A Appendix

A.1 StarCraft II

Starcraft is a real-time strategy game in which players compete to control a shared map by gathering resources and building units and structures. The game has several modes, such as team games or custom maps. For instance, the StarCraft Multi-Agent Challenge [38] is an increasingly popular task for Multi-Agent Reinforcement Learning and includes a collection of tasks. In this paper, we consider StarCraft II as a two-player game, as this is the primary setting for StarCraft II, and it is played at all levels, from casual online games to professional e-sport. It combines high-level reasoning over long horizons with fast-paced unit management. There are numerous strategies for StarCraft II with challenging properties presenting cycles and non-transitivity, especially since players start the game by selecting one of three alien *races*, each having fundamentally different mechanics, strengths and weaknesses. The game is played on large *maps* which have different terrain. Evaluation of the agents can be done by playing against human players, including professional players, the built-in bots, scripted bots from the community, or even the stronger online RL agents such as AlphaStar [50] or TStarBot [19].

While human players use their mouse and keyboard to play the game, the agents use an API, called the *raw interface*, which is detailed in the Appendix (Section A.3). In this interface, observations are split into three parts, namely `world`, `units` and `scalar_inputs` which are enough to describe the full observation. The actions are encoded into 7 parts, called *arguments*: `function`, `delay`, `queued`, `repeat`, `unit_tags`, `target_unit_tag` and `world`. It is important to note that the arguments are not independent of each other, in particular the `function` determines which other arguments are used.

A.2 Dataset

To train our agents we apply filtering to the publicly available 20 million StarCraft II games to get higher quality data. We use versions 4.8.2 to 4.9.2 and restrict the games to more skilled players with an MMR greater than 3500. This leaves us with around 1.4M games (Table 4). These replays span over two different balance patches, introducing some subtle differences in the rules of StarCraft II between the older and the more recent games, which are small enough to be ignored[8]. In addition, the map pool changed once during this period, so the games are played on a total of 10 different maps.

The average 2-player game is about 11 minutes long and has around 15482 frames in total which poses a significant modeling challenge by making both supervised and offline RL training harder and slower. Hence, the agent observes only the frames that have actions which cuts the effective length of the episode by 12 times. Additional statistics on the data used in the train set can be found on Table 4.

The replays are provided by Blizzard and hosted on their servers. The data is anonymized, and does not contain personal information about the players.

Table 4: Statistics of the replays used in the training set.

Number of replays	1388233
Number of frames across games	$2.1e10$
Number of frames with actions across episodes	$3.5e9$
Total game play time	30.4 years

A.3 StarCraft II Interface

StarCraft II features large maps on which players move their units. They can also construct buildings (units which cannot move), and gather resources. At any given time, they can only observe a subset of the whole map through the *camera*, as well as a coarse, zoomed-out version of the whole map called the *minimap*. In addition, units have a *vision* field such that any unit owned by the other player is hidden unless it is in a vision field. The player also receives some additional information, such as

[8] However, the version of the game is available for each episode, so one could decide to condition the agent on the game version.

the quantity of resources owned or some details about the units currently selected, and audible cues about events in the game. We call this the *standard* interface. Players issue orders by first selecting units, then choosing an ability, and lastly, for some actions, a target, which can be on the camera or the minimap.

Our agents use an API that is exposed as a *raw* interface. It consists of three categories of observations:

