Vector Quantized Models for Planning

Sherjil Ozair [*1,2] **Yazhe Li** [*1] **Ali Razavi** [1] **Ioannis Antonoglou** [1] **Aäron van den Oord** [1] **Oriol Vinyals** [1]

Abstract

Recent developments in the field of model-based RL have proven successful in a range of environments, especially ones where planning is essential. However, such successes have been limited to deterministic fully-observed environments. We present a new approach that handles stochastic and partially-observable environments. Our key insight is to use discrete autoencoders to capture the multiple possible effects of an action in a stochastic environment. We use a stochastic variant of *Monte Carlo tree search* to plan over both the agent's actions and the discrete latent variables representing the environment's response. Our approach significantly outperforms an offline version of MuZero on a stochastic interpretation of chess where the opponent is considered part of the environment. We also show that our approach scales to *DeepMind Lab*, a first-person 3D environment with large visual observations and partial observability.

1. Introduction

Making predictions about the world may be a necessary ingredient towards building intelligent agents, as humans use these predictions to devise and enact plans to reach complex goals (Lake et al., 2017). However, in the field of reinforcement learning (RL), a tension still exists between model-based and model-free RL. Model-based RL and planning have been key ingredients in many successes such as games like chess (Shannon, 1950; Silver et al., 2017a), Go (Silver et al., 2016b; 2017b), and Poker (Moravčík et al., 2017; Brown et al.). However, their applicability to richer environments with larger action and state spaces remains limited due to some of the key assumptions made in such approaches. Other notable results have not used any form of model or planning, such as playing complex video games Dota 2 (OpenAI et al., 2019) and StarCraft II (Vinyals et al., 2019), or robotics (OpenAI et al., 2018).

In this work we are motivated by widening the applicability of model-based planning by devising a solution which removes some of the key assumptions made by the MuZero algorithm (Schrittwieser et al., 2019). Table 1 and Figure 1 summarize the key features of model-based planning algorithms discussed in this paper. MuZero lifts the crucial requirement of having access to a perfect simulator of the environment dynamics found in previous model-based planning approaches (Silver et al., 2017a; Anthony et al., 2017). In many cases such a simulator is not available (eg., weather forecasting), is expensive (eg., scientific modeling), or is cumbersome to run (e.g. for complex games such as Dota 2 or StarCraft II).

However, MuZero still makes a few limiting assumptions. It assumes the environment to be deterministic, limiting which environments can be used. It assumes full access to the state, also limiting which environments can be used. The search and planning is over future agent(s) actions, which could be millions in environments with complex action spaces. The search occurs at every agent-environment interaction step, which may be too fine grained and wasteful.

Largely inspired by both MuZero and the recent successes of VQVAEs (van den Oord et al., 2017; Razavi et al., 2019) and large language models (Radford et al.; Brown et al., 2020), we devise VQ models for planning, which in principle can remove most of these assumptions.

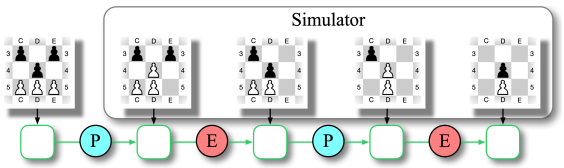
Our approach uses a state VQVAE and a transition model. The state VQVAE encodes future observations into discrete latent variables. This allows the use of *Monte Carlo tree search* (MCTS, (Coulom, 2006)) for planning not only over

Table 1. Key features of different planning algorithms.

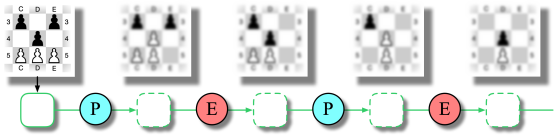
Method	Learned Model	Agent Perspective	Stochastic	Abstract Actions	Temporal Abstraction
AlphaZero	✗	✗	✗	✗	✗
Two-player MuZero	✓	✗	✗	✗	✗
Single-player MuZero	✓	✓	✗	✗	✗
VQHybrid	✓	✓	✓	✗	✗
VQPure	✓	✓	✓	✓	✗
VQJumpy	✓	✓	✓	✓	✓

[*]Equal contribution [1]DeepMind, London, United Kingdom [2]Mila, University of Montreal. Correspondence to: Sherjil Ozair <sherjilozair@deepmind.com>, Yazhe Li <yazhe@deepmind.com>.

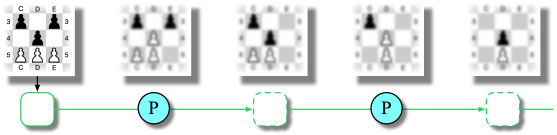
Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).



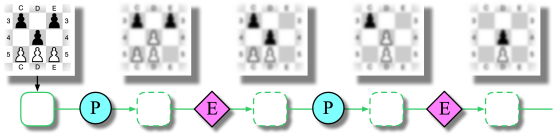
(a) AlphaZero



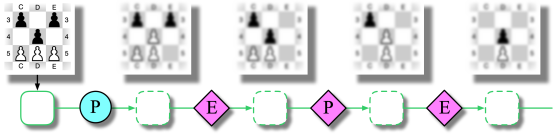
(b) Two-player MuZero



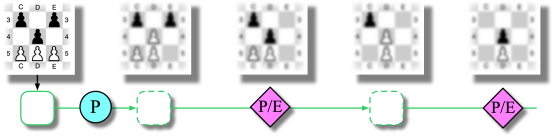
(c) Single-player MuZero



(d) VQHybrid



(e) VQPure



(f) VQJumpy

Figure 1. **Comparison of model-based planning formulations.** **(c)** AlphaZero plans with both player and opponent actions and the groundtruth states with the help of a simulator; **(b)** Two-player version of MuZero plans with both player and opponent actions; **(c)** Single-player MuZero plans only with the player actions; **(d)** VQHybrid plans with player actions and discrete latent variables; **(e)** VQPure plans with the player action for the first step and discrete latent variables thereafter; **(f)** VQJumpy plans with discrete latent variables that expand for more than a single agent step. **Notations:** The rounded squares denote states. Circles are actions in real action space. Diamonds are discrete latent variables. P denotes the player's action. E denotes the environment's action.

future actions, but also over future observations, thus allowing planning in stochastic or partially-observed environments.

We also propose a "pure" version of our model which encodes both future observations and actions into discrete latent variables, which would allow planning entirely in discrete latent variables. These discrete latent variables can be designed to have cardinality and time granularity independent of the actions, thus enabling planning in large action spaces over longer time horizons.

To demonstrate that the proposed solution works well in practice, we devise two evaluation frameworks using the game of chess. The first is the classic *two-player* chess framework where agents get to observe and plan over both their own and their opponent's actions. This is the framework used by techniques like MuZero and its previous iterations. In this setting, all four assumptions above perfectly hold, i.e. action and time granularity is already at the right level of abstraction, there is no stochasticity, and agents observe the full board. Then we remove the ability to enumerate opponent actions, and make the opponent's move part of the environment dynamics. That is agents can only observe their own states and actions. This makes the environment stochastic, since the transition is not a deterministic function but depends on an unknown opponent's action from a potentially stochastic policy. We refer to this framework as *single-player* chess.

We show that MuZero's performance drops catastrophically on *single-player* chess compared to *two-player* chess, demonstrating that MuZero depends on access to data from the opponent's perspective. Our approach which only uses player perspective data at training and playing time performs as well on *single-player* chess as MuZero does on *two-player* chess. This suggests that our approach is a promising way to generalize MuZero-style planning to partially-observable and stochastic environments.

To investigate how well our approach can scale, we also evaluate on DeepMind Lab (Beattie et al., 2016), which has a complex observation and action space with partial observability and stochasticity, showing that VQ planning models capture the uncertainty and offer the best results among all planning models we compare against.

