Deep Reinforcement Learning in Strategic Board Game Environments

Konstantia Xenou[1], Georgios Chalkiadakis[1], and Stergos Afantenos[2]

[1] School of Electrical and Computer Engineering, Technical University of Crete
`diaxenou@intelligence.tuc.gr, gehalk@intelligence.tuc.gr`
[2] Institut de recherche en informatique de Toulouse (IRIT), Université Paul Sabatier
`stergos.afantenos@irit.fr`

Abstract. In this paper we propose a novel Deep Reinforcement Learning (DRL) algorithm that uses the concept of "action-dependent state features", and exploits it to approximate the Q-values locally, employing a deep neural network with parallel Long Short Term Memory (LSTM) components, each one responsible for computing an action-related Q-value. As such, all computations occur simultaneously, and there is no need to employ "target" networks and experience replay, which are techniques regularly used in the DRL literature. Moreover, our algorithm does not require previous training experiences, but trains itself online during game play. We tested our approach in the Settlers Of Catan multi-player strategic board game. Our results confirm the effectiveness of our approach, since it outperforms several competitors, including the state-of-the-art *jSettler* heuristic algorithm devised for this particular domain.

Keywords: Deep Reinforcement Learning · Strategic Board Games

1 Introduction

Deep Reinforcement Learning (or DRL) is widely used in different fields nowadays, such as robotics and natural language processing [23]. Games, in particular, are a very popular testbed for testing DRL algorithms. Methods like Deep Q-Networks were found to be especially successful for video games, where one can learn using video frames and the instant reward.

Research on RL in strategic board games is particularly interesting, because their complexity can be compared to real-life tasks and their testbed allows comparison of many different players as well as AI techniques. The most known example of Deep RL use in this domain is perhaps AlphaGo [24], but other attempts have been made as well in games like chess and backgammon [15, 7].

Now, the popular board game "Settlers Of Catan" (SoC), has recently been used as a framework for machine learning and sequential decision making algorithms [28, 4]. Also, it has been used in the field of natural language understanding (parsing discourse used during multi-agent negotiations) [1], but such work has not dealt with strategic decision making.

In this paper we present a novel algorithm and a novel deep network architecture to approximate the Q-function in strategic board game environments. Our algorithm does not directly approximate the whole Q-function, like standard DRL approaches, but evaluates Q-values "locally": in our case, this means that the Q-value for each possible action is computed separately, as if it were the only possible next action. Standard techniques seen in DRL literature so far, like experience replay and target networks, are not used. Instead, we take advantage of the recurrency of the network, as well as the locality of our algorithm, to achieve stable good performance. Our generic Deep Recurrent Reinforcement Learning (DRRL) algorithm was adapted and tested in the SoC domain. Our results show that it outperforms Monte-Carlo-Tree-Search (MCTS) agents, as well as the state-of-the-art algorithm for this domain [28]. In addition, its performance gets close to that of another DRL agent found in the literature [4], though *it does not*—in contrast to that agent—use network pre-training for learning: as we detail later in the paper, our algorithm trains itself "on-line" while playing a game, using fewer than one hundred (100) learning experiences, as opposed to *hundreds of thousands* used by the DRL agent in [4]. Moreover, when we allowed training our network over a series of games, using $\sim 2,000$ learning experiences, our method's performance improves and matches that of the DRL agent in question.

2 Background and Related Work

In this section we provide the necessary background for our work, and a brief review of related literature.

2.1 Deep Reinforcement Learning

The main goal of DRL is to approximate an RL component, such as the Q-function, the value function or the policy. This function approximation is done by generalizing from samples.

The standard framework for RL problems is provided by Markov Decision Processes (MDPs). An MDP is a tuple of the form $(S, A_s, P^a_{ss'}, \gamma, R^a_{ss'})$ where S is a set of the possible states that represent the dynamic environment, A is the set of possible actions available in state s, $P^a_{ss'}$ is the probability to transit from state $s \in S$ to state $s' \in S$ by taking action $a \in A_s$, $\gamma \in [0,1]$ is a discount factor and $R^a_{ss'}$ is the reward function that specifies the immediate reward for transitioning from state $s \in S$ to state $s' \in S$ by taking action $a \in A_s$.

To measure the value (i.e. performance) of a state-to-action mapping, the fundamental Bellman Optimality Equations [3] are usually used. If we consider a policy $\pi(b)$ as the mapping of beliefs to actions, or else the probability of taking action $a \in A_s$, we describe the expected value of the optimal policy:

$$V^*(s) = \max_a \Big[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \Big] \tag{1}$$

An optimal policy is then derived as $\pi(s) = argmaxQ^*(s,a)$, where

$$Q^*(s,a) = R(s,a) + \gamma \max_{a'} Q^*(s',a')$$

With the optimal Q-function known, the optimal policy can be easily found, by choosing the action a that maximizes $Q^*(s,a)$ for state s. When the state-action spaces are very large and continuous, the need for function approximation arises [26], in order to compute the underlying functional form of the Q-function, from a finite set of state-action pairs in an environment. In recent years, using deep neural architectures for function approximation became possible.

