FernUniversität in Hagen
Faculty of Mathematics and Computer Science

AIG
Artificial Intelligence Group

Reinforcement Learning for Games on Hexagonal Grids

Bachelor's Thesis

in partial fulfillment of the requirements for
the degree of Bachelor of Science (B.Sc.)
in Informatik

submitted by
Miguel Álvarez Caro

First examiner:	Prof. Dr. Matthias Thimm Artificial Intelligence Group
Advisor:	Dr. Kai Sauerwald Artificial Intelligence Group

Statement

I declare that I have written the bachelor´s thesis independently and without unauthorized use of third parties. I have only used the indicated resources and I have clearly marked the passages taken verbatim or in the sense of these resources as such. The assurance of independent work also applies to any drawings, sketches or graphical representations. The work has not previously been submitted in the same or similar form to the same or another examination authority and has not been published. By submitting the electronic version of the final version of the bachelor´s thesis, I acknowledge that it will be checked by a plagiarism detection service to check for plagiarism and that it will be stored exclusively for examination purposes.

I explicitly agree to have this thesis published on the webpage of the artificial intelligence group and endorse its public availability.

Software created for this work has been made available as open source; a corresponding link to the sources is included in this work. The same applies to any research data.

Bonn, 14 June 2026

.....

(Place, Date) (Signature)

Zusammenfassung

In dieser Bachelorarbeit werden Kriegssimulationsspiele auf hexagonalen Gittern als vielversprechende Domäne für die Entwicklung und Evaluierung von *Reinforcement-Learning*-Algorithmen untersucht. Dabei werden grundlegende neuronale Netzwerkarchitekturen sowie Trainingsverfahren für diese Klasse von Spielen betrachtet. Es wird ein neuartiges Faltungsneuronales Netzwerk (*Convolutional Neural Network*, CNN) für hexagonale Gitter vorgestellt, das es ermöglicht, die in solchen Spielen inhärente Rotationsinvarianz auszunutzen. Darüber hinaus wird ein *Reinforcement-Learning*-Framework zur Optimierung neuronaler Netzwerke für Spiele auf hexagonalen Gittern entwickelt, das auf modernen Methoden des *Deep Reinforcement Learning* aufbaut. Abschließend wird das vorgeschlagene Framework auf ein einfaches Spiel auf einem hexagonalen Gitter angewendet, und die erzielte Leistungsfähigkeit des Frameworks wird evaluiert und analysiert.

Abstract

In this bachelor's thesis, war games on hexagonal grids are investigated as a promising domain for the development and evaluation of reinforcement learning algorithms. Fundamental neural network architectures and training methods for this class of games are explored. A novel convolutional neural network architecture for hexagonal grids is introduced, enabling the exploitation of the rotational invariance inherent in such games. Furthermore, a reinforcement learning framework for training neural networks on games played on hexagonal grids is developed, building upon state-of-the-art methods in deep reinforcement learning. Finally, the proposed framework is applied to a basic hexagonal-grid game, and the resulting performance is evaluated and analyzed.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Matthias Thimm, for granting me the freedom to explore this topic, for his continuous support, and for helping me define a research topic that remained within the scope of a bachelor's thesis.

I am also grateful to the FernUniversität in Hagen for the high-quality education it has provided throughout my studies. The excellent modules and the flexibility to define my own academic focus enabled me to acquire the fundamental knowledge required for this thesis.

Finally, I would like to thank my wife, Sokcheng, for encouraging me in my research and for her patience during the demanding final years of my studies; my daughter, Penélope, for the joy and happiness she brings to my life; and my mother-in-law, Marina, for her support during the final months of this thesis.

Contents

List of Figures

1 Introduction

Since the early application of neural networks to reinforcement learning (for example, Tesauro's *TD-Gammon* [Tes94]), games have, on the one hand, presented important challenges that motivated the development of new methods in reinforcement learning. On the other hand, games have also proven to be a good experimental area to test new algorithms. For example, the impact of the early success of neural networks on Atari games [MKS+13] did not merely show that these games could be mastered; it also established Atari as a benchmark domain for major later advances in reinforcement learning, including the asynchronous advantage actor-critic (A3C) algorithm [MPBM+16], the Importance Weighted Actor-Learner Architecture (IMPALA) architecture [ESM+18], and hierarchical reinforcement learning with FeUdal Networks [VOS+17].

Despite substantial progress in recent years, important challenges in reinforcement learning for games remain unresolved. DeepMind achieved major milestones in Go [SHM+16, SHS+18], chess [SHS+18], and the video game StarCraft II [VBC+19]. In their first applications [SHM+16, SHS+18], neural networks were used to guide the Monte Carlo tree search to find the best actions. However, this approach was limited to relatively simple environments for which a model of the state transitions could be built. A method applicable to a broader range of environments was the actor-critic approach used for StarCraft II [VBC+19], where the policy is represented directly by a neural network. Despite this success, that AI (Artificial Intelligence) still showed important limitations in initialization and exploration: initialization relied on imitation learning (with rewards based on divergence from expert human behavior), and the self-play phase was additionally guided by pseudo-rewards that encouraged human-derived strategies, so that exploration proceeded in efficient directions. The AI of DeepMind for StarCraft II was not able to find good exploration directions by itself, and it needed to be guided by human knowledge. Another limitation was the enormous training cost: hardware equivalent to 4,200 CPU (Central Processing Unit) cores and 128 TPU (Tensor Processing Unit) cores was used for more than 44 days to train the best neural network.

The high training cost is due in part to the large state representations of the games that are used by reinforcement learning algorithms for games such as StarCraft and Atari. StarCraft is much more complex than Atari games and is therefore useful for studying harder exploration problems, but its state representation is also significantly more complex, consisting of a dynamic set of features [VEB+17] rather than a single tensor as in Atari games. The size of the state representation, together with the frequency at which the neural network must be evaluated, leads to high training costs. These costs are a major limitation when testing and developing algorithms that are not intended to master real-time tasks.

This bachelor's thesis proposes another class of games as a more suitable domain for developing reinforcement learning algorithms: war games on hexagonal grids, a game type that has only recently been explored [PSD25]. Compared with real-

time games such as StarCraft, these games admit a more compact state representation, because they are discrete by design: space is divided into hexagonal tiles and time advances in turns. This compact representation enables more efficient neural network training. At the same time, these games still support a rich spectrum of strategies, making exploration challenging for AI systems. This combination of representational efficiency and strategic complexity makes them an excellent domain for developing and benchmarking reinforcement learning algorithms.

War games on hexagonal grids have a long history, ranging from early board games to modern video games, with the *Civilization* franchise as one prominent example. Because research on AI for this class of games is still limited, no particular game has become an established benchmarking standard (as happened in other genres with Atari games and StarCraft). For this reason, a new video game will be implemented with the explicit goal of developing and evaluating AI methods.

The main contribution of this bachelor's thesis is threefold: the development of a convolutional network framework for hexagonal grids, the design of a reinforcement learning method for training neural networks on games played on hexagonal grids, and an analysis of its application to a basic hexagonal grid game.

2 Scientific Background

In order to develop a machine learning method for training a neural network for war games on hexagonal grids, several scientific concepts and methods must be applied. These are introduced in the following sections.

2.1 Neural Networks

Neural networks are a class of machine learning models composed of weighted combinations of simple functions arranged in a layered structure, forming a complex composite function. This composition can be represented as a directed graph, where nodes correspond to inputs and intermediate outputs of the simple functions, while edges represent functional dependencies between them. The nodes are organized into layers such that the output of one layer serves as the input to the next. Typically, each node in a given layer contributes to multiple nodes in the subsequent layer. The influence of each simple function on the next layer is determined by learnable parameters, known as *weights*. A neural network is trained by adjusting these weights such that the network approximates a complex function able to solve a problem. In order to guide this adjustment of the weights, a loss function is used to quantify the discrepancy between the network output and the desired output, referred to as the *approximation error*. One of the most commonly used loss functions for regression problems–where continuous numerical values are predicted–is the mean squared error. It computes the average of the squared differences between predicted and target values, thereby penalizing larger errors more strongly.

Since the introduction of the first neural network with the *perceptron* [Ros58] (a neural network based on a single linear function), neural network units have commonly been based on linear functions that combine a vector of weights $\overline{W}$ with a vector of inputs $\overline{X}$ to produce a single output:

$$y = \overline{W} \cdot \overline{X}^T = \sum_{i=1}^{d} w_i x_i$$

To increase the expressive power of a neural network, its depth can be increased by adding more layers, each consisting of multiple linear functions. However, the composition of linear functions remains linear; therefore, a network composed solely of linear transformations is itself equivalent to a single linear function. To enable neural networks to model non-linear problems, non-linear functions must be introduced between layers. These functions are referred to as *activation functions*. Two of the most commonly used activation functions are the sigmoid function, which produces an S-shaped curve with output values in the range between 0 and 1:

$$\Phi(v) = \frac{1}{1 + e^{-v}}$$

and the rectified linear unit (ReLU), which outputs the input directly if it is positive and zero otherwise:

$$\Phi(v) = \max(0, v)$$

The training process of a neural network consists of adjusting the weights in several cycles, called *epochs*, in order to reduce the error of the output for a given input. At the end of the 1950s and in the 1960s, when the perceptron was introduced, this training process was not understood as a mathematical optimization problem. Instead, it was performed using a heuristic that updated the weights only for misclassified data. Due to the lack of a formal mathematical formulation of the training process, training multilayer neural networks was not feasible. The influential publication by Minsky and Papert [MP69] in 1969, in which the limitations of single-layer networks were highlighted and the problem of training multilayer neural networks was presented as an unsolved problem, initiated a period of general skepticism regarding research on neural networks.

The problem of training multilayer neural networks was formulated as an optimization problem only with the introduction of the backpropagation algorithm by Rumelhart, Hinton, and Williams [RHW86] in 1986: the gradient of the loss function with respect to the network weights is computed and used to determine the direction in which the weights are updated. This strategy is known as *gradient descent*. The derivative of the loss function with respect to the network weights is based on a deep composition of functions, since the loss function depends on the function represented by the neural network. To compute this gradient, the chain rule of differentiation is applied repeatedly along all paths in the computational graph between the output and each edge representing a weighted function composition. If a

node in the graph represents the function $f(g_1(x), \ldots, g_n(x))$ with n incoming edges from the previous layer functions $g_1(x), \ldots, g_n(x)$, then the derivative of this function with respect to a particular weight x can be computed using the multivariate chain rule:

$$\frac{\partial f(g_1(x), \ldots, g_n(x))}{\partial x} = \sum_{i=1}^{n} \frac{\partial f(g_1(x), \ldots, g_n(x))}{\partial g_i(x)} \frac{\partial g_i(x)}{\partial x}$$

The chain rule decomposes the derivative of a complex function into derivatives of its simpler subfunctions. However, a naive computation of the gradient of the loss function in a deep neural network is computationally inefficient, as the complexity of the resulting derivative grows exponentially with the depth of the network [Agg23, p. 32–33]. This is due to the fact that the number of paths in the computational graph is exponential in the network depth.

The main contribution of the *backpropagation algorithm* is the efficient application of the chain rule of differentiation. Backpropagation exploits the fact that the same subfunctions are reused across different paths in the computational graph, and that the derivative of a subfunction with respect to its immediate inputs can be stored and reused when computing derivatives along other paths, according to the chain rule. The algorithm traverses the computational graph of the neural network, starting from the output of the loss function, in order to compute and store derivatives with respect to progressively deeper nodes until the input nodes are reached. In this way, backpropagation applies a dynamic programming strategy to the computation of gradients: it stores intermediate results for the derivatives of the loss function with respect to each node in the graph and reuses them when computing derivatives for deeper nodes. Once the derivatives of the loss function with respect to all nodes are available, the computation of the derivatives with respect to the network weights is straightforward. The complexity of the backpropagation algorithm is linear in the number of edges of the computational graph representing the neural network.

2.2 Optimization Methods for the Training of Deep Neural Networks

Despite the fact that the backpropagation algorithm made the training of multilayer neural networks feasible, its application to several problems remained limited by training instability, computational cost, and the requirement for large amounts of data. The latter two issues have been mitigated over the past decades by advances in computational power –especially through the use of Graphics Processing Units (GPUs)–as well as by the increasing availability of data. However, the algorithmic challenges related to unstable training have multiple causes. In the following, we focus on the specific issues that required special attention in the implementation of the machine learning method for games on hexagonal grids of the present paper.

Performing a neural network update using a single data point (i.e., *stochastic gradient descent*) can make training highly unstable, as information learned from earlier

data points may be overwritten and thus lost. At the other extreme, performing updates based on the entire training dataset –by aggregating the error over all data points–is theoretically the most stable approach to training a neural network. However, it is in most cases computationally infeasible due to high memory and computational requirements. Between these two extremes, the use of a representative subset of data points that does not include the full training dataset is a common strategy to improve both stability and computational efficiency. To process such subsets efficiently, data points are not fed individually into the neural network; instead, they are grouped into a multi-dimensional array, forming a structure known as a *mini-batch*. In particular, the use of GPUs for backpropagation makes batch-based training significantly more efficient, as the parallel computation capabilities of GPUs are best utilized when operating on large tensor structures.

The first-order gradient provides information about the direction and magnitude of the local slope of the loss function, but it does not capture information about its curvature. This limitation of first-order methods can lead to slow convergence when both the gradient and curvature are small, or to oscillatory and unstable updates when the gradient varies sharply across dimensions. To address this issue, the *RMSProp* algorithm [Hin12] combines standard gradient descent with an adaptive approximation of second-order derivatives by maintaining an exponentially decaying average of past gradients and using it to rescale the current update direction. The application of RMSProp to the image classification model *AlexNet* in 2012 for the *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* represented a major breakthrough in the training of deep neural networks, as it significantly improved the stability and efficiency of training.

The gradient of a function converges to a minimum (or to a saddle point), but it does not necessarily converge to the global minimum of the function. The difference between these outcomes can be substantial, and the training of a neural network may therefore become trapped in a suboptimal region of the loss landscape. In order to mitigate the effects of local minima and improve convergence behavior, the use of momentum–analogous to the concept of inertia in physics–has been widely adopted in recent decades. Following the success of the RMSProp algorithm, the *Adam* algorithm [KB14] combines momentum with a similar adaptive second-moment estimation of gradients as used in RMSProp. It has become one of the most widely used methods for optimizing gradient-based updates in neural network training.

In this paper, strategies that improve standard gradient descent, such as the RMSProp and Adam algorithms, are referred to as *optimizers*. The application of these optimizers requires the use of batch-based gradient estimation, as they rely on sufficiently accurate gradient estimates [Agg23, p. 135]. Otherwise, if the gradient is computed from only a very small number of data points, the use of momentum and the approximation of second-order information may further increase the instability of stochastic gradient descent.

