Aalto University
School of Electrical
Engineering

Master’s Programme in Computer, Communication and Information Sciences

Investigating Different Neural Network Architectures for Discrete Codebook World Models

Erald Shahinas

2025

Aalto University
School of Electrical Engineering

Author Erald Shahinas

Title Investigating Different Neural Network Architectures for Discrete Codebook World Models

Degree programme Computer, Communication and Information Sciences

Major Machine Learning, Data Science and Artificial Intelligence

Supervisor Prof. Joni Pajarinen

Advisors Dr. Aidan Scannell, Jatan Shrestha

Date 6 January 2025 **Number of pages** 19 **Language** English

Abstract

In this project, we investigate the performance of different neural network architectures for Discrete Codebook World Models (DCWM). We compare the effectiveness of Transformer models to Multi-Layer Perceptron (MLP) and Recurrent Neural Network (RNN) models in predicting the dynamics of various environments. Our experiments demonstrate that Transformer models, when trained autoregressively, outperforms other training approaches. We also explore the impact of scaling the Transformer model. The results highlight the potential of Transformer models in enhancing the predictive capabilities of world models.

Contents

1 Symbols and abbreviations

Abbreviations

RL	Reinforcement Learning
DCWM	Discrete Codebook World Model
TD-MPC2	Temporal Difference Model Predictive Control 2
RNN	Recurrent Neural Network
MLP	Multi-Layer Perceptron
FSQ	Finite Scalar Quantization
LSTM	Long Short-Term Memory
FLOPS	Floating Point Operations Per Second
TC Loss	Temporal Consistency Loss

2 Introduction

World models are models that emulate the real world's or a particular environment's dynamics. They try to predict a future state given temporal and spatial data of the environment and of the actions taken by an agent. The rationale behind this type of model comes from ideas about the human mind [1]. Ha and Schmidhuber argue that one way to describe the predictive model inside a human mind is not to think of it as a future predicting machine but as one that tries to predict future sensory input given current motor actions [6]. Thus instead of predicting what is going to happen in the future, it predicts what information will it receive from the senses after a specific action. This explains how humans are able to act instinctively using this predictive model instead of planning a course of action [1], [6]. For example, in sports that require fast reflexes such as baseball the time that it takes the human to decide how to hit the ball is at times less than the time it takes for the visual information to reach the brain. The muscles are able to move on their own while relying solely on the internal world model's prediction of where everything is going to be [8].

Reinforcement learning agents that use world models based on this idea have been used in recent years in order to achieve a good level of performance in different tasks [2], [3]. DreamerV3 for example is a state-of-the-art algorithm that uses a world model to be able to generalize across different settings and environments without requiring hyperparameter-tuning [2]. This makes it remarkably useful as it decreases the barrier of entry for using reinforcement learning in different tasks and areas. It has already showcased good progress by beating specialized methods in 150 different tasks. DreamerV3 is the first agent to collect diamonds out of the box in Minecraft, a difficult task because the environment has sparse rewards, exploration difficulty and long time horizons [3]. TD-MPC2 is another example of a state-of-the-art reinforcement learning algorithm that is able to achieve good performance in complex environments by using a generalist world model without needing hyperparameter tuning [4]. TD-MPC2 is also robust to changes in scale unlike other models and its performance increases when using larger parameter models.

A typical world model is usually made up of multiple components most commonly: an encoder, dynamics predictor, reward predictor, and decoder which are usually different types of neural networks.

- The encoder encodes the input states and actions into the latent space to extract the most useful information from them.
- The dynamics model predicts how the dynamics of the environment work by predicting future latent states given latent state action pairs.
- Similarly, the reward model predicts the reward given latent state action pairs.
- The decoder maps everything back from latent space to the real space.

The dynamics predictor is a crucial element of a world model as it tries to emulate the dynamics of the environments. Neural networks models are used in order to learn the dynamics of a particular environment, as they can learn from large amounts of data and from complex states or state representations [1]. Different papers use different models for their dynamics functions, for examples DreamerV3 uses a Recurrent Neural Network (RNN) whereas TD-MPC2 uses a Multi Layer Perceptron (MLP) [3] [4].