DRL Algorithms Recent algorithms combat problems that were inherent in DRL (such as instability and inefficiency during learning). The first such algorithm, which was also able to be highly effective in a wide range of problems without using problem-specific knowledge or fine tunning, was the Deep Q-Network algorithm (DQN) [17].

DQN, implements Q-learning in a deep Convolutional Neural Network (CNN) and manages to master a range of Atari games with only raw pixels and score as input. For stability and better convergence, DQN also uses "experience replay", by storing state transitions. Furthermore the target Q-values are computed in a separate identical target Q network updated in a predefined number of steps.

Many extensions and improvements have been proposed in the literature (e.g [9, 19, 2]). One of the most significant contributions is AlphaGo [24], the first Go agent to have actually won a human professional player. Alpha Go combines Monte Carlo Tree Search (MCTS) for position evaluation and CNNs for move selection. Research has also focused lately in Deep Reinforcement Learning in continuous action spaces. In this case the DRL problem is approached with policy gradient methods rather than Q-learning [16].

Neural Network Types and Architectures The most known deep architectures are Deep Feed Forward (DFF), Convolutional (CNN), and Recurrent Neural Networks (RNN). One RNN type that is especially effective in practical applications is Long Short Term Memory (LSTM) [11]. LSTMs make it possible for current information to be processed by keeping in mind previous states' information, since they actually have internal recurrence, more parameters than normal RNNs, and a system of gating units that controls the information flow.

DRL has been successful in working with RNNs and especially LSTMs [18], because of their good performance in sequential data, or data with temporal relationships, and especially for their capability of avoiding the problem of vanishing gradients. Recent work in RNNs [10] also, showed that recurrency in deep networks provides good belief estimates in Partially Observable MDPs.

2.2 Action-Dependent Features and Q-Decomposition

The concept of "action-dependent state features" was introduced in [25], and was later generalized the idea of Q-function decomposition by [22] (and, later, others—see, e.g., [14]). In [25] the RL process is partitioned in multiple "virtual" sub-agents, and rewards realized by each sub-agent are independent given only the local action of each sub-agent.

In more detail, assume a factored representation of the state space, which entails a feature vector (that is, a vector of state variables) representing the state s. In general, when at a state one needs to select the appropriate action, execute it, and then further update the Q-function values given real-world reward gained by this action selection, and our estimates on the long-term effects of the action. Now, if we assume that the feature vector contains action-dependent features, and each specific instantiation of this vector ("feature value") is strongly related to an action a^i, then the long term reward for choosing action a^i depends only on the feature value related to a^i [25]. Since there is only one feature value related to each a^i, an agent can realize rewards independently, by performing only a specific action evaluated locally (i.e., given the current state).

To elaborate further, consider a possible action $a^i \in A$, let I denote $|A|$, and let $e(s, a^i)$ be a fixed function that takes a^i and s as input and outputs a specific instantiation of the feature vector (i.e., its "feature value"). This essentially relates the a^i action to the specific values of the state variables.[3] Intuitively, this means that the effect of using an action at a particular state is that the feature vector takes a specific value (e.g., because specific state variables take on specific values). The feature values can then be associated with the long-term effects of employing a^i at s via a Q-function over action-feature values pairs. That is, $Q(\langle e(s, a^1), ..., e(s, a^I \rangle), a^i)$ denotes the long-term value of a^i when s gives rise to (or, to use the language of [25], *generalizes to*) the vector $\langle e(s, a^1), ..., e(s, a^I)\rangle$ via a generalization function f. Now, since the feature values are action-dependent, we can assume that the expected value of employing a^i at s depends only on the feature value related to a^i: for a specific f, the value $Q(f(s), a^i)$ entirely depends on $e(s, a^i)$. That is:

$$Q\Big(\langle e(s, a^1), ..., e(s, a^I)\rangle, a^i\Big) = Q\Big(\langle e(s', a^1), ..., e(s', a^I)\rangle, a^i\Big) \tag{2}$$

whenever $e(s, a^i) = e(s', a^i)$. In other words, $Q(f(s), a^i)$ is entirely independent of (s, a^j) for $j \neq i$.

In our work we apply this idea in deep networks, considering that the association between each possible action and the feature values (networks' input) is based on the networks weights, which are different for each evaluation function.

2.3 The Settlers Of Catan (SoC) domain

The Settlers Of Catan (SoC) is a multi-player board game, where players attempt to build establishments while trading with other players to acquire the needed resources in order to do so. The actual board of the game is an island

[3] We remark that no factored state representation was assumed in [25]; rather, each state was linked to a single action-dependent feature (with its set of values).

representation, composed of hexagonal tiles (hexes), each one representing a different land type and resource (Clay, Wool, Wheat, Ore and Sheep). The players establishments can be settlements, cities and roads. Especially roads are used in order for the players to connect their holdings.