The final challenge considered in this section concerns the magnitude of gradi-

ents in deep neural networks. Due to the cumulative effect of activation functions across many layers, the gradients in the earlier layers of a deep network may become extremely small–if the derivative of the activation function is smaller than 1 (the *vanishing gradient problem*)– or extremely large–if the derivative is greater than 1 (the *exploding gradient problem*). One approach to mitigate these issues is the application of standardization after certain layers, in order to keep activation values within a limited range around 0, where many activation functions operate most effectively. However, simple standardization may lead to information loss, since centering the data around 0 can reduce information encoded in the magnitude and sign of the activations. A more advanced technique to address this issue is *batch normalization* [IS15], in which layer activations are normalized using the mean and standard deviation computed from the current mini-batch.

Another technique that helps mitigate the vanishing and exploding gradient problems is the use of residual connections (or skip connections) [HZRS16]. In this approach, a layer is connected not only to the subsequent layer, but also directly to deeper layers in the network. The intermediate layers learn additional transformations that are added to the original information from the earlier layer. As a result, the original information can be preserved throughout the network and the gradient of the loss function can propagate more directly through the skip connections to deeper layers. That reduces the impact of vanishing or exploding gradients in the intermediate layers. [1]

2.3 Convolutional Neural Networks

Most of the information in a game played on a hexagonal grid can be represented spatially, as the units and the terrain are positioned on the game board. To process this type of information efficiently, the neural networks proposed for games on hexagonal grids must be capable of handling spatial structures in a manner similar to that successfully employed in visual recognition tasks. The neural network architecture that has achieved the greatest success in visual recognition is the *convolutional neural network*.

The data representing an image is commonly structured as a four-dimensional tensor. Similar to a physical image, two of these dimensions correspond to the spatial height and width. In addition, the different color components of the image are represented in separate channels along another dimension. These three dimensions typically correspond to the last three dimensions of the tensor, although their order may vary depending on the neural network framework used. The first dimension corresponds to the size of the batch of data points processed simultaneously by the neural network.

Instead of applying a fully connected neural network blindly to any type of problem, regardless of its structure, a common strategy to improve training efficiency

[1] The last two introductory sections on neural networks were inspired by Chapters 1–3 of the book by Aggarwal [Agg23].

is to incorporate domain-specific information into the architecture of the network. The convolutional neural network architecture exploits two key properties of visual data:

- translation invariance: the property that the same object can be recognized at different locations within an image;
- local connectivity: the property that pixels that are close to each other are more likely to be related than pixels that are far apart.

Unlike fully connected networks, convolutional networks limit the connections of each neuron to a small local region of the previous layer, rather than connecting to all neurons in that layer. Given this restricted input, the convolutional operation performs a linear transformation of the input, similar in nature to a standard fully connected layer but applied locally. The region of the input processed by a neuron in a convolutional layer is called the *receptive field*, and the set of weights applied to this region is referred to as the *kernel*. The receptive field of a given neuron is typically composed of the surrounding spatial neighborhood in the input. In this way, convolutional networks exploit the local connectivity of visual data by processing inputs based on local regions. The kernel is commonly chosen as a small square filter, typically of size 3×3 or 5×5.

In order to avoid that pixels at the borders of the image are processed less frequently than those in the center –since they belong to fewer receptive fields– a common strategy is to expand the image with a border filled with a constant value, typically zeros. This technique is called *padding*. The width of the added border corresponds to the padding size and is usually chosen in relation to the size of the kernel.

The same kernel is applied uniformly across the entire image, with shared weights used for all spatial locations. In this way, the convolutional operation exploits the translation invariance property of visual data. The kernel can be applied at every spatial position of the input, in which case the spatial dimensions of the output remain comparable to those of the input (depending on padding). Alternatively, the kernel can be applied with a larger step size, skipping certain spatial positions of the input and thereby reducing the spatial dimensions of the output. This step size is referred to as the *stride*.

2.4 Reinforcement Learning

In Subsection 2.1, it was explained how neural networks are trained based on the quantification of the error between the network output and the desired output for a given input [2]. There are three main paradigms of machine learning, which are not limited to neural networks: supervised, unsupervised, and reinforcement learning. Each of these determines how the desired output is defined for each data point.

[2]The exposition of the following four sections is based on the canonical book by Sutton and Barto [SB18].

If the desired output is explicitly provided for each data point, the data is *labeled* and the learning paradigm is called *supervised learning*. Tasks such as classification (predicting the class to which an input belongs) and regression (predicting a numerical value) are examples of supervised learning. If the desired output is derived from the structure of the input data itself, the learning paradigm is called *unsupervised learning*. One of the most representative methods of unsupervised learning is clustering, although neural networks are not frequently applied to this type of task. A common unsupervised learning method that often involves neural networks is the autoencoder: an autoencoder learns to compress data and reconstruct it from the compressed representation. The loss function measures the reconstruction error, i.e., the difference between the original and reconstructed data. Under the third machine learning paradigm, *reinforcement learning*, an agent interacts with an environment from which it receives reward signals. The objective of the learning process is to learn a policy, i.e., the strategy by which the agent selects actions, that maximizes the expected cumulative reward. A key challenge of the reinforcement learning paradigm is that the reward signal is delayed in time. The quantification of the error–i.e., how much a given action contributes to future rewards–becomes a non-trivial problem that can make the learning process unstable, since the neural network cannot be trained directly using true target values for actions: a highly beneficial action may nevertheless lead to a poor outcome if it is followed by poorly chosen actions.

The interaction between an agent and an environment can be represented as a sequence consisting of:

- states of the environment,
- actions taken by the agent based on its (complete or incomplete) knowledge of the state,
- immediate rewards received by the agent:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

A complete sequence of states, actions, and rewards, from the initial state until a terminal condition is reached, is known as an *episode*. The reinforcement learning paradigm addresses the problem of selecting optimal actions under the assumption that the environment satisfies the *Markov property*: the distribution of the next states and rewards is entirely determined by the current state and action. If this property holds, the problem of choosing optimal actions can be formulated as a *Markov decision process*, a model defined by the following components: the set of possible states, the set of possible actions, the transition probabilities, and the reward function. In a Markov decision process, the transition probabilities and the reward function depend only on the previous state and action. More complex environments, for example, interactions between two or more agents that adapt to each other, go beyond the Markov decision process framework and are therefore not directly covered by standard reinforcement learning formulations.

2.5 Value-based Methods

A successful strategy for solving a reinforcement learning problem is to learn a value function that approximates the expected cumulative reward of a particular state or action. The value function takes a state as input in order to approximate its value. It is defined with respect to a particular policy, which determines the subsequent actions and thus allows it to approximate the expected cumulative reward. The training process approximates the function represented by a neural network to the true value function. Based on the value function, a new improved policy can be derived from the approximated value function, for example, a *greedy policy*, by selecting the action with the highest value in order to exploit the learned knowledge to obtain the maximal reward. To promote exploration and enable the discovery of optimal strategies (and not only exploitation of already discovered ones), a random action can also be taken by the derived policy with a small probability. Policies defined in this way are known as *ϵ-greedy policies*. The iterative process of improving the estimated value function and deriving a new policy is called *generalized policy iteration*.

In the context of neural networks, the gradient of the value function with respect to its weights provides the direction in which the value function should be updated in order to increase the estimated value for a given input. To determine whether the value should be increased or decreased, and by how much, the gradient is multiplied by the error between the estimated value and the return. The return is a function of the sequence of rewards from a given time onward; in the simplest case, it is defined as the cumulative sum of future rewards:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T$$

where R_n denotes the reward at time n and T is the final time step of the episode. *Monte Carlo predictions* wait until the episode has finished to compute the return for each state that occurred during the episode:

$$w_{t+1} = w_t + \alpha \cdot [G_t - \hat{v}(s_t)] \cdot \nabla_w \hat{v}(s_t)$$

where w are the weights of the estimated value function $\hat{v}$ and α is the *learning rate*, a constant that determines the step size of the updates. The learning process is slow, as the agent must wait until the end of the episode to learn from it. In addition, individual actions or states cannot be evaluated independently of the episode in which they occur (for example, an action may be beneficial even if subsequent suboptimal actions lead to a defeat).

To improve the convergence of the learning process, *temporal difference* (TD) methods allow learning from incomplete episodes. In order to approximate the value of a particular state at time t without waiting for the end of the episode, the estimated value function can be applied to the state n steps ahead in an incomplete episode that follows a given policy. In this way, the return for each time step t is constructed as follows:

$$G_t = R_{t+1} + \cdots + R_{t+n} + \hat{v}(s_{t+n})$$

The return G_t provides a better approximation of the value of the state at time t than directly using the estimated value function $\hat{v}(s_t)$. The difference between the two terms yields the temporal-difference error, which is used to update the neural network:

$$w_{t+1} = w_t + \alpha \cdot [R_{t+1} + \cdots + R_{t+n} + \hat{v}(s_{t+n}) - \hat{v}(s_t)] \cdot \nabla_w \hat{v}(s_t)$$

In this case, these methods are referred to as *n-step bootstrapping* methods.

Nevertheless, there are two important limitations of value-based methods:

- In order to derive a policy from the estimated value function, all possible actions of the visited states must be evaluated. Only reinforcement learning problems with a small number of actions are suitable for this strategy, as otherwise the computational cost becomes too high.
- Another problem of value-based methods is that the learning process can be unstable, as the policy may change abruptly from one action to another even when the value estimates change only slightly.

2.6 Policy-based Methods

In contrast to value-based methods, where the neural network approximates the value function, in policy-based methods, the neural network approximates the policy itself. In the case of discrete action spaces, where the number of actions is finite, the neural network outputs, for a given state, the probabilities of selecting each possible action. To ensure that all output probabilities sum to 1, the neural network first computes numeric values for each of the K possible actions, known as *logits*, which are then passed through a softmax function:

$$\pi(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$$

These methods can also handle continuous action spaces by outputting the parameters of a probability distribution.

Policy-based methods have several advantages over value-based methods, as they do not require the evaluation of every possible action separately. They also exhibit more stable convergence properties, since the policy changes smoothly at each learning step. Another advantage is that policy-based methods can represent stochastic policies, i.e., policies that select actions with probabilities between 0 and 1, in contrast to deterministic policies. The optimal policy for certain problems may require stochasticity, for example, in poker, both never bluffing and always bluffing may be suboptimal strategies. Because policies in policy-based methods are stochastic, they explore the state space in a natural way until a high-reward strategy is found, and may converge to the optimal stochastic policy, or even to a deterministic policy if it is optimal.

To train a neural network that represents a policy, the actions generated by that policy must be evaluated in order to decide which to reinforce and which to discourage. The foundation for evaluating an action lies in the expected value of the game resulting from taking that action. The performance of a policy θ, denoted $J(\theta)$, can be defined as the expected sum of action values for the initial states s_0 of all possible episodes:

$$J(\theta) = \mathbb{E}\left[\sum_a \pi_\theta(a \mid S_0) \cdot q_\pi(S_0, a)\right]$$

where π defines the policy by assigning a probability to each action given a state, and q_π denotes the value of an action in a given state under policy π with parameters θ. In order to improve the policy, the performance $J(\theta)$ must be maximized via gradient ascent with respect to the network weights. However, computing this gradient is non-trivial, since changes in the policy affect both the selected actions and the distribution of states in which those actions are taken. The *policy gradient theorem* [3] simplifies this computation by expressing the gradient in terms of the policy and the value function:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi\left[q_\pi(S, A) \cdot \frac{\nabla_\theta \pi_\theta(A \mid S)}{\pi_\theta(A \mid S)}\right] = \mathbb{E}_\pi\left[q_\pi(S, A) \cdot \nabla_\theta \ln \pi_\theta(A \mid S)\right]$$

where S and A are random variables representing states visited and actions taken under policy π. The structure of approximating algorithms that improve the policy via gradient ascent can be directly derived from the policy gradient theorem by replacing the expectation with samples generated by following the policy. For each sampled action, the neural network weights are updated via gradient ascent on the log-probability of the action, multiplied by the success of the action given by the value function.

Similarly to value-based methods, there are two different strategies to estimate the success of an action. Analogous to Monte Carlo methods, one approach consists of using the sum of future rewards over the whole episode as the return G_t, in order to update the policy based on the outcome of a particular action:

$$\theta_{t+1} = \theta_t + \alpha \cdot G_t \cdot \nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

This algorithm is known as *REINFORCE*. Although the policy gradient theorem shows that the REINFORCE algorithm improves the policy in expectation, the variance of the updates is high, since the probability of every taken action is always increased (even if the action was suboptimal), proportionally to the observed return. In the long term, successful actions are reinforced more strongly than unsuccessful ones, and the policy improves.

[3] For a detailed discussion and proof of the policy gradient theorem, the reader is referred to Sutton and Barto [SB18, Chapter 13].

In order to reduce variance, a value network can be trained in parallel with the policy network to approximate the value function. The estimate of the value network for state s_t can be subtracted from the return G_t and used in the update:

$$\theta_{t+1} = \theta_t + \alpha \cdot [G_t - \hat{v}_\pi(s_t)] \cdot \nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

The REINFORCE algorithm, augmented with the estimated value function, increases the probability of actions that improve the estimated value of the current state and decreases the probability of actions that reduce it. In this way, the variance is significantly reduced, while the expected value of the update remains unchanged.

The other possibility for updating the policy is based on temporal differences, similarly to value-based methods. The idea, similar to the REINFORCE algorithm with a value network, is to train a value network in parallel, which is used in this case to estimate the differences in value between successive states. These differences can be used to define the direction in which the gradient of the policy should be followed, without having to wait for the end of the episode:

$$\theta_{t+1} = \theta_t + \alpha \cdot [r_t + \hat{v}_\pi(s_{t+1}) - \hat{v}_\pi(s_t)] \cdot \nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

The credit assignment for each action within an episode is thus performed in a much more granular way, and policy updates become more efficient than in the REINFORCE algorithm, because the evaluation of each action depends only on the estimate of the state before and after the action, without the noise introduced by the remaining future actions until the episode ends. The value network can be trained simultaneously using temporal difference methods as explained in the previous section.

2.7 Eligibility Traces

On the one hand, as explained in Section 2.5, a disadvantage of Monte Carlo methods is that credit assignment to actions or states is influenced by subsequent actions and states occurring later in the episode. On the other hand, one-step temporal-difference methods assign credit in a more granular way, but at the cost of slower propagation of information to earlier actions or states: if an episode ends earlier than anticipated by the value network, only the final state or action of the episode is updated with the new information, while previous states and actions can only incorporate this information in subsequent training epochs. A way to balance the advantages and disadvantages of both approaches is to use n-step temporal-difference methods (cf. Section 2.5).

