

# Deep Learning for General Game Playing with Ludii and Polygames

Dennis J. N. J. Soemers\*, Vegard Mella\*\*,  
Cameron Browne\*, Olivier Teytaud\*\*

January 26, 2021

## Abstract

Combinations of Monte-Carlo tree search and Deep Neural Networks, trained through self-play, have produced state-of-the-art results for automated game-playing in many board games. The training and search algorithms are not game-specific, but every individual game that these approaches are applied to still requires domain knowledge for the implementation of the game’s rules, and constructing the neural network’s architecture – in particular the shapes of its input and output tensors. Ludii is a general game system that already contains over 500 different games, which can rapidly grow thanks to its powerful and user-friendly game description language. Polygames is a framework with training and search algorithms, which has already produced superhuman players for several board games. This paper describes the implementation of a bridge between Ludii and Polygames, which enables Polygames to train and evaluate models for games that are implemented and run through Ludii. We do not require any game-specific domain knowledge anymore, and instead leverage our domain knowledge of the Ludii system and its abstract state and move representations to write functions that can automatically determine the appropriate shapes for input and output tensors for any game implemented in Ludii. We describe experimental results for short training runs in a wide variety of different board games, and discuss several open problems and avenues for future research.<sup>1</sup>

## 1 Introduction

Self-play training approaches such as those popularised by *AlphaGo Zero* [27] and *AlphaZero* [26], based on combinations of Monte-Carlo tree search (MCTS) [10, 6, 3] and Deep Learning [13], have been demonstrated to be fairly generally applicable, and achieved state-of-the-art results in a variety of board games such as Go [27], Chess, Shogi [26], Hex, and Havannah [5]. These approaches require relatively little domain knowledge, but still require some in the form of:

---

<sup>1</sup>\*Department of Data Science and Knowledge Engineering, Maastricht University, the Netherlands.\*\*Facebook AI Research.

1. A complete implementation of a forward model for the game, for the implementation of lookahead search as well as automated self-play to generate experience for training.
2. Knowledge of which state features are required or useful to provide as inputs for a neural network.
3. Knowledge of the action space, which is typically used to construct the policy head in such a way that every distinct possible action has a unique logit.

The first requirement, for the implementation of a forward model, is partially addressed by research on using learned simulators for tree search as in *MuZero* [24], but in practice a simulator is actually still required for the purpose of generating trajectories outside of the tree search. For the board games Go, Chess, and Shogi, MuZero still requires the input and output tensor shapes (for states and actions, respectively) to be manually designed per game. We remark that MuZero was also evaluated on 57 different Atari games in the Arcade Learning Environment (ALE) [1], and it can use identical tensor shapes across all these Atari games because ALE uses the same observation and action spaces for all games in this framework.

The challenge posed by General Game Playing (GGP) [21] is to build systems that can play a wide variety of games, which makes the three forms of required domain knowledge listed above difficult. A number of systems have been proposed that can interpret and run any arbitrary game as long as it has been described in their respective game description language, such as the original *Game Description Language* (GDL) [16] from Stanford, *Regular Boardgames* (RBG) [11], and *Ludii* [20].

In this paper, we describe how we combine the GGP system Ludii and the PyTorch-based [17] state-of-the-art training algorithms in Polygames [5], with the goal of mitigating all three of the requirements for domain knowledge listed above. Section 2 provides some background information on these training techniques. Section 3 describes existing work and limitations in applying these Deep Learning approaches to general games. Section 4 presents the interface between Ludii and Polygames. Experiments and results are described in Section 5. We discuss some open problems in Section 6, and conclude the paper in Section 7.

## 2 Background

The basic premise behind AlphaZero and similar approaches in frameworks such as Polygames is that Deep Neural Networks (DNNs) take representations  $T(s)$  of game states  $s$  as input, and produce discrete probability distributions  $\mathbf{P}(s)$  with probabilities  $P(s, a)$  for all actions  $a$  in states  $s$ , as well as value estimates  $V(s)$ , as outputs. This is depicted in Figure 1. Both of these outputs are used to guide MCTS in different ways.

DNNs in general have a fixed architecture, requiring fixed and predetermined shapes for both the input and the output representations. The value output is

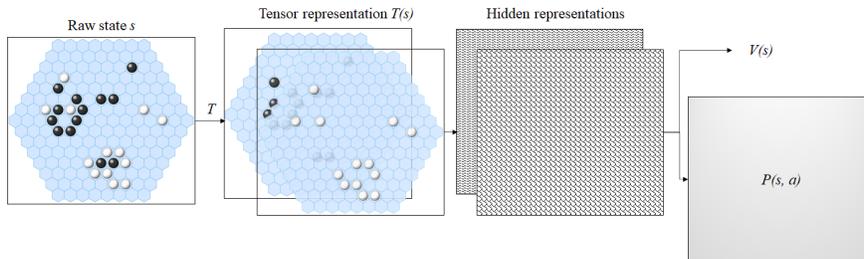


Figure 1: Basic architecture of DNNs for game playing. Raw game states  $s$  are transformed into a tensor representation  $T(s)$  of some fixed shape (often 3-dimensional). The DNN learns to compute hidden representations of its inputs in hidden layers. Finally, it computes a scalar value estimate  $V(s)$ , and a discrete probability distribution  $\mathbf{P}(s)$  with probabilities  $P(s, a)$  for all actions  $a$  in the complete action space.

always simply a scalar,<sup>2</sup> but determining the shapes of the input tensors  $T(s)$  and policy outputs  $\mathbf{P}(s)$  typically requires game-specific domain knowledge.