- **World**: A single tensor called *world* which corresponds to the minimap shown to human players. It is observed as 8 feature maps with resolution 128x128. The feature maps are:
  - `height_map`: The topography of the map, which stays unchanged throughout the game.
  - `visibility_map`: The area within the vision of any of the agent's units.
  - `creep`: The area of the map covered by Zerg "creep".
  - `player_relative`: For each pixel, if a unit is present on this pixel, this indicates whether the unit is owned by the agent, the opponent, or is a neutral construct.
  - `alerts`: This encodes alerts which show on the minimap of the standard interface, for instance when units start to fight.
  - `pathable`: The areas of the map which can be used by ground units.
  - `buildable`: The areas of the map which where buildings can be placed.
  - `virtual_camera`: The area currently covered by the virtual camera. The virtual camera restricts detailed vision of many *Unit* properties, as well as restricts some actions from targeting units/points outside the camera. It can be moved as an action.
- **Units**: A list of units observed by the agent. It contains all of the agent's units as well as the opponent's units when within the agent's vision and the last known state of opponent's buildings. For each unit, the observation is a vector of size 43 containing all the information that would be available in the game interface. In particular, some of the opponent's unit information are masked if they are not in the agent's virtual camera. In addition, the list also contains entries for *effects* which are temporary, localized events in the game (although effects are not units in a strict sense, they can be represented as such). In our implementation, this list can contain up to 512 entities. In the rare event that more than 512 units/effects exist at once, the additional observations will be truncated and not visible to the agent.
- **Scalar inputs**: *Scalar inputs* are global inputs available in game interface. They are:
  - `player_id`: The id of the player (0 or 1). This is not useful to the agent.
  - `minerals`: The amount of minerals currently owned.
  - `vespene`: The amount of vespene gas currently owned.
  - `food_used`: The current amount of food currently used by the agent's units. Different units use different amount of food and players need to build structures to provide more food.
  - `food_cap`: The current amount of food currently available to the agent.
  - `food_used_by_workers`: The current amount of food currently used by the agent's workers. Workers are basic units which harvest resources and build structures, but rarely fight.
  - `food_used_by_army`: The current amount of food currently used by the agent's non-worker units.
  - `idle_worker_count`: The number of workers which are idle. Players typically want to keep this number down.
  - `army_count`: The number of units owned by the agent which are not workers.
  - `warp_gate_count`: The number of warp gates owned by the agent (if the agent's race is Protoss).
  - `larva_count`: The number of larva currently available to the agent (if the agent's race is Zerg).
  - `game_loop`: The number of internal game steps since the beginning of the game.
  - `upgrades`: The list of upgrades currently unlocked by the agent.
  - `enemy_upgrades`: todo

- `unit_counts`: The number of each unit currently owned by the agent. This information is contained in the units input, but represented in a different way here.
- `home_race`: The race of the agent.
- `away_race`: The race of the opponent. If the opponent is has chosen a random race, it is hidden until one of their unit is observed for the first time.

In the raw interface, actions are also different. Each raw actions combine up to three standard actions: unit selection, ability selection and target selection. In practice, each raw action is subdivided into up to 7 parts, called *arguments*, which are taken successively:

1. **Function**: This corresponds to the ability part of the StarCraft II API, and specifies the action. Examples include: `Repair`, `Train_SCV`, `Build_CommandCenter` or `Move_Camera`.
2. **Delay**: In theory, the agent could take an action at each environment step. However, since StarCraft II is a real-time game, the environment steps are very quick[9] and therefore it would not be fair to humans, which cannot issue action that fast. In addition, it would make episodes extremely long which is challenging for learning. Therefore the agent specifies how many environment steps will occur before the next observation-action pair. Throttling is used to make sure the agent cannot issue too many actions per second.
3. **Queued**: This argument specifies whether this action should be applied immediately, or queued. This corresponds to pressing the Shift key in the game interface.
4. **Repeat**: Some repeated identical actions can be issued very quickly in the standard interface, by pressing keyboard keys very fast, sometimes even issuing more than one action per environment step. The repeat argument lets the agent repeat the action up to 4 times in the same step. This is mainly useful for building lots of Zerg units quickly.
5. **Unit tags**: This is the equivalent of a selection action in the standard interface. This argument is a mask over the agent's units which determines which units are performing the action. For instance for a `Repair` action, the unit tags argument specify which units are going to perform the repair.
6. **Target Unit Tag**: Which unit an action should target. For instance, a `Repair` action needs a specific unit/building to repair. There are no actions in StarCraft II that target more than one unit (some actions can affect more than one unit, but specify a target point).
7. **World**: Which point in the world this action should target. It is a pair of (x, y) coordinates aligned with the world observation. For example `Move_Camera` needs to know where to move the camera.

A.4 Architecture

The model we use for our baselines is illustrated on Figure 3. It is derived from the model used in [50], with some improvements. As described in Sections A.1 and A.3, the StarCraft II API has three types of inputs and output. We structure the model around them:

- Scalar inputs are embedded into a 1-d vector by the scalar encoder. The scalar arguments of the previous action taken are also embedded here. This vector is transformed by successive MLP blocks. This is a memory-less model, but in our ablation studies (and in [50]) this is also where we add the LSTM modules. Eventually, logits are produced from this vector to generate the `function` argument of the action. The sampled action is embedded again, and the next argument (`delay`) is sampled, followed by `queued` and `repeat`.
- Unit features, as well as the Units arguments of the previous action, are embedded into a 1-d vector per unit, resulting in a fixed-length list of embedding (as well as a mask indicating which elements of the list are not used). They are processed by three layers of transformers, then used to produce the `unit_tags` and `target_unit_tag` arguments, through pointer networks [52]. To generate the `unit_tags` argument, which can select up to 64 units, we iteratively apply the pointer network 64 times, so conceptually, the `unit_tags` represent 64 arguments.