Instead of seeking a trade-off between the disadvantages of Monte Carlo and temporal-difference methods, *eligibility traces* provide a mechanism to combine the strengths of both approaches by propagating updates to previously visited states and actions without producing too much noise in the updates. The idea behind eligibility traces is to maintain a temporary memory variable, called a *trace* (often

implemented as an exponentially decaying average), that keeps track of recent information about gradients or returns. This mechanism helps propagate newly acquired knowledge to previously visited states or actions within an episode. Figure 1 illustrates how the steps of an episode are updated with knowledge gained at the last step, comparing Monte Carlo methods, one-step temporal-difference methods, and eligibility traces.

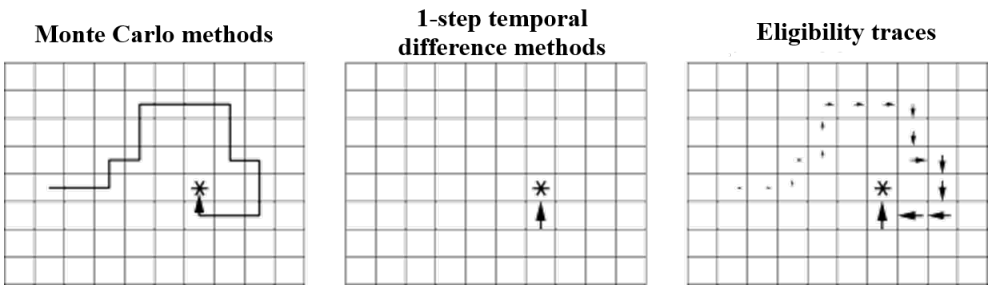

Figure 1: Credit assignment over past steps using Monte Carlo, one-step temporal-difference, and eligibility trace methods. Based on [SB18, Chapter 12]

The decay of the trace helps reduce noise while ensuring that information gained at each step propagates to multiple previous steps. Because state values are highly correlated across consecutive time steps –that is, nearby time steps usually have very similar values– eligibility traces are often used in conjunction with the value function. In contrast, consecutive actions may have very different effects on the expected return, so the one-step temporal-difference error is normally preferred over eligibility traces for updating the actor network in an actor-critic method. There are two main approaches to constructing eligibility traces:

- The *forward view* approach waits for the h subsequent steps in order to construct the return G_t^λ for each state or action. The return is defined as an exponentially weighted average of the n-step returns $G_{t:t+n}$ from time t up to $t+n$:

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1} \cdot G_{t:t+n} + \lambda^{h-t-1} \cdot G_{t:t+h}$$

  Here, λ is the coefficient of the exponentially decaying average, which controls the range of the propagated information. The n-step return $G_{t:t+n}$ is computed using the estimated value function $\hat{v}$:

$$G_{t:t+n} = R_{t+1} + \ldots + R_{t+n} + \hat{v}(s_{t+n})$$

  The difference between $G_{t:t+n}$ and $\hat{v}_\pi(s_t)$ is then used to update the weights in a manner similar to temporal-difference methods:

$$w_{t+1} = w_t + \alpha \cdot \left[G_t^\lambda - \hat{v}_\pi(s_t) \right] \cdot \nabla_w \hat{v}(s_t)$$

where w are the weights of the value function $\hat{v}$ Deviations from the expected value that occur closer in time are weighted more heavily than those occurring further in the future, as illustrated in Figure 2.

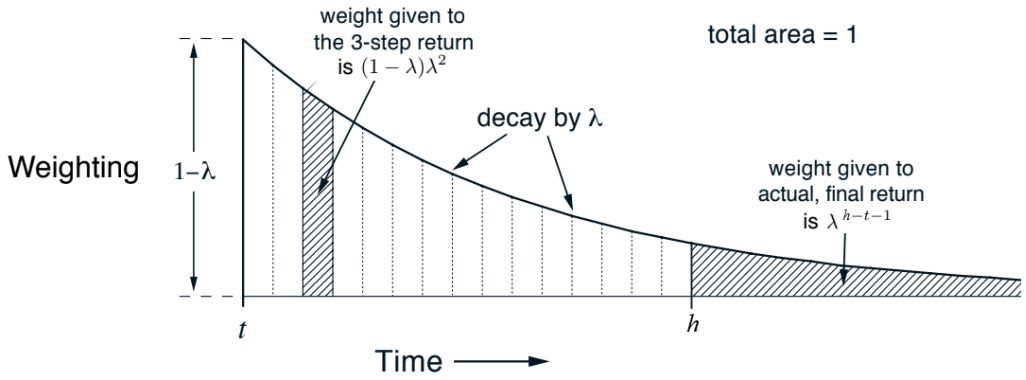

Figure 2: Exponential weighting of credit assignment over steps between t and $t+h$. Based on [SB18, Chapter 12]

The number of steps h used to construct the trace can be set to ∞ in order to wait until the end of the episode before computing the traces for each step. In this case, the family of algorithms is known as λ-*return* methods; otherwise, if h is finite, they are referred to as *truncated* λ-*return* methods.

- In contrast to forward-view methods, the *backward view* constructs the eligibility trace using gradients accumulated over time rather than multi-step returns. Instead of updating the weights based directly on future returns, backward-view methods maintain an exponentially decaying trace of past gradients:

$$z_{t+1} = \lambda z_t + \nabla_w \hat{v}(s_t)$$

This trace z assigns credit to recently visited states, allowing updates to incorporate both current and past information without waiting for future rewards. The parameters can then be updated online using temporal-difference errors:

$$w_{t+1} = w_t + \alpha\, \delta_t\, z_{t+1}$$

where

$$\delta_t = R_{t+1} + \hat{v}_\pi(s_{t+1}) - \hat{v}_\pi(s_t)$$

is the TD error. This backward-view formulation is analogous to the TD(λ) algorithm.

2.8 Extensive-Form Fictitious Play

In multi-agent games, transitions between states are determined both by the game environment and by the policies of the different agents. Therefore, a multi-agent game can only be considered a special case of a Markov decision process from the perspective of a single agent if the remaining agents are treated as part of the environment defining the state-transition dynamics. Since the environment is assumed to be stationary in reinforcement learning problems, the policies of the other agents must remain fixed in order to preserve the convergence properties of reinforcement learning methods. On the one hand, if several agents are trained simultaneously in a multi-agent game, the learning process of each agent continuously changes the state-transition dynamics experienced by the others, making the environment non-stationary from the perspective of every individual learner. On the other hand, if only a single agent is trained while the remaining agents follow fixed policies, the environment becomes stationary and standard reinforcement learning methods may converge. However, the learned policy is then only optimal with respect to the fixed behavior of the other agents. In either case, either because the environment becomes non-stationary or because the trained policy is optimized only with respect to fixed arbitrary policies, classical reinforcement learning algorithms cannot generally find stable policies for multi-agent games.

One model for simple multi-agent games is the *normal-form game*, in which all players choose their actions simultaneously, each player knows the possible actions and corresponding payoffs, and the outcome depends on the combination of chosen actions. The policy of each agent can be represented by a *strategy*, a choice of an action based on the state of the game. If the choice is deterministic (i.e., the same action is always taken in the same state), the strategy is known as a *pure strategy*. If, instead, the strategy does not specify a deterministic choice, it can be expressed as a probability distribution over pure strategies and is referred to as a *mixed strategy*. *Fictitious play* [Bro51] is a learning process for a set of strategies (a *strategy profile*) in normal-form games. The process consists of cycles, in which the game is played repeatedly with fixed strategies. After each cycle, the strategy of every agent is adjusted in order to form a best response to the observed behavior of the opponents. For specific classes of normal-form games, including two-player zero-sum games [Rob51] and potential games [MS96], the simultaneous training of agents using fictitious play converges to a Nash equilibrium, in which every strategy is a best response to the strategies of the other agents in the strategy profile.

Extensive-form games are a more expressive and general model of multi-agent games, in which agents take actions sequentially during a game, in contrast to normal-form games where each agent selects a single action at the beginning of the game. In extensive-form games, strategies–known as *behavioral strategies* –do not merely choose a single action at the beginning of the game, but assign an action to each of the different states in the course of a game. A behavioral strategy of an agent in an extensive-form game can be represented by a *realization plan*, i.e., a distribution over sequences of game states and actions induced by that strategy. Kuhn's Theo-

rem [Kuh53] shows that, for agents with perfect recall (the ability to remember all previously visited states), behavioral strategies are equivalent to mixed strategies. In other words, every extensive-form game can be represented in terms of an equivalent normal-form game. A complete contingent plan of an extensive-form game, that specifies a deterministic action for each of the game states, can be understood with Kuhn's Theorem as a pure strategy in a normal-form game, making both game models equivalent, because the same distributions over actions can be achieved by randomizing the strategy at the beginning of the game (mixed strategies) or at each state of the game (behavioral strategies). The convergence properties of fictitious play in normal-form games therefore, extend to extensive-form games [4].

Based on the results presented in the previous paragraphs, Heinrich, Lanctot, and Silver [HS16, HLS15] developed a machine learning framework for neural networks that implements fictitious play for extensive-form games. According to fictitious play, the framework consists of several cycles, in which each agent learns an approximate best response against the fixed set of agents from the previous iteration using classical reinforcement learning. At the end of each cycle, the set of agents is updated in the direction of these best responses and then fixed again for the next iteration. This framework is referred to as *Fictitious Self-Play*. Under the same conditions as fictitious play –namely, that the update step sizes α_t satisfy $\alpha_t \to 0$ for $t \to \infty$ and $\sum_{t=1}^{\infty} \alpha_t = \infty$ –Fictitious Self-Play converges to a Nash equilibrium [HLS15].

Nevertheless, the implementation of the framework by Heinrich and Silver [HS16] suffers from a limitation. As a reinforcement learning method, they use a variant of Q-learning, a value-based method. Q-learning was successfully applied by Silver and co-authors a few years earlier to train neural networks for Atari games [MKS+13]. As discussed in Section 2.5, value-based methods often exhibit abrupt changes in the derived policy. Consequently, such methods do not satisfy the fictitious play requirement of smooth policy updates. To address this issue, Heinrich and Silver introduce a policy network, which is trained in a second phase of each iteration of fictitious self-play, after training the value network using reinforcement learning. This policy network is updated via supervised learning (cf. Section 2.4), using labels obtained by applying the greedy policy derived from the Q-learning value network to the training data. The resulting framework resembles an actor-critic method, in which the value network guides the updates of the policy network, albeit with the additional overhead of the expensive process of deriving a policy from an approximated value-function. In contrast, policy-based methods, like the actor-critic method, provide a more suitable reinforcement learning paradigm for fictitious play, due to their ability to update policies in small steps. If policy-based methods are used within fictitious play, only iterative reinforcement learning phases

[4] In this short exposition of extensive-form games, we have ignored that in this model, the agents may have limited information about the state of the game, as this aspect has no implications for our experiments. Fictitious play can also be applied to extensive-form games with imperfect information, while preserving the same convergence properties.

are required to converge to a Nash equilibrium [5].

3 War Games on Hexagonal Grids

In this paper, the concept of "war games on hexagonal grids" refers to a class of strategy games that represent a battle between two or more armies on a hexagonal grid. The hexagonal grid aims to provide the most realistic discrete representation of the battlefield, in contrast to more abstract representations such as those used in chess. Two or more players play against each other by alternately giving commands to their units in their turns. There might be different types of units and terrains that influence the movement and fighting capabilities of the units. Typical winning conditions are the defeat of all enemy units or the occupation of important positions in the battlefield. A screen capture of an instance of this class of game with several units for each player is shown in Figure 3.

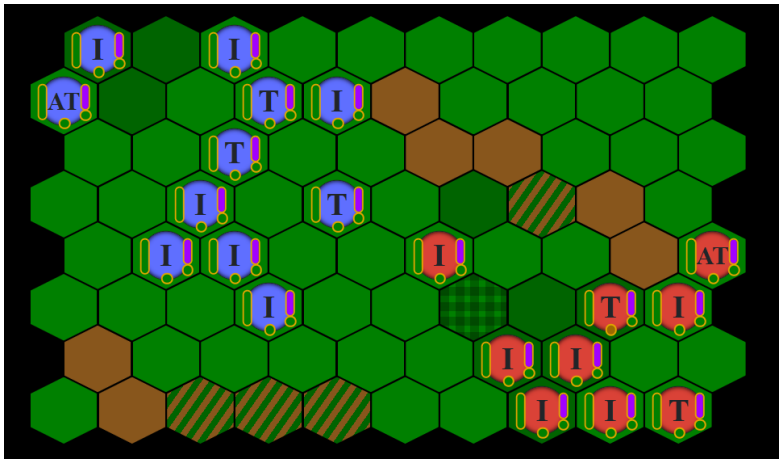

Figure 3: War game on hexagonal grids

Based on spatial relationships inside the grid, on the relation of strength between units, and on the cooperation of the different units, various strategies (fleeing, pursuing, occupying important positions in the battlefield...) can be more or less advantageous in different game states. An expert player can identify different game states to follow the most advantageous strategy in each of them.

This type of game can naturally be interpreted as a multi-agent system from two complementary perspectives:

- From a competitive perspective, the game represents the conflict between two opposing armies.

[5] This is the approach that Silver and the DeepMind team appear to have taken when they developed a learning algorithm for StarCraft [VBC+19] a few years later. A variant of an actor-critic method was trained using fictitious self-play, apparently without introducing a supervised learning stage in the fictitious play cycles.

- From a cooperative perspective, each army must coordinate the actions of its individual units in order to effectively defeat the opposing force.

3.1 Machine Learning Challenges in War Games on Hexagonal Grids

In general, reinforcement learning tasks present significantly greater convergence challenges than supervised or unsupervised learning problems. Unlike supervised learning, where the error can be computed directly from labelled data, or unsupervised learning, where the learning signal is derived from the intrinsic structure of the data, reinforcement learning relies primarily on reward signals obtained through interaction with the environment.

However, using rewards alone to estimate the learning error often produces poor results, especially when rewards are sparse or delayed in time. As discussed in the presentation of Monte Carlo methods in Section 2.5, delayed rewards make it difficult to correctly assign the contribution of individual actions to future outcomes. Alternatively, a value function can be introduced to estimate the expected long-term return associated with particular states or actions. In this case, beyond adding a new neural network that must be trained, an additional difficulty arises from the strong dependency between the value function and the policy itself. Whether the policy is derived directly from the value function or represented separately, as in actor-critic methods, both components remain tightly coupled. The value function can only be trained on the regions of the state space explored by the current policy, while policy updates depend on the accuracy of the value estimates. Furthermore, reinforcement learning data are inherently correlated, since consecutive states within an episode are sequentially dependent. This temporal correlation reduces the diversity of the training data used to update the machine-learning algorithm.