2. Related Work

Models for Planning Oh et al. (2015); Chiappa et al. (2017); Kaiser et al. (2019) use video prediction models as environment simulators. However, such models are not feasible for planning since they require observation reconstruction which would make planning prohibitively slow. van Hasselt et al. (2019) argue that experience replay can be regarded as a non-parametric model and that Dyna-based

methods are unlikely to outperform model-free methods. Schrittwieser et al. (2019); Oh et al. (2017) learn an implicit deterministic sequence model by predicting future reward, value and policy from current states and future actions. However, these models are in principle limited to deterministic or weakly stochastic environments such as Atari (Machado et al., 2018).

Stochastic models promise to capture uncertainty. PILCO (Deisenroth & Rasmussen, 2011) used Gaussian processes for transition model and achieves remarkable sample efficiency by capturing model uncertainty. However, it is not scalable to high dimensional state spaces. Depeweg et al. (2016) model uncertainty of the transition function with Bayesian neural networks (BNNs). Kurutach et al. (2018); Chua et al. (2018) use model ensembles to capture epistemic uncertainty that arise from scarcity of data. Variational autoencoders (VAE, Kingma & Welling (2013); Rezende et al. (2014)) have fuelled a range of stochastic models for RL. Moerland et al. (2017) builds models for RL with conditional VAE (Sohn et al., 2015). Buesing et al. (2018) investigate stochastic state-space models. Ha & Schmidhuber (2018) train a VAE to compress the observation into continuous latent variables and use an RNN to serve as the predictive model. Hafner et al. (2018; 2019) learn a full forward model using VAE framework. They incorporate multi-step prediction ("latent overshooting") to minimize compounding errors, which changes the optimization objective, while our approach uses data likelihood as the only objective. Hafner et al. (2020) propose a discrete autoencoders model and learn it with straight-through gradients. This is perhaps the most similar to our approach. However, the model is used to generate synthetic data and not used for planning. Lastly, Rezende et al. (2020) study environment models from a causal perspective. They propose adding stochastic nodes using *backdoors* (Pearl et al., 2016). This approach requires the backdoor variable to be observed and recorded during data generation. Our approach doesn't alter the data generating process, therefore works in offline RL setting.

Model-based policies Models can be used in different ways to materialize a policy. Oh et al. (2015); Kaiser et al. (2019) use environment models in the Dyna (Sutton & Barto, 2018) framework, which proposes to learn a policy with model-free algorithms using synthetic experiences generated by models. However, the accumulated error in synthesized data could hurt performance compared to an online agent. This loss of performance has been studied by van Hasselt et al. (2019). Ha & Schmidhuber (2018); Hafner et al. (2018) do policy improvement through black-box optimization such as CMA-ES, which is compatible with continuous latent variable models. Henaff et al. (2017) extends policy optimization to discrete action space. Following AlphaGo (Silver et al., 2016b) and AlphaZero (Silver et al., 2017a), Schrittwieser et al. (2019) prove that MCTS is scalable and effective for policy improvement in model-based learning. Our approach is a generalization of MuZero that is able to incorporate stochasticity and abstract away planning from agent actions. Continuous action spaces and MCTS have also been combined with some success, e.g. (Couëtoux et al., 2011) and (Yee et al., 2016). However, our choice of discrete latent space makes it possible to leverage all the recent advances made in MCTS. In a specific multi-agent setup, where the focus is to find policies that are less exploitable, models can be used with counterfactual regret minimization (Moravcík et al., 2017) or fictitious play (Heinrich & Silver, 2016) to derive Nash equilibrium strategy.

Offline RL While our model-based approach is applicable generally, we evaluate it in the *offline RL* setting. Previous model-based offline RL approaches (Argenson & Dulac-Arnold, 2020; Yu et al., 2020; Kidambi et al., 2020) have focused on continuous control problems (Tassa et al., 2020; Gulcehre et al., 2020). Our work focuses on environments with large observation spaces and complex strategies which require planning such as chess and DeepMind Lab (Beattie et al., 2016).

3. Background

Vector-Quantized Variational AutoEncoders (VQVAE, van den Oord et al. (2017)) make use of vector quantization (VQ) to learn discrete latent variables in a variational autoencoder. VQVAE comprises of neural network encoder and decoder, a vector quantization layer, and a reconstruction loss function. The encoder takes as input the data sample $\mathbf{x}$, and outputs vector $\mathbf{z}_u = f(\mathbf{x})$. The vector quantization layer maintains a set of embeddings $\{\mathbf{e}_k\}_{k=1}^K$. It outputs an index c and the corresponding embedding $\mathbf{e}_c$, which is closest to the input vector $\mathbf{z}_u$ in Euclidean distance. The decoder neural network uses the embedding $\mathbf{e}_c$ as its input to produce reconstructions $\hat{\mathbf{x}}$. The full loss is $\mathcal{L}^t = \mathcal{L}^r(\hat{\mathbf{x}}, \mathbf{x}) + \beta\|\mathbf{z}_u - sg(\mathbf{e}_c)\|^2$, where $sg(\cdot)$ is the stop gradient function. The second term is the commitment loss used to regularize the encoder to output vectors close to the embeddings so that error due to quantization is minimized. The embeddings are updated to the exponential moving average of the minibatch average of the unquantized vectors assigned to each latent code. In the backwards pass, the quantization layer is treated as an identity function, referred to as straight-through gradient estimation (Bengio et al., 2013). For more details, see van den Oord et al. (2017).

Monte Carlo Tree Search (MCTS, Coulom (2006)) is a tree search method for estimating the optimal action given access to a simulator, typically used in two-player games. MCTS builds a search tree by recursively expanding the tree

and assessing the value of the leaf node using Monte Carlo (MC) simulation. Values of leaf nodes are used to estimate the Q-values of all the actions in the root node.

The policy for child node selection during expansion is crucial to the performance of MCTS. The most popular method for this is UCT (stands for "Upper Confidence Bounds applied to trees", Kocsis & Szepesvári (2006)), which is based on Upper Confidence Bound (UCB, Auer et al. (2002)). UCT suggests that this problem can be seen as a Bandit problem where the optimal solution is to combine the value estimation with its uncertainty.

AlphaGo (Silver et al., 2016a) combined MCTS with neural networks by using them for value and policy estimations. The benefits of this approach are twofold: value estimation no longer incurs expensive Monte Carlo simulations, allowing for shallow searches to be effective, and the policy network serves as context for the tree expansion and limits the branching factor.

At each search iteration, the MCTS algorithm used in AlphaGo consists of 3 steps: selection, expansion and value backup. During selection stage, MCTS descends the search tree from the root node by picking the action that maximizes the following upper confidence bound:

$$\arg\max_a \left[Q(s,a) + P(a|s)U(s,a)\right], \tag{1}$$

where

$$U(s,a) = \frac{\sqrt{N(s)}}{1+N(s,a)}[c_1 + \log(\frac{N(s)+c_2+1}{c_2})], \tag{2}$$

and $N(s,a)$ is the visit counts of taking action a at state s, $N(s) = \sum_b N(s,b)$ is the number of times s has been visited, c_1 and c_2 are constants that control the influence of the policy $P(a|s)$ relative to the value $Q(s,a)$.

Following the action selection, the search tree receives the next state. If the next state doesn't already exist in the search tree, a new leaf node is added and this results in an expansion of the tree. The value of the new leaf node is evaluated with the learned value function. Finally, the estimated value is backed up to update MCTS's value statistics of the nodes along the descending path:

$$Q^{t+1}_{tree}(s,a) = \frac{Q^t_{tree}(s,a)N^t(s,a) + Q(s,a)}{N^t(s,a)+1}. \tag{3}$$

Here $Q^t_{tree}(s,a)$ is the action value estimated by the tree search at iteration t; $N^t(s,a)$ is the visit count at iteration t.

