Reinforcement learning in combinatorial action spaces with a partial simulator

Danielle Rothermel
Facebook, NYC
drotherm@fb.com

Jonas Gehring
Facebook, Paris
jgehring@fb.com

Daniel Haziza
Facebook, Paris
dhaziza@fb.com

Dan Gant
Facebook, NYC
danielgant@fb.com

Vegard Mella
Facebook, Paris
vegardmella@fb.com

Nicolas Carion
Facebook, Paris
alcinos@fb.com

Nicolas Usunier
Facebook, Paris
usunier@fb.com

Gabriel Synnaeve
Facebook, NYC
gab@fb.com

Abstract

The combination of tree search and reinforcement learning has recently been applied to board games with great success. However, these methods intrinsically rely on extensive rollouts in the environment. We consider the problem of extending these approaches to settings in which no simulator is available. Instead, we assume that we have access to a partial model that accurately predicts the short-term consequences of the agent's actions, disregarding the complex dynamics of the surrounding environment. We demonstrate the feasibility of using tree search within such a partial simulator for fixed horizon planning, learning the value function with reinforcement learning. We showcase our approach on a challenging task with a combinatorial action space, partial observations, and complex dynamics: the generation of high-level strategies in the real-time strategy game *StarCraft®: Brood War®*[1].

1 Introduction

Planning is an essential component of intelligent behavior, and tree search is at the core of breakthroughs in turn-based zero-sum games as achieved by DeepBlue, AlphaGo or AlphaZero [2, 18, 17]. It is a natural way to structure exploration and to build hierarchies of actions, but requires the possession of a simulator or forward model of the environment. In many settings, rollout procedures can be too computationally expensive, or simply unavailable due to properties of the environment: complex dynamics, very large state and action spaces or partial observability can result in combinatorial explosion of complexity. We can sometimes however obtain simulators that are only approximate or based on simple heuristics and that provide useful short-term predictions.

In this work we propose to study the usage of such imperfect, or partial, simulators to construct plans as a means to structure an otherwise combinatorial action space. In particular, we are interested in simulators or forward models that are able to predict the direct, *local* effects of actions with high accuracy. This is in contrast to forward models in the classical sense, as such partial models do not attempt to predict the whole future state.

We show how to employ a partial simulator for planning in combination with Monte-Carlo tree search (MCTS) in a partially observable environment, and propose to use the prediction of the partial simulator as an action abstraction. This makes it possible to learn effective value models in an

[1] *StarCraft* is a trademark or registered trademark of Blizzard Entertainment, Inc., in the U.S. and/or other countries. Nothing in this paper should be construed as approval, endorsement, or sponsorship by Blizzard Entertainment, Inc.

Preprint. Under review.

otherwise combinatorial action space. Our approach is validated on a build order planning task in the real-time strategy game *StarCraft: Brood War*.

2 Planning in Partially Observable Environments

2.1 Related Work

The heavy computational burden of planning in Partially Observable Markov decision processes (POMDP) motivates a variety of approximate techniques [8]. One of the main bottlenecks, the tracking of belief states across planning steps, was previously addressed by sampling belief points and for each storing its value and its derivative [15] and by maintaining state and action history MCTS nodes, which then enables belief state estimation via particle filtering, for example [16].

Efficient planning can also be achieved by exploiting structural properties of the problem at hand. In the context of RL, this is commonly done by abstraction at the level of states [13], along the temporal dimension by introducing macro-actions or options that consist of several successive actions [21], or at the level of actions [19, 25]. For POMDPs specifically, [14] decomposes the action space and obtains a hierarchy of POMDPs with reduced complexities.

2.2 Action Abstraction via Partial Models

It is often infeasible to produce good state abstractions for complex partially observable environments. This prevents learning accurate forward models for long-term predictions which are required for planning across several time-steps. But if an environment allows modeling the isolated, local consequences of our actions, we can leverage this to build meaningful abstractions over time. We refer to a model of these local effects as a *partial* model. Here we will determine necessary conditions for such action abstractions, and then detail the properties of a partial model to enable planning in said environments.

Consider an MDP represented as $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \tau, r, \gamma)$ with states $\mathcal{S}$, actions $\mathcal{A}$, a transition function $\tau : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$, an expected reward of a transition $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ and a discount factor γ. Without loss of generality we assume $r \in [0, 1]$. An abstraction over actions is provided by:

1. an exact cover of the original set of actions $\overline{\mathcal{A}} = (\overline{a}_1, ..., \overline{a}_{\overline{A}})$, i.e. $\overline{a}_j \subset \mathcal{A}$, $\bigcup_{j=1}^{\overline{A}} \overline{a}_j = \mathcal{A}$, and $\forall (j, j')\ j \neq j' \Rightarrow \overline{a}_j \cap \overline{a}_{j'} = \emptyset$.
2. a randomized strategy $\mu : \mathcal{S} \times \overline{\mathcal{A}} \to \mathcal{P}(\mathcal{A})$ that selects an action given an abstract action such that $\mu(s, \overline{a})(\overline{a}) = 1$ (i.e., μ outputs an action in $\overline{a}$ with probability 1 in every state when given $\overline{a}$).