While standard reinforcement learning methods typically converge toward equilibria corresponding to optimal policies for a given environment, machine learning approaches for extensive-form games based on fictitious self-play instead aim to converge toward a Nash equilibrium with respect to a strategy profile. In fictitious self-play, the strategy profile evolves continuously throughout the learning process as agents repeatedly adapt to the observed behavior of their opponents. As a consequence, the target equilibrium itself changes over time, making the training highly unstable. In this setting, neural networks must learn not only the dynamics of the environment, but also the evolving strategies of competing agents. The learning process with fictitious self-play must simultaneously cope with delayed rewards, correlated data, and continuously shifting opponent behaviors, all of which significantly complicate convergence toward stable and robust strategies.

Besides the general challenges involved in training neural networks for extensive-form games, war games played on hexagonal grids introduce additional difficulties for machine learning methods. In particular, action selection depends strongly on the spatial configuration of units and terrain on the grid. Two key aspects of action selection must be considered by an effective artificial intelligence system. First, the

type of action chosen for a specific unit should primarily depend on the local spatial configuration around that unit. The immediate neighborhood of the controlled unit should exert a significantly greater influence on the decision than distant regions of the map. For example, if powerful enemy units are located nearby, retreating may be the most appropriate strategy, even if defeatable enemy units are present elsewhere on the grid. To make effective decisions of this kind, the neural network requires an attention mechanism centered on the unit selected to execute the action. Second, actions for this class of games possess an intrinsic directional component that must remain consistent both with the selected action type and with the spatial arrangement of the game grid. For instance, when retreat conditions are met, the unit should move away from nearby enemy units in order to perform an effective withdrawal. Consequently, the neural network must solve two related tasks:

1. the selection of the action type, conditioned on the local spatial configuration around the controlled unit,

2. and the selection of the movement direction, conditioned both on the chosen action type and, again, on the spatial configuration of the grid.

In contrast, computer vision classification tasks do not involve such a coupled dependency between semantic decisions and spatially consistent directional outputs.

3.2 Rotation Invariance

The property whereby certain features of the data remain unchanged under rotations is known as *rotation invariance*. In many war games, strategies are invariant under rotations of the board along the axes of the grid. If a strategy yields a specific expected return for a given board configuration, then rotating the board and applying the correspondingly rotated strategy should produce exactly the same expected return (cf. Figure 4). Due to the discrete nature of the grid, perfect rotational invariance applies only to rotations aligned with the grid's axial symmetry. For example, in a hexagonal grid, only rotations by multiples of 60° preserve the invariance of the strategies. In contrast, rotation invariance is much less common in computer vision applications. It appears naturally in some cases, such as aerial imagery, but in most scenarios the orientation of the image carries semantic information. For instance, an image containing text cannot generally be rotated or mirrored without altering or losing part of its meaning.

4 Hexagonal Convolutional Networks

In order to process the hexagonal spatial features of the game board efficiently, a convolutional neural network must respect the geometry of the hexagonal grid. This requires adapting standard convolutional operations to the structure of the hexagonal grid:

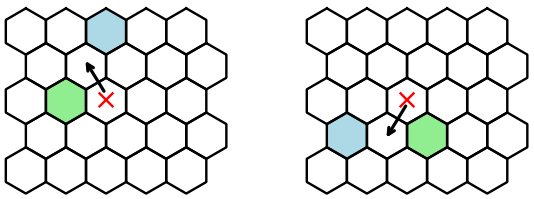

Figure 4: Actions selected by a rotation-invariant strategy under different rotations of the board

- construction of hexagonal kernels of different sizes,
- computation of homogeneous strides along the three hexagonal axes,
- definition of attention areas centered on a hexagonal tile,
- and support for weight sharing that encodes rotational symmetry over the six hexagonal axes.

By addressing these issues, the efficiency of the learning process is expected to improve, as the convolutional operation becomes better aligned with the underlying hexagonal structure. In general, hexagonal tilings offer several advantages over square tilings, as noted by Emiel Hoogeboom et al. [HPCW18]. First, in a hexagonal grid, all six neighboring tiles are located at the same distance from the center tile. In contrast, the eight neighbors of a square grid cell are divided into two groups with different distances to the center: cardinal neighbors and diagonal neighbors. Second, while square grids possess two principal axes (the x-axis and the y-axis), hexagonal grids have three principal axes, reflecting their sixfold symmetry. Together, these properties imply that different directions in hexagonal grids exhibit more uniform behavior than in square grids. For example, in square grids, distances along diagonal directions are significantly larger than those along cardinal directions defined by the axes. The characteristic of having similar properties in all directions is known as *isotropy*: in this sense, square grids exhibit greater *anisotropy* (or lower *isotropy*) than hexagonal grids.

The low anisotropy of hexagonal grids is one of the main reasons for their frequent use in board games. Furthermore, in several areas of physics where data are naturally represented on hexagonal grids, the use of hexagonal convolutions can be particularly beneficial [SH19]. Even in computer vision classification tasks, where the underlying data are not inherently hexagonal, the use of hexagonal convolutions instead of square convolutions has been shown to improve performance [HPCW18].

4.1 Strides and Kernel Sizes

The hexagonal convolutional layers must support different kernel sizes and strides. The kernels used in a hexagonal convolution are defined by the number of rings

surrounding a central tile. Figures 5 and 6 illustrate two examples of kernels of sizes 3 and 5, respectively.

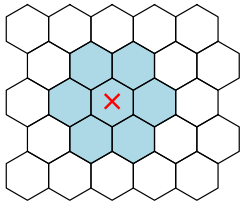

Figure 5: Hexagonal kernel of size 3

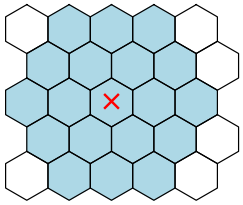

Figure 6: Hexagonal kernel of size 5

The strides in hexagonal convolutions cannot be defined along orthogonal directions, as in square grids, because the overlap between the different kernel applications would not be uniform along the three axes of the hexagonal grid, and the isotropy of the hexagonal convolutional network would be lost (cf. Figure 7, where the overlap is significantly higher between kernel application aligned in columns than between those aligned in rows). Instead, strides must follow the three hexagonal axes, as shown in Figure 8. In this way, the grid is covered more homogeneously and the output of the convolutional layer preserves its structure, while maintaining the rotation invariance of the data, independent of the stride length.

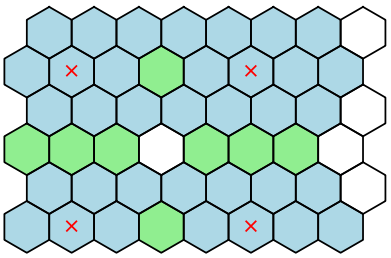

Figure 7: Orthogonal strides of length 3 with kernel size 5; the green tiles indicate the overlapping regions between different kernel applications.

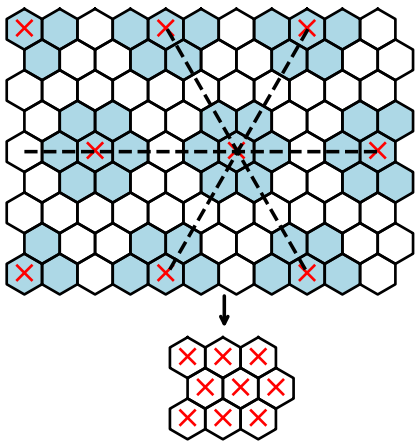

Figure 8: Strides of length 4 with kernel size 3, following the hexagonal axes

4.2 Attention Mechanism

Because action selection in war games on hexagonal grids is performed based on spatial data relative to the unit's position (cf. Section 3), the policy network requires an attention mechanism to focus on the region of the grid surrounding the unit to be moved. A basic attention mechanism is defined as a hexagonal area centered on the unit. It consists of concentric rings around the unit and applies convolutional operations that preserve the ring structure while using strides. The stride scheme following the hexagonal axes, described above, preserves this ring structure, as shown in Figure 9.

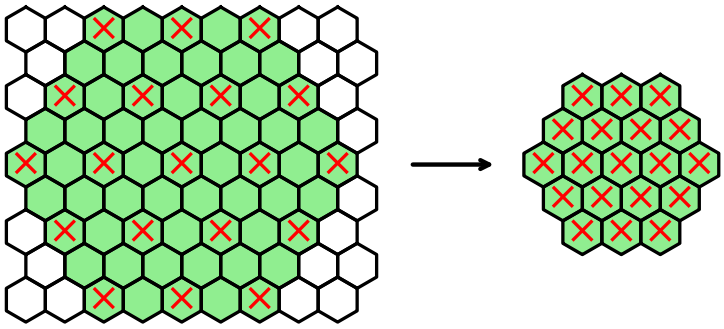

Figure 9: Stride of length 2 on an attention area with 4 rings

4.3 Weight Sharing for Rotational Symmetry

When the data exhibit rotational invariance, as is the case in many war games, neural networks can exploit this property to improve both training efficiency and generalization performance. Since the rotational symmetry in most war games is exact under rotations by multiples of $60°$, the most effective approach is to incorporate this prior knowledge directly into the architecture of the neural network, rather than requiring the neural network to learn it solely from data. Hexagonal convolutional networks are particularly well suited for exploiting rotational symmetry through weight sharing. The higher isotropy of hexagonal grids enables a more accurate representation of rotationally invariant patterns compared with square grids. Another consequence of the high isotropy of hexagonal grids is that they admit six rotational symmetries around the grid axes, whereas square grids admit only four. For each of the six rotations defined by the three axes of the hexagonal grid, the convolutional network performs the same computations while sharing weights. The only difference lies in the permutation of the kernel weights to account for rotations by multiples of $60°$. In this way, the outputs of the neural network remain identical whenever the input grid is rotated according to the symmetries of the hexagonal grid.

For each of the six directions of the hexagonal grid, a separate group of channels is constructed. During the forward pass, these channel groups remain independent and do not interact with one another until the final layer, which combines the information from all groups to produce a rotation-invariant output (i.e., the logits corresponding to the possible movement destinations or an estimated value for each tile of the grid). Figure 10 illustrates how the data propagate through the different layers of the neural network, including convolution, activation, and normalization layers, while remaining partitioned into the six directional channel groups. Although the figure depicts six parallel processing pipelines, the computations can be implemented efficiently using a single tensor at each layer with an additional dimension for the six directional groups.

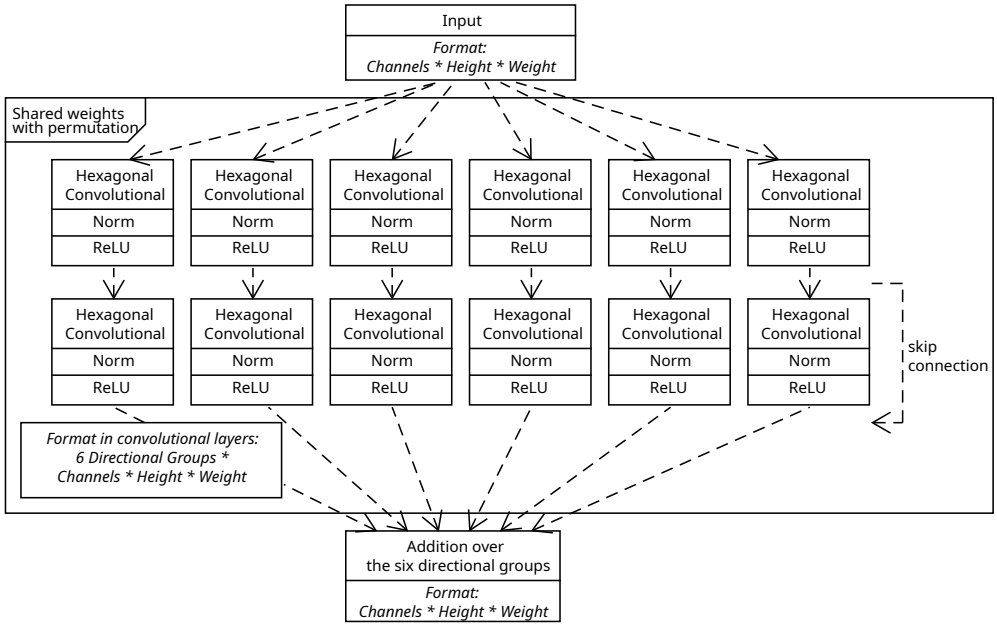

Figure 10: Multilayer convolutional network with weight sharing across directional groups

The permutation of the shared convolutional weights across the different channel groups must follow an ordering consistent with the rotational symmetry of the grid. The specific permutation pattern depends on the distance between the kernel ring and the center of attention (typically the position of the unit to be moved). Figure 11 illustrates how the permutation step length varies according to the ring on which the corresponding filter is applied.

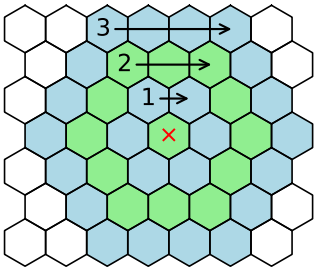

Figure 11: Permutation step lengths for shared weights at different kernel rings in order to preserve rotational symmetry

5 Uncorrelated Mini-Batch Updates in Reinforcement Learning

It has been noted that reinforcement learning methods combined with neural networks suffer from significant stability problems when naive stochastic gradient descent is used. This is due to the fact that the knowledge learned by the neural network is global, whereas the states from which learning is performed are local [Rie05, MPBM+16]: a change induced by a particular state can influence the evaluation of other states, thereby hindering or even reversing learning. This problem, which arises when only a limited region of the state space is explored, becomes especially severe when learning relies on correlated states generated from a single game instance.

5.1 Methods for Generating Uncorrelated Data

To address the above stability problems, the updates of the learning process must be performed using uncorrelated states; that is, the states must originate from different running instances of the game. The generation of uncorrelated data can be achieved through two main strategies:

1. Storing trajectories (sequences of states, actions, and rewards) and using them a posteriori during the training process. This technique is known as *experience replay* [Rie05];

2. Running multiple instances of the game in parallel threads during training and consuming the generated data immediately. This technique is known as *online learning*.