$T(s)$  is generally a 3-dimensional tensor, where 2 dimensions are spatial dimensions (corresponding to e.g. a 2-dimensional playable area in a board game). The third dimension is formed by a stack of different channels which each have different semantics. For example,  $T(s)$  in AlphaZero has a shape of  $19 \times 19 \times 17$  for the game of Go played on a  $19 \times 19$  board, with eight times two binary channels encoding the presence of the two players' pieces – for a history of up to eight successive game states ending in  $s$  – and one final channel encoding the current player to move. The spatial structure of the first two dimensions is typically assumed to be meaningful, which is exploited by the inductive bias of Convolutional Neural Networks (CNNs) [14].

For the policy head, it is customary for neural networks to first output real-valued *logits*  $L(s, a)$  for all possible actions  $a$ . These are subsequently converted into probabilities  $P(s, a)$  using a softmax over all legal actions  $a' \in \mathcal{A}(s)$  in  $s$ :

$$P(s, a) = \frac{\exp(L(s, a))}{\sum_{a' \in \mathcal{A}(s)} \exp(L(s, a'))}.$$

It is generally assumed that every distinct possible action  $a$  that may be legal in any game state  $s$  has a unique, matching logit  $L(s, a)$ . This means that domain knowledge of the game's action space is required to construct a DNN's architecture in such a way that distinct actions always have distinct logits. The logits are sometimes laid out in a structure of multiple 2-dimensional planes, like the inputs, but typically preceded by fully connected (as opposed to convolutional) layers. This is equivalent to all the logits being laid out in a single, flat vector with no spatial structure.

<sup>2</sup>Assuming 2-player zero-sum games; see [18] for relaxations of this assumption.

In addition to such typical architectures, Polygames [5] includes various different structures, such as:

- **Fully convolutional networks:** As in many cases, actions are spatially distributed in a manner somehow close to the pieces, the output has spatial coordinates matching the spatial coordinates of the input. This can be exploited in fully convolutional networks [25]: the policy head has no fully connected layer, and directly maps inputs to outputs through convolutional blocks. This has the advantage of being boardsize invariant: we can train in size  $13 \times 13$  and play in  $19 \times 19$ . Global pooling can be used to additionally make the value head size-invariant [15, 29].
- **U-networks:** It is usually considered that DNNs rephrase their data in an increasingly abstract manner, layer after layer. However, in fully convolutional networks, the output is dense; it has the same low-level nature as the input. The level of abstraction increases, and then decreases again. Then, one may consider that layers might benefit from a direct connection into a layer containing information at the same level of abstraction. This can be done by skip-connections, i.e. additional connections to layers symmetrically positioned in the network (Figure 2c): this is a U-network [22].

Some of these different structures are depicted and explained in Figure 2.

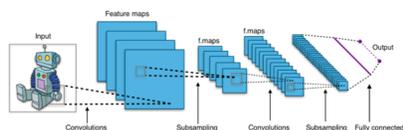
### 3 Deep Learning in General Game Playing

To some extent, all GGP systems mitigate the requirement for the implementation of complete forward models for every distinct game, in the sense that new games can be added and supported simply by defining them in a game description language. Ludii’s game description language in particular has been designed in such a way that game descriptions for new games are fast and easy to write and understand [20], which has allowed for a significantly larger library of distinct games<sup>3</sup> to be built up than would be feasible if they were all written in a programming language such as C++. Ludii’s predecessor has also already demonstrated that the “ludemic” approach to game description languages used by Ludii facilitates procedural generation of complete games [4], which can be used to easily extend the set of compatible benchmark problems.

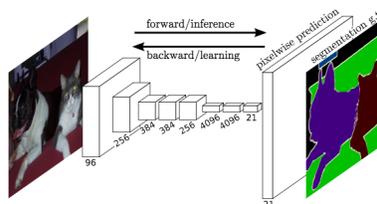
Similarly, we may argue that running games through a GGP system removes the requirements for *game-specific knowledge* about how to shape state inputs and action outputs, but introduces requirements for similar *knowledge about the GGP system*. Given any arbitrary game defined in a game description language of a GGP system, we require the ability to construct tensor representations of game states, and the ability to map from any index in a policy head to a matching action in any non-terminal game state.

---

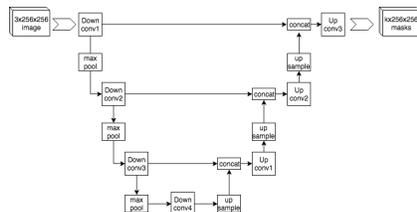
<sup>3</sup>Ludii has over 500 distinct built-in games at the time of this writing, with many of them having multiple variants for different board sizes, board shapes, variant rulesets, etc.



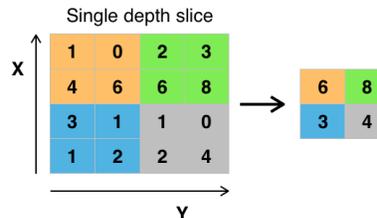
(a) Standard convolutional net: convolutional layers first, with their inductive bias towards spatial invariance, followed by fully connected layers. (source: [https://en.wikipedia.org/wiki/Convolutional\\_neural\\_network#/media/File:Typical\\_cnn.png](https://en.wikipedia.org/wiki/Convolutional_neural_network#/media/File:Typical_cnn.png))



(b) Fully convolutional net, e.g. for image segmentation: each output scalar (each logit in the case of games) is the output of the same net, applied on a moving window. No fully connected layers.