MuZero Schrittwieser et al. (2019) introduce further advances in MCTS, where a sequential model is learned from trajectories $\{s_0, a_0, \ldots, s_{T-1}, a_{T-1}, s_T\}$. At each timestep, the model uses the trajectory to formulate a *planning path* at timestep t: $s_t, a_t, \ldots, a_{t+M-1}, a_{t+M}$ with s_t being the root state. To simplify the notation, we omit the subscript and use superscript for indexing actions on the planning path. So the same sequence is written as $s, a^0, \ldots, a^{M-1}, a^M$. Given the starting root state s and a sequence of actions $a^{0:m}$, the model outputs a hidden state h^m and predicts action policy π^m, value v^m and reward r^m. The training objective is as follows:

$$\frac{1}{M}\sum_{m=1}^{M}[L^\pi(a^m, \pi(h^m)) + \alpha\mathcal{L}^v(v^m_{target}, v(h^m)) + \beta\mathcal{L}^r(r^m_{env}, r(h^m))] \tag{4}$$

where h^m is the hidden state on the planning path. $L^\pi(a^m, \pi(h^m))$ is the cross entropy loss between the action and learned parametric policy π. $\mathcal{L}^v$ is the value loss function, v_{target} is the value target and v is the value prediction. $\mathcal{L}^r$ is the reward loss function, r_{env} is the environment reward and r is the reward prediction. α and β are the weights.

During search, π is used as the prior for action selection; r gives the reward instead of using reward from the simulator; v estimates the value of the leaf state rather than using Monte Carlo rollouts.

Comparing to AlphaZero, MuZero model eliminates the need of a simulator to generate the groundtruth state along the planning path. In two player games, MuZero's planning path interleaves the player's action and opponent's action. Whereas in single player version, only player's actions are seen by the model.

4. Model-based Planning with VQVAEs

Our approach uses a state VQVAE model and a transition model. We refer to the full model as VQ Model (VQM), and the resulting agent when combined with MCTS as VQM-MCTS.

We first describe the components of VQM in detail. Then, we explain how the model is used with MCTS.

4.1. VQ Model

Our VQ model is trained with a two-stage training process[1]. We first train the state VQVAE model which encodes the observations into discrete latent variables (Figure 3a). Then we train the transition model using the discrete latent variables learned by the state VQVAE model (Figure 3b).

Notation We use s_t, a_t, and r_t to denote the state at time t, the action following state s_t, and the resulting reward, respectively. An episode is a sequence of interleaved states, actions, and rewards $(s_1, a_1, ..., s_t, a_t, ..., s_T)$.

[1]Training the full model end-to-end is a promising future research direction.

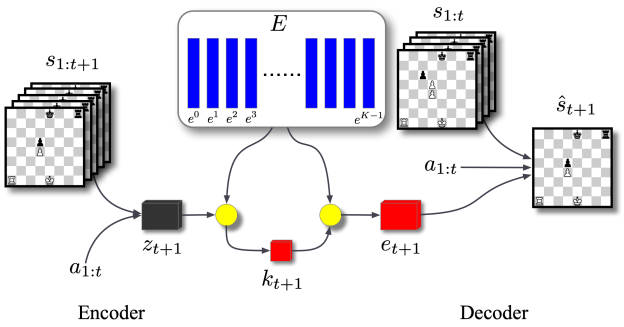


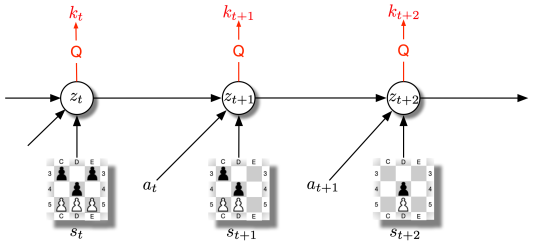
Figure 2. **Complete encoder/decoder architecture of the state VQVAE.** Encoder compresses $s_{1:t+1}$ and $a_{1:t}$ to a continuous latent z_{t+1}. The quantization layer returns the nearest code e_{t+1}, as well as the corresponding index k_{t+1}, in its codebook E. Decoder uses $s_{1:t}$, $a_{1:t}$ and the code $e_{t+1} = E[k_{t+1}]$ to reconstruct s_{t+1}.

State VQVAE The purpose of the state VQVAE is to encode a sequence of states and actions into a sequence of discrete latent variables and actions that can reconstruct back the original sequence. This is done by learning a conditional VQVAE encoder-decoder pair. The encoder takes in states $s_1, ..., s_t, s_{t+1}$ and actions $a_1, ..., a_t$, and produces a discrete latent variable $k_{t+1} = f_{enc}(s_{1:t+1}, a_{1:t})$. The decoder takes in the discrete latent variable and the states and actions until time t and reconstructs the state at time $t+1$, i.e. $\hat{s}_{t+1} = f_{dec}(s_{1:t}, a_{1:t}, k_{t+1})$. Thus, k_{t+1} represents the additional information in the state s_{t+1} given the previous states and actions.

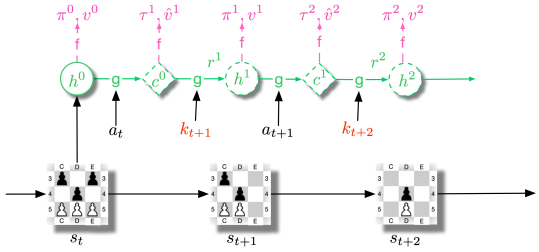
The state VQVAE is trained using the VQVAE technique introduced in van den Oord et al. (2017) (reviewed in Section 3). The cardinality of the discrete latent variable is a design choice. Larger values are more expressive but potentially expensive at search time. Lower values would lead to lossy compression but could potentially yield more abstract representations. We show the effect of cardinality size on reconstruction quality in the supplementary material. Figure 2 depicts the state VQVAE.

Transition Model We obtain latent variable k_t from state s_t using the state VQVAE. We construct a *planning path* which comprises of a state followed by a sequence of interleaved actions and latent variables until a maximum depth M is reached, i.e. $s, a^0, k^1, a^1 \ldots, a^{M-1}, k^M$. Thus, instead of planning over only the agent's actions, this allows us to also plan over the outcomes of those actions.

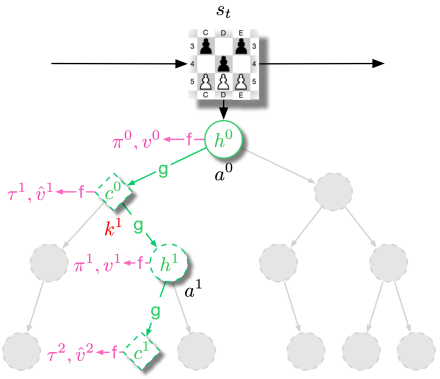
Similar to the environment model of MuZero, our transition model predicts reward and value function at every step. Unlike MuZero, our model not only has a policy head π but also a discrete latent code head τ. We alternate the prediction of action and discrete latent code along the sequence. Note that the value prediction after the discrete latent code



(a) The state VQVAE encodes a sequence of observations s and actions a into discrete latent variables k.



(b) An autoregressive transition model outputs a policy π over the actions, a policy τ over the discrete latent codes and a value function v.



(c) MCTS branches over both actions and state latent variables.

Figure 3. The main components of the proposed agent.

corresponds to an estimate of the state value functions, while value prediction after the action corresponds to an estimation of the Q-function.