In order to be able to build, a player must spend an amount of resources. The available resources are Clay, Wool, Wheat, Ore and Sheep and are represented by *resource cards.* In each turn, depending on the dice roll, it is decided which hexes produce resources, thus the player with a settlement (city) in this hex gains one (two) resource card of the corresponding resource type. To provide an example, if a number corresponding to a clay hex is rolled, the player who owns a city and a settlement adjacent to this hex, will get three clay resource cards.

There are also five kinds of *development cards* (i.e. knight, victory point, monopoly, road building and year of plenty). When a development card is played, it has a positive outcome for the player. For example, the "road building" card allows the player to build two roads with no cost.

Another way for a player to gain resources is by trading with other players or the game bank. Thus in each turn the player rolls the dice and all players gain resources based on their establishments. Then it is up to the player if she wants to propose a trade, if she is in need of some particular resources. Afterwards she can decide whether she wants to build a settlement, road or city, buy or play a development card. When a trade is offered, the other players can accept, reject the offer or make a new counter-offer.

Each time a player expands its territory, she gains victory points. A settlement (resp. city) awards the player 1 (resp. 2) victory point. Furthermore, the player with longest uninterrupted road is awarded 2 victory points. The game ends when a player gets 10 victory points (at least).

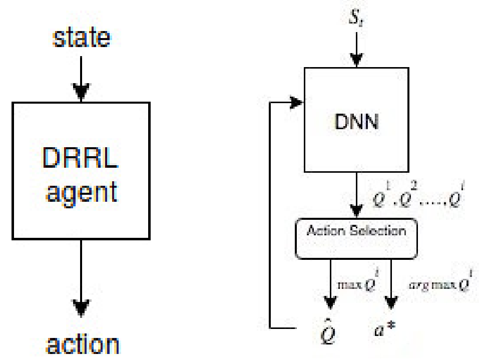
Agents for SoC The Java Settlers game as well as two jSettler agents included in it, was originally created in [28]. The jSettlers use business negotiation strategies for evaluating offers and trading during a negotiation. For other decision making in the game, [28] implemented two separate algorithms for computing the building speed (each corresponds to an agent respectively). The "fast" strategy takes into account the immediate reward, and the "smart" one evaluates actions beyond that. Both switch between three different strategies (road-building, city-building and monopolizing) for deciding their build plan. Then, [8] altered the original jSettler by improving aspects regarding initial placement of builds, favoring specific types of build plans and the purchase of development cards. In [21], model trees and linear regression were used for Q-function approximation.

SoC has also been a popular testbed for MCTS methods. Several implementations have been put forward [27, 20] but without being tested according to the complete game rule set, or without being able to make trade offers. Three different MCTS agents capable of trading, while playing under the full set of rules are introduced in [12]. Specifically, one uses the known bandit family method UCT, the second agent is an extension of UCT using Bayesian inference (BUCT), and the third employs the VPI RL method [5]. All these agents also use parts of the original jSettler, for tasks not supported by the MCTS (e.g. playing development cards). Although the MCTS agents could not outperform the jSettler, the VPI one appeared to be competitive. In [6], an extension of UCT incorporating knowledge mining from a corpus of human game-play was proposed. It had promising results, but was not tested against the jSettler agent.

According to [13], the only approaches so far capable of outperforming the jSettler, are specific heuristic improvements in agent negotiation strategies, and also, interestingly, a DRL approach [4]. That agent trains and tests a fully-connected neural network using DQN against several opponents, focusing on mapping game instances to dialogue actions. The DRL agents trained playing against jSettlers and supervised agents, both outperformed the jSettler. In contrast to [4], our approach does not require a long sequence of training experiences ranging over a series of games, but actually learns to play effectively within a single SoC game instance, in true "on-line" RL style. As such, learning occurs within a realistically small number of rounds.

3 Our Approach

In this section we explain the algorithm and architecture of the novel agent we designed for decision making in strategic board games. This agent observes the environment (i.e. the game state), and returns an appropriate action, one with maximal Q-value. To do this, we built a deep structure. Figure 1 provides a graphical overview of our approach.



(a) (b)

Fig. 1: (a) *A top-level view of the DRRL agent algorithm, with the state as input and the preferable action as output.* (b) *The network gets S_t as input at time t and outputs all the Q^is. Then the action with the maximum Q^i for this state is returned as output of the algorithm and the maximum Q^i is used in next network iteration.*

We decided to exploit the advantages of recurrent networks and specifically LSTMs, along with the concept of Q-decomposition. This is novel for DRL:

in this way, our network does not approximate the Q-function itself, but the different local Q-values. Considering that the neural network architecture we have implemented uses recurrency, we can employ the Bellman Q-value function as if dealing with an MDP problem [10].