This gives rise to a new MDP $\overline{\mathcal{M}} = (\mathcal{S}, \overline{\mathcal{A}}, \overline{\tau}, \overline{r}, \gamma)$ where $\overline{\tau} : \mathcal{S} \times \overline{\mathcal{A}} \to \mathcal{P}(\mathcal{S})$ is such that $\overline{\tau}(s'|s, \overline{a}) = \sum_{a \in \overline{a}} \mu(a|s, \overline{a}) \tau(s'|s, a)$ and $\overline{r}(s, \overline{a}, s') = \sum_{a \in \overline{a}} \mu(a|s, \overline{a}) r(s, \overline{a}, s')$.

A direct result of the simulation lemma (Strehl and Littman [20, Lemma 6], [27]) is that if the abstraction preserves the transition and reward structure, the optimal policy in the new MDP $\overline{\mathcal{M}}$ is nearly optimal in the original MDP. Hence, assuming

$$\begin{aligned} \exists \epsilon, \epsilon' > 0, \forall s, \quad & \max_{\overline{a} \in \overline{\mathcal{A}}} \max_{a, a' \in \overline{a}} \sum_{s' \in \mathcal{S}} \left| \tau(s'|s, a) - \tau(s'|s, a') \right| \leq 2\epsilon \\ \text{and} \quad & \max_{\overline{a} \in \overline{\mathcal{A}}} \max_{a, a' \in \overline{a}} |r(s, a, s') - r(s, a', s')| \leq \epsilon' \,, \end{aligned} \tag{1}$$

we can state:

Observation 1. *Let $\overline{Q}^*$ be the optimal Q-function in $\overline{\mathcal{M}}$, and let π be the policy in $\mathcal{M}$ inferred from the optimal policy in $\overline{\mathcal{M}}$ defined as*

$$\begin{aligned} \overline{\mathcal{A}}^*(s) &= \operatorname*{argmax}_{\overline{a} \in \overline{\mathcal{A}}} \overline{Q}^*(s, \overline{a}) \\ \pi(a|s) &= \frac{1}{|\mathcal{A}^*(s)|} \sum_{\overline{a} \in \overline{\mathcal{A}}^*(s)} \mu(a|s, \overline{a}) \,. \end{aligned}$$

Then, under assumption (1), denoting with Q^π the true Q-function of the stochastic policy π in $\mathcal{M}$ and with Q^* the optimal Q-function in $\mathcal{M}$, we obtain:

$$\max_{s,a} Q^*(s,a) - Q^\pi(s,a) \leq 2\frac{(1-\gamma)\epsilon' + \gamma\epsilon}{(1-\gamma)(1-\gamma+\epsilon)}\,.$$

Proof. We create an MDP $\mathcal{M}' = (\mathcal{S}, \mathcal{A}, \tau', r', \gamma)$ from $\overline{\mathcal{M}}$ with the action space of $\mathcal{M}$, i.e. for an action a and its abstraction $\overline{a}$, let $\tau'(s'|s,a) = \overline{\tau}(s'|s,\overline{a})$ and $r'(s,a,s') = \overline{r}(s,\overline{a},s')$. The policy π as defined above is optimal in $\mathcal{M}'$ since it is optimal in $\overline{\mathcal{M}}$. Conditions (1) make sure that τ' and r' satisfy $\max_{s,a}|r'(s,a) - r(s,a)| \leq \epsilon'$ and $\max_{s,a}\|\tau'(.|s,a) - \tau(.|s,a)\|_1 \leq 2\epsilon$. Writing $Q^* - Q^\pi \leq Q^* - Q^{'\pi^*} + \underbrace{Q^{',\pi^*} - Q^{'*}}_{\leq 0} + Q^{'*} - Q^\pi$, we can apply [20, Lemma 6], which compares the Q-value of a policy in two similar MDPs, to both $|Q^* - Q^{'\pi^*}|$ and $|Q^{'*} - Q^\pi|$ to obtain the result. □

For deriving the requirements of a partial model, we assume an environment where short-term local effects of an agent's actions can be modeled in isolation. For illustration, consider a task allocation problem of assigning work items to multiple processors. Let us assume that the overall system dynamics are stochastic: processors operate at variable, uncontrollable speeds, and an outside process continuously schedules work items. Even without considering the specific behavior of processors, we can predict with high accuracy the *local* effects of a single assignment action, e.g. a change to our work queue (one item is gone) or to the state of the designated processor (item assigned). Modeling the full state of our system after taking an action is, in contrast, extremely challenging. It requires considering that our selected processor could fail, and requires estimating which processors are available and how many work items would be scheduled next.