(c) U-networks: this fully convolutional network also connects some layers to their symmetric counterpart supposed to work at a similar level of abstraction.



(d) Max pooling: here we downsize from 4x4 to 2x2. In case of global pooling, the size of tensors after the global pooling layer is  $1 \times 1 \times \#channels$ , independently of the input size.

Figure 2: Convolutional neural network (a). Fully convolutional counterpart (b, image from [25]; other images from Wikipedia), typically used in Image segmentation: image segmentation is related to policy heads in games in that the output has the same spatial coordinates at the input. U-networks (c): only convolutional layers, and skip connections symmetrically connecting layers. Global pooling (d): here we down-sample to a spatial size 1x1 in the value head: this is boardsize invariant. Global pooling can use channels for mean, standard deviation, max, etc: the number of channels is not necessarily preserved. (b+d) or (c+d) allow boardsize-invariant training [5].

GDL [16] is a low-level logic-based game description language, where games are described as logic programs consisting of many low-level propositions. Many GDL-based agents convert such a GDL description into a propositional network [23, 7, 28], which can more efficiently process the games than Prolog-based reasoners or other similar techniques. Such propositional networks can be automatically constructed from GDL descriptions, and the structure of such a network remains constant across all game states of the same game. [9] therefore proposed using the internal state of a game’s propositional network as the input state tensor for a deep neural network. A downside of this approach is that the state input tensor is a flat tensor, and there is no possibility to use inductive biases such as those of CNNs for inputs with spatial semantics. *Galvanise Zero* [8] does exploit knowledge of spatial semantics through CNNs, but it only supports a limited selection of GDL-based games because it requires a handwritten Python function to create the mapping from game states to input tensors for every game that it supports. The action space can automatically be inferred from GDL descriptions, which means that these approaches require no extra domain knowledge with respect to the output policy heads.

In the game description language of Ludii [20], common high-level game concepts such as boards, piece types, etc. are all “first-class citizen” of the language, as opposed to GDL where every separate game description file encodes such concepts from scratch in low-level logic. Based on these concepts, Ludii also has an object-oriented game state representation that it uses internally, which remains consistent across all games. This enables us to write a single function that automatically constructs input tensors from Ludii’s internal state representation, using our domain knowledge of Ludii as a whole instead of domain knowledge of every individual game. Unlike GDL, it is not straightforward (if at all possible) to infer the action space from game description files in Ludii. However, actions in Ludii do have an object-oriented structure, and at least an approximation of the action space can be constructed based on these properties – again, based on domain knowledge of Ludii rather than any individual game. In many games, this is sufficient to distinguish most or all legal actions from each other.

## 4 Interface Between Ludii and Polygames

Based on the insights described above, we developed an interface between the Ludii general game system, and the Polygames framework with state-of-the-art AI training code. In Polygames, different games are normally implemented from scratch in C++. The basic idea of this interface is that there is a single “Ludii game” in Polygames, with C++ code that interacts with Ludii’s Java-based API through Java Native Interface. Polygames command-line arguments can be used to load different games and variants from Ludii into this wrapper. This section provides details on how Ludii automatically constructs tensor representations of its state and action spaces, based on its own internal representations, for any

arbitrary game implemented in Ludii.<sup>4</sup>

## 4.1 Constructing the Spatial Dimensions

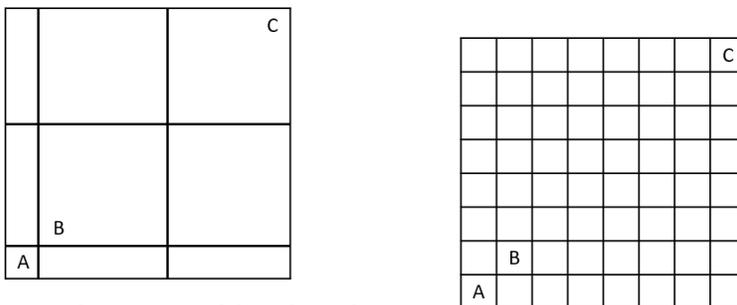
CNNs normally operate on grid structures of “pixels”, such that every position can be indexed by a row and column, and every position has a square of up to eight neighbour positions around it. This structure resembles the game boards of games such as Chess, Shogi, and Go most closely. Some other boards, such as the tilings of hexagonal cells used in games like Hex and Havannah, can also be “packed” into such a grid. This approach is used for the game-specific C++ implementations of those games in Polygames. However, Ludii supports games with arbitrary graphs as boards, and hence requires a generic solution that can map positions from graphs with any arbitrary connectivity structure into a grid structure that CNNs can work with.

For every game in Ludii, there is at least one (and possibly more than one) *container*, which specifies a playable “area” with positions that may contain pieces, have corresponding clickable elements in Ludii’s GUI, etc. [20, 19]. The first container typically corresponds to the board that a game is played on, and is often the largest. Any other containers represent auxiliary areas, such as players’ hands to hold captured pieces in Shogi. Even games that are not generally thought of as being played on a board are still modelled in this way in Ludii. For instance, Rock-Paper-Scissors is modelled as a board with two (initially empty) cells, and two hands for the two players, each containing rock, paper, and scissors “pieces” which players can drag onto their designated cells on the board to make their move.