[9] 22.4 steps per second.

- World features are embedded into feature planes. The world argument of the previous action is not embedded, as we found this leads to overfitting. The embedded world features are processed by Residual Convolutional Networks (ResNet) [20]. Part of the ResNet downscales the input into 128 feature planes with size 16×16. The world action is obtained by upscaling the feature planes into a single plane of logits, with resolution 256×256 (to match the resolution of the game world).

Information flows between the scalar, units and world part of the model through special operations. After the transformer layers, units vectors are aggregated (averaged) and merged into the embedding of the scalar inputs. Units are also embedded into the world feature planes through a *scatter* operation, which merges the embedding of each unit with the corresponding (x, y) coordinate of the feature planes. Feature planes are down-scaled and reshaped into a 1-d vector to be merged with the embedding of the scalar inputs. Pointer networks naturally merge scalar embeddings (they form the query) with units embeddings (they form the keys). Finally, the scalar embedding, along with the selected `unit_tags` argument, are reshaped into the world feature planes. Note that the `world` argument is not conditioned on the `target_unit_tag`, because no action in the API can use a `target_unit_tag` and a `world` argument at the same time.

We will open-source our architectures and the exact hyperparameters and details of our architecture can be found in the released agent.

A.5 Details on Evaluation Metric

Let us assume that we are given an agent to evaluate p and a collection of reference agents

$$\mathrm{Q} = \{\mathrm{q}_j\}_{j=1}^{N}.$$

Each of these players can play all three races of StarCraft II:

$$\mathrm{R} = \{\mathrm{terran}, \mathrm{protoss}, \mathrm{zerg}\}.$$

We define an outcome of a game between a player p and a reference player q as

$$\mathrm{f}(\mathrm{p}, \mathrm{q}) \stackrel{\text{def}}{=} \mathbb{E}_{\mathrm{r_p}, \mathrm{r_q} \sim U(\mathrm{R})} \mathrm{P}[\mathrm{p\ wins\ against\ q} | \mathrm{r}(\mathrm{p}) = \mathrm{r_p}, \mathrm{r}(\mathrm{q}) = \mathrm{r_q}],$$

where $\mathrm{r}(\cdot)$ returns a race assigned to a given player, and the probability of winning is estimated by playing matches over uniformly samples maps and starting locations.

A.5.1 Robustness computation

We define robustness of an agent p with respect to reference agents Q as

$$\mathrm{robustness}_{\mathrm{Q}}(\mathrm{p}) \stackrel{\text{def}}{=} 1 - \min_{\mathrm{q} \in \mathrm{Q}} \mathrm{f}(\mathrm{p}, \mathrm{q}).$$

Note, this is 1 minus exploitability [10], simply flipped so that we maximise the score. In particular, Nash equlibrium would maximise this metric, if Q contained every mixed strategy in the game.

A.5.2 Elo computation

We follow a standard Chess Elo model [12], that tries to predict f by associating each of the agents with a single scalar (called Elo rating) $\mathrm{e}(\cdot) \in \mathbb{R}$ and then modeling outcome with a logistic model:

$$\widehat{\mathrm{f_{Elo}}}(\mathrm{p}, \mathrm{q}) \stackrel{\text{def}}{=} \frac{1}{1 + 10^{[\mathrm{e}(\mathrm{p}) - \mathrm{e}(\mathrm{q})]/400}}.$$

For consistency of the evaluation we have precomputed ratings $\mathrm{e}(\mathrm{q})$ for each $\mathrm{q} \in \mathrm{Q}$. Since ratings are invariant to translation, we anchor them by assigning $\mathrm{e}(\mathrm{very\_hard}) := 1000$.