To formalize a partial model, we assume a POMDP setting $(\mathcal{S}, \mathcal{A}, \tau, r, \Omega, \mathcal{O}, \gamma)$ in which we can represent a state s_t and observations o_t in a joint feature space $\mathbb{R}^d$ obtained via $\Phi(s_t)$ and $\Psi(o_t)$, respectively. We now assume the overall system dynamics f can be expressed as a function of a (true) partial model ρ from the effect of our action and independent noise components u_t, v_t as follows:

$$\Phi(s_{t+1}) = f(s_t, u_t, \rho(\Psi(o_t), a_t, v_t))$$

Considering resource management tasks such as our task allocation problem from above, local changes to state features (e.g. the reduction of items in the work queue by assignment) can be modeled in deterministic manner, and we can treat, in feature space, the system dynamics as an additive term. With a deterministic partial model g', we thus have:

$$\Phi(s_{t+1}) = f'(\Phi(s_t), u_t) + g'(\Psi(o_t), a_t)\,.$$

When planning across several time-steps, we desire a partial model that can be applied recursively. g' is a function of the features of observations $\Psi(o_t)$, which raises the need for another partial model $g : \mathbb{R}^d \times A \to \mathbb{R}^d$ that is able to predict those in a reliably up to an error term α:

$$|g(\Psi(o_t), a_t) - \Psi(o_{t+1})| < \alpha\,.$$

When generating rollouts, applying the partial model g multiple times will accumulate prediction errors. In realistic settings, long-term prediction errors of local effects are dominated by the unaccounted global dynamics f'. Over shorter time spans, the possession of g enables planning by treating local effects of a sequence of actions $(a_j)_{j=t}^{t+h}$ up to a planning horizon h as an action abstraction:

$$\begin{aligned}\overline{a} &= g_h\left(\Psi(o_t), (a_j)_{j=t}^{t+h}\right)\\ &= g\left(g_{h-1}(\Psi(o_t), (a_j)_{j=t}^{t+h-1}), a_{t+h}\right), \text{ with } g_1 = g\end{aligned}$$

The planning process now consists of finding a sequence of actions that maximizes the expected return of the resulting effects as predicted by the partial model, given the current state s_t. In our experiments, we realize this with MCTS and a Q-function $Q(s, \overline{a})$ that evaluates the resulting features. Ignoring the overall system dynamics f' during the planning phase will prevent us from executing the resulting action sequences exactly as planned without loss of return; as a remedy, we re-plan at regular intervals.

We emphasize that by considering local consequences rather than raw actions, we can effectively marginalize over multiple action sequences that produce the same effect. Depending on the environment, this marginalization can have a significant impact on the exploration and generalization capabilities of our model. In the task allocation example above it might be possible to assign two work items to two processors in the same environment step. The order of the assignments does not matter here and modeling both $Q(s_t, (a_i, a_j))$ and $Q(s_t, (a_j, a_i))$ explicitly increases the sample complexity of the learning algorithm and can negatively impact generalization performance.

3 Build Order Planning in *StarCraft*

3.1 Background

To showcase our proposed combination of a partial simulators and RL for planning, we apply them to the real-time strategy (RTS) game *StarCraft: Brood War* for the task of build order planning. In this context, a build order prescribes a set of characters (units) or technologies to produce at each time step. The desired result of executing a build order is a composition of units that – subject to each player's skill at handling them – makes it possible to defeat the opponent. The key challenge in producing a winning build order is accurately allocating resources to economic, technological, or military investments, subject to payoffs which vary based on the opponent's partially hidden strategy. For example, producing more gatherers can result in affording a larger army later, but producing too few combat units may result in immediate defeat.

The implementation of a build order plan, i.e. the translation to concrete game actions, is commonly done using a heuristic planner obeying the constraints imposed by the game mechanics: resource requirements, production time or prerequisite units or technologies [3]. In our experiments, we're utilizing the build order planner in the open-source *StarCraft* bot CherryPi[2]. Starting from a list of items to produce, the planner emits a plan up to a specified horizon, specifying which of the items can be produced at which point in the future and which units and upgrades will have been completed or will still be in production at the planning horizon (see Figure 4 for an example plan). During the search phase, the planner resolves dependencies and performs optimizations by scheduling productions in parallel when feasible. Resource income is approximated with linear functions based on the number of available worker units and resource depots.

It is important to point out that build order planners employed in *StarCraft* (including the one we use here) do *not* model state changes due to actions of the opponent player but assume unhindered execution of the plan. In reality, the combat with the opponent may eliminate some of our units or interfere with resource acquisition and unit construction. Adapting to these changes, or to newly gained information on the opponent's strategy, requires either frequent re-evaluation of the current plan or for re-planning from scratch in short intervals. The planner we are considering here implements the latter method and re-plans at regular intervals.