Every site in any such container in Ludii has  $x$  and  $y$  coordinates in  $[0, 1]$ , which are used by Ludii for purposes such as drawing game states in the GUI for human players. We construct a grid structure simply by sorting all the distinct  $x$ - and  $y$ -coordinates across all sites in the board in increasing order, and assigning distinct columns and rows, respectively, to distinct  $x$ - and  $y$ -coordinates. Coordinates that are within a tolerance value of  $10^{-5}$  are treated as equal, to avoid generating excessively large and sparse tensors due to small differences resulting from floating-point arithmetic. Note that this approach is not equivalent to directly overlaying a sufficiently fine-grained grid over the  $[0, 1]^2$  space, because we only add rows and columns that each contain at least one site. This is depicted in Figure 3. Our approach may lose some information concerning the relative distances between sites, but because these  $x$ - and  $y$ -coordinates are only used for the graphical user interface – not for game logic – we expect the smaller and less sparse grids to be preferable due to improved computational efficiency. Note that the vast majority of games in Ludii use boards defined by regular or semiregular tilings, and for these the two approaches will have similar results.

---

<sup>4</sup>The source code for building tensors from Ludii’s internal state and action representations is available from <https://github.com/Ludeme/LudiiAI>. All the source code of Polygames is available from <https://github.com/facebookincubator/Polygames>.



(a) A  $3 \times 3$  grid is computed based on the distinct  $x$ - and  $y$ -coordinates of three sites A, B, and C. (b) An  $8 \times 8$  grid is overlaid on the  $[0, 1]^2$  space containing three sites A, B, and C.

Figure 3: Two different approaches for computing a grid based on a playable space defined by three sites A, B, and C, each with distinct  $x$ - and  $y$ -coordinates. The approach we use is depicted in (a). This approach results in smaller, more dense tensors, but information of the relative distances between all sites is not necessarily preserved. The alternative approach, depicted in (b), preserves more of this information, but can result in large and sparse tensors.

In the current version of Ludii, containers other than the first one (corresponding to the “main” board) never have more than one meaningful dimension; they are always a single, contiguous sequence of cells. Each of those containers is concatenated to the grid constructed for the first container, either using one extra column or one extra row per extra container (whichever results in the lowest increase in total size of the tensor). Additionally, one extra dummy row or column is inserted to create a more explicit separation between the main board (for which we expect there to be meaningful spatial semantics) and the other containers (for which there is no expectation that any meaningful spatial semantics exist). For example, Shogi is played on a  $9 \times 9$  board, but each of the two players also has a “hand” of 7 cells as extra containers to potentially hold captured pieces. This results in a  $12 \times 9$  grid for Shogi. A screenshot of Shogi being played in Ludii’s user interface is depicted in Figure 4, with cells of the different containers labelled by numbers. The mapping from these positions to positions in the tensor representation is depicted in Figure 5.

## 4.2 Representing Ludii Game States as Tensors

Let  $s$  denote a raw game state in Ludii’s object-oriented state representation [19], for a game  $\mathcal{G}$ . Based on the properties of  $s$ , we construct a tensor representation  $T(s)$  – which can be used as input for a DNN – of shape  $(C, W, H)$ , where  $C$  denotes the number of channels (variable, depends on  $\mathcal{G}$ ),  $W$  denotes the width (i.e., number of columns), and  $H$  denotes the height (i.e., number of rows). The channels are constructed as follows:

▲	73	▲	▲	▲	76	▲	78	▲	▲	▲
63	64	65	66	▲	68	▲	▲	▲	71	
▲	▲	▲	▲	▲	59	▲	▲	▲	▲	
▲	46	47	48	49	▲	51	52	53		
36	▲	38	39	40	41	42	43	44		
▲	28	▲	30	31	32	33	34	35		
18	19	20	▲	▲	▲	▲	▲	▲		
▲	10	11	▲	▲	14	15	16	17		
0	▲	▲	3	▲	▲	▲	▲	▲		

1 Player 1  
Human ▼ 82 83 84 85 86 87

2 Player 2  
Human ▼ 88 89 90 92 93 94

---

Status
Moves
Turns
Analysis
Ludeme
Rules
Info

Description:  
 Shogi is a capturing game that is native to Japan. Early evidence suggests it was played in some recognizable form during the Heian Period. It has been hugely popular since then in Japan, and today is known throughout the world. It is originated ultimately from Indian Chaturanga, like many other similar games.

Aliases:  
 [Sho-Shogi]

Origin:  
 This game was played in Japan, around 1058CE.

Classification:  
 board/war/shogi

Credit:  
 Eric Plette

Version:  
 1.1.1

Figure 4: Shogi being played in Ludii’s user interface. The game board is on the left-hand side, and each player has a “hand” with seven slots to hold captured pieces on the right-hand side. Figure 5 shows how the numbered positions get mapped to positions in a tensor.

72	73	74	75	76	77	78	79	80		81	88
63	64	65	66	67	68	69	70	71		82	89
54	55	56	57	58	59	60	61	62		83	90
45	46	47	48	49	50	51	52	53		84	91
36	37	38	39	40	41	42	43	44		85	92
27	28	29	30	31	32	33	34	35		86	93
18	19	20	21	22	23	24	25	26		87	94
9	10	11	12	13	14	15	16	17			
0	1	2	3	4	5	6	7	8			

Figure 5: Mapping from positions in Shogi’s three containers to positions in a single tensor. Numbers 0 through 80 correspond to positions on the board, 81 through 87 are positions in the hand of Player 1, and 88 through 94 are positions in the hand of Player 2 (see Figure 4).

