parent chance nodes, they will both contest the importance of being expanded during the UCT step of selection, and as the resources are distributed to them equally, the resulting approximation of the chance node subtrees will be less detailed than if only one of these chance nodes were focused for expansion.

7.2.2 Realization

For deterministic environments, where chance nodes always have exactly one child decision node, one can immediately recognize a duplicate state after generation (by storing all decision nodes in a hashset), and assigning some arbitrary negative value to that decision node, like -1000. As the value propagates upwards, the parent chance node will most likely never be picked in the UCT step again, which is a desirable result. However, in stochastic environments, since the agent does not possess knowledge about $\mathcal{T}$ and $\mathcal{R}$, it can never know whether two chance nodes result in the same state distribution for certain. This forces us to devise a custom metric, which determines how equivalent any two chance nodes are and whether any of them is worth expanding.

We claim that if two state distributions are equivalent, they must have an identical domain (For clarity, a chance node distribution domain can yield that '*executing chance node Z can result in states A, B or C*', as it is a set, in contrast to a chance node distribution, which yields information of type '*executing chance node Z can result in state A with probability 0.1, state B with probability 0.85, and state C with probability 0.05*', as it is a function/mapping).

The essential structure amplifying the original MCT is a new hashmap,

which maps chance node state distributions domains to a) the first such explored distribution domains (called *prime*), and b) all other such explored distribution domains. When any chance node is expanded and a decision node is added to its children, its underlying state distribution domain changes and so does the domain hashmap: if the old domain was *prime*, a new *prime* is appointed choosing randomly from the corresponding *non-prime* nodes, if the new domain doesn't exist in the hashmap, it becomes a new *prime*, otherwise it is added to the corresponding *non-prime* set, etc. The hashmap always keeps track of what state distribution domains exist in the MCT, and always has a clearly appointed *prime* chance node for each domain equivalence class.

With all chance nodes separated into equivalence classes with a representing *prime* node, we arrive at a new UCT formula:

$$UCT(c) = V_r(c) + P(c) \cdot (E\frac{\sqrt{T(d)}}{T(c)+1} - R(c)), \tag{7.1}$$

$V_r(c)$ being the *relative value* from equations 4.6 and 4.8, we added the term R called *redundancy* to the chance node's exploration value, where if $p =$ corresponding prime distribution, $q =$ chance node distribution,

$$sim = \frac{\sum_i (p_i q_i)}{||p|| \cdot ||q||} \tag{7.2}$$

$$confidence = \min_{k \in \{p_i\} \cup \{q_i\}} k \tag{7.3}$$

$$R(c) = \begin{cases} 0 & \text{if c is } \textit{prime} \\ sim + ((sim > 0.9) - sim) \cdot \frac{confidence}{1+|confidence|} & \text{otherwise,} \end{cases} \tag{7.4}$$

assuming the reader is familiar with the notion that, unless specified otherwise, boolean expressions like $a > b$ are equal to 1 if true and 0 otherwise. Through this expression, we claim that the distributions are equal if their cosine similarity is higher than 0.9 and different if it's lower, being more confident in the claim if we have more data supporting the similarity. For example, if the state frequencies of two chance nodes are [1, 2] and [10, 22], then although very similar, we are less confident that the distributions are equal compared to frequencies like [40, 80] and [100, 221], which are less similar. The higher the confidence, the strongly is the final result shifted towards 0 (non-redundant) if the similarity is below or equal to 0.9, and towards 1 (redundant) if the similarity is above 0.9.

The redundancy of chance nodes is propagated to their parents as well. When the redundancy of any node changes, then its parent's redundancy becomes $\min_c R(c)$ **for each child chance node c if the parent is a decision node**, since decision nodes are only as redundant as the least redundant chance node, and $\sum_d (R(d) \cdot T(d)) / \sum_d (T(d))$ **for each child decision node d if the parent is a chance node**, since selecting a child of a chance node is random and so its redundancy is the expected redundancy of its children. Keep in mind that the redundancy of a decision node can be changed only by this propagation process.

If a redundant node is selected even despite the heavy bias against this (by having high relative value and being scarcely explored), the *redundancy checking* is not applied for any of its children, in case a new distribution domain would be explored, which would result in an unreachable prime, locking out entire branches of the state tree.