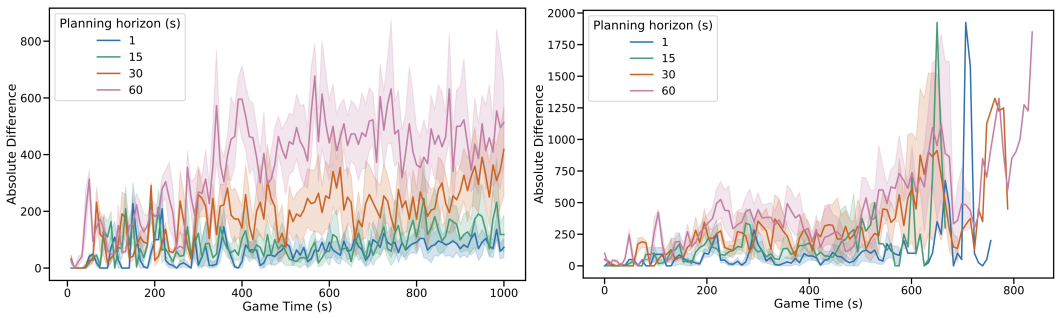
To further illustrate the point above, we measure the accuracy of CherryPi's planner in two scenarios using several planning horizons (Figure 5). In each setup, we construct build orders at step t with a horizon h up to $t + h$. We then execute the resulting build order and re-plan frequently as usual. At game time $t + h$, we measure the absolute difference in unit counts and upgrades between the target state of the planner from time t and the real state at time $t + h$ with a weighted sum account for respective production costs[3] of an item. In Figure 1a we play without an opponent so that inaccuracies of the plan stem solely from the approximations performed by the planner. For comparison, Figure 1b shows the average planning accuracy over 10 two-player games against another bot. Planning errors are now also a result of the opponent's actions and we can observe higher discrepancies.

3.2 Related Work

Traditionally, build order research in RTS games like *StarCraft* focused on efficient planning engines [3], generating fixed strategies [1, 11] or reactive planning to combat derivations from the initial plan due to opponent actions [26]. For continuous planning over the course of a game, [6] proposes an evolutionary algorithm to build orders, which are then evaluated with a hand-crafted

[2] `https://github.com/TorchCraft/TorchCraftAI`

[3] Production costs in *StarCraft* are measured in minerals and gas; the total cost is the sum of both amounts.



(a) Autobuild in isolation

(b) Autobuild in a real game

Figure 1: Accuracy of the build order planner used in CherryPi and with our model. We compare the difference between the target state and the true state when playing (a) without any opponent interaction and (b) against another bot. Note the difference in scale on the y-axis.

fitness function. In [23] the authors predict the build order of the *opponent* player using a Bayesian model and motivate its usage for guiding future strategic decisions.

Recently, RL approaches have been used to switch between several rule-based build orders during a game [4] and to directly predict build actions for each environment step using Q-Learning [24]. A supervised learning approach using replays of human games was presented in [7]: a neural network is trained on pairs of states and build actions, and during evaluation a build order manager queries the model for a next action to perform (multiple actions can be performed in one game step).

The approach presented in this work uses a tree-search planning component to efficiently construct valid and optimized build orders spanning up to a minute of game time. Re-planning the build order frequently enables adaptation to the current game situation. Instead of evaluating future unit compositions with hand-crafted heuristics, we learn a value function with RL as described in the following section. We will demonstrate in the experimental section that both the planning and the learning component are crucial for obtaining good performance.

3.3 A Partial Simulator for Build Order Planning

Considering the formulation introduced in §2.2, we consider a native action space $\mathcal{A}$ that is a subset of all possible build actions, e.g. for producing a unit, researching an upgrade or expanding at another base location; each action is subject to different resource constraints and duration (see Appendix A for a full listing). We define the feature space $\mathbb{R}^d$ as a d-dimensional vector of descriptive features that our partial model can predict, e.g. unit counts; see §4.1 for an exhaustive enumeration. The true state features $\Phi(s_t)$ are unavailable due to partial observability while $\Psi(o_t)$ is obtained directly from the game interface. We differentiate between the unit counts of our player $\Psi^o(o_t)$ and the (observable) enemy unit counts $\Psi^e(o_t)$ as our partial model g only accounts for feature changes directly caused by our own build actions. The build order planner provides the partial model g_h, i.e. the planner implicitly unrolls a sequence of build actions up to a horizon h and produces $\Psi^o(o_{t+h})$. As detailed above, $\Psi^o(o_{t+h})$ is, for large h, subject to high uncertainty in real games.

Our planning procedure now consists of a tree search among possible action sequences $(a_i)_{i=1}^n$. Note that our environment allows parallel execution of actions when feasible, because the *StarCraft* game engine can be instructed to perform a different command for each controllable unit at each time step. The length of action sequences obtained via planning is thus dependent on the planning horizon as well as the specific state s_t. The value of a tree node is determined by performing rollouts with a random succession of additional actions, applying g_h to the resulting sequences of build actions and scoring the predicted features $\Psi^o(o_{t+h})$ for each rollout with a Q-Function $Q(s_t, \Psi^o(o_{t+h}))$. To improve the valuation of future unit compositions, we model $Q(s_t, \Psi^o(o_{t+h}))$ with a two-stage neural network which first integrates $(\Psi(o_j))_{j=1}^t$ into an embedding e_t using a recurrent network and then predicts Q from the concatenation of e_t and $\Psi^o(o_{t+h})$. The search procedure is terminated after a fixed number of traversals with UCT, or when the tree is fully explored. To accelerate the planning

procedure, we leverage our planner's ability to construct build orders and the corresponding rollouts incrementally. As a result, each node in the search tree is assigned a corresponding planner state.