2.1 Problem Statement

The main focus of this project is to compare different architectures used for the dynamics model and to see how they can affect the performance of the algorithm. We will be using a baseline MLP architecture and comparing it to RNN and Transformer models. By exploring architectures which perform better when working with sequential data such as RNNs and Transformers, we want to determine if it improves the world-model predictive capabilities and the agent's decision-making.

The algorithm which we are using is DCWM developed by researchers at Aalto University [9]. DCWM is a model based RL algorithm similar to TD-MPC2, but unlike TD-MPC2 it uses quantized discrete codebook encodings. DCWM is able to outperform both TD-MPC2 and DreamerV2 on control tasks [9].

The quantized representations will be used as inputs for our dynamics models based on the architectures above. We will be testing the performance of the algorithm on different environments in Deep Mind Control Suite.

3 Methodology

In this section we describe the dataset preparation and preprocessing. We introduce an MLP, an LSTM, and a Transformer model and we detail different traning approaches for Transformer model.

3.1 Dataset and Preprocessing

The dynamics model is fed replayed data while the agent is interacting with the environment. The state and action data that comes from the replay buffer is encoded and then discretized using Finite Scalar Quantization (FSQ) . The shape of the data used is [T B L Q] where we can control the time dimension [T], batch dimension [B], the latent dimension [L], and the quantization levels [Q].

To train the model we are going to use a time dimension of 5 episodes, a batch size of 256, a variable latent dimension depending on the environment, and two FSQ levels with 5 values and 3 values for each level. FSQ also normalizes the data between -1 and 1.

We concatenate the discrete latent state and action data along the time dimension to form a single state action input as can be seen in Figure 1. The data is then fed to the dynamics function.

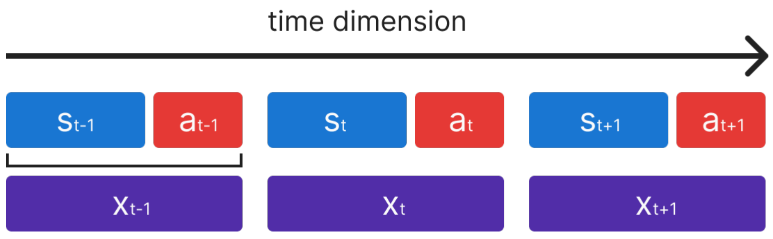

Figure 1: Input formatting for the dynamics model. Each timestep consists of a state (s) in blue and an action (a) in red. These are concatenated along the time dimension to form the input sequence (X) in purple.

3.2 MLP

The DCWM algorithm uses an MLP for the dynamics model which will be used as a baseline [9]. The model is similar to TD-MPC2 and it is made of linear layers where each intermediate linear layer is followed by LayerNorm.

The MLP is trained autoregressively. Firstly, we encode and quantize a batch of 5 timesteps of replayed experience of state action pairs. The transition MLP is fed the first state action pair t_0 and it outputs logits which represent the distribution of the following quantized latent t_1 state. We then compare the logits to the actual next state in order to calculate the cross-entropy loss. Finally, we sample from the logits and use the sampled t_1 state as the next input for the MLP. Generation is done autoregressively in the same manner.

3.3 RNN

A simple LSTM model will be used as another baseline for our Transformer model; however, the LSTM model will not be the focus of this report. The LSTM model we will be using is the Pytorch LSTM model, followed by a linear layer to shape the output. The input sequence is:

$$\mathbf{X} = [x_1, x_2, \ldots, x_5], \tag{1}$$

where $x_t \in \mathbb{R}^d$. The input is transformed on the LSTM as follows:

Input Gate:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{2}$$

Forget Gate:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{3}$$

Cell Candidate:

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{4}$$

Cell State Update:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{5}$$

Output Gate:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{6}$$

3.4 Transformer

The Transformer model will be the main focus of this report. The Transformer we are using is a Decoder only Transformer modeled after the Pytorch's DecoderTransformer. The Decoder Layers and Multi Head Attention blocks are modeled after barebone implementations of their equivalent Pytorch implementations. Each decoder layer is made of a self-attention, multihead attention and feedforward block in that order. All blocks are followed by a dropout layer, a skip connection and a layer normalization.

For the Transformer training, we compared two different approaches in order to see if there are any benefits or drawbacks in any specific way of training.