The redundancy checking being the only addition to EXPI, we name this agent "Expecti-max Policy Iteration with Redundancy", or EXPIRE. By ignoring redundant action histories and further optimizing the state tree approximation, we expect this agent to outperform the previous ones.

8 Experiments

8.1 Network model

For all our experiments, we use a 20-layer *residual network*, or ResNet, built according to [2]. It is a direct extension of a convolutional network - those are great for multidimensional data processing - which utilizes *skip connections*, allowing for automatic depth selection.

The input to this network is always a rank 4 tensor of shape $[?, 13, 13, 8]$, the first dimension being the *batch dimension*, allowing us to input any number of states to the network and receive their corresponding outputs in one forward propagation thanks to broadcasting. Axes 1 and 2 represent the X and Y dimensions on the map respectively, while the third axis represents field states at $[X, Y]$. If there is a building at $[X, Y]$, then $[X, Y, type]$ is set to $\lfloor \frac{n}{16} \cdot max_hp \rfloor$, where $type = 0, 1, 2, 3,$ or 4 for building types main, house, turret, tesla, or wall respectively; n being the amount of sixteenths of health received from the environment. If there are infected at $[X, Y]$, their amount clipped to 127 is shown at $[X, Y, 5]$. For target location, $[tX, tY, 6]$ is set to 1, and $\forall x, y : [x, y, 7] = \lfloor gold \rfloor$.

For DQN, the network outputs a rank 2 tensor of Q-values with shape $[?, |\mathcal{A}|]$, and for EXPI/RE, it outputs a rank 2 tensor of policy probabilities with shape $[?, |\mathcal{A}|]$ and a rank 1 tensor of current state values with shape $[?]$.

For value outputs, we minimize MSE, and for probability outputs, we minimize *categorical cross-entropy*, which is equal to $-\sum_{x \in T} p(x) \cdot log\ q(x)$. We use this loss function instead of MSE, because it is applicable for measuring distances between probability distributions. When both losses are present in a model at the same time, they have the same weight, and we apply the *adaptive momentum* SGD technique for their reduction, with parameters $learning_rate = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and batches of size 32.

8.2 Agent parameters

8.2.1 DQN

During the DQN training procedure, we set the experience replay circular buffer size to 5000 and $\epsilon : f(episode) = 2 - 0.0005 \cdot episode$ with 10 epochs of

training for each candidate training step and the target network parameters being set to the candidate's each 100 episodes.

8.2.2 EXPI/RE

For our EXPI/RE experiments, we let $\epsilon = 0.25$ and $\gamma = 0.03$ like in the original AZ implementation at [32], while setting $N = 50$ and $M = 500$. We found these values to balance the training stability and time well.