4 Experiments

4.1 Model and Features

We model the Q-Function over our action abstractions as a two-stage neural network as described in §3.3. We first featurize observations into $\Psi(o_t)$ by accumulating per-type unit counts in disjoint channels for allied and enemy units, scaled by their approximate in-game value [22] and a factor of 10^{-3}. Upgrades and technologies are represented as a binary vector. The number of bases is passed in numerical form. Resources are provided in a logarithmic scale of $\log(0.2x + 1)$. As further categorical features we consider the current game time (in minutes) and the index of the map in the pool. The featurized action abstraction $\Psi^o(o_{t+h})$ produced by our partial model is limited to allied unit counts and resources and additionally includes the presence of a base expansion action in the current plan. We differentiate between completed units and units being produced using disjoint channels. Multiple units can be produced within planning horizons we examine; to discriminate between producing units earlier and later we include $\Psi^o(o_{t+h'})$ and $\Psi^o(o_{t+h''})$ for $h', h'' \leq h$ in the featurization[4].

The state encoder consists of dedicated 8-dimensional embeddings for each categorical feature, which are then concatenated with the remaining features. A 512-cell LSTM [5] forms the core component of the encoder, and its output is subject to a linear projection resulting in a 128-dimensional state embedding e_t. A new observation is presented to the encoder every 5 seconds of game time. The second part of the model estimates $Q(e_t, \Psi^o(o_{t+h}))$ and consists of a single linear combination followed by a sigmoid activation function.

4.2 Training

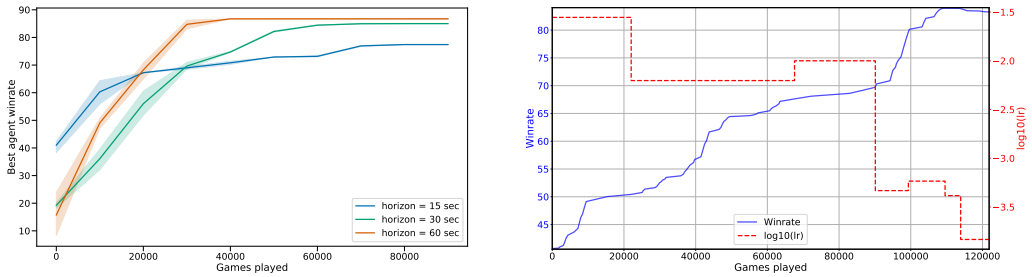
We collect experience for training the Q-Function by playing full games against a variety of rule-based community bots, considering Zerg vs Zerg matchups only. We play on a variety of maps and provide opponents with the opportunity to adapt between games as to maximize the strategies and game situations the model gets exposed to (we refer to Appendix B for a detailed description of the setup for training games). During training games, we use the latest model for planning as-is and rely solely on MCTS for exploration. The model is trained to predict the return of each $(s_t, \Psi^o(o_{t+h}))$ pair that results from the final build order plan, which corresponds to 1 for wins and 0 for games that are lost or timed out (i.e. we only supply a final reward and use $\gamma = 1$). Gradients are computed on batches of 256 state-action sequences corresponding to entire games. Truncated back-propagation through time (BPTT) is not applied as the timeout imposed on training games limits the sequence length of samples to 360. We optimize a binary cross-entropy loss and use Adam [9] to compute parameter updates.

We tune the learning rate via population-based training (PBT) [12] with a population size of 8. The fitness of models is estimated based on the rate of winning their respective training games. Models check-point their parameters every 60 updates; after all but one population member produces 2 check-points (this corresponds to 30720 games with a batch size of 256), we rank all members according to their fitness and cull the worst-performing 50% of models. In parallel to each training run, we continuously evaluate the performance of the best check-points as determined by training win rates (Appendix B). New members are initialized from the best-performing model check-point according to the parallel evaluation runs, and their learning rate is sampled from log-space in $[10^{-5}, 10^{-2})$. A plot of PBT families (or lineages) can be found in Figure 5. Finally, we use the result of the evaluation runs to determine the best check-points for the entire PBT run.

4.3 Baselines

In order to better judge the efficacy of our approach, we present two baseline methods for comparison. The *heuristic* baseline uses MCTS settings that are identical to the experiments described above.

[4] specifically, we include features at 2, 5 or 15 and 5, 15 or 30 seconds when planning for 15, 30 or 60 seconds, respectively.



(a) Performance of the best models found so far during PBT training, across planning horizons 15, 30, 60 seconds. Horizon 15sec starts by learning faster, 30 and 60 seconds converge to a higher win rate.

(b) Learning rate and (smoothed) win rate of the one model (the best overall) during PBT training.

Figure 2: Win rate during population-based training, and evolution of learning rate.

Instead of learning a Q-Function we estimate the value of the planner output with a heuristic score, which is a (scaled-down) weighted sum of the predicted unit counts at the planning horizon by their respective approximate in-game value [22]. The second baseline does not plan into the future, instead solely estimating the value of action sequences which can be started within the next second. MCTS fully explores all possible action sequences in this case. We refer to this variant as a *policy* baseline, even though it has a planning horizon of 1 second. We prefer this baseline to a pure policy model as it has exactly the same model and action space (which is an advantage over a pure policy model as it can take structured simultaneous actions) as when we plan to further horizons.