Our main input X is a sequence of 5 timesteps with shape [H, B, FL] where H is our time dimension, B is our batch dimension, and FL is our flattened latent dimension. The ground truth sequence targets (T) corresponds to X shifted by one timestep.

$$X = [x_0, x_1, \ldots, x_4], T = [x_1, x_2, \ldots, x_5], \quad x_i \in \mathbb{R}^{d_{\text{model}}} \tag{7}$$

where n is the sequence length and d_{model} is the dimensionality of the model input.

3.4.1 Training with teacher forcing

In the first approach we are using teacher forcing by forcing the ground truth values as inputs to the model as can be seen on Figure 2. In order to make sure the model does not peak into the future we use masking on the target. The target will be used again in the end in order to compute the cross entropy loss between the output of the model and the ground truth.

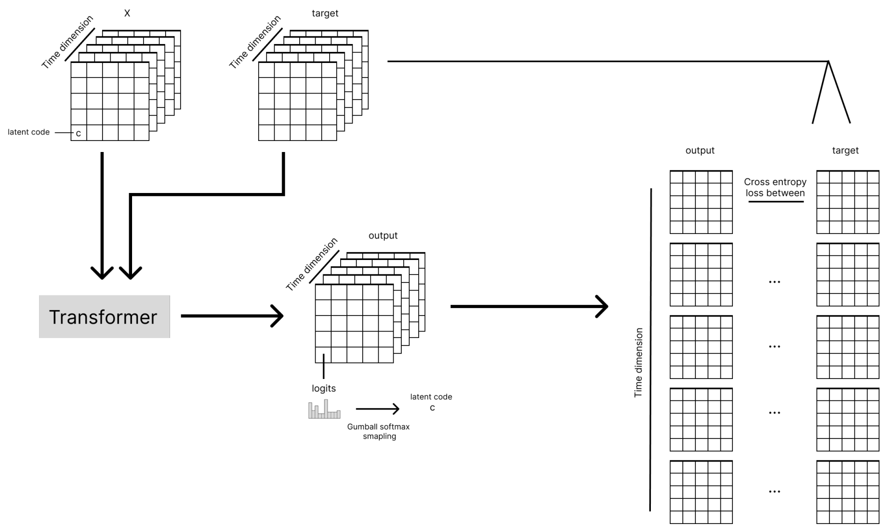

Figure 2: This approach uses teacher forcing by providing state-action pairs (left) and ground truth states (right). The ground truth targets are used to compute the loss by comparing the model's predictions with the actual values.

The model attends over its inputs as follows: For a single attention head with dimensionality d_k

$$Q = TW^Q, \quad K = XW^K, \quad V = XW^V \tag{8}$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ are learnable projection matrices. Q is our query which comes from the taget, K is our key and V is our value matrix which come from the input X.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V \tag{9}$$

where M is our attention Mask

$$M_{ij} = \begin{cases} 0, & \text{if } j \leq i, \\ -\infty, & \text{if } j > i. \end{cases} \tag{10}$$

For multiple heads, the outputs are concatenated and linearly transformed:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{11}$$

where W^O is the output projection matrix.

3.4.2 Autoregressive training

Figure 3 showcases the second approach. This approach is similar to the first approach, but this time we only input the training data X without the target. We will only use the target for calculating the loss. In this approach, we do not need to use masking, as we are not feeding the model with future data. Worth noting is that self-attention and multi-head attention blocks become equivalent in this approach, thus one of them could be reduced, leading to a more concise model. During self attention we only need to attend over X, making out projected vectors as follows:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \tag{12}$$