To train all the components, and again following MuZero, we use teacher forcing of trajectories generated by a behavior policy (human experts or other agents in all our experiments) based on the observed states s, actions a, and latent variables k. The total loss combining all the prediction losses is

$$\frac{1}{M}\sum_{m=0}^{M-1}\text{CE}(a^m, \pi(h^{2m})) + \frac{1}{M}\sum_{m=1}^{M}\text{CE}(k^m, \tau(h^{2m-1}))$$
$$+\frac{\alpha}{2M}\sum_{m=0}^{2M}\mathcal{L}^v(v^m_{target}, v(h^m)) + \frac{\beta}{2M}\sum_{m=0}^{2M}\mathcal{L}^r(r^m_{env}, r(h^m)),$$

where CE is the cross entropy loss. The total loss is similar to MuZero loss. The main difference is that we also predict latent variables at every odd timestep.

Throughout this Section, we explain our VQM-MCTS based on the VQHybrid *planning path* (Figure 1d). However, VQHybrid is not the only choice available. Depending on the information encoded by the state VQVAE, the *planning path* can be structured differently.

VQPure (Figure 1e) requires a factorized state VQVAE which provides two sets of latent space: one for environment stochasticity, same as in VQHybrid, and the other for the agent's actions. VQPure allows the transition model to decouple from the action space and unroll purely in discrete latent spaces. In contrast, VQHybrid interleaves between state latent variables and actions. For VQJumpy (Figure 1f), the state VQVAE makes a jumpy prediction of the state s_{t+m} instead of predicting the immediate state s_{t+1}.

This enables "temporal abstractions" during planning and provides a principled way to unlock planning in a stochastic environment and decoupling the action space tied to environment both in branching and in time. Although we provide some result for VQPure in our chess experiment, these two *planning path* alternatives remain largely unexplored and left for future works.

4.2. Monte Carlo Tree Search with VQ Planning Model

In order to use the VQ planning model, we modify *Monte Carlo Tree Search* (MCTS) algorithm to incorporate the VQ "environment action". The main difference with the MCTS used in MuZero (reviewed in Section 3) is that, instead of predicting agent (or opponent) actions, our MCTS also predicts next discrete latent variables k given the past. Unlike classical MCTS which anchors in the real action space and imposes an explicit turn-based ordering for multi-agent environment, our MCTS leverages the abstract discrete latent space induced by the state VQVAE.

The search tree in MCTS consists of two types of nodes: action node and stochastic node. During selection stage, MCTS descends the search tree from the root node: for action nodes, Equation 1 is used to select an action; for stochastic nodes,

$$\arg\max_k \left[\hat{Q}(s,k) + P(k|s)U(s,k)\right]$$

is used to select a discrete latent code. We obtain $U(s,k)$ by replacing a with k in $U(s,a)$ from Equation 1. $P(k|s)$ is computed with the learned policy τ of the discrete latent code. $\hat{Q}(s,k)$ can be 0 for a neutral environment, $Q(s,k)$ if the environment is known to be cooperative or $-Q(s,k)$ if the environment is known to be adversarial. When $\hat{Q}(s,k) = 0$, our algorithm is similar to Expectimax search (Michie, 1966; Russell & Norvig, 2009) where the expectation of children's Q-value is computed at the stochastic nodes. In zero-sum games like Chess and Go, we can use the adversarial setting. As we will see in Section 5.1, adding this prior knowledge improves agent performance.

5. Experiments

Our experiments aim at demonstrating all the key capabilities of the VQ planning model: handling stochasticity, scaling to large visual observations and being able to generate long rollouts, all without any performance sacrifices when applied to environments where MCTS has shown state-of-the-art performance.

We conducted two sets of experiments: in Section 5.1, we use chess as a test-bed to show that we can drop some of the assumptions and domain knowledge made in prior work, whilst still achieving state-of-the-art performance; in Section 5.2, with a rich 3D environment (DeepMind Lab), we probe the ability of the model in handling large visual observations in partially observed environment and producing high quality rollouts without degradation.

5.1. Chess

Chess is an ancient game widely studied in artificial intelligence (Shannon, 1950). Although state transitions in chess are deterministic, the presence of the opponent makes the process stochastic from the agent's perspective, when the opponent is considered part of the environment.

5.1.1. Datasets

To evaluate our approach, we follow the two-stage training of VQM and use MCTS evaluation steps illustrated in Section 4. We use the offline reinforcement learning setup by training the models with a fix dataset. We use a combination of Million Base dataset (2.5 million games) and FICS Elo >2000 dataset (960k games)[2]. The validation set consists of 45k games from FICS Elo>2000 from 2017. The histogram of player ratings in the datasets is reported in the supplementary material.

5.1.2. Model Architectures

The state VQVAE uses feed-forward convolutional encoder and decoder, along with a quantization layer in the bottleneck. The quantization layer has 2 codebooks, each of them has 128 codes of 64 dimensions. The final discrete latent is formed by concatenating the 2 codes, forming a 2-hot encoding vector.

The transition model consists of a recurrent convolutional model which either takes an action or a discrete latent code

[2]https://www.ficsgames.org/download.html

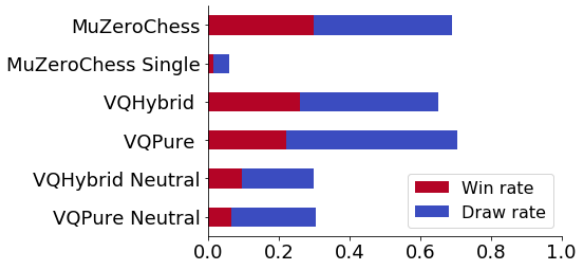


Figure 4. Performance of agents playing against Stockfish 10 skill level 15.

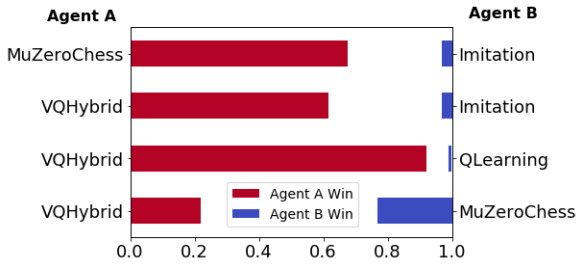


Figure 5. Performance of agents playing against each other.

as input at each unroll step. The model predicts the policies over action and discrete latent code, as well as the value function. We use Monte Carlo return of the game as the target value for training.

5.1.3. Evaluations

The performance of our agent is evaluated by playing against:

1. Stockfish version 10 (T. Romstad) (44 threads, 32G hash size and 15s per move);
2. Q-value agent: Action with the highest Q value is picked. The Q value is computed by unrolling the model for one step and using the learned value function to estimate the value of the next state.
3. Imitation agent: Agent chooses the most probable action according to the learned policy.
4. MuZeroChess agent: MuZero with the same backbone architecture as VQ planning model.

Each agent is evaluated for 200 games playing as white and black respectively for 100 games. We present results for both the worst case and neutral scenario of VQM-MCTS with VQHybrid and VQPure planning path.