In order to compute rating of a newly evaluated agent p we minimise the cross entropy between true, observed outcomes f and predicted ones (without affecting the Elo ratings of reference agents):

$$\mathrm{Elo}_{\mathrm{Q}}(\mathrm{p}) \stackrel{\text{def}}{=} \arg\min_{\mathrm{e}(\mathrm{p})} \left[-\sum_{\mathrm{q}} \mathrm{f}(\mathrm{p}, \mathrm{q}) \log \left(\widehat{\mathrm{f_{Elo}}}(\mathrm{p}, \mathrm{q}) \right) \right] = \arg\max_{\mathrm{e}(\mathrm{p})} \sum_{\mathrm{q}} \mathrm{f}(\mathrm{p}, \mathrm{q}) \log \left(\widehat{\mathrm{f_{Elo}}}(\mathrm{p}, \mathrm{q}) \right).$$

Note, that as it is a logistic model, it will be ill defined if $\mathrm{f}(\mathrm{p}, \mathrm{q}) = 1$ (or 0) for all q (one could say Elo is infinite). In such situation the metric will be saturated, and one will be in a need of expanding Q to continue research progress.

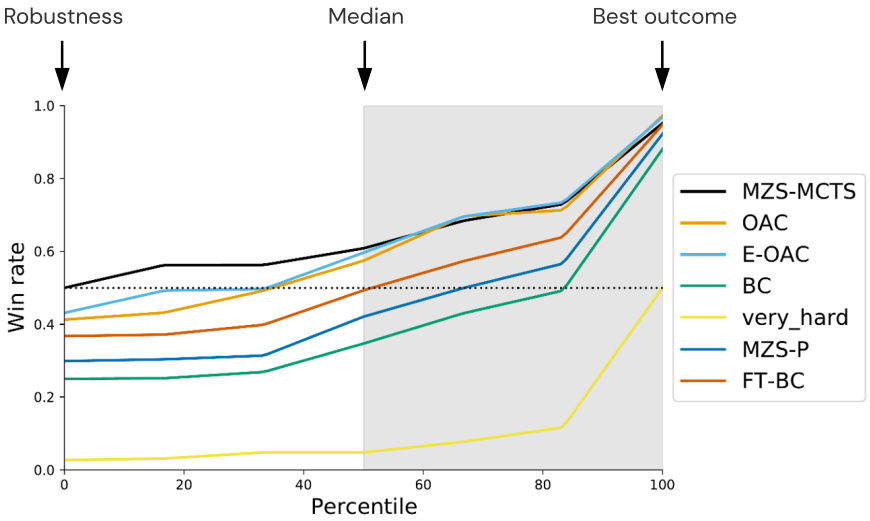

Figure 4: Percentile curve of reference agents.

B Percentile curve

A percentile curve [47] allows us to look in a single picture at the whole spectrum of outcomes of our agents. Instead of aggregating scores $\mathrm{f}(\mathrm{p}, \mathrm{q})$ we plot percentiles of these values in Figure 4, and can define that an agent is better than another agent if and only if it pareto dominates it over percentiles between 0 and 50 (domination beyond 50, which represents median score would simply put weight on overspecialisation towards a tiny fraction of reference opponents). In particular robustness metric from previous sections is a percentile 0 on this curve. It is worth noting, that this approach imposes a partial ordering, some agents might be not comparable, if one is more robust, and the other is better in median. Maximisation of the progress using it would thus encourage development of agents which are both robust, and good on average. When it comes to our reference agents, they can be fully ordered using percentile curves, placing MZS-MCTS at the top, followed by E-OAC, OAC, FT-BC, MZS-P, BC and finally `very_hard` bot, an ordering consistent with the one given by robustness.

B.1 Implementation details

In this section we provide pseudo-code for our methods, as well as the hyperparameters used for training. In the algorithms, the data at a time step t is denoted as X. We refer to different part of the data X: X_s is the observation at time t, X_a is the action at time t, X_r is the reward at time t, X_R is the MC return (in the case of StarCraft II, this is equal to the reward on the last step of the episode), and $X_{\text{game loop delta}}$ is the number of internal game steps between $t-1$ and t (and 0 on the first step).

B.1.1 Behaviour Cloning

Algorithm 1 Behaviour Cloning (with rollout length K set to 1)

Inputs: A dataset of trajectories $\mathcal{D}$, a mini batch size M, an initial learning rate λ, the total number of observations processed n_{frames}, and the initial weights used θ to parameterise the estimated policy $\hat{\mu}_\theta$.
for $i = 0..n_{\text{frames}}/M - 1$ **do**
  Set the gradient accumulator $g_{\text{acc}} \leftarrow 0$.
  for $j = 0..M - 1$ **do**
    Sample a trajectory $T \sim \mathcal{D}$
    Sample k in $[0, length(T) - K]$
    Set $X \leftarrow T[k]$
    Set $g_{\text{acc}} \leftarrow g_{\text{acc}} + \frac{1}{M} \cdot \frac{\partial L_{\text{cross entropy}}(\hat{\mu}_\theta(\cdot|X_s), X_a)}{\partial \theta}$,
    where X_s, X_a are the observation and action parts of X, respectively.
  end for
  Set $\lambda_i \leftarrow \frac{\lambda}{2}\left(\cos\left(\frac{i\pi}{n_{\text{frames}}/M-1}\right) + 1\right)$
  Set $\theta \leftarrow \theta - \lambda_i \cdot Adam(g_{\text{acc}})$
end for
return θ