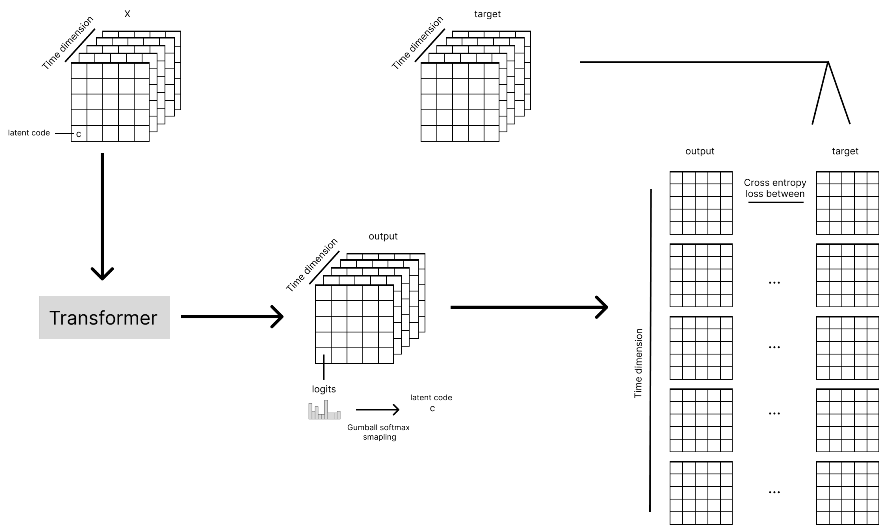

Figure 3: This approach differs from the first one by only the training data X as input to the transformer, excluding the target data. The target is used solely for loss computation.

During generation, the Transformer generates new states in an autoregressive manner because model-predictive path integral control (MPPI) requires the model to predict a state for the algorithm to sample policy trajectories.

3.4.3 Hyperparameters

As the Transformer is a complex architecture we want to see how different hyperparameters affect the Transformer and the performance metrics. Two hyperparameters which we will be exploring are the number of layers and the size of the feedforward layer. Decoder layers are connected sequentially, thus the number of layers affects the depth of the Transformer. Feedforward layers are present in each layer, thus increasing the size of these layers can be thought of as a way to increase the width of the Transformer.

Both these hyperparameters have a big impact on the scale of the model. In this report we will also be investigating scaling up the model and how that affects performance.

4 Experiments

In this section we document the different experiments conducted and their outcomes.

4.1 Comparing different training approaches

For the first experiment we compared the two different training approaches for the same Transformer architecture. Each approach is tested using three seeds in the walker-walk environment.

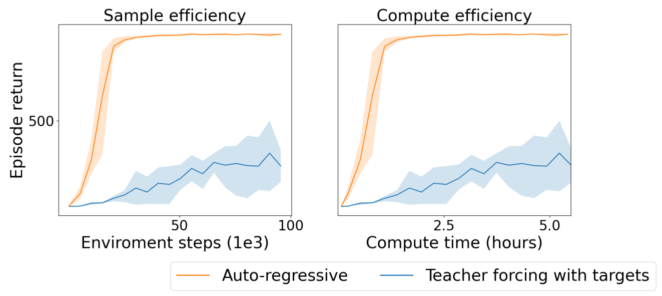

Figure 4: Episodic return for the two training approaches, using three seeds and 95 % confidence intervals. On the x axis we have the number of rollout steps that the agent experienced from the environment (left) and the total computation time for the agent (right).

From Figure 4 we can see the second approach, which trains autoregressively, performs the best. The teacher forcing approach lags behind and underperforms. The first approach is both sample efficient compared and time efficient as it achieves a good performance given less enviroment steps and time. This is the only approach that does not see the targets during training. The other approach could be suffering from over-reliance on the correct context making it unable to generalize to new situations.

Aside from the peformance metrics such as the episodic return we can also take a look at the world model loss. DCWM uses a consistency loss instead of a reconstruction loss as it can have a detrimental effect on the performance of model-based agents. That is why the world model does not have a decoder and instead we use a self-supervised consistency loss. The role of the temporal consistency loss (TC loss) is to check that output states are consistent with ground truth states, thus the dynamic model is working correctly. For it we use cross-entropy, which is a classification loss that checks that the outputted discrete values belong to the same class as the ground truth ones. In the plot in Figure 5, we can see the loss of the autoregressive model has a sharp incline which is followed by a steady decline. On the other hand the worst performing model has a small almost trivial TC loss throughout. This model is most likely overfitting on this loss, leading to a model which is overreliant on the targets as inputs.