The use of experience replay requires a mechanism known as *importance sampling* [SB18, Chapter 5] to reuse data collected under a policy different from the policy currently being trained. The policy that generated the data is known as the *behavior policy*, while the policy being trained is known as the *target policy*. The actions

taken under the behavior policy follow a different probability distribution from those taken under the target policy. Therefore, updates of the target policy based on actions generated by the behavior policy must be scaled by a weight known as the *importance sampling ratio*. For one-step temporal-difference methods, the importance sampling ratio ρ represents the likelihood ratio between an action being taken under the target policy π and the behavior policy b:

$$\rho_{t:t+1} = \frac{\pi(A_t \mid S_t)}{b(A_t \mid S_t)}$$

Instead, if an n-step bootstrapping update is used, the ratio between all actions taken in the trajectory from time t to $t+n$ must be multiplied to obtain the overall importance sampling ratio:

$$\rho_{t:t+n} = \prod_{k=t}^{t+n} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}$$

Using this importance sampling ratio, the n-step bootstrapping update can be written as:

$$w_{t+1} = w_t + \alpha \cdot \rho_{t:t+n} \cdot [R_{t+1} + \cdots + R_{t+n} + \hat{v}(s_{t+n}) - \hat{v}(s_t)] \cdot \nabla_w \hat{v}(s_t)$$

where α is the learning rate.

Importance sampling is necessary when training is performed across multiple machines, because the target policy cannot be distributed immediately to all machines after each update. As a result, a mismatch arises between the current version of the target policy and the versions used to generate data on the different machines.

In the early 2000s, experience replay was widely used for batch training in reinforcement learning [Rie05]. Following the introduction of the asynchronous advantage actor-critic algorithm in 2016 [MPBM$^+$16], which relies on a single machine, online learning methods gained popularity for single-machine training.

5.2 Online Training Strategies

Performing a series of stochastic gradient descent updates using single data points from independently generated game instances, rather than from a single correlated trajectory, already improves the stability of reinforcement learning methods, as the training data is no longer correlated. Nevertheless, each update is still based on a particular state from the overall state space, and the direction of the update is driven by local information specific to that state. Global knowledge of the full state space of the game is not gained immediately in each epoch and is difficult to achieve with consecutive updates, since consecutive updates tend to correct the locality of previous updates without properly generalising [Agg23, p. 58–60]. A way to obtain more global information in each update is to base the gradient on several (uncorrelated)

data points using mini-batch gradient descent, as explained in Section 5, so that the state space of the game is better represented in each update.

There are two main architectures for computing gradients from multiple data points generated online by several agents running different instances of the game:

- One possibility is to generate data and compute separate gradients for multiple data points across several CPU cores. The gradients are then aggregated to update a shared neural network at regular intervals. This is the approach used in the *asynchronous advantage actor-critic (A3C)* algorithm [MPBM$^+$16]. In this setting, the use of a GPU is not beneficial, since the conditional logic of the game runs more efficiently on the CPU, and the gradients are computed from single data points, so GPU parallelism cannot be effectively exploited.
- The other possibility, which has recently shown better empirical performance, is to run only the game environment on CPU cores to generate data, while a separate learner process receives trajectories from the game instances, constructs batches, and computes the gradients on a GPU to update the neural network. Because gradients are computed from large batches of data, GPU acceleration becomes highly effective, as has been standard since the 2010s in supervised and unsupervised deep learning. This is the approach used in the *Importance Weighted Actor-Learner Architecture (IMPALA)* [ESM$^+$18].

5.3 Eligibility Traces in Conjunction with Scalable Learning Methods

Beyond improving training stability, scalable learning methods that compute gradients from large batches of data, such as the IMPALA architecture, also provide significant runtime advantages. Large batches enable parallel computation of gradients, which is especially beneficial when using a GPU. In addition, batching improves memory efficiency. Since the data in a batch is stored contiguously in memory, cache misses are reduced and memory accesses become more regular. Furthermore, the data used for running the game environment and the data used for gradient computation do not need to be accessed intermittently, which further improves memory and computational efficiency.

Nevertheless, the parallel computation of gradients makes the use of eligibility traces more challenging, because traces must necessarily be computed through an iterative process (cf. Section 2.7). In particular, the recursive dependency between consecutive trace values prevents full parallelisation along the temporal dimension of a trajectory.

A forward-view method for computing eligibility traces, used in the IMPALA architecture, consists of first calculating the one-step temporal-difference errors along a trajectory:

$$\delta_t \stackrel{\text{def}}{=} r_t + V(x_{t+1}) - V(x_t)$$

The computation of temporal-difference can still be performed in parallel across timesteps, since each δ_t depends only on local quantities (x_t, x_{t+1}, r_t).

These temporal-difference errors are then used to compute the λ-return G_t^λ for each timestep:

$$G_t^\lambda \stackrel{\text{def}}{=} V(x_t) + \sum_{k=0}^{T-t} \lambda^k \cdot \delta_{t+k}$$

where T denotes the terminal timestep of the trajectory.

The recursive term in the λ-return allows it to be computed iteratively backwards from the terminal state, since:

$$\sum_{k=0}^{T-(n-1)} \lambda^k \cdot \delta_{(n-1)+k} = \delta_{n-1} + \lambda \cdot \sum_{k=0}^{T-n} \lambda^k \cdot \delta_{n+k}$$

Once the returns G_t^λ have been computed for all timesteps, the mini-batch gradient can be calculated efficiently in parallel. Because the iterative computation of the λ-return cannot be avoided, the batch should not be constructed as a one-dimensional collection of independent data points, but rather as a two-dimensional structure. The first dimension of size N corresponds to the different partial trajectories contained in the batch while the second dimension of size T corresponds to the temporal order of steps within a trajectory.

For the construction of this two-dimensional batched tensor, the following procedure is used: the collected trajectories are stored column-wise along the temporal dimension. When a trajectory terminates, the remaining steps of the trajectory are stored in a following column, as if the remainder of the trajectory were an independent one. If the temporal dimension is exhausted before a trajectory terminates, the remaining entries of the column of the tensor are filled with a new trajectory. Figure 12 illustrates the construction of such a two-dimensional batch using multiple trajectories.

Because the batch contains multiple independent partial trajectories, the computation of λ-returns can be parallelised across trajectories, even though it cannot be parallelised along the temporal dimension of an individual trajectory.

The exposed version of the forward view is based on the *Generalized Advantage Estimation (GAE)*[SML$^+$15]. The organization of the batch tensor into two dimensions –one for partial trajectories and another for temporal order within trajectories– has become standard in reinforcement learning tasks with the IMPALA architecture [ESM$^+$18], in order to optimize computation on GPUs.

The adjusted Generalized Advantage Estimation algorithm with a two-dimensional batch tensor is presented in Algorithm 1:

- The input $rewards$, $mask$, and $values$ are batch tensors with two dimensions of sizes N and T. We use NumPy-style indexing, where $values[:, 1:]$ denotes all elements except the first time step across all partial trajectories, and $values[:, :-1]$ denotes all elements except the last time step across all partial trajectories. The parameter λ is the decay coefficient of the eligibility trace.

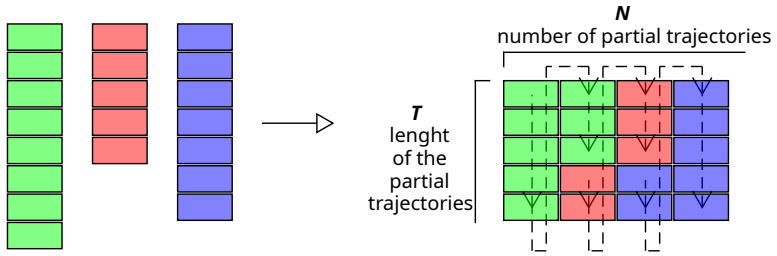

Figure 12: Construction of a two-dimensional batch tensor for the efficient computation of eligibility traces. Different trajectories are represented with different colors. Entries within each column of the tensor are temporally dependent. Rows of the tensor can be processed in parallel, since their entries are treated as independent samples, even if they originate from the same trajectory.

- In line 1, the one-step temporal-difference errors δ_t (deltas) are computed in parallel. At the final step of a trajectory, the estimated value of the first step of the next partial trajectory should not be used. This is handled via the $mask$, which prevents propagation beyond episode boundaries.
- The $advantages$ correspond to the backward-recursive term $\sum_{k=0}^{T-t} \lambda^k \cdot \delta_{t+k}$ for each time step t within the partial trajectories. The variable gae stores the running accumulation of this sum. Both $advantages$ and gae are initialized to zero in lines 2 and 3.
- Lines 4 to 6 describe the part of the algorithm that cannot be fully parallelized. Instead, the temporal dimension of the batch tensor is processed sequentially, while computation across the dimension corresponding to the partial trajectory is parallelized. Thus, a small temporal dimension combined with a large number of partial trajectories improves GPU utilization. However, choosing a temporal dimension that is too small would limit the effective range of the eligibility trace.
- The λ-returns are computed fully in parallel by adding the estimated state values to the corresponding $advantages$. The last entry of the $values$ tensor can only be used as the second term of the final one-step temporal-difference error. To use this last entry as the first term of the next temporal-difference error, it must be carried over as the first entry of the next partial trajectory segment. Therefore, the temporal dimension of the output is reduced by one compared to the input.

Algorithm 1: Generalized Advantage Estimation (GAE)

Input: $rewards$, $mask$, λ, $values$
Output: $advantages$, $returns$

$deltas \leftarrow rewards[:, :-1] + (1 - mask[:, :-1]) \cdot values[:, 1:] - values[:, :-1]$
$advantages \leftarrow 0$
$gae \leftarrow 0$
for $t = T - 1$ ***downto*** 0 **do**
    $gae \leftarrow deltas[:, t] + \lambda \cdot (1 - mask[:, t]) \cdot gae$
    $advantages[:, t] \leftarrow gae$
$returns \leftarrow advantages + values[:, :-1]$
return $advantages, returns$

Although backward-view eligibility traces combined with stochastic gradient descent (i.e., using a single data point for each update) have the advantage of enabling online updates at every timestep, without requiring access to future rewards (cf. Section 2.7), the combination of forward-view eligibility traces with scalable reinforcement learning methods, such as the adaptation of the GAE algorithm within the IMPALA architecture, has become highly popular in recent years.

Nevertheless, it is conceivable to adapt the presented version of the GAE algorithm to a backward-view formulation:

- A two-dimensional batch tensor is used, as in the presented forward-view algorithm.
- Gradients for each timestep are computed using one-step temporal differences while preserving the two-dimensional structure of the batch tensor.
- The backward-view eligibility traces are then computed iteratively, across dimension of the batch tensor corresponding to the the partial trajectory, in the forward temporal direction (in contrast to GAE), using the gradients. With eligibility traces, the neural network parameters can be updated in a fully parallelized manner.

If eligibility traces are preserved between epochs for unfinished trajectories (which constitute the majority of trajectories), then the traces can span a much longer temporal range than the relatively short horizon represented by the temporal dimension of the batch tensor (see Figure 13). If eligibility traces are not preserved between epochs, the updates are based on exactly the same batch data and they are performed at the same timesteps as in the forward-view version. Consequently, the backward view would not provide significant advantages over GAE.

When eligibility traces are preserved across epochs, only one trajectory segment can be processed in each epoch, as backward-view eligibility traces require sequential updates. As a result, the state associated with each environment instance must be maintained between epochs, leading to increased memory consumption.

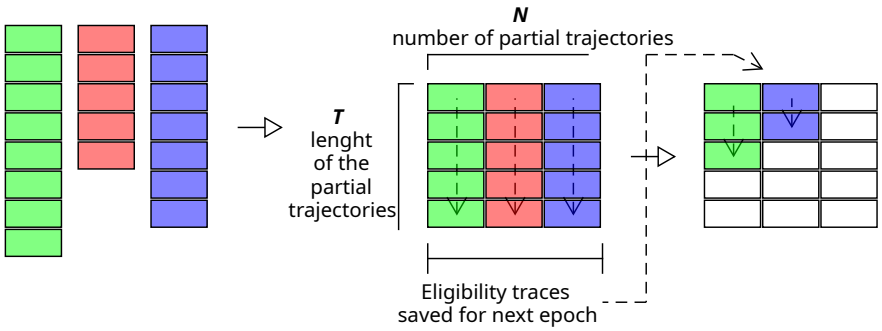

Figure 13: Construction of two-dimensional batch tensors for backward-view eligibility traces

Whether the potential benefits of longer-horizon eligibility traces outweigh these additional memory costs remains an open research question. The effectiveness of backward-view eligibility traces is likely to depend strongly on the characteristics of the game environment, since the resulting memory requirements may vary considerably across different settings.

6 Experimental Setup

This section describes the experimental setup, consisting of a new game environment developed from scratch and several techniques that make the implementation of the reinforcement learning method efficient for war games on hexagonal grids. At the end of this section, the neural network architectures used in the experiments are also described.

The experimental setup is designed to validate the effectiveness of convolutional neural networks on hexagonal grids and to assess the efficiency gains achieved through mini-batch gradient descent compared to stochastic gradient descent in reinforcement learning.

Developing a new game environment in parallel with the reinforcement learning method helped guide the research and enabled errors to be identified at an early stage. An iterative development approach was adopted in which the learning algorithms were tested and validated after each iteration, and further research could be conducted to meet the increasing complexity of the challenges. Using an existing game would have fixed the rules and mechanics from the outset, preventing earlier versions of the reinforcement learning algorithms, which worked only in simpler environments, from being evaluated during the development process. As a result, identifying the sources of errors would have been considerably more difficult.

6.1 Version of the Game for Experimentation

The version of the game used for experimentation retains almost all challenges of general war games on hexagonal grids, as described in Section 3.1. First, it is a competitive multi-agent game with sparse and delayed rewards, which means that an artificial intelligence for this game cannot be trained effectively with a standard reinforcement learning approach and instead requires an approach adapted to fictitious play. Second, action selection in this game has a twofold dependency on the spatial configuration of the grid: the choice of action type depends on the local spatial configuration around the unit to be moved, and each action also has a directional component that depends on the same spatial configuration.

Only the cooperative aspect between units of the same player is omitted in this implementation, by limiting each player to a single unit. Implementing a game in which each player controls multiple units exceeds the scope of this paper, as it would introduce non-homogeneous game steps: some actions would trigger a change of turn, allowing the opponent to command its units, while after other actions the same player would just continue commanding the remaining units.

In the basic implementation of the war game, illustrated in Figure 14, the hexagonal board consists of a 10×8 grid. Each player controls a single unit initialized with a random health level. Each unit can move one tile in any of the six directions or remain still. When a unit attempts to move onto the enemy unit's position, it first deals a fixed amount of damage and, if the enemy unit is not killed, receives the same amount of damage in return. No terrain or unit types that modify movement or damage are included. The game ends when a player's unit reaches negative health (win condition) or when a turn limit is reached (draw condition).