The BC agent is trained by running Algorithm 1 with the following inputs: $\mathcal{D}$ is the full set of episodes with MMR>3500, $M = 32,768$, $\lambda = 10^{-3}$, $n_{\text{frames}} = 10^{10}$, θ is initialized randomly.

The FT-BC agent is trained by running Algorithm 1 with the following inputs: $\mathcal{D}$ is the set of episodes with MMR>6200 and a winning outcome, $M = 32,768$, $\lambda = 10^{-5}$, $n_{\text{frames}} = 10^{9}$, θ is initialized set as the weights of the trained BC agent.

B.1.2 Offline Actor-Critic

Algorithm 2 Behaviour Cloning with value function training (with rollout length K set to 1)

Inputs: A dataset of trajectories $\mathcal{D}$, a mini batch size M, an initial learning rate λ, the total number of observations processed n_{frames}, and the initial weights used θ to parameterise the estimated policy $\hat{\mu}_\theta$ and value function $V^{\hat{\mu}_\theta}$.
for $i = 0..n_{\text{frames}}/M - 1$ **do**
  Set the gradient accumulator $g_{\text{acc}} \leftarrow 0$.
  for $j = 0..M - 1$ **do**
    Sample a trajectory $T \sim \mathcal{D}$
    Sample k in $[0, length(T) - K]$
    Set $X \leftarrow T[k]$
    Set $g_{\text{acc}} \leftarrow g_{\text{acc}} + \frac{1}{M} \cdot \left(\frac{\partial L_{\text{cross entropy}}(\hat{\mu}_\theta(\cdot|X_s), X_a)}{\partial \theta} + \frac{\partial L_{\text{MSE}}(V^{\hat{\mu}}(X_s), X_r)}{\partial \theta}\right)$,
    where X_s, X_a are the observation and action parts of X, respectively.
  end for
  Set $\lambda_i \leftarrow \frac{\lambda}{2}\left(\cos\left(\frac{i\pi}{n_{\text{frames}}/M-1}\right) + 1\right)$
  Set $\theta \leftarrow \theta - \lambda_i \cdot Adam(g_{\text{acc}})$.
end for
return θ

Algorithm 3 Offline Actor-Critic (with fixed critic V^{μ})

Inputs: A dataset of trajectories $\mathcal{D}$, the mini batch size M, the rollout length K, an initial learning rate λ, the weights of the estimated behavior policy and value function from BC θ_0 (such that $\hat{\mu}_{\theta_0}$ is the BC policy, and $V^{\hat{\mu}_{\theta_0}}$ is the behaviour value function), the total number of observations processed n_{frames}, the bootstrap length N, the IS threshold $\bar{\rho}$, a per-game step discount γ_0.
Set $\theta \leftarrow \theta_0$
for $i = 0..n_{\text{frames}}/M - 1$ **do**
  Set the gradient accumulator $g_{\text{acc}} \leftarrow 0$.
  for $j = 0..M - 1$ **do**
    Sample a trajectory $T \sim \mathcal{D}$
    Sample k in $[0, length(T) - K]$
    Set $X \leftarrow T[k : k + K - 1]$
    Compute the TD errors δ and clipped IS ratios $\bar{\rho}$ with clipping threshold $\hat{\rho}$
    for $t = 0..K - 2$ **do**
      Set $\delta[t] \leftarrow X_r[t+1] + \gamma_{t+1} V^{\hat{\mu}_{\theta_0}}(X_s[t+1]) - V^{\hat{\mu}_{\theta_0}}(X_s[t])$
      Set $\bar{\rho}[t] \leftarrow \min(\bar{\rho}, \frac{\pi_\theta(X_a[t]|X_s[t])}{\hat{\mu}_{\theta_0}(X_a[t]|X_s[t])})$
      Set $\gamma[t] \leftarrow \gamma_0^p$ where $p = X_{\text{game\_loop\_delta}}[t]$.
    end for
    Compute the V-Trace targets v:
    for $t = 0..K - N - 1$ **do**
      Set $v[t+1] \leftarrow V^{\hat{\mu}_{\theta_0}}(X_s[t+1]) + \sum_{u=t}^{t+N-1} (\prod_{k=t}^{u-1} \bar{\rho}[k]\gamma[k+1])\, \bar{\rho}[u]\delta[u]$
    end for
    Set $g_{\text{acc}} \leftarrow g_{\text{acc}} + \sum_{t=0}^{N-1} \bar{\rho}[t](X_r[t+1] + \gamma[t+1]v[t+1] - V^{\hat{\mu}_{\theta_0}}(X_s[t])) \frac{\partial \log \pi_\theta(X_a[t]|X_s[t])}{\partial \theta}$.
  end for
  Set $\lambda_i \leftarrow \frac{\lambda}{2}\left(\cos\left(\frac{i\pi}{n_{\text{frames}}/M-1}\right) + 1\right)$
  Set $\theta \leftarrow \theta - \lambda_i \cdot Adam(g_{\text{acc}})$
end for
return θ