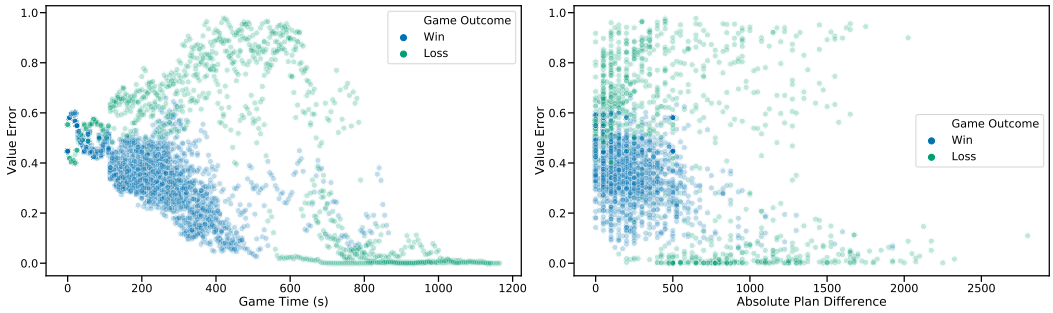
4.4 Results

Influence of the MCTS exploration and planning horizon: We display in Table 1 the evaluation win rates for a trained model across varying values of c, the exploration hyperparameter from the UCT formula $v_i + c\sqrt{\frac{\ln N}{n_i}}$ (as in [10]); and across varying values of r, the number of rollouts performed by MCTS. While more rollouts generally improve performance, 10k shows diminishing returns vs. 5k so we use 5k in all other experiments. As for c, given a high enough r (and $c > 0$) there is no strong difference in evaluation win rates. But, we found it helpful to consider c a hyperparameter of optimization, similar to the learning rate, which we did in all of our other experiments.

Influence of the planning horizon and comparison to baselines: In Table 2 we show the training and evaluation win rates of the baseline *CherryPi (bot)* (using its built-in, rule-based build orders), of the same planning procedure used with a *heuristic* value, of the same model but planning only one second ahead (that we call a direct *policy* model), and of our *model* across different planning horizons, planning frequencies (how often we replan), and with pre-trained value models or not. First, we notice that the best results are achieved with the approach we present in this paper, that our models perform better than the planner with a *heuristic* value and that a trained value model (used as *policy*) without planning. Note that the 29% win rate (that autobuild with a heuristic value function can achieve) is already non trivial to reach. We observe no significant difference between 30 and 60 seconds for our best models at test time, although Figure 2a shows a marginally faster training time with a longer horizon. We observed that pre-training the value model (off-policy, on dumped game states and build order planner predictions) improves training speed as well, but pre-trained models do not achieve as high a final win rate.

Table 1: Evaluation (test) win rates across levels of exploration (c) in MCTS and number of rollouts (r), with a model trained with $c = 2.82$, at planning horizon 30sec, replanning frequency 5sec.

c	r	WR
0	500	53.4
0.7	500	72.2
1.41	500	64.7
2.1	500	62.2
2.82	500	62.6
0	1000	69.9
0.7	1000	81.6
1.41	1000	80.9
2.1	1000	79.0
2.82	1000	76.6
0	5000	83.7
1.7	5000	84.4
1.41	5000	85.7
2.1	5000	84.3
2.82	5000	84.9
0	10000	83.9
0.7	10000	85.4
1.41	10000	86.7
2.1	10000	84.1
2.82	10000	84.3



(a) Errors by game time

(b) Errors by simulation error

Figure 3: Visualization of Q-Function errors depending on game time and error of the features predicted by the build order planner. 45 of the 50 games used to generate the samples are were won. Details on planning error estimation are provided in §3.1.

Table 2: Results of our value model across different planning horizon and with and without pre-training, compared to baseline methods. We report the best win rate of the top 3 models after population-based training.

Name	Does it plan?	Learned value?	Planning Horizon	Planning Frequency	Pre-trained	Win rate Train	Win rate Test
CherryPi	yes	no	variable	625ms	-	-	84.4
Heuristic	yes	no	15s	5s	-	-	29.8
Heuristic	yes	no	30s	5s	-	-	24.8
Heuristic	yes	no	60s	5s	-	-	29.4
Policy	no	yes	1s	5s	no	58.9	62.3
Policy	no	yes	1s	2.5s	no	68.1	63.1
Policy	no	yes	1s	5s	yes	61.3	59.8
Model	yes	yes	15s	5s	no	80.6	86.6
Model	yes	yes	30s	5s	no	84.6	**90.6**
Model	yes	yes	30s	5s	yes	80.7	75.8
Model	yes	yes	60s	5s	no	87.1	**90.5**

Do the value estimation errors come from the partial simulator or value model? Figure 3 visualizes the errors of our best model across 50 games against the "Microwave" bot. While initial and final predictions are accurate, losses are misjudged in the middle of the game, and this error in value estimation obviously has a consequence on our behavior. Related to that, in Figure 3b, the green points (losses) at the bottom (low value error, high absolute plan difference) grossly correspond to the green points at time of more than 600 seconds in Figure 3, when we are sure to have lost already. The other mass of green points in Figure 3b (high value error) denotes that the losses are significantly more frequent in states spaces where our partial simulator induces errors (w.r.t. the true state in the future), hinting that our value model is somewhat dependant on a consistent partial state estimation – if not on a correct one.