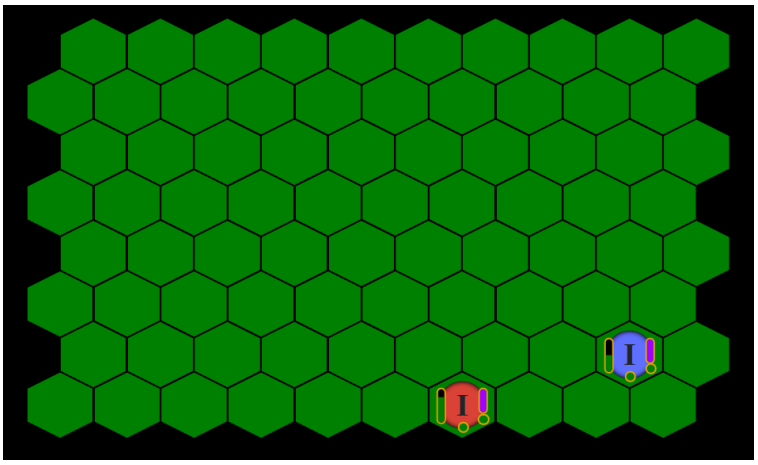

Figure 14: Game for experimentation

An advantage of this basic implementation is that, despite the machine learning challenges it presents, it enables a clear evaluation of the learning process, since the optimal strategies can be identified relatively easily by a human expert. The

critic and neural network can be evaluated automatically using data annotated by a human expert, as it will be explained in Section 7.

In the experimental implementation of the game, there are only three optimal strategies, determined by the relative health levels of the two units. If the agent's unit has higher health than the enemy unit, the optimal strategy is to pursue and attack the enemy until it is defeated. If the agent's unit has lower health than the enemy unit, the optimal strategy is to flee. The most difficult case to learn is when both units have similar health levels. In this situation, the AI should ensure that, when both health levels are close to zero, its own unit does not move close to the enemy unit, in order to prevent the enemy from dealing the final damage first.

The game state at a given time can be represented as a multidimensional tensor of spatial features, similar to representations used in image processing with convolutional neural networks. In this representation, different channels encode different features of tiles and units (for example, the terrain type of a tile or the health of a unit located on that tile).

6.2 Architecture and Responsibilities of the Main Modules

The code for the complete project, including the machine learning method and the game, is organized into four modules, each managed in a dedicated GitHub repository [AC26a, AC26b, AC26c, AC26d]. Each module has distinct responsibilities, illustrated in Figure 15:

1. **Game logic**: implements the game rules and the random generation of game instances.

2. **Neural network**: provides different architectures for the actor and critic networks, the implementation of a hexagonal convolutional neural network, and machine learning methods for training an actor-critic network using mini-batch gradient descent. The hexagonal convolutional network can be instantiated with arbitrary stride lengths and kernel sizes, with or without directional groups that share weights, offering great flexibility for use in different domains. The module also provides functionality to save and load a trained neural network from files in order to continue training at a later time, evaluate results, or run games against the artificial intelligence.

3. **HTTP API**: exposes HTTP endpoints for generating games between human players or against trained AI instances, and for receiving commands and forwarding them to the game logic. It can run multiple game instances in parallel while ensuring that users are mapped to their corresponding games.

4. **Web app (user interface)**: provides a user interface for manually evaluating the game and the trained AI instances. It is implemented using the Model-View-ViewModel (MVVM) architecture. Both the HTTP API and the web app

support different versions of the game rules, as these modules are agnostic to the specific rule set.

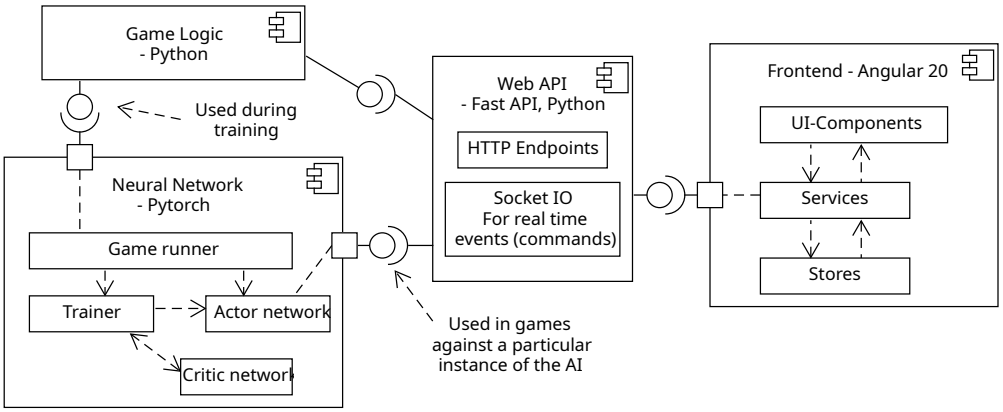

Figure 15: Architecture of the four modules for the game and AI

6.3 Scalable Machine Learning Method

Section 5 explained the importance of using uncorrelated data from different game instances when training reinforcement learning algorithms with neural networks. While neural network updates can be performed efficiently on a GPU using large batches of data, training data generation is more efficiently performed on CPU cores, since the game logic contains numerous conditional statements that are not well suited for GPU parallelism. To improve the efficiency of the data generation process, multiple game instances can be executed in parallel across several CPU cores. Furthermore, both data generation and gradient computation can be scaled across multiple machines. However, for the experiments presented in this paper, a single machine provided sufficient computational resources. The quantity of game instances executed in parallel can be adjusted according to the computational resources available. Because the experiments were conducted on a single machine, there was no need for an offline learning framework in which training data are collected independently of the current policy. Instead, the generated experience was used directly for neural network updates, following an online learning paradigm.

On the one hand, the processes responsible for generating data from game instances must be synchronized with the process that updates the neural network. A centralized component collects the generated data and initiates training only once a batch is complete. This configuration can be modeled as a bounded producer-consumer system with multiple producers and a single consumer. Additional synchronization is required to prevent producers from overwriting unprocessed data in shared memory and thereby causing data corruption. If the trainer is slower than

the data generation processes, the latter must be temporarily paused until the current batch has been processed. To avoid interrupting data generation whenever a training step is performed –not only in cases where the trainer is slower– the next batch of data can be collected concurrently while the trainer processes the current batch.

On the other hand, once a training step has been completed, the updated policy network must be distributed to the data generation processes to ensure that new data are generated using the latest policy. To achieve this, the updated neural network is written to a shared file, and the data generation processes are notified to load the new version.

The components involved in this synchronized workflow are illustrated in Figure 16.

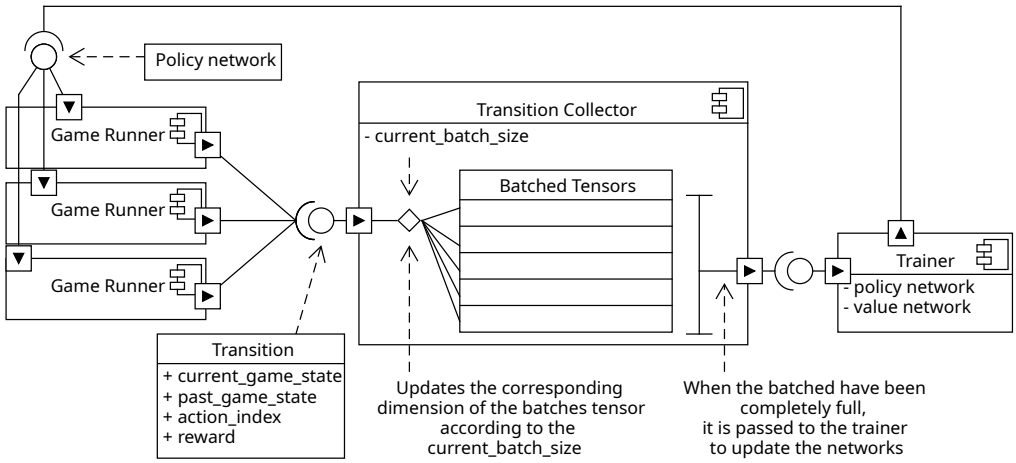

Figure 16: Components of the scalable training architecture. Multiple *Game Runner* components execute game instances in parallel on separate CPU cores and generate training data. The *Transition Collector* gathers the generated transitions, assembles training batches, and triggers a training step once a batch is complete. The *Trainer* performs gradient computations on the GPU and updates the neural network parameters.

6.4 Optimization and Regularisation Methods

To quantify the magnitude of updates in the value and policy networks, the divergence between the original and updated outputs was measured. While computing the divergence for the value network is straightforward, since its output consists of a single scalar value, the policy network produces a probability distribution over actions, requiring a more sophisticated measure of divergence. The *Kullback–Leibler*

(KL) Divergence is a widely used measure of the difference between a reference distribution (the original policy) P and the updated distribution (the new policy) Q. It is defined as:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

The learning rate was adjusted based on the measured divergences. For a temporal-difference error of 1 (corresponding to the difference between an expected draw and either a win or a loss), the learning step was calibrated to yield a divergence of 0.015 for the value network and 0.02 for the policy network.

To make the learning method more robust with respect to the choice of learning rate, the use of *optimizers*, i.e., adaptive learning rate methods such as Adam and RMSProp (cf. Section 2.2), was considered. During an early stage of development, when the update procedure was still based on stochastic gradient descent, both optimizers were evaluated as potential improvements to the learning process. To assess their effect, the divergences of the value and policy networks were measured after each update step. It was observed that updates associated with terminal states were weaker than those associated with non-terminal states, although the temporal-difference error of terminal states is often much larger.

In the following, an explanation of the phenomenon described above is provided. The Adam optimizer, when used in conjunction with stochastic gradient descent, accumulates momentum throughout the trajectory of an episode, based on temporal-difference errors. Since the reward signal appears only at the end of the episode, the momentum at the beginning of the episode is based entirely on the predictions of the value network, and thus reinforces presumably the inaccurate predictions of the untrained value network. In terminal states, however, substantial new information becomes available through the final reward. This information often contradicts the predictions of the value network, since the reward signal conflicts with the momentum accumulated from previous updates based on these predictions. As a result, the accumulated momentum partially counteracts the newly acquired information, leading to weaker updates in terminal states, where the most reliable learning signal is available. Consequently, the updates fail to fully incorporate the new information, preventing the networks from being adjusted in the appropriate direction. Since the value network is not sufficiently improved, the momentum continues to accumulate in a suboptimal direction throughout the learning process. Empirically, the use of the Adam and RMSProp optimizers in conjunction with stochastic gradient descent resulted in poorer performance with the experimental setup than standard stochastic gradient descent.

In contrast, with the use of mini-batch gradient descent with uncorrelated data, as explained in Section 5, multiple steps from several trajectories are incorporated within a single update step. This helps avoid the accumulation of momentum within a single trajectory based on inaccurate predictions from an untrained value network. First, the gradients computed from uncorrelated data yield much higher quality. Second, the momentum is not built across single steps from a trajectory,

since multiple steps of each trajectory are incorporated within a single update step. With mini-batch gradient descent, the momentum is formed by the common direction of updates across different game trajectories; thus, the relevant information from different trajectories is identified, and the learning process converges. The choice of the Adam optimizer for the experiments was motivated by its widespread use in combination with reinforcement learning methods [VBC+19, SWD+17, TPK18]. Once a scaled mini-batch gradient descent approach was adopted, Adam could be applied without destabilizing the learning process.

Another method to improve the stability of the learning process is to restrict the magnitude of parameter updates. Since data generation is often more expensive than gradient computation, multiple training steps can be performed on the same batch of data. However, this approach has the limitation that performing several epochs on the same data can increase the variance of policy update magnitudes. Excessively large updates in the policy can destabilize learning. To address this issue, the technique known as *proximal policy optimization* [SWD+17] introduces a constraint on policy updates by measuring the probability ratio $r_t(\theta)$ between the updated and the original policy:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

To limit deviations from the original policy that lead to the probability ratio $r_t(\theta)$ exceeding a certain threshold ϵ, the policy loss is scaled using a clipping function applied to the probability ratio:

$$\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$$

The clip function over the interval $[-1, 1]$ is illustrated in Figure 17.

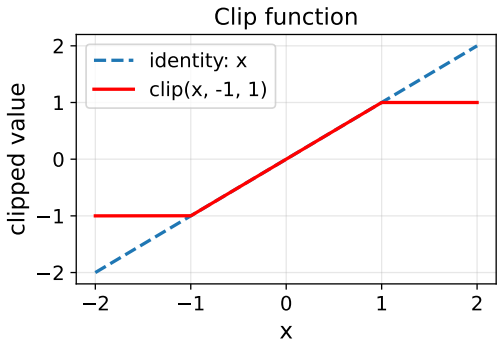

Figure 17: Clip function for the interval $[-1, 1]$

The *entropy regularisation*, the final technique discussed in this section, is used to encourage exploration in the policy. As noted in Section 2.6, policy-based methods

can naturally balance exploration of the state space when no effective strategy has yet been identified, and exploitation once convergence begins. Nevertheless, in the early stages of training, the policy may converge prematurely to a local optimum, resulting in a suboptimal strategy that is difficult to escape. To mitigate this issue, the entropy of the policy can be added as a regularisation term to the policy loss. This entropy-based regularisation encourages the policy to maintain a certain level of randomness in action selection, provided that such stochasticity does not lead to a significant decrease in expected reward. In this way, exploration is promoted in directions that the value network deems promising. During training, the weight of the entropy regularisation term can be gradually reduced, allowing the policy to converge towards a more deterministic strategy. The use of entropy regularisation has become a standard practice in reinforcement learning since its inclusion in the Asynchronous Advantage Actor-Critic algorithm by Mnih et al. [MPBM+16].

6.5 Neural Network Architectures

It is expected that both the convolutional neural network on hexagonal grids and the reinforcement learning method described in the previous sections can be applied to complex implementations of war games, without much need for adjustment. In contrast, the design of the neural network architectures is more limited to the application in the basic setting used for experimentation. In this section, the architectures of the actor and critic networks are described, focusing on the features that address specific challenges of war games on hexagonal grids.

6.5.1 Value Network Architecture

The value network is divided into two branches. The first branch is based on the difference between the health values of the two opposing units. This simple value is passed through several ReLU activation layers in order to learn the non-linearity of the relationship. This fundamental value is then modified by the second branch, which focuses on the spatial configuration of the grid.