8.3 Training procedure

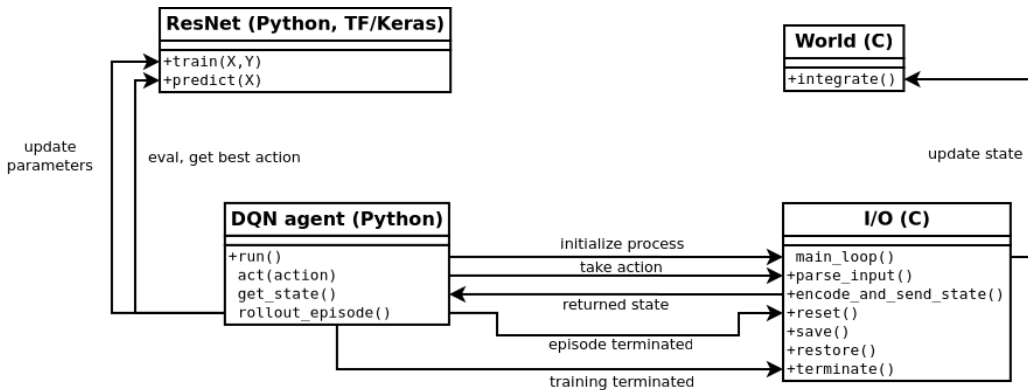


Figure 8.1: DQN - TAT process interaction diagram

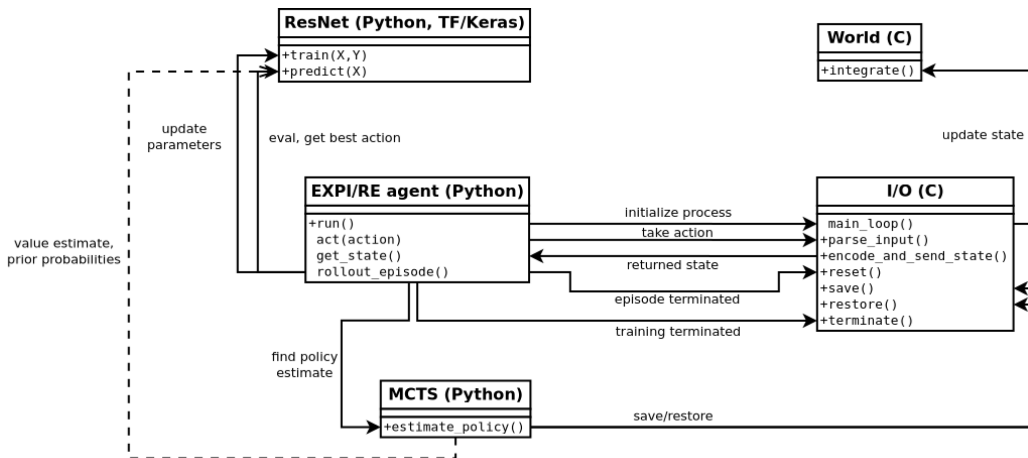


Figure 8.2: EXPI/RE - TAT process interaction diagram

The agent system, written in Python, creates a subprocess of TAT and interacts with it as shown in Figures 8.1, 8.2.

Considering the evaluation metric for agents, we claim that the best agent is one that achieves the highest evaluation reward after a set amount of training time. Following this approach, we let each agent train for ~ 96 hours total (as each agent improves only each N seconds, the real training time is always smaller, since the training loop is terminated mid-way). After finishing the training period, evaluating the agents as they are updated and logging their results, the agents' performance is visualized by Figure 8.3. The Y values show the average reward received by the respective agents during 50 evaluation episodes, where the policy used is defined by $\mathcal{Q}_\theta(s, .)$ and $\mathcal{P}_\theta(s)$ for DQN and EXPI/RE respectively. Unlike during the experiments in [32], where AZ was evaluated using the same policy used for training (MCTS guided by $\mathcal{P}_\theta$ and $\mathcal{V}_\theta$), our goal was to create agents whose policies would be represented purely by neural networks.

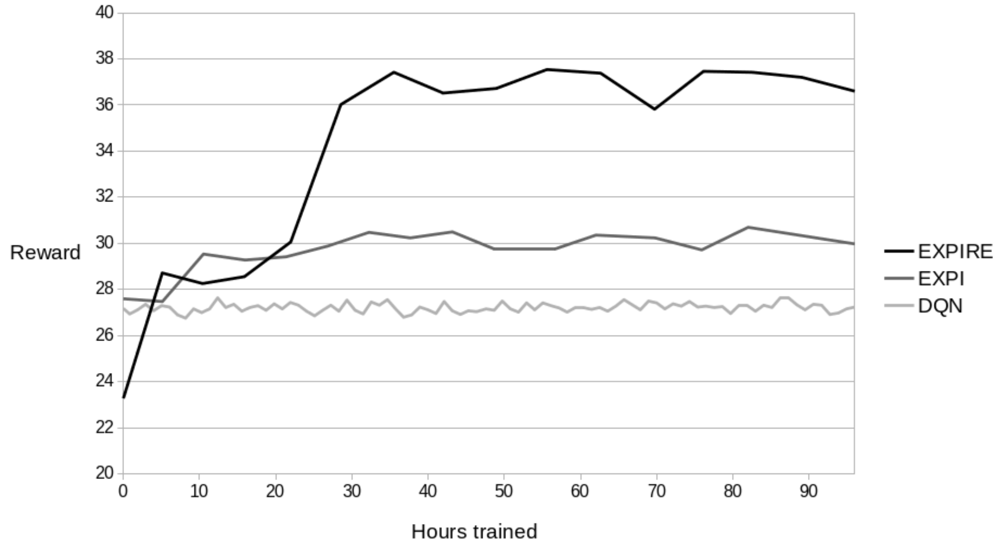


Figure 8.3: Agent performance

Agent	Average time per episode	Average time per update	Final reward
DQN	37s	1h 1min 35s	27
EXPI	8min 53s	7h 24min 10s	30
EXPIRE	8min 58s	7h 28min 29s	37