5.1.4. Results

Figure 4 reports our main results. Performance of Single-player MuZeroChess (which doesn't observe opponent actions) is considerably worse than two-player MuZeroChess. In addition, Figure 6 shows that using more MCTS simulations hurts single-player MuZeroChess performance, because the model is not correctly accounting for the stochasticity introduced by the unobserved opponent's actions. Both VQHybrid and VQPure agents with worst case chance nodes are able to recover the performance to the same level as the two-player MuZeroChess agent. We then further remove the assumption of the adversarial environment by using neutral case chance nodes during MCTS. The resulting agents, VQHybrid Neutral and VQPure Neutral, don't perform as well as VQHybrid and VQPure. This shows that

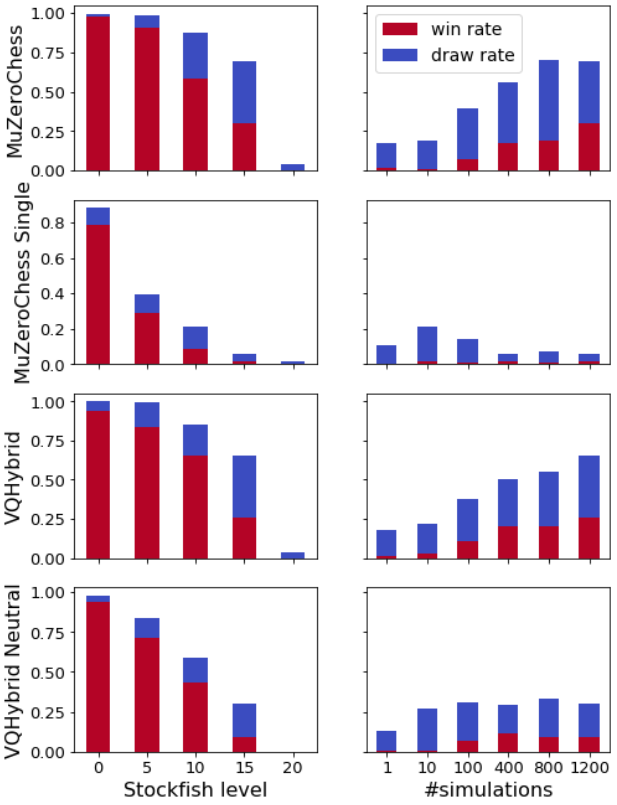


Figure 6. **Agent performance as function of Stockfish strength and simulation budget.** Left column shows win and draw rates of the agent evaluated by playing against different levels of Stockfish 10 with a fixed simulation budget of 1200 per move. Right column shows the impact of simulation budget on agent performance playing against level 15.

prior knowledge of the environment can indeed help the agent perform better.

We also note that when assuming the environment is neutral, increasing the simulation budget doesn't seem to improve the performance (Figure 6). This is because MCTS searches over the expected behavior of players from a broad range of Elo ratings as human data is used for training. This

expectation could deviate significantly from the Stockfish agent, especially for higher skill levels. The full results for the agents can be found in the supplementary material.

In Figure 5, we report win and draw rates of a VQHybrid agent playing against other baseline agents. The results confirm that the VQ agent's performance is on par with two-player MuZeroChess: VQHybrid and MuZeroChess achieve very similar performance playing against the same imitation agent; when playing directly against each other, VQHybrid and MuZeroChess reach similar win rate.

5.2. DeepMind Lab

DeepMind Lab is a first-person 3D environment that features large and complex visual observations, and stochasticity from procedural level generation and partial observability. Due to these properties, it makes for a good environment to test the scalability of our VQ planning model.

5.2.1. Dataset

We used an A2C agent (Mnih et al., 2016) to collect a dataset of $101,325,000$ episodes from the `explore_rat_goal_locations_small` level in *DeepMind Lab*, $675,500$ of which is held out as the test set. Each episode has 128 timesteps in which the agent is randomly spawned in a maze-like environment, which it observes from first-person view. The agent is rewarded when it captures one of the apples that are randomly placed in the environment, at which point it is transported to a new random location in the map. The collection of episodes is started with a randomly initialized agent and is continued as the training of the agent progresses and it learns about the environment. The collected dataset thus comprises a variety of episodes corresponding to different experience level of the A2C agent.

5.2.2. Model Architecture and Training

In addition to the approach described in Section 4, we add a frame-level VQVAE training stage at the start, which uses feed-forward convolutional encoder and decoder trained to map observed frames to a frame-level latent space. The codebook has 512 codes of 64 dimensions. Then as discussed in Section 4, in the second stage, we train the state VQVAE on top of the frame-level VQ representations. This model captures the temporal dynamics of trajectories in a second latent layer, consisting of a stack of 32 separate latent variables at each timestep, each with their separate codebook comprising of 512 codes of 64 dimensions. The architecture for this component consists of a convolutional encoder torso, an encoder LSTM, a quantization layer, a decoder LSTM and finally a convolutional decoder head that maps the transition discrete latents back to the frame-level VQ space. Finally in the third stage, we fit the transition model with a hybrid "planning path" using a deep, causal Transformer (Vaswani et al., 2017).

To generate new samples from the model, we first sample state latent variables from the prior network. These are then fed to the state VQVAE decoder for mapping back to the frame-level discrete latent space. The resulting frame-level codes are mapped to the pixel space by the frame-level VQVAE decoder.

5.2.3. Baselines

We compare our proposed approach based on VQM with several baselines. As the simplest baseline, we train a deterministic next-frame prediction model with an LSTM architecture closely mimicking the state VQVAE architecture except for the quantization layer. The network is trained to reconstruct each frame given the preceding frames with mean-squared-error (MSE). Additionally, we train several sequential continuous VAE baselines with different posterior and prior configurations to compare our discrete approach with continuous latent variables. We use GECO (Jimenez Rezende & Viola, 2018) to mitigate the well-known challenges of training variational autoencoders with flexible decoder and prior models. In particular, we assign a target average distortion tolerance for the decoder and minimize, using the Lagrange multiplier method, the KL-divergence of the posterior to the prior subject to this distortion constraint, as described in Jimenez Rezende & Viola (2018). We choose two different distortion levels of 25dB and 33dB PSNR. The former is based on our experiments with the deterministic LSTM predictor (the first baseline described above), which achieves reconstruction PSNR of about 24dB, and the latter is set slightly higher than the reconstruction PSNR of our frame-level VQVAE decoder at 32dB.

5.2.4. Evaluation Metric

For every trajectory in the test set we take $k(=1000)$ sample episodes from the model using the initial prefix of $T_0(=16)$ frames from each ground-truth trajectory. For each ground-truth trajectory, we find the sample with minimum (cumulative) reconstruction error (measured as Mean Squared Error or MSE) from the end of the prefix up to a target timestep $t(=128)$, and average the error for this *best-match* sample over the test trajectories. We refer to this metric as *Mean Best Reconstruction Error* or `MBRE` defined as

$$\texttt{MBRE}(S, G, T_0, T) = \frac{1}{|G|}\sum_{i=1}^{|G|}\min_{s\in S_i}\sum_{t=T_0}^{T}\|s(t) - G_i(t)\|^2,$$

where G is a set of ground-truth trajectories, and S is a set of sampled episodes–k episodes sampled using the initial T_0 frames of each ground-truth episode G_i as prefix. We compare sampled episodes with their corresponding ground truth episode at the target frame T. An ideal model that

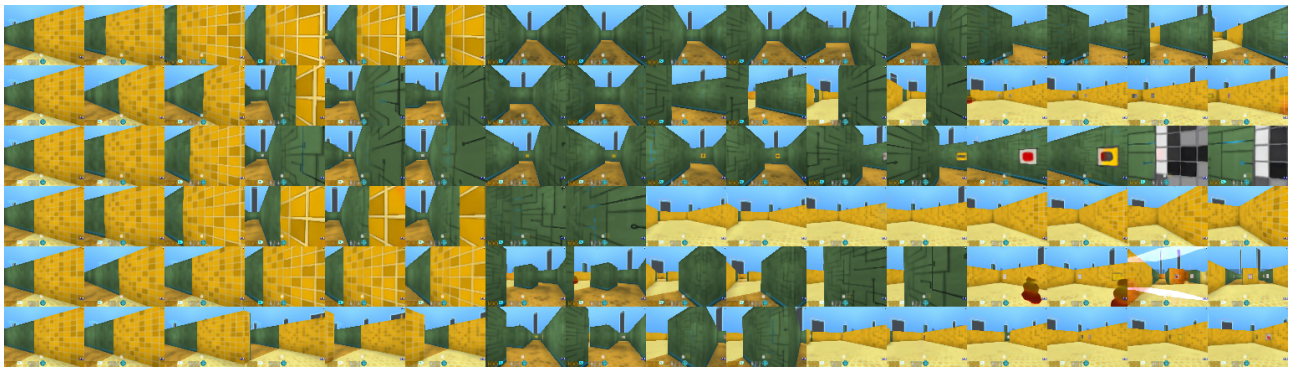


Figure 7. Different random rollouts from the same prefix video in *DeepMind Lab*. The first row is a ground-truth episode from the validation set. Each of the following rows show a sample from the model conditioned on the same 16 starting frames (not shown in the figure). Videos of more samples can be seen in https://sites.google.com/view/vqmodels/home.

generalizes well to the test set would place a non-negligible probability mass on the ground-truth trajectory, and thus an episode close to the ground-truth should be sampled given a sufficiently large number of trials.