The second branch contains a single convolutional neural network on hexagonal grids with several ReLU activation and normalization layers. The convolutional layers should be able to differentiate the following situations based on the spatial configuration of the grid: when both units are almost dead, and the opponent is close enough to deal the final damage, the value of the state is very low; in the opposite case, the estimated value should be higher, although in both situations the difference in health levels is the same.

To be able to process grids of different sizes, a global pooling layer is applied after the convolutional layers. *Global pooling layers* were first introduced by Lin et al. [LCY13] and popularized by the publication on the neural network GoogLeNet [SLJ+15] for image recognition tasks. A global pooling layer aggregates the spatial information of the feature maps produced by the convolutional layers into a single or few values per channel by averaging or taking the maximum value across the

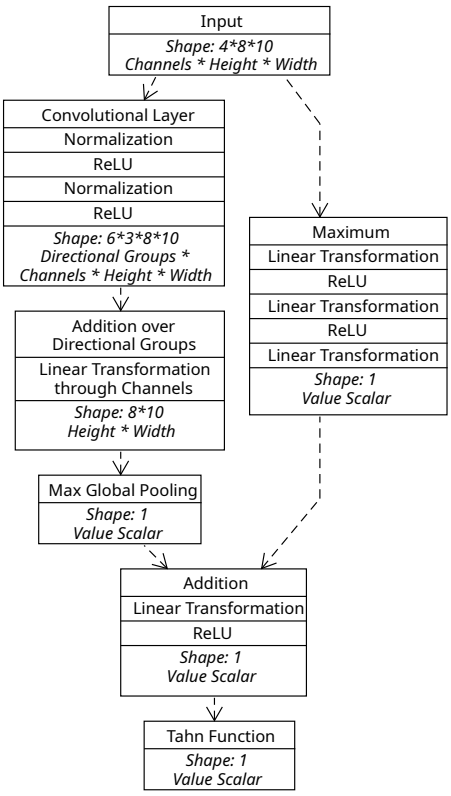

Figure 18: Architecture of the critic network

spatial dimensions. In the case of the presented experiment, taking the maximum value is more appropriate because it allows the identification of a critical situation without diluting it with the average of the other values.

Finally, a Tanh function is applied to the sum of the outputs of the two branches to bound the final value to the interval $[-1, 1]$. Figure 18 illustrates the architecture of the critic network.

6.5.2 General Design Principles of the Policy Network Architecture

The policy network architecture has to face more difficult challenges, as explained in Section 3.1: it must decide the correct action type (fleeing or pursuing) based on the local spatial configuration and implement this action type by selecting the correct direction of movement, which also depends on the spatial configuration of the grid relative to the unit to move.

A first design of the policy network was based on applying convolutional layers with directional groups of channels to the input state and using each of the six different directional groups for the evaluation of the six possible movement directions.

For the specific action of remaining still, an independent branch was used. In this way, each of the seven possible actions is evaluated by a different branch in the network, either through an independent branch or through a branch corresponding to one of the directional groups.

This approach has two main limitations. First, the use of different branches for different action types is not very generalizable, since it is limited to settings where the action space is finite and not too large, in order to keep the computational cost of the network manageable. Second, the selection of the action type and the directional component of the action are not separated in the design of the neural network. The same elements of the network are responsible for learning both aspects of the action selection. Additional depth is required in order to learn both aspects effectively. The convergence of the learning process is more difficult, not only because of the increased depth, but also because of the interdependence of the two aspects of action selection: adjustments to select the correct action type can lead to changes in the directional component of the action, and vice versa. It was not possible to successfully train this architecture.

An alternative to the previous approach, in which the neural network directly chooses the action, is to let the policy network evaluate to which tile in the grid the unit should move, and to derive the specific action from that choice. An advantage of this approach is that the neural network can evaluate the spatial configuration of the grid in order to solve the single task of determining how beneficial it is to move to each particular tile. The directional component of the action can be derived by computing the path between the unit's location and the target tile. This path can be computed with the A* algorithm, a variant of Dijkstra's algorithm. Because reaching a particular target typically requires performing a sequence of movements over multiple turns, this approach makes it very difficult for the neural network to consider the path to the target tile. For example, if a unit is weaker than the enemy, the artificial intelligence could choose to flee to the furthest tile away from the enemy. If that tile happens to be behind the enemy, then the artificial intelligence would lead its weak unit directly into the enemy's attack range, which is a very poor strategy. The problem of not considering the path to the target tile is illustrated in Figure 19.

The third and final version of the policy network, which was successfully trained, adopts from the previous approach the idea of evaluating tiles instead of actions, but instead of evaluating the whole grid, it is restricted to the tiles that can be reached in a single turn, thereby avoiding the problem of not considering the path to the target tile and simplifying the derivation of the action from the target tile.

In order to be able to consider the enemy unit when it lies outside the movement range of the unit, the distance to the enemy unit is encoded in a separate channel of the neural network input. Figure 20 illustrates the encoding of the distance to the enemy unit and its influence on the evaluation of the neighboring tiles around the own unit. With this encoding, the first step in the shortest path to the enemy unit in a homogeneous grid can be identified. The objective is to provide the neural

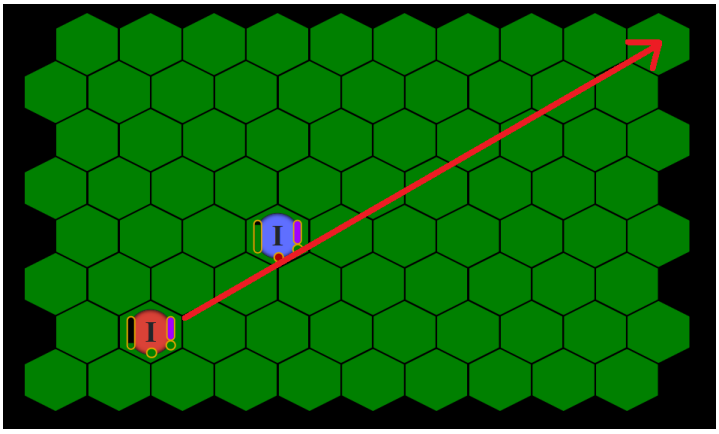

Figure 19: Incorrect execution of a fleeing strategy: the red unit, with less health than the blue unit, chooses to flee to the furthest tile away from the enemy, in the direction of the blue unit, which is a very poor strategy.

network with the information needed to learn the correct direction, depending on the location of the enemy unit relative to the own unit.

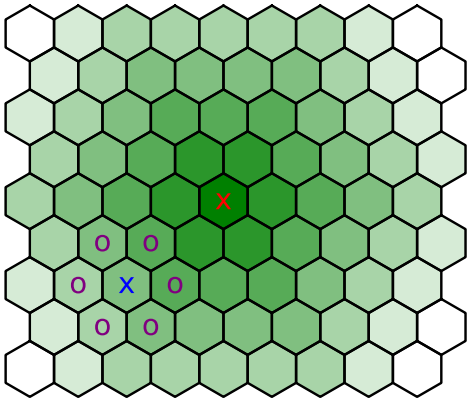

Figure 20: Radial distance encoding around the enemy unit, marked with a red cross, and its influence on the evaluation of the neighboring tiles around the own unit, marked with a blue cross.

Nevertheless, the optimal direction is conditioned not only by the location of the enemy unit, but also by the type of action. The direction to flee is the opposite of the direction required to perform an attack. To be able to learn this kind of conditional knowledge, an attention mechanism, a simplified version of the transformer architecture [VSP+17], is used. In contrast to standard fully connected neural networks or convolutional neural networks, this type of neural network does not perform a

linear transformation of the input data with learnable weights. Instead, it is based on matrix multiplications of the input data. The input is split into a *query* matrix Q of size $m \times n$ and a *key* matrix K of size $n \times p$, and the output is computed by multiplying the query matrix with the key matrix. To increase the expressiveness of the attention mechanism, an additional multiplication can be applied to the result of the first step, using a *value* matrix V of size $p \times q$. In the *transformer architecture*, linear transformations are applied to each of the three matrices, all derived from the same input data. The terminology of keys, queries, and values is taken from the analogy of a search engine.

For the present experiment, in order to determine the correct direction for an action type, only a basic attention mechanism is necessary. The radial distance encoding with respect to the enemy unit is used as the key matrix. The query consists of an encoding representing how much power the own unit can exert on each particular tile of the grid with respect to the enemy unit. A convolutional neural network should compute this encoding before the matrix multiplication. A positive value indicates that the own power surpasses the enemy power on that tile, while a negative value indicates the opposite. The highest result of the elementwise multiplication between the query and the radial distance encoding indicates the tile with the most beneficial power relationship.

For the basic experiment, only an elementwise multiplication between the query and the radial distance encoding is needed. As the game becomes more sophisticated, it is expected that the complexity and conditional nature of the optimal policy will also increase. In these cases, it is expected that several layers of self-attention with query, key, and value matrices will need to be stacked in order to model a policy based on complex conditions. The idea of using a multiplicative interaction, based on the self-attention mechanism, to determine the optimal direction for different action types was newly conceived by the author for this experiment.

6.5.3 Implementation of the Policy Network Architecture

The specific architecture of the neural network for the policy in the present experiment does not require much depth and instead makes greater use of width, making the learning process easier to converge. Diagram 21 illustrates the architecture of the actor network. The policy network begins with a single convolutional layer with a normalization layer and a ReLU activation layer. The directional groups of channels with shared weights are used only to reflect the rotation invariance of the game from the perspective of the six possible rotations of the grid. The directional groups do not correspond to the six possible movement directions, like in the first design of the policy network. Each of the six directional groups has four channels, resulting in 24 channels for the convolutional layer in total. The values of all directional groups are summed after the convolutional layer to obtain four total channels that are rotationally invariant with respect to the input.

An elementwise multiplication is applied between the output of the convolutional

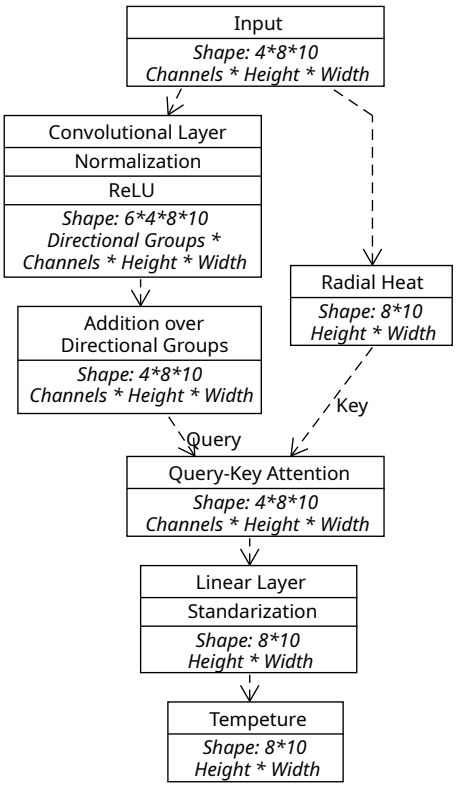

Figure 21: Architecture of the actor network

layer and the radial distance encoding. At the end, all four channels are reduced to a single one with a standard linear layer, yielding a single scalar for the evaluation of each individual tile. To enable the neural network to learn different degrees of randomness by selecting different actions, the evaluation values of the tiles are standardized and a learnable temperature is applied to the standardized values. The temperature acts as a scaling factor applied equally to all evaluation values of the tiles. To avoid the learned temperature converging to zero or infinity, which would cause numerical instability during training, the temperature is bounded using a tanh function. The output consists of numerical values for each neighboring tile, which are transformed into probabilities using a softmax function.

7 Evaluation

To evaluate the proposed techniques for training war games on hexagonal grids, different variations of the described reinforcement learning method were assessed and compared with each other. A further goal of the evaluation is to demonstrate

that even basic implementations of war games on hexagonal grids present significant machine learning challenges that require the use of the proposed techniques to be successfully trained. Indirectly, the correct functioning of the hexagonal convolutional networks is also corroborated, since they are a crucial part of both the value and policy networks, and the training process depends on these convolutional layers to converge to optimal results.

7.1 Methodology

To evaluate the results of training with the developed reinforcement learning method, two approaches were considered. The first and most basic approach was based on data annotated by a human expert in order to measure the distance between the output of the trained neural network and the optimal strategy identified by the expert. The second approach was not based on any prior knowledge and instead consisted of evaluating the trained AI by letting it play against different versions of itself.

The first approach, based on human knowledge, can only be successfully applied to sufficiently simple games whose challenges have already been completely mastered by a human expert, ensuring that optimal strategies discovered by an artificial intelligence are not misclassified. This human knowledge is used only for evaluation; the reinforcement learning method itself does not use prior knowledge. The annotated data consists of different game scenarios, where the expected output is determined by a human expert. For the value network, the mean squared error between the estimated value and the expected value was measured. For the policy network, the mean squared error between the probability assigned to the optimal action and 1 was measured.

The second approach, based on the results of matches between different versions of the AI, has fewer limitations and can be successfully applied to more complex games. This approach evaluates a set of different versions of the AI by letting them play against each other in a round-robin tournament (a tournament in which every participant plays against every other participant, similar to national football leagues) and by measuring the performance of each version against the others using the Elo rating system, a popular rating system used in chess and other competitive games. The Elo rating system assigns a higher rating to a player who wins against a strong opponent and a lower rating to a player who loses against a weak opponent. This system computes the expected outcome of a match for player A, E_A, based on the ratings of both players, R_A and R_B:

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

Based on the expected outcome of the match, the rating of player A is updated after the match using the following formula:

$$R_A^1 = R_A^0 + K(S - E_A)$$

where S is the actual outcome of the match (1 for a win, 0.5 for a draw, and 0 for a loss), and K is a development factor that determines how quickly ratings change.

In the Elo system, it is important to evaluate players in fair matches, where the skill of the players is the main factor determining the outcome of the match. Otherwise, the differences between the ratings of different players would be very small and heavily affected by the variance of the results. Since the game environment creates asymmetric scenarios during training, as the health levels of the two opposing units are initialized randomly, the expected outcome of these randomly generated scenarios is often skewed in favor of one of the sides. Therefore, a selection of fair scenarios has to be made. Similar to the *divide and choose* procedure in game theory, in a first stage, the most balanced scenarios are selected by one artificial intelligence, and in a second stage, the other contestant chooses which side to play, red or blue. In this way, no expert knowledge is needed for the selection of fair scenarios. The disadvantage of this approach is that not only the policy network, but also the value network is evaluated, since the value network determines the initial decision.