3.1 Local Q-Function Approximation

In order to approximate the Q-function, we will use the Bellman equation as well as the concept of Q-decomposition. We define $s_t = \langle s_t^j \rangle$ as a factored state at time t, with s_j^t being one of N variables (state features) that takes a value in its domain. Then, a_t^i as the action $i = \{1, 2, ...I\}$ selected at t, where $a_i \in \{a^1, a^2, ..., a^I\} \in A$ (and $I = |A|$). S includes all possible states, and A all possible actions. We also consider a reward function $r(s_t, a_t^i)$ which denotes the returned reward when selecting action a_t^i in state s_t at time step t. The real Q-function can be approximated by some $\hat{Q}$ at time t, whose form can be expressed recursively, in terms of the $\hat{Q}$ of the next state, as follows:

$$\hat{Q}(s_t, a_t^i; \theta) = r(s_t, a_t^i) + \gamma \max_a \hat{Q}(s_{t+1}, a; \theta) \tag{3}$$

where θ are some network weight parameters.

Following the concept of Q-decomposition presented in Section 2, we assume that the Q function can be decomposed into $i \in [1, I]$ partitions, where $I = |A|$. In this way we can compute the short-term effect of every action a_t^i based on the current environment state, i.e., a local Q-value which is action dependent. Since we try to approximate these Q-values, we will also associate a different set of weights for each one of them, denoted as θ^i, such that:

$$Q^i(s_t, a_t^i; \theta^i) = \begin{bmatrix}\phi_1(s_t) \ \phi_2(s_t) \cdots \phi_N(s_t)\end{bmatrix} \begin{bmatrix}\theta_1^i \\ \theta_2^i \\ \vdots \\ \theta_N^i\end{bmatrix} = \sum_{j=1}^{N} \phi_j(s_t) \cdot \theta_j^i \tag{4}$$

where Q^i is the Q-value estimated for action i at timestep t, and the $\phi_j(s_t)$ basis functions are of the form:

$$\phi_j(s_t) = (1 - \sigma(s_t)) \cdot \phi_j(s_{t-1}) + \sigma(s_t) \cdot \tanh(s_t) \tag{5}$$

where σ is the sigmoid function and tanh the hyperbolic tangent function, and each ϕ_j is actually applied to a corresponding state variable s_t^j (the state variables for the SoC domain are listed in Table 1).

Note that with $\phi(s_t)$ being an $1 \times N$ vector, each θ^i, is a $N \times 1$ vector. By multiplying these two vectors, we end up with a unique value for each Q^i.

Now, to come up with better Q^i estimates, we naturally aim to minimize the difference between the "real" Q-value estimate, and Q^i in each time step. This can be framed as an optimization problem that minimizes a loss function with respect to the θ^i parameters, and solved with stochastic gradient descent (SGD) via back-propagation.

Given these calculated Q^is, the action that will now be considered "best" for the state that was observed, is the one with maximal (locally) estimated Q-value:

$$arg \max_i Q^i = arg \max_i Q^i(s_t, a_t^i; \theta^j) = arg \max \begin{cases} Q^1(s_t, a^1; \theta^1) \\ Q^2(s_t, a^2; \theta^2) \\ ... \\ Q^n(s_t, a^I; \theta^I) \end{cases} \quad (6)$$

The Q-value of that a^* action will constitute the new $\hat{Q}$ value estimate for an $s = s_t, a = a^*$ pair: whenever $s = s_t$ is encountered, $a = a^*$ is the *currently assumed* action of choice, and has a value of $\hat{Q}(s, a^*)$; of course, this can be updated in future iterations.

We note again that in our case, there is a separate θ^i for each a^i. Given this, notice that the evaluation for the Q-values at s_t can be computed locally in parallel[4] for all possible actions and corresponding parameters θ.

3.2 Deep Architecture

We implemented a deep network (see Figure 2) consisting of $I = |A|$ parallel recurrent recursive neural networks (RNN). Each RNN is an LSTM [11] (i.e. LSTM layer) followed by a soft-max activator (soft-max Layer) and outputs a Q-value. The LSTM layer practically summarizes a state input vector retrieved from the environment as a single representation, that contains information about the entire input sequence. The LSTM cells also provide an internal memory to the network, regarding useful information from the previous seen states of the environment, in order for them to be used in the next time step. Those aspects of the LSTM provide us with the necessary mathematical transformations to compute the basis function presented in Section 3.1 based on the current state input as well as the previous state's one.

By iteratively optimizing a selected loss function, the corresponding parameters θ^i for each RNN can be computed. The inner product of the LSTM output with each θ^i vector, is normalized with the soft-max activation function [26], in order to keep values scaled between $[0, 1]$. This information is actually a Q-value estimate. The actual value of each Q^i corresponding to an action is then approximated after the new gradients are computed, (i.e. the θ^i updates from the loss function minimization with SGD) as described earlier. The output of the Q-values at one time step is actually the output of the whole network.