5.2.5. Results

Rollouts generated by the model can be seen in Figure 7. Figure 8 shows the results of evaluating MBRE for our proposed approach against the aforementioned baselines. The LSTM baseline performs significantly worse than all latent variable models. This is expected because the generative model is not expressive enough for stochastic and multi-modal data, and as a result predicts the mean of all possible outcomes, which results in poor best matches against ground-truth. Comparisons with the two sequential VAE baselines demonstrate the trade-off between the reconstruction quality of a model and its predictive performance as measured by MBRE. Furthermore, it shows that our VQM approach is able to achieve competitive performance with respect to both reconstruction quality and long range predictive performance.

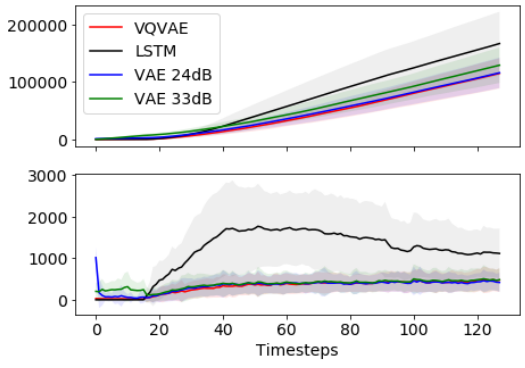


Figure 8. Mean Best Reconstruction Error (MBRE) for our proposed VQVAE approach and baselines. X-axis is length of trajectory. Top: computed over frames up to the target frame, Bottom: computed for the target frame only.

6. Conclusion

In this work, we propose a solution to generalize model-based planning for stochastic and partially-observed environments. Using discrete autoencoders, we learn discrete abstractions of the state and actions of an environment, which can then be used with discrete planning algorithms such as MCTS (Coulom, 2006). We demonstrated the efficacy of our approach on both an environment which requires deep tactical planning and a visually complex environment with high-dimensional observations. Further we successfully applied our method in the offline setting. Our agent learns from a dataset of human chess games and outperforms model-free baselines and performs competitively against offline MuZero and Stockfish Level 15 while being a more general algorithm. We believe the combination of model-based RL and offline RL has potential to unlock a variety of useful applications in the real world.

Acknowledgements

We'd like to thank Ivo Danihelka and Nando de Freitas for providing valuable feedback on early drafts of the paper. We'd like to thank Julian Schrittwieser for helping with the MuZero baselines. We'd also like to thank Sander Dieleman, David Silver, Yoshua Bengio, Jakub Sygnowski, and Aravind Srinivas for useful discussions and suggestions.

References

Anthony, T., Tian, Z., and Barber, D. Thinking fast and slow with deep learning and tree search. *arXiv preprint arXiv:1705.08439*, 2017.

Argenson, A. and Dulac-Arnold, G. Model-based offline planning. *arXiv preprint arXiv:2008.05556*, 2020.

Auer, P., Cesa-Bianchi, N., and Fischer, P. Finite-time

analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Brown, N., Sandholm, T., and Machine, S. Libratus: The superhuman ai for no-limit poker.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Buesing, L., Weber, T., Racaniere, S., Eslami, S. M. A., Rezende, D., Reichert, D. P., Viola, F., Besse, F., Gregor, K., Hassabis, D., and Wierstra, D. Learning and querying fast generative models for reinforcement learning, 2018.

Chiappa, S., Racaniere, S., Wierstra, D., and Mohamed, S. Recurrent environment simulators. *arXiv preprint arXiv:1704.02254*, 2017.

Chua, K., Calandra, R., McAllister, R., and Levine, S. Deep reinforcement learning in a handful of trials using probabilistic dynamics models, 2018.

Couëtoux, A., Hoock, J.-B., Sokolovska, N., Teytaud, O., and Bonnard, N. Continuous upper confidence trees. In *International Conference on Learning and Intelligent Optimization*, pp. 433–445. Springer, 2011.

Coulom, R. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pp. 72–83. Springer, 2006.

Deisenroth, M. and Rasmussen, C. Pilco: A model-based and data-efficient approach to policy search. In Getoor, L. and Scheffer, T. (eds.), *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pp. 465–472, New York, NY, USA, June 2011. ACM. ISBN 978-1-4503-0619-5.

Depeweg, S., Hernández-Lobato, J. M., Doshi-Velez, F., and Udluft, S. Learning and policy search in stochastic dynamical systems with bayesian neural networks. *arXiv preprint arXiv:1605.07127*, 2016.

Glickman, M. E. Example of the glicko-2 system.

Gulcehre, C., Wang, Z., Novikov, A., Paine, T. L., Colmenarejo, S. G., Zolna, K., Agarwal, R., Merel, J., Mankowitz, D., Paduraru, C., et al. Rl unplugged: Benchmarks for offline reinforcement learning. *arXiv preprint arXiv:2006.13888*, 2020.

Ha, D. and Schmidhuber, J. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, pp. 2450–2462, 2018.

Hafner, D., Lillicrap, T., Fischer, I., Villegas, R., Ha, D., Lee, H., and Davidson, J. Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551*, 2018.

Hafner, D., Lillicrap, T., Ba, J., and Norouzi, M. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

Hafner, D., Lillicrap, T., Norouzi, M., and Ba, J. Mastering atari with discrete world models, 2020.

Heinrich, J. and Silver, D. Deep reinforcement learning from self-play in imperfect-information games. *CoRR*, abs/1603.01121, 2016. URL http://arxiv.org/abs/1603.01121.

Henaff, M., Whitney, W. F., and LeCun, Y. Model-based planning with discrete and continuous actions. *arXiv preprint arXiv:1705.07177*, 2017.

Jimenez Rezende, D. and Viola, F. Taming VAEs. *ArXiv e-prints*, October 2018.

Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R. H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

Kidambi, R., Rajeswaran, A., Netrapalli, P., and Joachims, T. Morel: Model-based offline reinforcement learning. *arXiv preprint arXiv:2005.05951*, 2020.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2014.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Kocsis, L. and Szepesvári, C. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Kurutach, T., Clavera, I., Duan, Y., Tamar, A., and Abbeel, P. Model-ensemble trust-region policy optimization. *arXiv preprint arXiv:1802.10592*, 2018.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M., and Bowling, M. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

Michie, D. Game-playing and game-learning automata. 1966.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Moerland, T. M., Broekens, J., and Jonker, C. M. Learning multimodal transition dynamics for model-based reinforcement learning. *arXiv preprint arXiv:1705.00470*, 2017.

Moravčík, M., Schmid, M., Burch, N., Lisỳ, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., and Bowling, M. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

Moravcík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., and Bowling, M. H. Deepstack: Expert-level artificial intelligence in no-limit poker. *CoRR*, abs/1701.01724, 2017. URL http://arxiv.org/abs/1701.01724.

Oh, J., Guo, X., Lee, H., Lewis, R., and Singh, S. Action-conditional video prediction using deep networks in atari games, 2015.