- Binary channels indicating the presence (or absence) of every piece type defined in  $\mathcal{G}$ . Most games have one channel per piece type, where values of 1 indicate the presence of a piece of that type in a position. If  $\mathcal{G}$  is a “stacking” game, meaning that it allows for multiple pieces to form a stack on a single position, we use  $M + N$  binary channels per piece type, instead of just one.  $M$  channels are used to indicate presence of a piece type in the bottom  $M$  layers of every stack on every position, and  $N$  channels indicate the same for the top  $N$  layers. In our implementation, we use  $M = N = 5$ . If a single stack contains more than  $M + N$  pieces, this representation is not sufficient to provide information about some of the middle layers to the DNN, but this is rare in practice.
- If  $\mathcal{G}$  is a “stacking” game, we include an additional non-binary channel containing the height of every stack in every position.
- If  $\mathcal{G}$  is a game where positions can contain a “count” of more than one piece, we include a non-binary channel denoting the count of pieces on that position. This channel is semantically similar to the one described above for stack heights. In Ludii, positions in these games are still restricted to containing only a single piece *type* at a time. This is most notably used for *mancala* games. Games where pieces of different types can share a single position are modelled as stacking games instead.
- Ludii’s state representation can include an “amount” value per player, primarily intended to represent money for games that involve betting or other similar mechanisms. If  $\mathcal{G}$  uses this, we add one non-binary channel per player, such that every position in the channel for player  $p$  contains the amount value of  $p$  in  $s$ .
- If  $\mathcal{G}$  is played by  $n > 1$  players, we include  $n$  binary channels, such that the  $n^{th}$  channel is filled with values of 1 if and only if  $n$  is the current player to move in state  $s$ . This also accounts for swap rules. For example, the first player normally plays red, and the second blue, in *Hex*. If  $s$  is a state where the red player is the next to make a move, and a swap has occurred, the second of these channels will be filled with 1 entries instead of the first.
- In some games, every position has a “local state” variable, which is an integer value. Different games can use this in different ways to store (temporary) auxiliary information about positions. For instance, local state values of 1 are used for positions that contain pieces that are still in their initial position, and values of 0 otherwise (this is used for castling). Most games only use low local state values, if any at all. Hence, we use separate binary channels to indicate local state values of 0, 1, 2, 3, 4, and  $\geq 5$ .
- If the game uses a swap rule (or “pie rule”), such as Hex, we include a binary channel that is filled with values of 1 if and only if a swap has occurred in  $s$ .

- For every distinct container in  $\mathcal{G}$ , we include one binary channel that has values of 1 for entries that correspond to a position in that container, and values of 0 everywhere else.
- For each of the last two moves  $m$  played prior to reaching  $s$ , we add one binary channel with only a single value of 1 in the entry corresponding to the “from” position of  $m$  (typically the location that a piece moves away from), and a similar channel for the “to” position of  $m$  (typically the location that a piece is placed in).

With these channels, we did not yet exhaustively cover all the state variables in Ludii’s game state representation [19], but we covered the most commonly-used ones. Whenever new variables are added to Ludii’s game state representation, engineering effort for including these in the tensor representations is only required once for Ludii as a whole – not once per game added to Ludii.

### 4.3 Representing Ludii Actions as Tensors

In contrast to GDL [16, 8, 9], it is not straightforward – if at all possible – to automatically infer the complete action space for any arbitrary game described in Ludii’s game description language. This is because in Ludii’s game description language, the function that generates lists of legal moves is defined as a composite of many simple functions (*ludemes*), which may be arranged in any arbitrary tree structure. While each of these ludemes in principle has some domain for its possible inputs, and range for its possible outputs, these are not strictly defined in logic-based or other formats that permit automated inference.

Similar to its state representation, Ludii has an object-oriented move<sup>5</sup> representation [19]. However, in contrast to the state representation, the most important variables of the move representation are arbitrarily-sized lists (of primitive modifications to be applied to a game state) and arbitrarily-sized trees (of ludemes to be evaluated after applying the initial primitive modifications). The arbitrary sizes of these variables make them difficult to encode in a fixed-size tensor representation. Hence, we ignore these properties, and only distinguish moves based on some simple properties that can easily be used for this purpose. We construct the space of output tensors to map moves to for a game  $\mathcal{G}$  as follows:

- The action space is organised as a stack of 2-dimensional planes, with the spatial dimensions being identical to those of the state tensors (see Subsection 4.2). Every action will map to exactly one position in this space – i.e., one location in the 2-dimensional area, and one channel.
- Pass and swap moves have been identified as special cases that are sufficiently common, important, and semantically different from any other

---

<sup>5</sup>In this document we use the terms “move” and “action” interchangeably, to refer to complete decisions that players make. Within Ludii, these are referred to only as moves, and actions are smaller parts of moves.

kind of move that they warrant the inclusion of their own dedicated channels.