The OAC agent is trained by first running Algorithm 2 to obtain θ_V, with the following parameters: $\mathcal{D}$ is the full set of episodes with MMR>3500, $M = 32,768$, $\lambda = 10^{-4}$, $n_{\text{frames}} = 10^9$, θ is initialized with the weights of the BC agent.
Then we run Algorithm 3 with the following parameters: $\mathcal{D}$ is the full set of episodes with MMR>3500, $M = 512$, $K = 64$, $\lambda = 2 \cdot 10^{-5}$, $n_{\text{frames}} = 10^9$, θ_0 is set to θ_V, $N = 32$, $\bar{\theta} = 1$, $\gamma_0 = 0.99995$.

B.1.3 N-steps Emphatic Traces

Algorithm 4 Emphatic Offline Actor-Critic (with fixed critic V^{μ})

Inputs: A dataset of trajectories $\mathcal{D}$, the mini batch size M, the rollout length K, an initial learning rate λ, the weights of the estimated behavior policy and value function from BC θ_0 (such that $\hat{\mu}_{\theta_0}$ is the BC policy, and $V^{\hat{\mu}_{\theta_0}}$ is the behaviour value function), the total number of observations processed n_{frames}, the bootstrap length N, the IS threshold $\bar{\rho}$, a buffer $\mathcal{B}$ containing M empty lists, a per-game step discount γ_0, initial emphatic traces $\forall j < N, F[j] = 1$.
Set $\theta \leftarrow \theta_0$
for $i = 0..n_{\text{frames}}/M - 1$ **do**
  Set the gradient accumulator $g_{\text{acc}} \leftarrow 0$.
  for $j = 0..M - 1$ **do**
    if $\mathcal{B}[j]$ has less than $K + 1$ elements **then**
      Sample $T \sim \mathcal{D}$
      $\mathcal{B}[j] \leftarrow \text{concatenate}(\mathcal{B}[j], T)$
    end if
    Set $X \leftarrow \mathcal{B}[j][0 : K + 1]$
    Set $\mathcal{B}[j] \leftarrow \mathcal{B}[j][K :]$
    Compute the TD errors δ, clipped IS ratios $\bar{\rho}$ and V-Trace targets v with clipping threshold $\hat{\rho}$ as in Alg. 3
    Compute the emphatic trace F:
    for $t = 0..K - N - 1$ **do**
      Set $F[t] = \prod_{p=1}^{N}(\gamma[t - p + 1]\hat{\rho}[t - p])F[t - N] + 1$
    end for
    Set $g_t = \bar{\rho}[t](X_r[t + 1] + \gamma[t + 1]v[t + 1] - V^{\hat{\mu}_{\theta_0}}(X_s[t]))\frac{\partial \log \pi_\theta(X_a[t]|X_s[t])}{\partial \theta}$
    Set $g_{\text{acc}} \leftarrow g_{\text{acc}} + \sum_{t=0}^{N-1} F[t]g_t$
  end for
  Set $\lambda_i \leftarrow \frac{\lambda}{2}\left(\cos\left(\frac{i\pi}{n_{\text{frames}}/M-1}\right) + 1\right)$
  Set $\theta \leftarrow \theta - \lambda_i \cdot Adam(g_{\text{acc}})$
end for
return θ

The E-OAC agent uses the same BC agent as the OAC training in the previous section, that is, θ_0 is also set to θ_V. Then we run Algorithm 4 with the same hyper-parameters as the OAC agent. However, unlike the OAC agent, it uses sequentially ordered trajectories in their order of interactions with the MDP, and reweight the policy gradient updates with the emphatic trace F.