Oh, J., Singh, S., and Lee, H. Value prediction network. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 6120–6130, 2017.

OpenAI, Andrychowicz, M., Baker, B., Chociej, M., Józefowicz, R., McGrew, B., Pachocki, J. W., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., Schneider, J., Sidor, S., Tobin, J., Welinder, P., Weng, L., and Zaremba, W. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018. URL http://arxiv.org/abs/1808.00177.

OpenAI, :, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. Dota 2 with large scale deep reinforcement learning, 2019.

Pearl, J., Glymour, M., and Jewell, N. P. *Causal inference in statistics: A primer*. John Wiley & Sons, 2016.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners.

Razavi, A., van den Oord, A., and Vinyals, O. Generating diverse high-fidelity images with vq-vae-2. In *Advances in Neural Information Processing Systems*, pp. 14866–14876, 2019.

Rezende, D. J., Mohamed, S., and Wierstra, D. Stochastic backpropagation and approximate inference in deep generative models, 2014.

Rezende, D. J., Danihelka, I., Papamakarios, G., Ke, N. R., Jiang, R., Weber, T., Gregor, K., Merzic, H., Viola, F., Wang, J., Mitrovic, J., Besse, F., Antonoglou, I., and Buesing, L. Causally correct partial models for reinforcement learning, 2020.

Russell, S. J. and Norvig, P. *Artificial Intelligence: a modern approach*. Pearson, 3 edition, 2009.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

Shannon, C. E. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016a. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016b.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. Mastering chess and shogi by self-play

with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017a.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017b.

Sohn, K., Lee, H., and Yan, X. Learning structured output representation using deep conditional generative models. *Advances in neural information processing systems*, 28: 3483–3491, 2015.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

T. Romstad, M. Costalba, J. K. Stockfish: A strong open source chess engine.

Tassa, Y., Tunyasuvunakool, S., Muldal, A., Doron, Y., Liu, S., Bohez, S., Merel, J., Erez, T., Lillicrap, T., and Heess, N. dm_control: Software and tasks for continuous control, 2020.

van den Oord, A., Vinyals, O., et al. Neural discrete representation learning. In *Advances in Neural Information Processing Systems*, pp. 6306–6315, 2017.

van Hasselt, H., Hessel, M., and Aslanides, J. When to use parametric models in reinforcement learning?, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782): 350–354, Nov 2019. ISSN 1476-4687. doi: 10.1038/s41586-019-1724-z. URL https://doi.org/10.1038/s41586-019-1724-z.

Yee, T., Lisỳ, V., Bowling, M. H., and Kambhampati, S. Monte carlo tree search in continuous action spaces with execution uncertainty. In *IJCAI*, pp. 690–697, 2016.

Yu, T., Thomas, G., Yu, L., Ermon, S., Zou, J., Levine, S., Finn, C., and Ma, T. Mopo: Model-based offline policy optimization. *arXiv preprint arXiv:2005.13239*, 2020.

A. Appendix

A.1. Chess datasets

The histogram of Glicko-2 ratings (Glickman) of the players in the training set is shown in Figure 9.

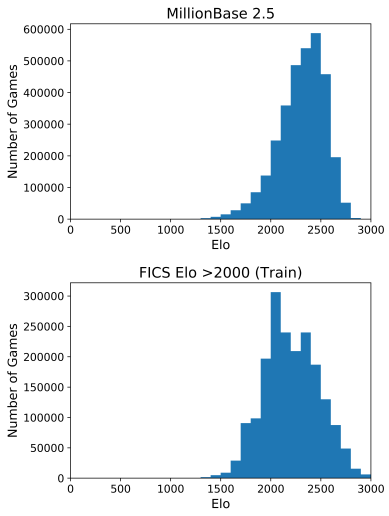


Figure 9. Histogram of Elo rating for players in Million Base dataset and FICS Elo2000 training set.

A.2. Details for Chess Experiments

A.2.1. Input Representations

We use symbolic representation for the chess board where each piece is represented as an integer. Following the same protocol as Silver et al. (2017a), the action space is represented as $8 \times 8 \times 73$ discrete actions.

A.2.2. Baseline Models

MuZeroChess The baseline MuZeroChess model is similar to that of the MuZero (Schrittwieser et al., 2019). Instead of using a deep convolutional network extracting features from the board state, we simply embed the board state to get the initial state s. The dynamics function is consist of 2 convolutional networks g and f. It works as follows: set $z^0 = s$ for the initial state; $z^{m+1} = g(z^m, a^m)$, g takes in the embedded action input a^m and previous state z^m and outputs the next state z^{m+1}; $h^{m+1} = f(z^{m+1})$, f produces the final state for prediction heads. We have two prediction heads v and π for value and policy respectively. Both g and f are 20-layer residual convolution stacks with 1024 hiddens and 256 bottleneck hiddens. All the heads have a hidden size 256. The action predictor uses kernel size 1 and strides 1 and its output is flattened to make the prediction.

We train with sequence length 10. If the training sequence reaches the end of the game, then after game terminates, we pad random actions and target value, as well as mask the action prediction loss.

We use a batch size of 2048 for training and use Adam optimizer (Kingma & Ba, 2014) with learning rate $3e^{-4}$ and exponential decay of with decay rate 0.9 and decay steps 100000. To stabilize the training, we apply gradient clipping with maximum clipping value 1. The model is trained for 200k steps.

Q-value and Imitation Agents The same MuZeroChess model as described above is used. For Q-value agent, we unroll the dynamics function for 1 step with each legal action. The estimated action value of the Q-value agent is the value prediction of the next state. For imitation agent, we use the policy prediction of the model as the imitation agent's policy.

A.2.3. VQ Model

State VQVAE The state VQVAE for chess has a encoder and a decoder, which are both 16-layer residual convolution stack with 256 hiddens and 64 bottleneck hiddens. The quantization layer has 2 codebooks, each of them has 64 codes of 256 dimensions. Before feeding into the quantization layer, we apply spatial mean pooling on the features. The reconstruction of the board state is cast as a classification problem where the model predicts the piece type at each position. Therefore, the reconstruction loss is the cross entropy loss. The training batch size is 1024. We use Adam optimizer (Kingma & Ba, 2014) with learning rate $3e^{-4}$ and exponential decay of with decay rate 0.9 and decay steps 100000. The model is trained for 1.2 million steps.

Transition model Our transition model for chess is similar to the model used for MuZeroChess. In addition to the dynamics function g which takes the action as input at each step, we introduce another dynamics function g' which takes the input of discrete latent codes. At alternating steps, instead of using g, we obtain the next state by $z^{2m+1} = g'(z^{2m}, k^{2m})$ where $m > 0$. We also introduce an additional prediction head τ to predict the discrete latent codes. The additional function g' is a 30-layer residual convolution stack with 1024 hiddens and 256 bottleneck hiddens. Unlike for actions, discrete latent codes after termination of the game don't need random padding or masking.

We use the same training setup and optimizer parameters as MuZeroChess. VQHybrid model is trained for 200k steps; VQPure model is trained for 400k steps.

MCTS The hyperparameters used for the MCTS are the same for baseline MuZeroChess and VQM-MCTS: $discount = 1.0$, UCB parameters $c_{base} = 19652.0$ and $c_{init} = 1.25$. No temperature is applied on the acting visit count policy. Same as MuZero (Schrittwieser et al., 2019),

we limit the agent action to all legal moves. MuZeroChess and VQHybrid agents don't have terminal state. VQPure agent reaches terminal state when the same VQ code is taken for 10 consecutive steps in the search tree branch.