- Many games only involve moves that can be identified by just a single position in the spatial dimensions; these are generally games where players place stones (Go, Hex, Havannah, etc.), but may in theory also be games like Chess if they have been defined in a way such that movements are split up into two separate decisions (picking a source and picking a destination). These games can be automatically discovered in Ludii. For these games, we only add one more channel in addition to the pass and swap move channels, to encode all other moves based on their positions in the spatial dimensions. In Ludii, this position is referred to as the “to” position.
- In all other games, moves may have distinct “from” and “to” positions; typical examples are standard implementations of Chess, Amazons, Shogi, etc. For moves that have an invalid “from” position, we assume that it is equal to the “to” position. For games that involve stacking, moves may additionally have  $l_{min}$  and  $l_{max}$  properties which refer to the levels within a stack at which a move operates; both are assumed to equal 0 if the game does not allow stacking. The “to” position of a move is used to map the move to a location in the spatial dimensions, and the remaining properties are used to index into one of multiple channels based on the relative “distance covered” by the move. More specifically, we create  $(2M+1) \times (2M+1) \times (N+1) \times (N+1)$  channels, where we use  $M = 3$ , and  $N = 2$  if  $\mathcal{G}$  involves stacking, or  $N = 0$  otherwise. Let  $dx$  and  $dy$  denote the differences in rows and columns, respectively, between the “to” and “from” positions of a move. Let  $[a]_b^c$  denote a value of  $a$  clipped to lie in the interval  $[b, c]$ . Then, this move gets mapped to the channel given by the 0-based index  $(([dx]_{-M}^M \times (2M+1) + [dy]_{-M}^M) \times (N+1) + [l_{min}]_0^N) \times (N+1) + [l_{max} - l_{min}]_0^N$ .

Note that this is simply one approach to constructing tensor representations of moves that we implemented, but we may envision other approaches as well. For instance, in a game like Chess, it may be more important to encode the type of the piece that makes a move, rather than encoding the distance and direction covered by a move. This could be accomplished by creating channels that are indexed based on the type of piece in the “from” location of a move, instead of the distance between “from” and “to” positions.

While we find this approach to be sufficient to distinguish moves from each other in many cases, there are cases where multiple distinct moves that are legal in a single game state will end up being represented by exactly the same logit. When multiple distinct moves are represented by the same logit in a DNN’s output, we say that they are *aliased*. DNNs cannot distinguish between aliased moves, and hence always provide the same advice (in the form of the prior probabilities  $P(s, a)$ ) to MCTS for these different moves. However, in Polygames [5], the MCTS itself *can* still distinguish between the different moves by different representing them as distinct branches in the search tree, and backing up (po-

tentially) different values throughout the tree search. This is an important difference with other frameworks, such as OpenSpiel [12], where the MCTS itself requires every possible distinct action that may ever be legal in a game to be assigned a unique integer upfront. When subsequently using the visit counts to compute the standard cross-entropy loss as proposed by [27], the visit counts for all moves that share a single logit are summed up. The softmax over the logits only counts every distinct logit once.

## 5 Experiments

In this section we describe experiments<sup>6</sup> intended to demonstrate the potential for the approach described in the previous section to facilitate training and research in general games. We picked fifteen different games, all as implemented with their default options in Ludii [20] v1.1.6, and trained a model of the `ResConvConvLogitPoolModelV2` type from Polygames [5] in each of these games. The selected games are depicted in Figure 6.

We used the same training hyperparameters across all games. Every training run used 20 hours of wall time, with 8 GPUs, 80 CPU cores, and 475GB memory allocated per training job. Every training job used 1 server for model training, and 7 clients for the generation of self-play games. The MCTS agents used 400 MCTS iterations per move in self-play.

The final model checkpoint of every training run is evaluated in a set of 300 evaluation games played against a pure MCTS – a standard UCT agent [10, 3] without any DNNs. In evaluation games, the MCTS with a trained model used 40 iterations per move, whereas the pure MCTS used 800 iterations per move – where at the end of every iteration, the average outcome of 10 random rollouts is backed up. The final column of Table 1 reports the win percentages of the trained MCTS against the untrained MCTS. The table also provides further details on the number of trainable parameters in each of the DNNs, and for some games summarises unusual properties that these games have which we did not yet observe in much of the existing literature on learning through self-play in games.

In the majority of the evaluated games, the trained MCTS easily outperforms the untrained one, even using 20 times fewer MCTS iterations (or 200 times fewer if the number of random rollouts performed by the untrained MCTS is counted). Note that, in comparison to work that focuses on achieving super-human playing strength [26, 5], we focused on short training runs using fewer resources and smaller networks. Our primary aim is to demonstrate the possibility of training effectively using a single implementation without game-specific domain knowledge.

---

<sup>6</sup>The code used by Ludii to construct state and move tensors for any game is available from <https://github.com/Ludeme/LudiiAI>. All the training and evaluation code of Polygames is available from <https://github.com/facebookincubator/Polygames>. Checkpoints of models used in these experiments are available from [http://dl.fbaipublicfiles.com/polygames/ludii\\_checkpoints/list.txt](http://dl.fbaipublicfiles.com/polygames/ludii_checkpoints/list.txt).

Table 1: Data for a variety of different games, all implemented in Ludii v1.1.6, for which we trained models in Polygames over a duration of 20 hours using 8 GPUs and 80 CPU cores per model. The second column lists some interesting properties for games that we have not yet often seen (if at all) in existing literature using AlphaZero-like training approaches. The third column lists the number of trainable parameters in the model (we used identical Polygames hyperparameters for the DNN architecture across all games, but in Polygames by default the number of channels in hidden convolutional layers scales with the number of input channels). The last column lists the win percentages of MCTS with the trained model using 40 iterations per move, against MCTS without any trained model using 800 iterations per move – where every iteration backs up the average outcome of 10 random rollouts.