5 Conclusion

In this work, we demonstrated the usage of an imperfect, partial simulator to enable planning in an adversarial and partially observable environment, and for coping with a combinatorial action space via action abstraction. This action abstraction can be represented by featurizing the predicted state or state changes that are consequences of a plan. Our results show that planning can be effective even if the output of the partial model used for simulation introduces inaccuracies; however, large simulation errors usually result in wrong valuation of plans. By comparing representative baselines and several variants of our model, we underline that, for high-level strategy generation in RTS games, estimating

a value function via reinforcement learning and planning with a sufficient horizon are crucial for good performance.

In the current setup, the model is not informed about the opponent it is facing, or about its past games and results against it. This prevents the planning component from generating a better informed opening build order to start a new game with, until the model gets some information about the opponent's state (by scouting). It would be interesting to investigate how adaptation across games can be added to our approach to alleviate this issue, for instance by feeding the model with some (summary) state of past games in a series. Future extension of this work in the setting of real-time strategy games would benefit from self-play during training time, as there is a visible ceiling in performance that can be obtained by training purely against rule-based bots.

References

[1] Jason Blackford and Gary Lamont. The real-time strategy game multi-objective build order problem. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[2] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2): 57–83, 2002.

[3] David Churchill and Michael Buro. Build order optimization in StarCraft. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[4] Jonas Gehring, Da Ju, Vegard Mella, Daniel Gant, Nicolas Usunier, and Gabriel Synnaeve. High-Level Strategy Selection under Partial Observability in StarCraft: Brood War. In *Reinforcement Learning under Partial Observability workshop, NeurIPS*, 2018.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] Niels Justesen and Sebastian Risi. Continual online evolutionary planning for in-game build order adaptation in StarCraft. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 187–194. ACM, 2017.

[7] Niels Justesen and Sebastian Risi. Learning macromanagement in starcraft from replays using deep learning. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 162–169. IEEE, 2017.

[8] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014. URL `http://arxiv.org/abs/1412.6980`. arXiv: 1412.6980.

[10] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[11] Matthias Kuchem, Mike Preuss, and Günter Rudolph. Multi-objective assessment of pre-optimized build orders exemplified for starcraft 2. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[12] Ang Li, Ola Spyra, Sagi Perel, Valentin Dalibard, Max Jaderberg, Chenjie Gu, David Budden, Tim Harley, and Pramod Gupta. A generalized framework for population based training. *arXiv preprint arXiv:1902.01894*, 2019.

[13] Michael L Littman and Richard S Sutton. Predictive representations of state. In *Advances in neural information processing systems*, pages 1555–1561, 2002.

[14] Joelle Pineau, Nicholas Roy, and Sebastian Thrun. A hierarchical approach to pomdp planning and execution. In *Workshop on hierarchy and memory in reinforcement learning (ICML)*, volume 65, page 51, 2001.

[15] Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. Point-based value iteration: An anytime algorithm for pomdps. In *IJCAI*, volume 3, pages 1025–1032, 2003.

[16] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, pages 2164–2172, 2010.

[17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[18] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[19] Jost Tobias Springenberg, Karol Hausman, Martin Riedmiller, Nicolas Heess, and Ziyu Wang. Learning an embedding space for transferable robot skills. In *International Conference on Learning Representations (ICLR)*, 2018.

[20] Alexander L Strehl and Michael L Littman. A theoretical analysis of model-based interval estimation: Proofs. Technical report, Forthcoming tech report, Rutgers University, 2005.

[21] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[22] Gabriel Synnaeve. *Bayesian programming and learning for multi-player video games*. PhD thesis, Université de Grenoble, 2012.

[23] Gabriel Synnaeve and Pierre Bessiere. A Bayesian model for plan recognition in RTS games applied to StarCraft. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[24] Zhentao Tang, Dongbin Zhao, Yuanheng Zhu, and Ping Guo. Reinforcement learning for build-order production in starcraft ii. In *2018 Eighth International Conference on Information Science and Technology (ICIST)*, pages 153–158. IEEE, 2018.

[25] Guy Tennenholtz and Shie Mannor. The natural language of actions. *arXiv preprint arXiv:1902.01119*, 2019.

[26] Ben George Weber, Michael Mateas, and Arnav Jhala. Applying goal-driven autonomy to StarCraft. In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2010.

[27] Ronald J Williams and Leemon C Baird. Tight performance bounds on greedy policies based on imperfect value functions. Technical report, Citeseer, 1993.

A Action Space for Build Order Planning

We consider a subset of all possible build actions as we target a single matchup (Zerg vs Zerg). table 3 contains a full listing and includes construction time (i.e. the duration of the action) and requirements. "Supply" is provided by Overlord units that are added to the build order by the planner when necessary. All units require a Larva unit to be morphed from, which are automatically provided by Hatcheries. Structures (Hatchery, Spawning Pool, Spire) require a Drone to be produced (the Drone will morph into the structure). The Lair is morphed from an existing Hatchery which will continue to function as a resource depot and source of Larvae afterwards.