3.3 The DRRL agent

In this section we present the Deep Recurrent Reinforcement Learning agent, an algorithm to approximate the local Q-values using the deep network above.

[4] More accurately, in our implementation in a pseudo-parallel manner: all LSTMs are executed independently and the final action is selected given their outputs.

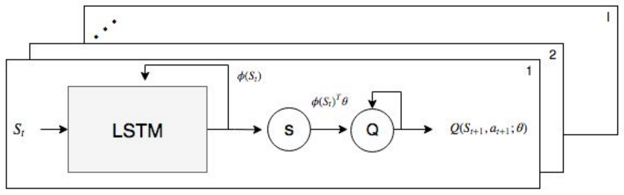


Fig. 2: *A visualization of the network for one action. The deep network consists of $I = |A|$ parallel RNNs with LSTMs (LSTM layer). A soft-max activator (soft-max layer) follows each one of them in order to normalize weights θ^i and the Q-value is updated (Q-value layer). Each RNN outputs a Q-value.*

In each time step t (or a round of a game), the algorithm receives a state of the environment as an input and returns the action corresponding to the maximum Q-value approximated by the network. In response it receives a reward and observes a state transition.

In every t all the Q^is corresponding to actions are updated, as well as the weights θ^i relevant to each one of them. This means, that an optimization step occurs for every every Q^i separately, where previous estimated parameters θ^i for a^i are updated. Thus evaluations for all the θ^is are conducted in parallel.

Since all the θ^is are updated, the Q^is are updated too. We preserve each $Q^i(s_t, a^i_t; \theta^i)$ approximation in the Q-values layer as the output of the RNN, and when the back-propagation optimization steps take place, those Q values are updated with the corresponding θ^is. Each Q^i represents the mapping from all states seen so far and parameters to a specific action; thus it is a part of a local Q-function approximator that contains the previous time step Q-value approximation for this action with the corresponding θ^i. The action a^i_t extracted from the relevant Q^i which maximizes it locally, is considered to be the one that maximizes the actual Q-function. So this action is returned by the algorithm. The appropriate $\hat{Q} = Q^i$ for the selected action is fed back in the RNN iteratively as part of the input, representing the Q-value of the previous time-step.

To summarize, the algorithm learns through the network different set of parameters regarding the loss functions, and also saves parameter-dependent function approximators. Thus, there are $I = |A|$ Q-approximators, each one using a different instance of the LSTM layer to update its own parameters in every iteration. A pseudo-code for the algorithm is provided in Algorithm 1.

4 Evaluation

The DRRL agent was evaluated in the Settlers of Catan (SoC) domain, using the jSettlers Java application.[5] In this section we explain how DRRL was instantiated in this domain, and present our evaluation results.

[5] http://nand.net/jsettlers/

Algorithm 1 DRRL Algorithm

```
 1: procedure DRRL
 2:     Input = $s_t$, $\hat{Q}$
 3:     // Returns a "best" action $a^*$ and an estimate of its value at $s_t$
 4:     Initialize $Q^i$ to zero, $\forall i \in \{1, 2, ..., |A|\}$
 5:     // at t=0 only, all $\hat{Q}$s and $\theta^i$s are also initialized to zero
 6:
 7:     for $\forall i \in \{1, 2, ..., |A|\}$ do        ▷ All computations and updates in parallel
 8:         Set temporary $Q^i(s_t, a_t^i; \theta^i) = \phi(s_t)\theta^i$
 9:         Perform SGD with respect to $\theta^i$:
10:         $L(\theta^i) = \nabla_{\theta^i}\left(r(s_t, a_t^i) + \gamma\hat{Q}((s'|a_t, s_t), a') - Q^i(s_t, a_t^i; \theta^i)\right)^2$
11:         Update $\theta^i$, $Q^i$
12:     $a^* = arg\max_i Q^i(s_t, a_t^i; \theta^i)$
13:     $\hat{Q}(s_t, a^*) \leftarrow \max_i Q^i(s_t, a_t^i; \theta^i)$
14:     return $\hat{Q}(s_t, a^*), a^*$
```

4.1 Domain State Representation and Action Set

Our decision making algorithm requires a state of the environment to be fed as an input to the network. The state captures the board of the game, and the information known to the player at time step t. Each state feature (state element) is assigned to a range of integers, and thus the state is an integer vector of 161 elements (see Table 1). In more detail, the first 5 features (hasClay, hasOre, hasSheep, hasWheat, hasWood) represent the available resources of the player whose turn is to move. The board consists of 19 main hexes. Each hex has 6 nodes, thus there are in total 54 nodes. Also there exist 80 edges connecting the nodes to each other. The player can build roads to connect her settlements and cities on those edges. The robber is placed in the desert hex when the game begins, and can be moved according to the game rules.