Table 8.1: Average training times and final rewards of each agent

9 Conclusion

After the training was completed, we've noticed that the standard DQN struggled to improve the random policy by any significant value. Regression analysis shows that the DQN agent actually does improve linearly, roughly by 4 reward points per 3000 hours of training, which is much too slow to be given any further notice. As this result was backed by [20], we see that as clever an agent the DQN might be, it fails to solve the more intricate environments, such as TAT.

The EXPI agent, which is itself a variant of AlphaGo Zero designed to solve non-adversary stochastic environments, did find an improvement over DQN, but has been stuck on a local minimum since hour 30. The likely cause is the choice of a too small M compared to [32], who used 1600 instead of our 500. This limitation caused a lack of think-ahead for EXPI, which resulted in its inability to find a better policy.

However, the effects of this MCTS limitation have been significantly negated by EXPIRE, which used the *redundancy checking* technique we have introduced to optimize its state tree search algorithm, surpassing EXPI's final evaluation reward advantage over DQN three times, with its final reward falling slightly short of 37. Since EXPIRE is *de-facto* an extension of AZ, our results hint at the fact that the original mini-max AZ too might benefit from a similar redundancy checking procedure, which would be much easier to implement (see Section 7.2.2).

Our work can be followed up on by *parallelizing* the MCTS procedure in EXPI/RE agents, which is non-trivial considering the complexity of our redundancy checking implementation, or by extending these agents to handle *infinite* episodes with state loops and a discount factor, detecting loops using similar technique to the redundancy check, and calculating the expected loop value. Apart from balancing the entity parameters in TAT, repeating the training procedure several times to test the consistency of our results would be very useful, which is something we have unfortunately been forced to omit as a single pass of the training procedure takes 12 days, but it is something we will look into in the future.

10 Common abbreviations

Abbreviation	Meaning
AI	artificial intelligence
RL	reinforcement learning
NN	neural network
SGD	stochastic gradient descent
TAT	They Are Trillions
LB	linearity-breaking
PRNG	pseudo-random number generator
DQN	Deep Q-learning
AZ	AlphaGo Zero
UCT	Upper Confidence bound applied to Trees
MCTS	Monte Carlo Tree Search
EXPI	Expecti-max Policy Iteration
EXPIRE	Expecti-max Policy Iteration with REdundancy

Bibliography

[1] ADL. An introduction to q-learning: reinforcement learning, 2019. URL https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/. Q learning 3.

[2] admin. Introduction to resnet in tensorflow 2, -. URL https://adventuresinmachinelearning.com/introduction-resnet-tensorflow-2/. ResNet in TF/Keras.

[3] Anas Al-Masri. What are overfitting and underfitting in machine learning?, 2019. URL https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690. Overfitting explained.

[4] Mohammad Ashraf. Reinforcement learning demystified: Markov decision processes (part 1), 2018. URL https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690. MDP basics.

[5] Phillipa Avery, Julian Togelius, Elvis Alistar, and Robert Leeuwen. Computational intelligence and tower defence games. pages 1084 – 1091, 07 2011. doi: 10.1109/CEC.2011.5949738.

[6] Marc G. Bellemare, Will Dabney, and Rémi Munos. A Distributional Perspective on Reinforcement Learning. *arXiv e-prints*, art. arXiv:1707.06887, July 2017.

[7] David Blackman and Sebastiano Vigna. Scrambled Linear Pseudorandom Number Generators. *arXiv e-prints*, art. arXiv:1805.01407, May 2018.

[8] Branko Blagojevic. Reinforcement learning with sparse rewards, 2018. URL https://medium.com/ml-everything/reinforcement-learning-with-sparse-rewards-8f15b71d18b. Sparse reward.

[9] Amar Budhiraja. Dropout in (deep) machine learning, 2016. URL https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-mach Dropout explained.