Table 3: List of build actions considered during planning.

		Resources			
Action	Time (s)	Minerals	Gas	Supply	Dependencies
Unit_Drone	13	50	0	1	
Unit_Zergling	18	25	0	0.5	Spawning Pool
Unit_Scourge	19	25	75	0.5	Lair, Drone, Spire
Unit_Mutalisk	25	100	100	2	Lair, Drone, Spire
Unit_Extractor	25	50	0	0	Drone
Unit_Hatchery	75	300	0	0	Drone
Upgr_Metabolic_Boost	63	100	100	0	Drone, Spawning Pool
Expand	Unit_Hatchery placed at the next base location				

B Game Protocols

We consider different protocols for running games during training, continuous check-point evaluation and for obtaining final performances:

- Training games are played in series of length 25. Between games, opponents have the opportunity to adapt by storing and loading data from disk after and before a game, respectively. This ensures exposure to a variety of game situations during training as opponents may adapt their strategy based on previous games.
- When continuously evaluating check-points during training, we use shorter series of two games only to ensure a fast turn-around time and run 15 such series for each opponent in parallel.
- The above evaluation method is still subject to relatively high variance; the final performance for a run is thus obtained by running 45 two-game series against each opponent with the 3 best check-points as determined by continuous evaluation. We then report the maximum win rate of these runs.

All games are played on the AIIDE tournament (yearly *StarCraft* AI competition) map pool comprising of 10 maps[5] and with a game time limit of 30 minutes during training and on-the-side evaluation. Games exceeding the time limit are stopped and registered as losses.

[5] `https://skatgame.net/mburo/sc2011/rules.html`

C Opponent Bots

Table 4: List of opponent bots considered. SSCAIT versions were obtained from the public SSCAIT ladder[6]. Bots denoted as "SSCAIT*" have been been removed or replaced by newer versions by their respective authors.

Bot Name	Version/Source
AILien	SSCAIT/AIIDE2017
Arrakhammer	SSCAIT/AIIDE2017
BlackCrow	SSCAIT
Microwave	SSCAIT*
NLPRBot_CPAC	SSCAIT/AIIDE2017
Pineapple_Cactus	SSCAIT
Proxy	SSCAIT*
Steamhammer	SSCAIT*
Tscmoo	Provided by author
ZZZKBot	SSCAIT

D Example Build Order Plan

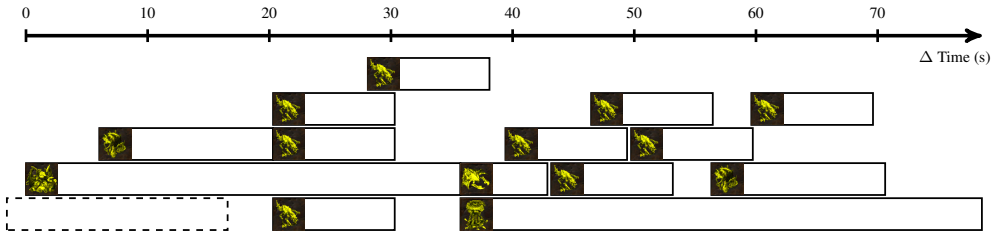


Figure 4: Example build order plan for a planning horizon of 60 seconds. The time scale represents the distance of actions from current game time. This shows dependencies among items, e.g. the supply provided by the Overlord unit that is built starting from 6s is required for producing the 3 Zerglings at 20s. The dotted rectangle depicts an item that is already in production at the start of planning.

E Example of a Population Based Training Run

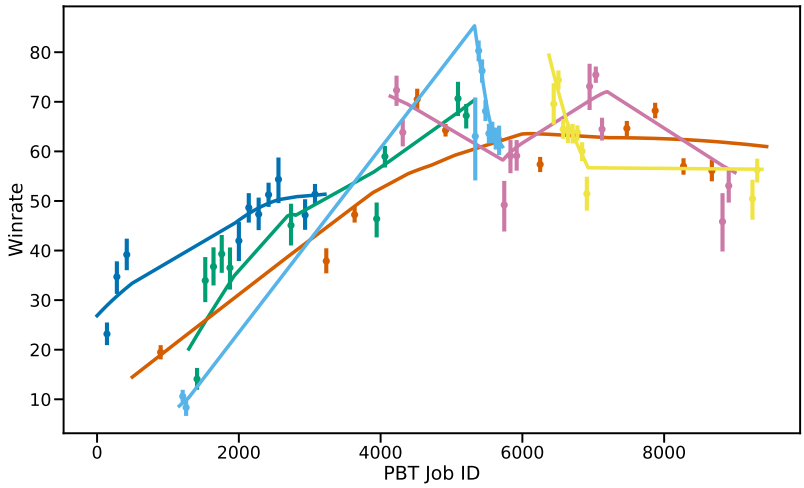


Figure 5: Training win rates of of lineages of agents (that can have different hyperparameters but continue/fork training from the same model parameters) during population based training. Error bars depict 95% confidence intervals, lines are locally weighted scatterplot (linear) smoothing.