Game	Unusual Properties	Trainable Parameters	Win Percentage
Breakthrough	-	188,296	100.00%
Connect6	-	180,472	75.67%
Dai Hasami Shogi	-	188,296	99.33%
Fanorona	Move aliasing due to choice of capture direction.	188,296	50.00%
Feed the Ducks	Moves have global effects across entire board.	231,152	83.00%
Gomoku	-	180,472	91.00%
Hex	-	222,464	100.00%
HeXentafl	Asymmetry in piece types, initial setup, and goals.	231,152	98.67%
Konane	-	188,296	98.00%
Lasca	Pieces (of multiple different types) can stack.	5,450,268	3.50%
Minishogi	-	2,009,752	97.00%
Pentalath	-	180,472	95.33%
Squava	Lines of 4 win, but lines of 3 lose.	222,464	96.67%
Surakarta	Loops around board allow for unique move patterns.	188,948	100.00%
Yavalath	Lines of 4 win, but lines of 3 lose.	222,464	97.33%

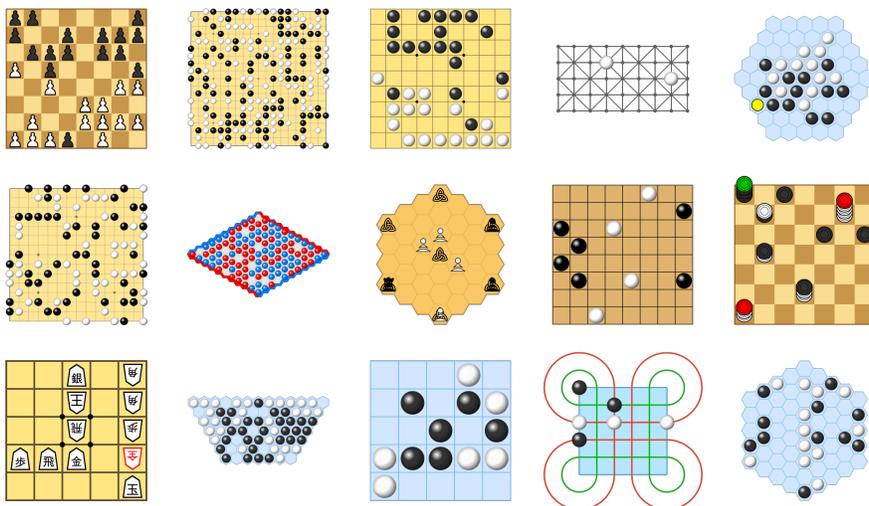


Figure 6: Screenshots of all the Ludii-based games included in our experiments. **First row:** Breakthrough, Connect6, Dai Hasami Shogi, Fanorona, Feed the Ducks. **Second row:** Gomoku, Hex, HeXentaffl, Konane, Lasca. **Third row:** Minishogi, Pentalath, Squava, Surakarta, Yavalath.

The two results that stand out most are for *Lasca* and *Fanorona*. The win percentage of 3.50% for *Lasca* indicates that this model is not trained nearly as well as the others. *Lasca* is the only game among those tested that involves stacking of multiple pieces on a single site. Our procedures for the construction of input and output tensors lead to a significantly larger numbers of channels in this game compared to the other games, which is also reflected in the large number of trainable parameters that this model has. Further research is required to establish whether it would be sufficient to reduce the size of the model, or whether entirely different approaches for constructing the tensors would be more appropriate. In *Fanorona*, the win percentage of 50% for the trained model is not necessarily a poor level of performance (considering the large difference in number of MCTS iterations), but it appears to be noticeably worse than in the other games. One possible explanation for this may be that *Fanorona* has a more severe degree of move aliasing, because there are situations where there are multiple different legal moves with identical “to” and “from” positions, but different effects in that a player can choose in which direction they wish to capture opposing pieces. Such moves are all represented by a single, shared logit in our output tensors – which means that only the MCTS can distinguish between them, but the trained policy head cannot.

## 6 Open Problems

Thanks to the large library of games available in Ludii [20], we can get a clear picture of categories of games that are open problems to various extents; some that are simply not supported yet by Polygames [5] and require more engineering effort, and some that appear to have been neglected across the majority of recent game AI literature. All of these types of games are supported by Ludii:

- *Stochastic games*: these were not included in this paper because they are temporarily unsupported by the MCTS implementation of Polygames, but were supported in earlier versions of Polygames and will be again in future versions.
- *Games with more than 2 players*: support for these can be added relatively easily [18], but is not yet available in Polygames.
- *Imperfect-information games*: there has been some recent work towards AlphaZero-like training approaches that support imperfect-information games [2], but tractability is still a concern for games with little common knowledge.
- *Simultaneous-move games*: simultaneous-move games will at least require significant changes in the MCTS component [3] as it is typically used in AlphaZero-like training setups.
- *Games with excessively large state or move tensors*: games such as *Taikyoku Shogi*, with a  $36 \times 36$  board and 402 pieces per player of 209 different types, can be modelled and run in Ludii, but produce excessively large tensors which quickly lead to memory issues when training with standard hyperparameter values that work well for “normal” games. These issues do not appear straightforward to resolve with current hardware and large DNNs.
- *Games played on a mix of cells, edges and/or vertices of graphs*: while games like *Chess* are only played on cells, and games like *Go* only on vertices, there are also games such as *Contagion* that are played on a mix of multiple different parts of a graph. It is not clear how to directly support these with the standard CNNs.
- *Games without an explicitly defined board*: games such as *Andantino* or *Chex* are not played in a limited area that is defined upfront, but in a playable area that grows dynamically as play progresses. The standard DNN architectures require these spatial dimensions to be predefined and fixed.
- *Games with more than 2 spatial dimensions*: games such as *Spline* have a third spatial dimension, which cannot be handled by the standard 2D convolutional layers. While a straightforward extension to 3D convolutional layers may be sufficient, we are not aware of any existing research towards this for games, and also imagine that a third spatial dimension

can rapidly lead to tensors becoming excessively large again for many non-trivial games.