[10] Aneek Das. The very basics of reinforcement learning, 2017. URL https://becominghuman.ai/

the-very-basics-of-reinforcement-learning-154f28a79071. RL basics.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv e-prints*, art. arXiv:1810.04805, October 2018.

[12] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified Language Model Pre-training for Natural Language Understanding and Generation. *arXiv e-prints*, art. arXiv:1905.03197, May 2019.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[14] Ray Heberer. Why going from implementing q-learning to deep q-learning can be difficult, 2019. URL https://towardsdatascience.com/why-going-from-implementing-q-learning-to-deep-q-learning-can-be-difficult-36e7ea Q learning.

[15] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, art. arXiv:1502.03167, February 2015.

[16] Pierre Jaumier. Backpropagation in a convolutional layer, 2019. URL https://towardsdatascience.com/backpropagation-in-a-convolutional-layer-24c8d64d8509. Conv. backprop.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[18] Conor Lazarou. Reward hacking in evolutionary algorithms, 2019. URL https://towardsdatascience.com/reward-hacking-in-evolutionary-algorithms-c5bbbf42994b. Reward hacking.

[19] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. *NIPS 2017*, pages 6231–6239, 2017. URL http://papers.nips.cc/paper/7203-the-expressive-power-of-neural-networks-a-view-from-the-width.pdf.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, art. arXiv:1312.5602, December 2013.

[21] Thanh Thi Nguyen, Cuong M. Nguyen, Dung Tien Nguyen, Duc Thanh Nguyen, and Saeid Nahavandi. Deep Learning for Deepfakes Creation and Detection. *arXiv e-prints*, art. arXiv:1909.11573, September 2019.

[22] Michael Nielsen. Neural networks and deep learning, 2019. URL `http://neuralnetworksanddeeplearning.com/chap1.html`. Cost function.

[23] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press; eBook(NeuralNetworksAndDeepLearning.com), San Francisco, CA, USA, 2015. ISBN N/A.

[24] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time Limits in Reinforcement Learning. *arXiv e-prints*, art. arXiv:1712.00378, December 2017.

[25] Aditya Prasad. Lessons from alphazero (part 3): Parameter tweaking, 2018. URL `https://medium.com/oracledevs/lessons-from-alphazero-part-3-parameter-tweaking-4dceb78ed1e5`. AZ temperature.

[26] Schartz Rehan. The shape of tensor, 2019. URL `https://mc.ai/the-shape-of-tensor/`. Tensor shapes.

[27] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015. URL `http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-network` `pdf`.

[28] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016. URL `https://ruder.io/optimizing-gradient-descent/`. Gradient descent variants.

[29] Ihab S. Mohamed. *Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques*. PhD thesis, Université Côte d'Azur, 09 2017.

[30] Irhum Shafkat. Intuitively understanding convolutions for deep learning, 2018. URL https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1. Convolution explained.

[31] Chathurangi Shyalika. A beginners guide to q-learning, 2019. URL https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c. Q learning 2.

[32] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv e-prints*, art. arXiv:1712.01815, December 2017.

[33] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

[34] Alexander Trott, Stephan Zheng, Caiming Xiong, and Richard Socher. Keeping Your Distance: Solving Sparse Reward Tasks Using Self-Balancing Shaped Rewards. *arXiv e-prints*, art. arXiv:1911.01417, November 2019.

[35] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A Generative Model for Raw Audio. *arXiv e-prints*, art. arXiv:1609.03499, September 2016.

[36] Joel Veness, Kee Siong Ng, Marcus Hutter, William Uther, and David Silver. A Monte Carlo AIXI Approximation. *arXiv e-prints*, art. arXiv:0909.0801, September 2009.

[37] John Tunnicliffe Wesley Bourne, Robin Gallimard. Environments, 2006. URL https://www.doc.ic.ac.uk/project/examples/2005/163/g0516302/environments/environments.html. Environment properties.

[38] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876*, 7 (8), 2015.

[39] Renchun You, Yuan Yao, and Jia Shi. Tensor-based ultrasonic signal processing for defect detection in fiber reinforced polymer (frp) structures. *2017 6th International Symposium on Advanced Control of Industrial Processes (AdCONIP)*, pages 312–317, 2017.

[40] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv e-prints*, art. arXiv:1611.03530, November 2016.