B.1.4 *MuZero*

In this section we will provide further details about our *MuZero* algorithm applied in the Offline RL setting, which we call *MuZero Supervised*.

MuZero Supervised is trained using supervised learning on the dataset described in Section A.2. Trajectories of length K are sampled from games from the dataset consisting of K consecutive observations, actions, and rewards starting at some time t, i.e. $(x_t, a_t, x_{t+1}, a_{t+1}, ..., x_{t+K}, a_{t+K})$, along with the player-relative game result $r \in \{1, 0, -1\}$ corresponding to whether the player won, drew, or lost the game. For simplicity we remove t and use superscripts to refer to this trajectory, $(x^0, a^0, x^1, a^1, x^2, a^2, ..., x^k, a^k)$.

The architecture consists of a representation function $s^0 = h_\theta(x^0)$ which computes the internal state given the initial observation, a dynamics function $s^{k+1} = g_\theta(s^k, a^k)$ which computes the next internal state given the current internal state and action, and a prediction function $\pi^k, v^k = f_\theta(s^k)$ which predicts the policy and the value function.

We used AlphaStar's encoders and action prediction functions in the *MuZero* architecture. AlphaStar's action prediction functions are notably autoregressive to handle the complex and combinatorial action space of StarCraft II. We predict and embed the full action in the representation function, *i.e.* for the root node in MCTS. To improve inference time, we only predict the function and delay components

of the action in the prediction heads in the model, *i.e.* for non-root nodes in MCTS. We found that using as few as 20 actions for *Sampled MuZero* worked well, and increasing it did not improve performance.

The model parameters θ are estimated by minimizing the following loss

$$\mathcal{L}_{\textit{MuZero}}(\theta, x^0, a^0, ..., a^{k-1}, x^n) = \sum_{k=0}^{K-1} \left[l^v(z^k, v^k) + l^\pi(a^k, \pi^k) \right]. \tag{2}$$

Here, a^k is the action taken at time $t+k$ in the dataset. z^k is the n-step bootstrapped return. Since StarCraft II does not have non-terminal rewards, this is simplified into either the value function prediction at time $t+n$, v^-_{t+n}, if the game length is at least $t+n$, or the final game result r if the game length is smaller than $t+n$. We use a target network θ^- which is updated to θ every 100 steps, to compute the bootstrapped target value v^-_{t+n}. We found that $n = 512$ worked best, which is notably large and roughly corresponds to half of the average game length. Unlike the offline actor critic, we only estimate the behavioural value function v_μ and thus don't need any off-policy corrections.

Note that here we differ from *MuZero* and *MuZero Unplugged* by training with action and value targets obtained from the offline dataset. *MuZero* and *MuZero Unplugged* use the result of MCTS as the action and value targets.

We use Adam optimizer [27] with additive weight decay [30], with a cosine learning rate schedule starting from 1×10^{-3} and dropping to zero at the end of training.

Algorithm 5 MuZero Supervised

Inputs: A dataset of trajectories $\mathcal{D}$, the mini batch size M, the rollout length K, the temporal difference target distance n, an initial learning rate λ, the total number of observations processed n_{frames}, and the initial weights used θ to parameterise the representation function h_θ, the dynamics function g_θ, and the prediction function f_θ.
for $i = 0..n_{\text{frames}}/M - 1$ **do**
  Set the gradient accumulator $\nabla_{\text{acc}} \leftarrow 0$.
  for $j = 0..M-1$ **do**
    Sample a trajectory $T \sim \mathcal{D}$
    Sample k in $[0, length(T))$
    Set $X \leftarrow T[k : k+K]$
    Set $X_{\text{TD}} \leftarrow T[k+n]$
    Set $\nabla_{\text{acc}} \leftarrow \nabla_{\text{acc}} + \frac{1}{M} \cdot \frac{\partial \mathcal{L}_{\textit{MuZero}}(\theta, X_s[k], X_a[k:k+K], X_{\text{TD}})}{\partial \theta}$.
  end for
  Set $\lambda_i \leftarrow \frac{\lambda}{2}\left(\cos\left(\frac{i\pi}{n_{\text{frames}}/M-1}\right)+1\right)$
  Set $\theta \leftarrow \theta - \lambda_i \cdot Adam(g_{\text{acc}})$.
end for
return θ