## 7 Conclusions

We have described our approach for constructing tensor representations of states and moves for any game implemented in the Ludii general game system, and used this to implement a bridge between Ludii and the Polygames framework. This allows for the state-of-the-art tree search and self-play training techniques implemented in Polygames to be used for training game-playing models in any game described in Ludii’s general game description language. Whereas AlphaZero-like approaches typically require game-specific domain knowledge to define a Deep Neural Network’s architecture and its input and output tensors, we only require such domain knowledge at the level of the general game system as a whole, and can now leverage Ludii’s wide library of games – which can quickly grow thanks to its user-friendly game description language – to facilitate more general game AI research with minimal requirements for game-specific engineering efforts. We have identified a series of “open problems” in the form of classes of games that are already supported by Ludii, but not yet by Polygames. For some of these is a clear path that merely requires additional engineering effort, but others are likely to require a more significant amount of extra research.

## Acknowledgments

This work was partially supported by the European Research Council as part of the Digital Ludeme Project (ERC Consolidator Grant #771292), led by Cameron Browne at Maastricht University’s Department of Data Science and Knowledge Engineering. We thank Éric Piette for his image editing skills, and Matthew Stephenson for his mastery of the English language.

## References

- [1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.
- [2] N. Brown, A. Bakhtin, A. Lerer, and Q. Gong. Combining deep reinforcement learning and search for imperfect-information games. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, 2020.
- [3] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of

- Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–49, 2012.
- [4] C. B. Browne. *Automatic Generation and Evaluation of Recombination Games*. PhD thesis, Queensland University of Technology, 2009.
- [5] T. Cazenave, Y.-C. Chen, G. Chen, S.-Y. Chen, X.-D. Chiu, J. Dehos, M. Elsa, Q. Gong, H. Hu, V. Khalidov, C.-L. Li, H.-I. Lin, Y.-J. Lin, X. Martinet, V. Mella, J. Rapin, B. Roziere, G. Synnaeve, F. Teytaud, O. Teytaud, S.-C. Ye, Y.-J. Ye, S.-J. Yen, and S. Zagoruyko. Polygames: Improved zero learning. *ICGA Journal*, 2020. To appear.
- [6] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games*, volume 4630 of *LNCS*, pages 72–83. Springer Berlin Heidelberg, 2007.
- [7] E. Cox, E. Schkufza, R. Madsen, and M. R. Genesereth. In *Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)*, pages 13–20, 2009.
- [8] R. Emslie. Galvanise zero. [https://github.com/richemsie/galvanise\\_zero](https://github.com/richemsie/galvanise_zero), 2019.
- [9] A. Goldwaser and M. Thielscher. Deep reinforcement learning for general game playing. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence*, pages 1701–1708. AAAI Press, 2020.
- [10] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *LNCS*, pages 282–293. Springer, Berlin, Heidelberg, 2006.
- [11] J. Kowalski, M. Maksymilian, J. Sutowicz, and M. Szykuła. Regular boardgames. In *The Thirty-Third AAAI Conference on Artificial Intelligence*, pages 1699–1706. AAAI Press, 2019.
- [12] M. Lanctot, E. Lockhart, J.-B. Lespiau, V. Zambaldi, S. Upadhyay, J. Pérolat, S. Srinivasan, F. Timbers, K. Tuyls, S. Omidshafiei, D. Hennes, D. Morrill, P. Muller, T. Ewalds, R. Faulkner, J. Kramár, B. de Vylder, B. Saeta, J. Bradbury, D. Ding, S. Borgeaud, M. Lai, J. Schrittwieser, T. Anthony, E. Hughes, I. Danihelka, and J. Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games. <http://arxiv.org/abs/1908.09453>, 2019.
- [13] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

- [14] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.
- [15] M. Lin, Q. Chen, and S. Yan. Network in network, 2014.
- [16] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing: Game description language specification, 2008.
- [17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [18] N. Petosa and T. Balch. Multiplayer alphazero. In *Workshop on Deep Reinforcement Learning at the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, 2019.
- [19] É. Piette, C. Browne, and D. J. N. J. Soemers. Ludii game logic guide. *CoRR*, abs/2101.02120, 2021.
- [20] É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne. Ludii – the ludemic general game system. In G. D. Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, editors, *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 411–418. IOS Press, 2020.
- [21] J. Pitrat. Realization of a general game-playing program. In A. J. H. Morrel, editor, *Information Processing, Proceedings of IFIP Congress 1968, Edinburgh, UK, 5-10 August 1968, Volume 2 - Hardware, Applications*, pages 1570–1574, 1968.
- [22] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015.
- [23] E. Schkufza, N. Love, and M. Genesereth. Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In W. Wobcke and M. Zhang, editors, *AI 2008: Advances in Artificial Intelligence*, volume 5360 of *LNCS*, pages 56–66. Springer, Berlin, Heidelberg, 2008.

- [24] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588:604–609, 2020.
- [25] E. Shelhamer, J. Long, and T. Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):640–651, 2017.
- [26] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.
- [27] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, 2017.
- [28] C. F. Sironi and M. H. M. Winands. Optimizing propositional networks. In *Computer Games*, pages 133–151. Springer, 2017.
- [29] D. J. Wu. Accelerating self-play learning in go. *CoRR*, abs/1902.10565, 2019.