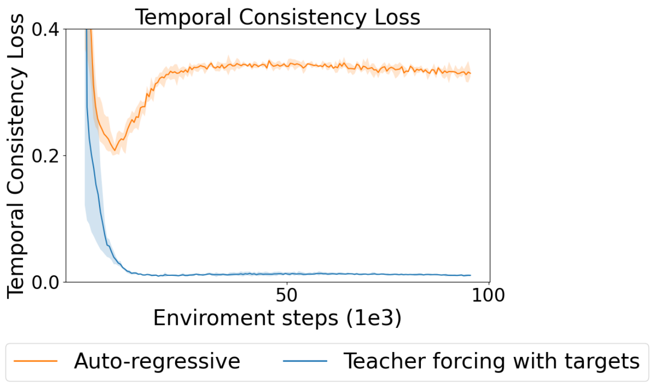

Figure 5: World model temporal consistency loss for each training approach

4.2 Investigating hyperparameters and model size

We compared the performance of different size transformer models in 2 difficult environments ‘Humanoid Walk‘ and ‘Dog Run‘. Figure 6 showcases the performance of each model after 750,000 environment steps and three seeds. Each seed took 96 hours of GPU time to run. We can see that the Transformer model’s performance decreases as the model scales. Thus our current Transformer architecture does not scale well with increases in model parameters.

One reason that for why it does not scale well could be that larger models require larger amounts of data. However that would require more computational resources in order to test.

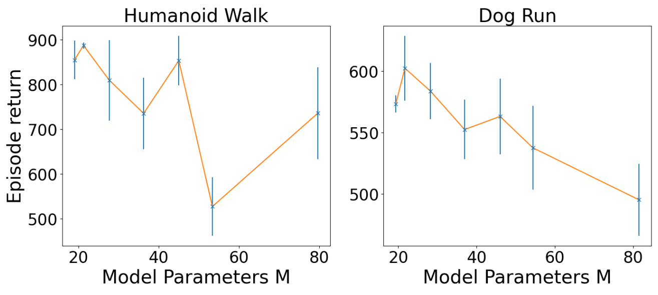

Figure 6: Comparing performance of different size transformer models given the same amount of environment steps in different environments. The blue line represents the standard deviation.

4.3 Comparing with alternative architectures

After finding the best performing hyperparameters for our Transformer model we compare the performance with the MLP baseline which was used on the DCWM paper [9]. We compare both model on 5 different environments ranging from easy to difficult. We use 5 seeds for each model and showcase the mean and 95 percentile confidence intervals on Figure 7.