Num	Feature	Domain	Description
1	Clay	$\{0, .., 10\}$	Player's number of Clay Units
1	Ore	$\{0, .., 10\}$	Player's number of Ore Units
1	Sheep	$\{0, .., 10\}$	Player's number of Sheep Units
1	Wheat	$\{0, .., 10\}$	Player's number of Wheat Units
1	Wood	$\{0, .., 10\}$	Player's number of Wood Units
49	Hexes	$\{0, .., 5\}$	Type of resource in a hex (unknown element=0, clay=1, ore=2, sheep=1, wheat=4, wood=5)
54	Nodes	$\{0, .., 4\}$	Builds ownership on each node (settlements and cities): (no builds=0, opponents' builds=1,2, agents' builds=3,4)
80	Edges	$\{0, .., 2\}$	Roads ownership on each edge: (no road=0, opponents' road=1, agents' road=2)
1	Robber	$\{0, .., 5\}$	Robber's location (unknown element=0, clay=1, ore=2, sheep=1, wheat=4, wood=5)
1	Turns	$\{0, .., 100\}$	Number of game turns
1	VP	$\{0, .., 10\}$	Number of player's victory points

Table 1: *State features retrieved from jSettlers. Number is the number of elements needed to describe this feature and Domain includes the feature's possible values.*

Since SoC is a board game with numerous possible actions, we focused our learning on a constrained set of actions, in order to reduce computational complexity. Since trading is key to the game, this is the action set we selected. Therefore, we select an action set $A^0 \in A$, containing *(i)* actions specifying "trade offers", and *(ii)* "reply" actions towards proposers. For all other actions, the agent simply adopts the choices of the jSettler. Thus, the jSettler part of the agent is responsible for choosing actions like "*build a road*").[6]

In more detail, the DRRL agent is responsible for "reply" actions to opponent offers (i.e. accept, reject, counter-offer), and "trade offers" for giving up to two resources (same or different kind) and receiving one. The resources available are Clay, Ore, Sheep, Wheat and Wood—thus the "trade offers" include all possible combinations of giving and receiving those resources. Some trade offers examples are: *Trade 1 Clay for 1 Wheat, Trade 2 Woods for 1 Clay, Trade 1 Ore and 1 Sheep for 1 Wood.* Overall, we have 72 actions $\in A^0$, 70 for trade offers, and 2 for reply actions. The counter-offer reply action is not considered as a separate action in the actual implementation, since the agent directly makes a new offer instead of accepting or rejecting a trade.

4.2 DRRL in the SoC Domain

In SoC, the DRRL agent is responsible for the following during game-play:

1. Decide whether to accept, reject or make a counter-offer to a given trade proposal from another player.
2. At each round decide if a new trade offer should be made or not.
3. Select the preferred trade offer, if such an offer is to be made.

Given our discussion above, we consider a_t^i as the action $i = \{1, 2, 3, ...72\}$ selected at t, where $a_i \in \{a^1, a^2, a^3, ..., a^{72}\} \equiv A^o$. Furthermore, assuming action-dependent state features, we consider $Q^i(s_t, a_t^i; \theta^i)$ as a partition of the Q-function, for taking action a^i in state s at t, given the network parameters θ^i. Thus we assign a different $\theta^i \in \{\theta^1, \theta^2, ..., \theta^{72}\}$ for each possible action.

The function $r(s_t, a_t^i)$ gives the returned reward when executing action a_t^i in state s_t at time step t. In our setting, the only reward signal provided by the jSettlers at a state is VPs accumulated so far. So, we formulate the reward function as follows:

$$r(s_t, a_t^i) = \begin{cases} \overline{VP}(s_t, a_t^i) \cdot k & \text{if } \overline{VP}(s_t, a_t^i) > 0 \\ -VP \cdot k & \textit{otherwise} \end{cases} \tag{7}$$

where VP are the accumulated victory points gained in the game so far, and $\overline{VP}(s_t, a_t^i)$ are the victory points gained in t by acting with a^i—i.e., the immediate reward provided by the difference between VPs in the previous and current state (which actually can sometimes be a negative number). The k parameter was set to 0.01 after trials, since we wanted the reward to have an impact on the Q-values, without creating a huge deviation among them.

[6] In general, our action and game set up follows [4].

The DRRL algorithm is taking action every time the agent wants to decide whether to trade or reply to an offer. In each round, the network receives the state of the game as an input. To achieve this, the architecture of the neural network is formed accordingly. This means that the LSTM layer consists of 72 independent LSTM units, and also 72 different Q^i and sets of θ^i are produced.

4.3 Simulations and Results

The DRRL agent was implemented in Python using the Tensorflow library. To measure agent's performance we manipulated the jSettlers Java API, in order to support python-based clients, and specifically ones with deep neural network architecture. We tested our algorithm both in CPU (Intel(R) Core(TM) i3-2120 @ 3.30GH) and GPU (NVIDIA GeForce GTX 900), and actually it was performing slightly faster in the CPU. This probably happens since learning is taking place in batches of one, thus the GPU's accelerating aspects are not used.