B.2 Additional Figures

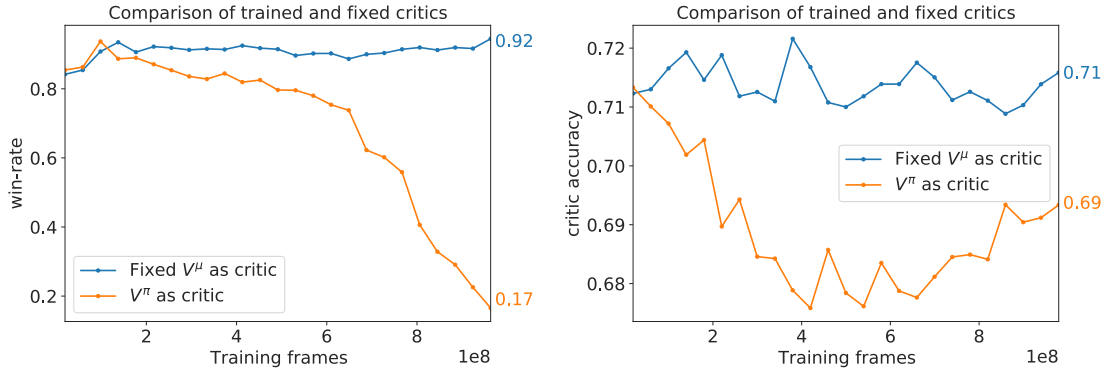

Figure 5: Comparison of critics for the OAC agent. Keeping the critic fixed as V^{μ} leads to stable training, whereas using V^{π} degenerates. Left: performance against the `very_hard` bot, the peak performance is similar. Right: critic accuracy.

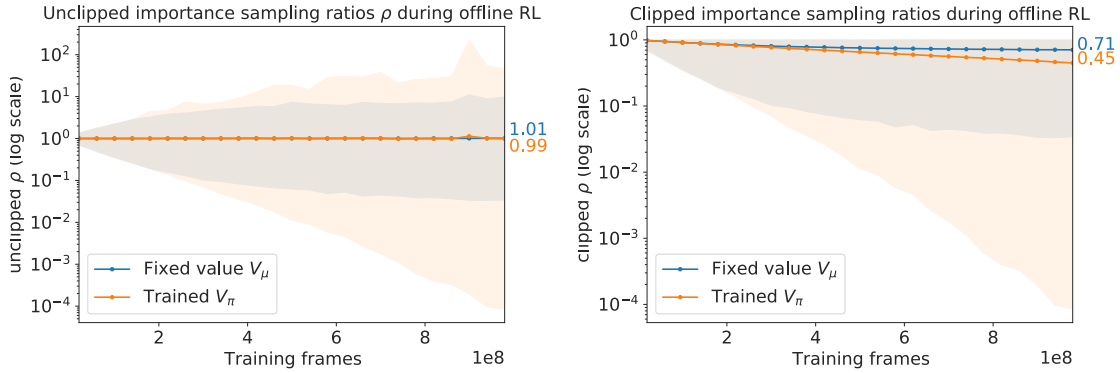

Figure 6: Evolution of the importance sampling ratios over the course of offline RL using V-Trace Actor-Critic. The solid part represents the spanning interval between the lowest and highest values. Left: unclipped $\hat{\rho}$. Right: clipped $\hat{\rho}$, used to train the policy π.

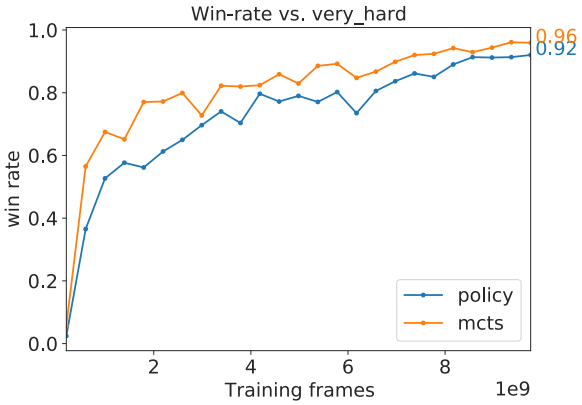

Figure 7: Win-rate of the *MuZero* agent against the `very_hard` bot of the behavioural policy (μ) and MCTS using the behvaioural policy μ and behavioural value function v_μ, as a function of data consumed while training. Notably MCTS performance exceeds policy performance throughout training.