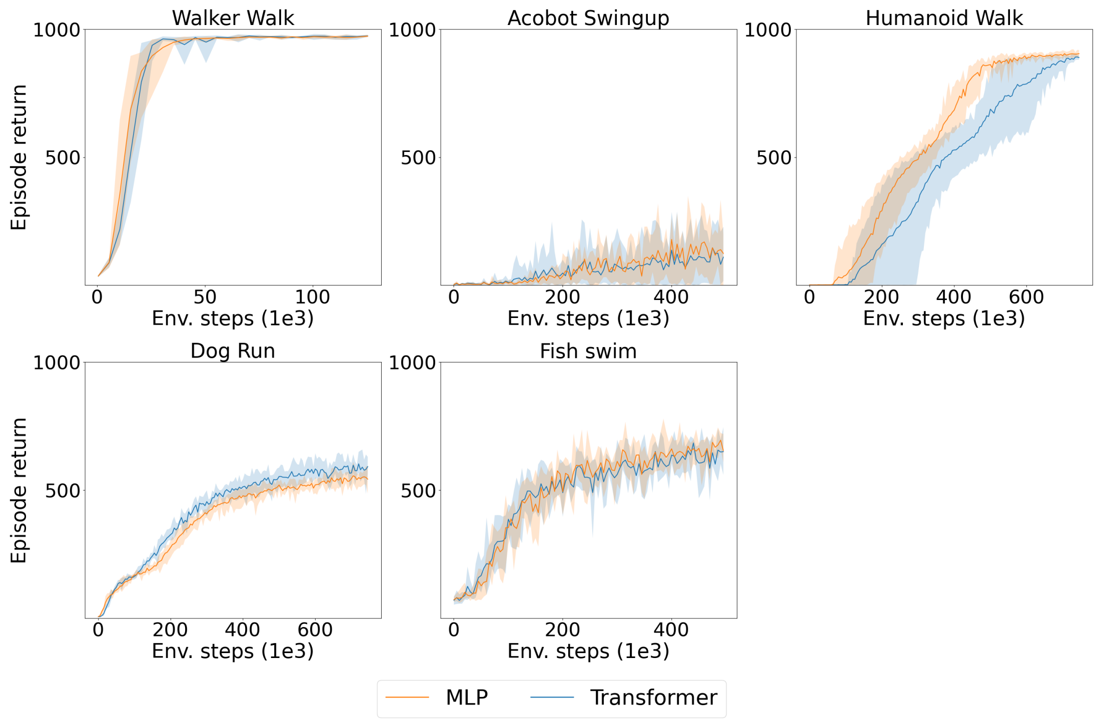

Figure 7: Comparing performance of the Transformer model to the MLP baseline

We can see that the performance of both the Trasnformer and MLP models are similar in all tasks. The MLP model outperforms in the ‘Humanoid Walk‘ environment while the Transformer model slightly outperformes in the ‘Dog Run‘ environment. We decided to also compare the Trasnsformer model to the LSTM model as can be seen on Figure 8 using three seeds.

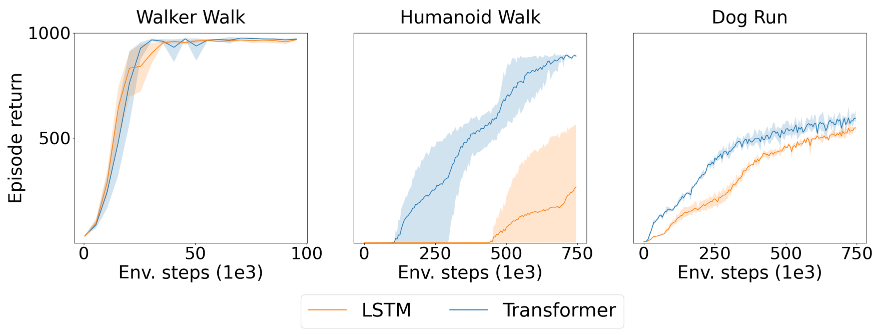

Figure 8: Comparing performance of the Transformer model to the LSTM baseline

While both models perform similarly on a simple environment like walker, the Transformer model seem to perform much better on hard environments like ‘Humanoid Walk‘ and ‘Dog Run‘.

5 Conclusions

In this project, we compared the performance of MLP, RNN, and Transformer architectures for the dynamics model in a world model-based reinforcement learning algorithm. Our experiments demonstrated that the Transformer model performs best when it is trained autoregressively. We also experimented with scaling up the Transformer model to no avail. The hyperparameter-tuned Transformer model was able to outperform the RNN architecture and to match the performance of the MLP model from the DCWM paper.

6 Limitations and future work

To further enhance the performance and capabilities of our world model neural network, several improvements can be considered. One promising direction is the incorporation of linear attention mechanisms. Linear attention can significantly reduce the computational complexity of the model, making it more efficient and scalable [10]. This improvement can lead to faster training times and the ability to handle larger datasets, ultimately improving the model's predictive accuracy and generalization capabilities.

Additionally, exploring more complex models such as Mamba could provide substantial benefits. Mamba is able to handle diverse and complex tasks and could enhance the model's ability to learn from intricate patterns in the data [11]. By leveraging Mamba we can potentially achieve higher performance in various environments and tasks, pushing the boundaries of what our world model can accomplish.

References

[1] D. Ha and J. Schmidhuber, "Recurrent World Models Facilitate Policy Evolution," *arXiv preprint arXiv:1809.01999*, Sep. 2018. [Online]. Available: https://arxiv.org/abs/1809.01999

[2] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Mastering Atari with Discrete World Models," *arXiv preprint arXiv:2010.02193*. [Online]. Available: https://arxiv.org/pdf/2010.02193

[3] D. Hafner *et al.*, "Mastering Diverse Domains through World Models," *arXiv preprint arXiv:2301.04104v2*. [Online]. Available: https://arxiv.org/pdf/2301.04104v2

[4] J. Fu *et al.*, "TD-MPC2: Scalable, Robust World Models for Continuous Control," *arXiv preprint arXiv:2310.16828*. [Online]. Available: https://arxiv.org/abs/2310.16828

[5] A. Scannell, K. Kujanpää, Y. Zhao, M. Nakhaei, A. Solin, and J. Pajarinen, "iQRL – Implicitly Quantized Representations for Sample-efficient Reinforcement