Usually the training of deep neural networks needs thousands of iterations. For instance, in [4], they trained their DRL agent for SoC with $500,000$ learning experiences, until it converges to an effective policy. In our approach, we do not train the neural network, thus we had to find an efficient way to optimize the network parameters within the given game limitations. To this end, we experimented with different learning rates for Stochastic Gradient Descent (SGD). For very small learning rates (i.e. exponential), the algorithm appeared to be incapable of exploring the action space within the course of one game, and alternated among using 3 or 4 actions only. On the other hand, for higher learning rates, the algorithm gets stuck at local minima, and ended up exploring during the first game rounds, but then kept selecting the same preferred action. After some experimentation, we set the learning rate to 0.0023.

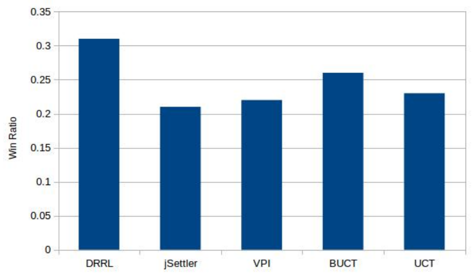
The θ^i model parameters were initialized with a truncated normal distribution. We also tested the network initialized with random, normal, and uniform distributions, but this did not help, because it introduced a bias towards specific actions in some cases. Finally, the $\hat{Q}$ (i.e. the Q-value of the previous chosen action) as well as the local Q^is were initialized with zeros.

We pit our agent against the "standard" jSettler agent, and also against the three MCTS implementations from [12]: BUCT, UCT, and VPI (Section 2.3). A SoC game involves four agents, thus in each game four of the aforementioned five agents face each other, chosen randomly and also in random order. We made sure that every agent faces every possible combination of others an equal number of times: in total, we ran five 4-agent tournaments of 20 games each, and each agent participated in four of those tournaments.

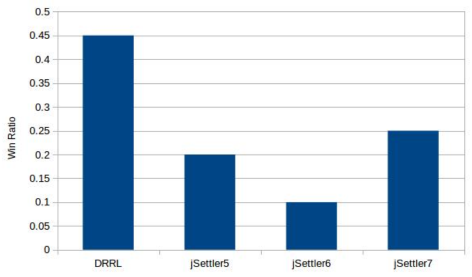
In every new game, all network parameters are initialized, as if the agent has never played before. Each game lasts about 15-26 rounds, and the agent gains at most about 70 *learning experiences* (since an experience is gained in every turn where she has to propose an offer, or respond to one).[7] The DRRL

[7] Compare this number to the $500,000$ learning experiences required by the DRL agent in [4].

algorithm itself runs in about 56 seconds, and the whole game in approximately half to one hour and a half (depending on the number of rounds). As mentioned above, in total we ran 100 games, by combining the agents in five different pools of four agents (each corresponding to a "tournament"). Thus, every agent faces all different combinations of opponents, itself participating in 80 games in total. We used as our principal comparison metric, the agents' *win ratio*, calculated for each agent over the number of games that agent participates in.



(a)



(b)

Fig. 3: (a) *The agents' winning ratio over 80 games against all opponents.* (b) *The agents' winning ratio over 20 games against jSettlers.*

Our evaluation results shown in summarized in Figure 3a show that the DRRL agent is capable to outperform all others without pre-training, but by only training itself in real time while playing a game, and rapidly adapts to the game environment. Some instability is noticed from game to game, but this is normal due to the fact that each game is a first time experience for the agent, yet we plan to further investigate this in the future by learning parameters over a sequence of games. We can also report that the DRRL agent was consistently proposing significantly more offers (and counter-offers) than its opponents (specifically, 10-40 per game), and actually in a very early stage of the game, while other agents did not. This increased trading greatly benefited the agent. Yet, it was rarely accepting other players' trades offers—it would rather counter-offer instead. We also report that the MCTS agents perform much better when the DRRL agent is in the game, since it favors trading, while the jSettler does not.

We also ran experiments in a pool with only jSettlers as opponents. This allows us to make an indirect comparison of our DRRL agent against the "best" DRL agent built in [4], namely a DRL agent whose policy was pre-trained using about $500,000$ learning experiences while facing jSettler opponents. Once trained, that DRL agent was able to achieve a win ratio of 53.36% when playing against three jSettler agents. We see in Figure 3b that our DRRL agent achieves a win ratio of 45% against jSettlers, using at most 70 learning experiences; we note however that 45% is higher than the results achieved by most other DRL agents in [4]. We then increased our training horizon, allowing our DRRL agent to train itself across a series of 30 games; that is, the agent continued to update its network weights across 30 games ran sequentially, and not just one. This corresponds to $\sim 2,000$ learning experiences. We can report that now the win rate of our agent rises to 56%, with DRRL now winning 17/30 games, and with the agent winning nine out of the fifteen last games played. Therefore our DRRL agent *(a)* matches (and slightly surpasses) the results of the DRL agent in [4]; and *(b)* we can also reasonably deduce that adding more training experience benefits the agent.