For experiments where VQ agent is playing against Q-value, imitation and MuZeroChess agents, we employ similar strategy used for data generation in Schrittwieser et al. (2019). This is because both agents are deterministic, playing among them would result in deterministic games making the evaluation less meaningful. Specifically, instead of selecting the action with the highest visit count at the end of the tree search, we use a stochastic variant. We keep a pool of possible actions which have an action count of at least 1% the total count $\mathcal{A} = \{a : N(a) > 0.01 N_{max}\}$, and sample an action a according to the probability induced by visit counts $p(a) = \frac{N(a)}{\sum_{a' \in \mathcal{A}} N(a')}$. This stochastic variant of the tree policy is used for the first 30 steps of the game for all MCTS-based agents.

A.3. Quasirandom Sampling

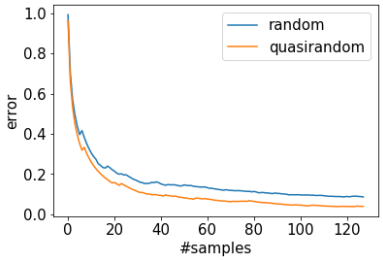


Figure 10. Quasirandom sampling produces empirical distributions closer to the true distribution than random sampling. The plotted error is the Euclidean distance between the probability distributions. For this analysis, we sampled the probabilities for the Multinomial distrbution from a Dirichlet distribution with $\{\alpha_i\}_{i=1}^{N} = 1$ where $N = 64$.

As explained in Section 4.2, our MCTS implementation takes the following form:

$$\arg\max_k \hat{Q}(s,k) + P(k|s)U(s,k)$$

where

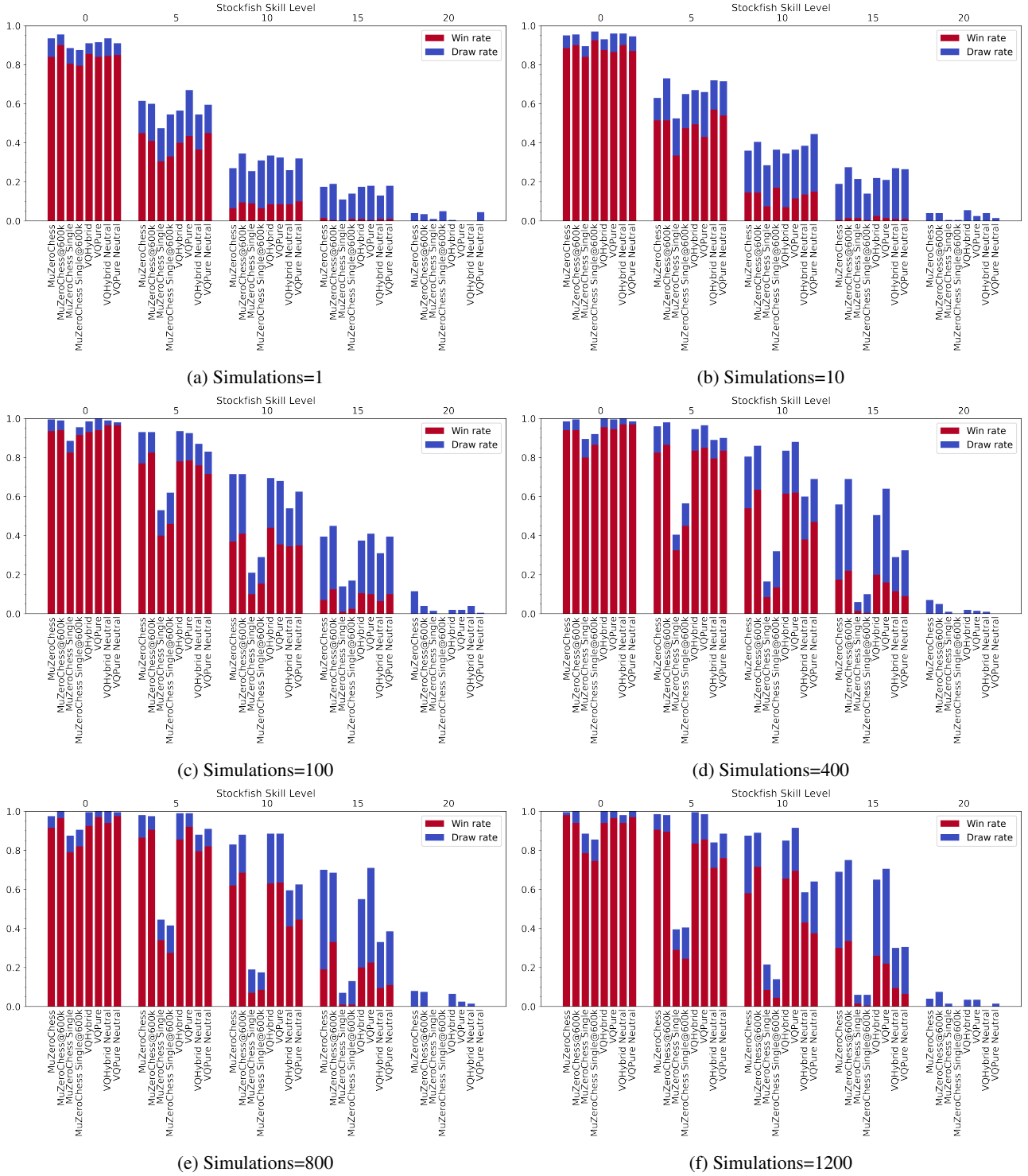
$$U(s,a) = \frac{\sqrt{N(s)}}{1+N(s,a)}\left[c_1 + \log\left(\frac{N(s)+c_2+1}{c_2}\right)\right],$$

$$\hat{Q}(s,k) = \begin{cases} Q(s,k) & \text{cooperative} \\ 0 & \text{neutral} \\ -Q(s,k) & \text{adversarial} \end{cases}$$

If we assume the environment to be neutral, the empirical distribution of the selected discrete latent codes at planning time should match the estimated distribution of codes $P(k|s)$. A straightforward way to obtain such a distribution is to sample i.i.d from the estimated distribution. However, in practice, we found that using the above equation with $\hat{Q}(s,k) = 0$ works better. This corresponds to a quasi-random Monte Carlo sample of a multinomial distribution p_i where we simply select the value which has the largest value of $\frac{p_i}{N(i)+1}$, where N is the number of times i has been sampled already.

A.4. Chess Agents Performance against Stockfish

Because VQ Models have a pre-training phase, a confounding factor for the performance comparison is the amount of computational resources required. We increase the computational resources for MuZeroChess baselines for 3 times. We call the resulting agents MuZeroChess@600k and MuZeroChess Single@600k. This increase is more than that of the pre-training phase since our state VQVAE is trained on a pair of observations whereas the transition model is trained on a sequence of 10 steps. Figure 11 reports the full results of MuZeroChess and VQ agents playing against Stockfish. As Figure 11 shows, with the increase of computational budget, MuZeroChess agent does have a slight performance improvement. However, this is not significant enough to affect the conclusions. Most importantly, MuZeroChess Single agent does not perform better even with significantly more compute. In fact, we see a decrease in win rate across all the Stockfish levels with 1200 simulations.



(a) Simulations=1

(b) Simulations=10

(c) Simulations=100

(d) Simulations=400

(e) Simulations=800

(f) Simulations=1200

Figure 11. **Agent performances evaluated against Stockfish 10 with 1, 10, 100, 400, 800 and 1200 simulations per step.** Compare agent performance with different number of simulation budget per move. Reported for agents: two-player MuZeroChess, two-player MuZeroChess@600k, single-player MuZeroChess, single-player MuZeroChess@600k, VQHybrid and VQPure searching over worst case scenario, VQHybrid and VQPure searching over neutral scenario. Stockfish 10 skill levels is varied between 0, 5, 10, 15 and 20 to control the strength of the engine.Red bar shows the win rate; blue bar shows the draw rate of the agents.