Learning," *arXiv preprint arXiv:2406.02696*. [Online]. Available: https://arxiv.org/pdf/2406.02696

[6] D. Ha and J. Schmidhuber, "World Models," *arXiv preprint arXiv:1803.10122*. [Online]. Available: https://arxiv.org/pdf/1803.10122

[7] D. Mobbs, C. C. Hagan, T. Dalgleish, B. Silston, and C. Prévost, "The ecology of human fear: survival optimization and the nervous system," *Frontiers in Neuroscience*, vol. 9, no. 55, 2015. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnins.2015.00055/full

[8] G. W. Maus, J. Fischer, and D. Whitney, "Motion-Dependent Representation of Space in Area MT+," *Neuron*, vol. 78, no. 3, pp. 554–560, 2013. [Online]. Available: https://www.cell.com/neuron/fulltext/S0896-6273(13)00257-2

[9] Anonymous authors, "Discrete cookbook world models for continous control" 2024. [Online]. Available: https://openreview.net/pdf?id=lfRYzd8ady

[10] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention," *arXiv preprint arXiv:2006.16236*, 2020. [Online]. Available: https://arxiv.org/abs/2006.16236

[11] A. Gu and T. Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces," *arXiv preprint arXiv:2312.00752*, 2024. [Online]. Available: https://arxiv.org/abs/2312.00752

7 Appendix

7.1 Architectures

7.1.1 Transformer

```
(_trans): TransformerDecoderModel(
    (transformer_decoder): LocalTransformerDecoder(
      (layers): ModuleList(
        (0-1): 2 x LocalTransformerDecoderLayer(
          (self_attn): LocalMultiheadAttention(
            (q_proj): Linear(in_features=1045, out_features=1045, bias=True)
            (k_proj): Linear(in_features=1045, out_features=1045, bias=True)
            (v_proj): Linear(in_features=1045, out_features=1045, bias=True)
            (out_proj): Linear(in_features=1045, out_features=1045, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (multihead_attn): LocalMultiheadAttention(
            (q_proj): Linear(in_features=1045, out_features=1045, bias=True)
            (k_proj): Linear(in_features=1045, out_features=1045, bias=True)
            (v_proj): Linear(in_features=1045, out_features=1045, bias=True)
            (out_proj): Linear(in_features=1045, out_features=1045, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (feed_forward): Sequential(
            (0): Linear(in_features=1045, out_features=2048, bias=True)
            (1): ReLU()
            (2): Dropout(p=0.1, inplace=False)
            (3): Linear(in_features=2048, out_features=1045, bias=True)
          )
          (norm1): LayerNorm((1045,), eps=1e-05, elementwise_affine=True)
          (norm2): LayerNorm((1045,), eps=1e-05, elementwise_affine=True)
          (norm3): LayerNorm((1045,), eps=1e-05, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
    (fc_layer): Linear(in_features=1045, out_features=512, bias=True)
    (fc_out): Linear(in_features=512, out_features=7680, bias=True)
```

7.1.2 LSTM

```
(_trans): LocalLSTM(
    (lstm): LSTM(1030, 1024, num_layers=2, dropout=0.2)
    (fc): Linear(in_features=1024, out_features=7680, bias=True)
)
```

7.1.3 MLP

```
(_trans): Sequential(
    (0): NormedLinear(in_features=1030,         out_features=512,
bias=True,          act=Mish)
    (1): NormedLinear(in_features=512,         out_features=512,
bias=True,          act=Mish)
    (2): Linear(in_features=512, out_features=7680, bias=True)
)
```

7.2 Experimental setups

7.2.1 Comparing different training approaches

The results were obtained using the following script arguments and hyperparameters.

```
python train.py -m +env=walker-walk ++use_wandb=True agent=fsqmpc_td3
hydra/launcher=slurm_24hrs_5cpu ++agent.trans=transformer
```

```
++agent.latent_dim=512 ++agent.mpc=True
++agent.transformer_training=ar,tf ++seed=1,2,3
++hydra.launcher.constraint="ampere"
```