5 Conclusions and Future Work

In this paper we presented a novel approach for deep RL in strategic board games. We optimize the Q-value function through a neural network, but perform this locally for every different action, employing recurrent neural networks and Q-value decomposition. Our algorithm managed to outperform state-of-the-art opponents, and did so with minimal training.

Regarding future work, we intend to examine performance differences when using stacked LSTMs compared to the vanilla ones we used now, or even GRUs. Furthermore, we plan to extend the algorithm to larger or different action sets, and also create DRRL agents for other games. A final goal is to incorporate natural language processing, in order for the deep network to take into account the players' conversations during the game.

References

1. Afantenos, S., Kow, E., Asher, N., Perret, J.: Discourse parsing for multi-party chat dialogues. In: Proceedings of EMNLP 2015. pp. 928–937 (2015)
2. Anschel, O., Baram, N., Shimkin, N.: Deep reinforcement learning with averaged target DQN. CoRR **abs/1611.01929** (2016)
3. Bellman, R.: Dynamic programming. Courier Corporation (2013)
4. Cuayáhuitl, H., Keizer, S., Lemon, O.: Strategic dialogue management via deep reinforcement learning. In Proceedings of the NIPS Deep Reinforcement Learning Workshop (NIPS 2015)
5. Dearden, R., Friedman, N., Russell, S.: Bayesian q-learning. In: AAAI/IAAI. pp. 761–768 (1998)
6. Dobre, M.S., Lascarides, A.: Online learning and mining human play in complex games. In: Computational Intelligence and Games (CIG), 2015 IEEE Conference on. pp. 60–67. IEEE (2015)
7. Finnman, P., Winberg, M.: Deep reinforcement learning compared with q-table learning applied to backgammon (2016)
8. Guhe, M., Lascarides, A.: Game strategies for the settlers of catan. In: Computational intelligence and games (CIG). pp. 1–8. IEEE (2014)
9. van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning. CoRR **abs/1509.06461** (2015)
10. Hausknecht, M., Stone, P.: Deep recurrent q-learning for partially observable mdps. CoRR, abs/1507.06527 **7**(1) (2015)

11. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
12. Karamalegkos, E.: Monte carlo tree search in the "settlers of catan" strategy game (2014), Senior Undergraduate Diploma Thesis, School of Electrical and Computer Engineering, Technical University of Crete ; https://goo.gl/rU9vG8
13. Keizer, S., Guhe, M., Cuayáhuitl, H., Efstathiou, I., Engelbrecht, K.P., Dobre, M., Lascarides, A., Lemon, O.: Evaluating persuasion strategies and deep reinforcement learning methods for negotiation dialogue agents. In: Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers. vol. 2, pp. 480–484 (2017)
14. Kok, J.R., Vlassis, N.: Collaborative multiagent reinforcement learning by payoff propagation. Journal of Machine Learning Research **7**(Sep), 1789–1828 (2006)
15. Lai, M.: Giraffe: Using deep reinforcement learning to play chess. arXiv preprint arXiv:1509.01549 (2015)
16. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. CoRR **abs/1509.02971** (2015)
17. Mnih, V., Kavukcuoglu, K., Silver, et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (Feb 2015)
18. Oh, J., Guo, X., Lee, H., Lewis, R.L., Singh, S.: Action-conditional video prediction using deep networks in atari games. In: Advances in neural information processing systems. pp. 2863–2871 (2015)
19. Osband, I., Blundell, C., Pritzel, A., Roy, B.V.: Deep exploration via bootstrapped DQN. CoRR **abs/1602.04621** (2016)
20. Panousis, K.P.: Real-time planning and learning in the "settlers of catan" strategy game (2014), Senior Undergraduate Diploma Thesis, School of Electrical and Computer Engineering, Technical University of Crete ; https://goo.gl/4Hpx8w
21. Pfeiffer, M.: Reinforcement learning of strategies for settlers of catan. In International Conference on Computer Games: Artificial Intelligence (2018)
22. Russell, S.J., Zimdars, A.: Q-decomposition for reinforcement learning agents. In: Proceedings of the 20th International Conference on Machine Learning (ICML-03). pp. 656–663 (2003)
23. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural networks **61**, 85–117 (2015)
24. Silver, et al.: Mastering the game of go without human knowledge. Nature **550**, 354– (Oct 2017)
25. Stone, P., Veloso, M.: Team-partitioned, opaque-transition reinforcement learning. In: Proceedings of the third annual conference on Autonomous Agents. pp. 206–212. ACM (1999)
26. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (1998)
27. Szita, I., Chaslot, G., Spronck, P.: Monte-carlo tree search in settlers of catan. In: Advances in Computer Games. pp. 21–32. Springer (2009)
28. Thomas, R.S.: Real-time decision making for adversarial environments using a plan-based heuristic. Ph.D. thesis, Northwestern University (2003)