As we will see in the following, the actor-critic pairs were often trained with weaker agents, causing the value network to converge to overly optimistic estimates, as these estimates depend on the performance of the set of opponent agents. The selection of fair matches by an agent was based on estimates close to 0 in a range between -1 and 1 from the perspective of one player, which indicates an expected draw. Nevertheless, as a skilled agent can avoid defeat against a weak opponent, which would not be possible against a stronger opponent, the selected matches were often not fair, and one of the sides was favored due to these optimistic estimates. These estimates were biased by the weak performance of the opponent agents used during training. The evaluation of the matches with an Elo system was therefore very unstable, as the outcome of the game was already largely determined by the incorrect selection of fair matches.

Possible solutions to this problem include comparing the estimates using the value network of each agent –as in the present approach– but evaluating it from both perspectives, in order to select games that present similar estimations for both sides, even if these are far from 0. Another possible solution would be to train a separate, more accurate value network using data generated by all policy versions, and use it for the selection of fair matches.

7.2 Results

In the following, the results of several variations of the training process using the described scalable reinforcement learning method with mini-batch gradient descent are presented. The best training results were obtained with a fictitious play method comprising six cycles, in which the actor-critic pairs are trained against past versions of a set of six agents. Unlike the standard fictitious play method, in every cycle, each agent was trained not only against the latest version of the other agents, but also, at a lower frequency, against all previous versions of the other agents, in order to use a

larger set of opponent agents and to ensure that the trained policies did not cycle or forget strategies learned against earlier opponents. Each fictitious play cycle consists of 180 seconds of training for each agent. In each cycle, the entropy regularisation is reduced by half, starting from a high value of 0.2.

The training data is generated using 14 different threads on 8 CPU cores, and the gradient computation is performed on a GPU. The mini-batches have a temporal dimension of size 24, and the size of the partial trajectory dimension is 240, resulting in a total batch size of 5760 states per training step. The only reliable evaluation of the training process was based on annotated scenarios provided by a human expert, as explained above. These scenarios cover all three strategies described in Section 6.1. The results shown in Diagram 22 indicate a clear improvement in the training error with this configuration, which reaches almost optimal performance for all six instances [6]. The training time with the optimal configuration was the same as that of the other configurations; only the evaluation frequency was higher.

A configuration without mini-batches was also evaluated, in which the actor-critic pairs are updated after each generated state. The results shown in Diagram 23 indicate that learning is not detectable. Partly, these poor results were caused by an inefficient use of computational resources, as only one thread was used for data generation. Nevertheless, the training process stagnated at a very unsatisfactory level in all trained instances, because the computed gradients were based on a single data point and were therefore of very low quality, making the learning process highly unstable.

To show the importance of the fictitious play method, the actor-critic pairs were trained against a set of eight fixed random agents. Since the opponent agents are not updated, the training process consists of a standard reinforcement learning setup, where the environment, which also includes the fixed opponent agents, is static. The set of random agents is larger than in other configurations, in order to promote generalization and to prevent the trained policies from overfitting to a small set of opponents, which would lead to poor performance against unseen opponents. As Figure 24 shows, the training process achieves good results. In particular, the large set of opponent agents provides substantial stability during training. Nevertheless, even for this basic implementation of the game, fictitious play yields better results. For more complex settings, the importance of fictitious play is expected to be even higher.

To illustrate the importance of training against a large set of opponent agents for achieving good performance, the last results are compared with those of a configuration that uses fictitious play with a very small set of opponent agents, trained only against the latest version of the agents from the previous cycle (cf. Figure 25).

[6] The instability observed in two of the instances can be attributed to missing information in the input features of the neural network: the network has access only to information about the difference in health, but not to the absolute health values. Thus, in scenarios where both units are nearly defeated, the strategy of avoiding the final strike from the enemy while still attempting to kill the opponent when he enters the own unit's attack range is much more difficult to learn, and the training process becomes unstable.

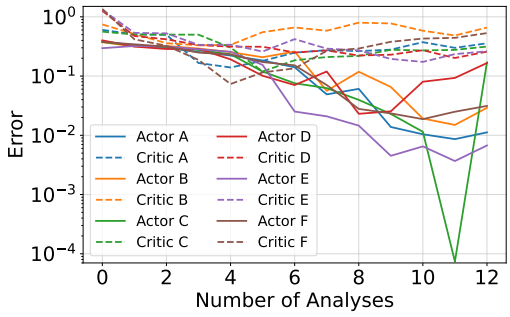

Figure 22: Error of six instances of an artificial intelligence trained with the optimal configuration

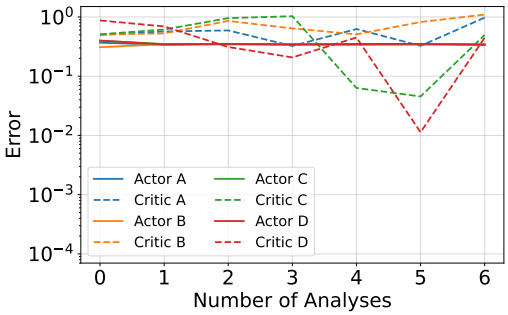

Figure 23: Error of training without mini-batches

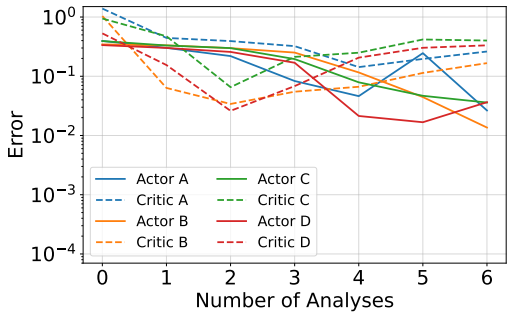

Figure 24: Error of four instances trained against eight fixed random agents

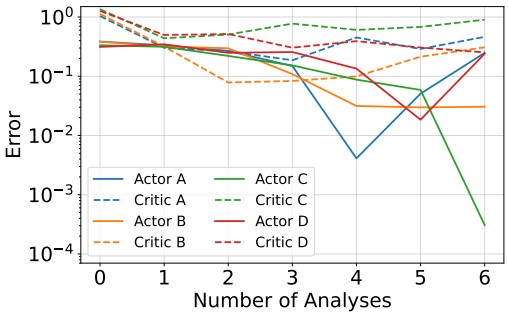

Figure 25: Error of four instances trained in pairs against previous versions

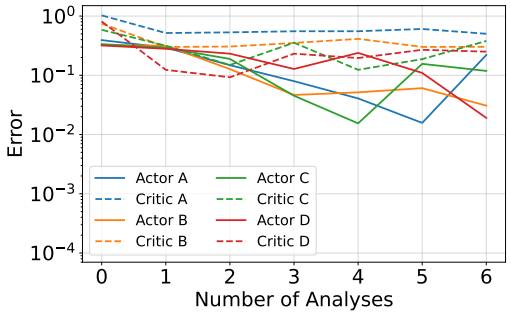

Figure 26: Error of training with smaller mini-batches

Although the results for one agent are slightly better than those of the previous configuration, the training process is considerably less reliable.

Finally, to show the importance of using sufficiently large mini-batches for computing high-quality gradients and thus improving the training process, an experiment was performed with a reduction of the mini-batch size to 60%, achieved by reducing the size of the partial trajectory dimension from 240 to 140, which leads to a significant deterioration of the training process, as shown in Figure 26.

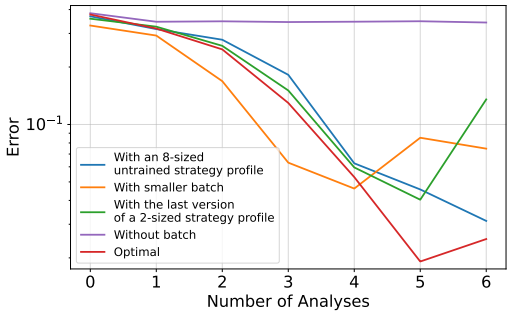

Figure 27: Median error over all instances for each training method configuration

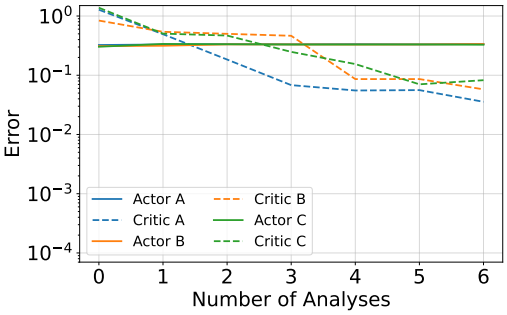

Figure 28: Error of training with high-entropy policies

A comparison of the five different configurations of the training process is shown in Figure 27, where the median error over all instances for each of the 5 training method configurations is plotted. The use of the median is intended to reduce the effect of outliers.

In the results of all five configurations of the training process, it can be observed that the error of the critic network barely decreases. As mentioned in the previous section, the value network learns overly optimistic estimates, since the actor-critic pairs are trained not only against the latest version of the agents but also against a large set of past versions, which are significantly weaker than the current version. These optimistic estimates create discrepancies with the expected values obtained from annotated scenarios provided by a human expert.

In contrast, when the entropy regularisation is set to a very high value, promoting exploration at the expense of policy optimisation, the value network is able to produce much more accurate estimates (cf. Figure 28), because the initial random policies are not substantially weaker than the trained policies, while still ensuring sufficient exploration.

8 Conclusion

This bachelor's thesis investigated war games on hexagonal grids as a domain for the development and evaluation of reinforcement learning algorithms. The goal was to explore the challenges that this class of games poses for machine learning

methods while designing a framework that enables the efficient training of neural network-based agents.

To this end, a reinforcement learning framework based on actor-critic methods with eligibility traces and fictitious self-play was developed. The reinforcement learning method was designed to use mini-batch gradient descent and to scale efficiently on a single machine with multiple CPU cores and a GPU. A main contribution of this thesis is the design of a novel convolutional neural network architecture specifically tailored for hexagonal grids, capable of exploiting symmetries that naturally occur in many hexagonal-grid games through the incorporation of rotational weight sharing. An architecture for the actor network was designed to address the specific challenge of action selection in war games, where multiple dependencies on the spatial configuration of the grid must be taken into account.

The experimental results demonstrate that the developed framework is capable of learning competitive strategies in a basic implementation of a war game on hexagonal grids. The experiments also highlighted the importance of the employed techniques, such as fictitious play and mini-batch gradient descent, for achieving good performance, even in a relatively simple game environment. These results support the idea that the proposed class of games presents significant challenges for machine learning methods that can motivate the development of new reinforcement learning techniques and serve as a benchmark for evaluating them.

Future work could extend the framework to more sophisticated war games featuring multiple units, different win conditions, and richer tactical interactions. In particular, modeling the spatial relationships among multiple units with respect to the unit to move is expected to be a significant challenge. Multiple units would also introduce non-homogeneous game steps, as several actions are performed within a single turn, making the problem of credit assignment more difficult. A more complex action space would also pose a challenge for exploration, especially in the absence of guidance from human-generated data.

Overall, the results suggest that war games on hexagonal grids are a promising domain for the development and evaluation of reinforcement learning techniques. The proposed framework demonstrates that mini-batch gradient descent, actor-critic methods, and fictitious self-play can be successfully combined to learn competitive strategies in this class of games. Furthermore, the developed hexagonal convolutional architecture provides a foundation for exploiting the unique structural properties of hexagonal grids in future research. The presented framework therefore serves as a basis for further investigations into more complex war games and advanced reinforcement learning methods.

References

[AC26a] Miguel Alvarez Caro. Hexagonal grid game: Ai project, 2026. Available: `https://github.com/MiguelAlvCar/bachelor-neuralNetwork` (Accessed: 2026-02-18).

[AC26b] Miguel Alvarez Caro. Hexagonal grid game: Angular frontend, 2026. Available: `https://github.com/MiguelAlvCar/bachelor-gameFrontend` (Accessed: 2026-02-18).

[AC26c] Miguel Alvarez Caro. Hexagonal grid game: game logic, 2026. Available: `https://github.com/MiguelAlvCar/bachelor-gameLogic` (Accessed: 2026-02-18).

[AC26d] Miguel Alvarez Caro. Hexagonal grid game: HTTP API, 2026. Available: `https://github.com/MiguelAlvCar/bachelor-gameApi` (Accessed: 2026-02-18).

[Agg23] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, 2023.

[Bro51] George W. Brown. Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 1951.

[ESM+18] Lasse Espeholt, Hubert Soyer, Remi Munos, et al. IMPALA: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

[Hin12] Geoffrey Hinton. Neural networks for machine learning, 2012. Coursera online course.

[HLS15] Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. *International conference on machine learning*, pages 805–813, 2015.

[HPCW18] Emiel Hoogeboom, Jorn W. T. Peters, Taco S. Cohen, and Max Welling. Hexaconv. *arXiv:1803.02108*, 2018.

[HS16] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Kuh53] Harold W Kuhn. Extensive games and the problem of information. *Contributions to the Theory of Games*, 2(28):193–216, 1953.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MP69] Marvin Minsky and Seymour Papert. *An Introduction to Computational Geometry*. The MIT Press, 1969.

[MPBM+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, et al. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[MS96] Dov Monderer and Lloyd S Shapley. Potential games. *Games and economic behavior*, 14(1):124–143, 1996.

[PSD25] Guilherme Palma, Pedro A. Santos, and João Dias. Playing hex and counter wargames using reinforcement learning and recurrent neural networks. *arXiv:2502.13918*, 2025.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[Rie05] Martin Riedmiller. Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method. In João Gama, Rui Camacho, Pavel B. Brazdil, Alípio Mário Jorge, and Luís Torgo, editors, *Machine Learning: ECML 2005*, pages 317–328, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Rob51] Julia Robinson. An iterative method of solving a game. *Annals of Mathematics*, 54(2):296–301, 1951.

[Ros58] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[SB18] Richard Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2 edition, 2018.

[SH19] Constantin Steppa and Tim L. Holch. HexagDLy—Processing hexagonally sampled data with cnns in pytorch. *SoftwareX*, 9:193–198, January 2019.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, et al. Mastering the game of Go with deep neural networks and tree search, 2016.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[SML+15] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, et al. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

[Tes94] Gerald Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play, 1994.

[TPK18] Arash Tavakoli, Fabio Pardo, and Petar Kormushev. Action branching architectures for deep reinforcement learning. *Proceedings of the AAAI conference on artificial intelligence*, 32(1), 2018.

[VBC+19] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning, 2019.

[VEB+17] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, et al. Starcraft II: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[VOS+17] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, et al. FeUdal networks for hierarchical reinforcement learning. In *International conference on machine learning*, pages 3540–3549. PMLR, 2017.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.