Hyperparameters	Value	Description
DCWM and training hyperparameters	Default values	-
World Model		
Fsq levels	[5, 3]	
Horizon	5	
Latent dimension	512	
Transistion archtecture	Transformer	
Training approach	autoregressive	
Local Transformer	True	Uses local implementation of decoder layers instead of Pytorch'
Transformer layers	2	
Transformer heads	2	
Transformer feedforward	2048	width of feedforward layer inside the decoder layer
Transformer dropout	0.1	

7.2.2 Investigating hyperparameters and model size

The results were obtained using the following script arguments.

```
python train.py -m +env=dog-run ++use_wandb=True agent=fsqmpc_td3
hydra/launcher=slurm_96hrs ++agent.trans=transformer
++agent.latent_dim=512 ++agent.mpc=True
++agent.transformer_training=ar ++seed=1,2,3 ++agent.trasformer_layers=1,2,4,8
++agent.trasformer_feedforward=2048
++hydra.launcher.constraint="volta"

python train.py -m +env=dog-run ++use_wandb=True agent=fsqmpc_td3
hydra/launcher=slurm_96hrs ++agent.trans=transformer
++agent.latent_dim=512 ++agent.mpc=True
++agent.transformer_training=ar ++seed=1,2,3
++agent.trasformer_layers=2
++agent.trasformer_feedforward=512,4096,8192
++hydra.launcher.constraint="volta"

python train.py -m +env=humanoid-walk ++use_wandb=True agent=fsqmpc_td3
hydra/launcher=slurm_96hrs ++agent.trans=transformer
++agent.latent_dim=512 ++agent.mpc=True
++agent.transformer_training=ar ++seed=1,2,3 ++agent.trasformer_layers=1,2,4,8
```

```
++agent.trasformer_feedforward=2048
++agent.transformer_heads=5
++hydra.launcher.constraint="volta"

python train.py -m +env=humanoid-walk ++use_wandb=True agent=fsqmpc_td3
hydra/launcher=slurm_96hrs ++agent.trans=transformer
++agent.latent_dim=512 ++agent.mpc=True
++agent.transformer_training=ar ++seed=1,2,3
++agent.trasformer_layers=2
++agent.trasformer_feedforward=512,4096,8192
++agent.transformer_heads=5
++hydra.launcher.constraint="volta"
```

7.2.3 Comparing with alternative architectures

The results were obtained using the following script arguments.

```
python train.py -m +env=walker-walk,humaoid-walk,dog-run,
    acrobot-swingup,fish-swim
++use_wandb=True
agent=fsqmpc_td3
hydra/launcher=slurm_96hrs ++agent.trans=mlp
++agent.latent_dim=512 ++agent.mpc=True
++seed=1,2,3,4,5
++hydra.launcher.constraint="volta"

python train.py -m +env=walker-walk,dog-run
++use_wandb=True agent=fsqmpc_td3
hydra/launcher=slurm_96hrs
++agent.trans=transformer
++agent.latent_dim=512
++agent.mpc=True
++agent.transformer_layers=2
++agent.transformer_feedforward=512
++agent.transformer_heads=2
++seed=1,2,3,4,5
++hydra.launcher.constraint="volta"

python train.py -m +env=humanoid-walk,acrobot-swingup
++use_wandb=True agent=fsqmpc_td3
hydra/launcher=slurm_96hrs
++agent.trans=transformer
++agent.latent_dim=512
++agent.mpc=True
++agent.transformer_layers=2
```

```
++agent.transformer_feedforward=512
++agent.transformer_heads=5
++seed=1,2,3,4,5
++hydra.launcher.constraint="volta"

python train.py -m +env=fish-swim ++use_wandb=True
agent=fsqmpc_td3
hydra/launcher=slurm_96hrs
++agent.trans=transformer
++agent.latent_dim=512
++agent.mpc=True
++agent.transformer_layers=2
++agent.transformer_feedforward=512
++agent.transformer_heads=5
++seed=1,2,3,4,5
++hydra.launcher.constraint="volta"


python train.py -m +env=walker-walk,dog-run,humanoid-walk ++use_wandb=True
agent=fsqmpc_td3
hydra/launcher=slurm_96hrs
++agent.trans=lstm
++agent.latent_dim=512
++agent.mpc=True
++seed=1,2,3
```

7.3 Github

https://github.com/eralds/discrete-world-models