Solving the Rubik's Cube via Sequence Modeling using Transformer-based Models

Mark Bedaywi, Longtai (Ted) Deng, Shaul Francus

Abstract

Transformer-based models have been proven to be extremely effective at sequence modelling. In this paper, we pivot some variants of transformers-based models via miscellaneous techniques and leverage their sequence modelling power to solve the Rubik's cube, traditionally a RL task. More specifically, we first introduce the techniques for enabling the regular transformer (RT), the decision transformer (DT) and the online decision transformer (ODT) to solve the Rubik's cube, then provide some intuitive explanations of their strengths and weaknesses as well as the technical analysis based on their performances.

1 Introduction

Transformers have dominated the field of NLP in recent years due to their incredible power of sequence modelling. Various studies have been done in designing variants of the transformer architecture to apply in other fields. We will be tackling the Rubik's cube, using transformer-based models, for the following reasons: 1) Training data is clean and easy to generate; 2) There is a good measure of model performance (i.e. by shuffle length the model can restore); 3) It is a sparse reward problem in which many RL algorithms fail, so we look for alternatives.

Our work divides into three parts: First, we[1] evaluate the performance of RT applied with specific techniques; second, we compare RT with DT introduced by Chen et al. [2021] and examine various settings including the choice of the reward function and model capacity and their influence to the model performance; third, we enable DT to be online DT and evaluate against the previous two.

2 Related work

The Rubik's cube has been studied before in the literature, but most approaches focus on alpha-go style Monte Carlo tree search like in Agostinelli et al. [2019]. However, there is no accepted, published work applying transformers to the Rubik's cube. The transformer itself was introduced by Vaswani et al. [2017]. DT was introduced by Chen et al. [2021]. DT are known to perform poorly in stochastic environments in Paster et al. [2022], hence our setting of a deterministic environment.

Moreover, work has been done in generalizing the DT to the online setting in Zheng et al. [2022]. Our approach is different in that it is better suited for problems like the Rubik's cube where there is a clear measure of difficulty, and allows a greater deal of control over exploration and exploitation. The idea of controlling difficulty has been studied before, and is known as curriculum learning (Wang et al. [2021]). The idea of training on past experiences is common in RL, and is known as experience replay (Zhang and Sutton [2017]). Our approach of only training on solutions that solve the cube was inspired by Oh et al. [2018] that focus training on good examples in an actor-critic algorithm.

[1] Please see appendix A for the contributions of each member, and visit `https://github.com/tedtedtedtedtedted/Curious-Transformer-On-Rubik-Cube` to see our code base.

Preprint. Under review.

3 Methods

3.1 Regular Transformer

We begin with the regular transformer architecture (decoder-only) and feed it with many sequences[2] of states and actions to the restored state. Ideally the model shall only predict actions, however, due to an architectural limitation[3] of the transformer, RT must consider all tokens in token space.
We next propose a workaround for this, using the weighted cross entropy loss (WCE):

$$L = \sum_{c=1}^{C} \left(w_c \cdot \log \left(\frac{\exp(x_c)}{\sum_{i=1}^{C}(\exp(x_i))} \right) \cdot y_c \right)$$

where C is the number of classes to predict, x is logits, y is target indicator, w is weights for each class (for siplicity, one batch). This helps with focusing on predicting the correct action tokens rather than being satisfied with predicting correctly the state tokens, considering that our dataset is highly imbalanced, where there is only one action token to predict for every 29 consecutive tokens in the sequence. Our weighting scheme used $1 : 28$ for state token v.s. action token.
It is clearly wrong punish the model for not being able to predict the starting state. Hence, We adjust the mask of the attention matrix from lower triangular to lower triangular everywhere except the top left block that corresponds to the starting state, where we do not mask at all. Another scheme is to apply a lower triangular, staircase-like mask where we do not hide the preceding states but we do hide for the action tokens and subsequent states.
For model to not mislead itself, we augment the transformer architecture with the transition function outputting states for the autoregressive step during inference time. We also find that RT performs better during inference time with the masked attention matrix which is originally designed for training time. This is surprising, but we suspect that it is because the model changes with the mask removed.

3.2 Decision Transformer

The main difference between DT and RT is that now we include the undiscounted returns-to-go in the trajectory. Define the returns-to-go, R_t, to be the sum of all future rewards. Then, the training data is of the form: $(s_0^{(i)}, a_0^{(i)}, R_0^{(i)}, s_1^{(i)}, a_1^{(i)}, R_1^{(i)}, \ldots, s_N^{(i)}, a_N^{(i)}, R_N^{(i)})$. During inference, we start with (s_0, R_0), where s_0 is the initial state, and R_0 is the reward we are trying to achieve, then we predict the next action.
Define $\mathcal{F}_{M:N}^{(i)} = (s_M^{(i)}, a_M^{(i)}, R_M^{(i)}, s_{M+1}^{(i)}, a_{u+1}^{(i)}, R_{u+1}^{(i)}, \ldots, s_N^{(i)})$, where we take the trajectory from the i'th data point starting at M and up to N, and we don't include the final action and reward. Let θ be the parameters of the transformer. The transformer then implements a policy $\pi_\theta(a \mid \mathcal{F})$ that depends on all the previous states and actions (here, we don't rely on the Markov property).
We change the loss to only take into account the actions. In this setting, a cross entropy loss is suitable, and we use the following, when $\mathcal{D}$ is our data-set of trajectories:

$$\mathbb{E}_{(\mathcal{F}_{M:N}, a_N) \sim \mathcal{D}} \left[-\log(\pi_\theta(a_N \mid \mathcal{F}_{M:N})) \right]$$

Note that we absorb the probability of seeing a trajectory into the expectation. Moreover, we learn a different embedding for states, actions, and rewards respectively. We modify the decision transformer by removing the tanh activation for actions at the end and replacing this with a softmax layer, so that the output is the logits, instead of the moves as in Chen et al. [2021].
Finally, we need a good choice of reward. We will investigate two rewards, $r_0(s)$ that is 10 if s is the solved state, and 0 otherwise and $r_{-1}(s)$ that is 10 when s is the solved state and -1 otherwise.

3.3 Decision Transformer with Self-Imitation

We propose a method of converting offline DT to online as online RL is the most typical in the literature. The key idea is to store a replay buffer $\mathcal{D}$ (which in the RL literature, is the name given to the history of encountered trajectories) and train on that repeatedly. To generate the replay buffer, we heavily rely on random exploration, using an ϵ-greedy policy, π'_θ that picks a random move with probability ϵ and otherwise from the policy π_θ. This removes reliance on expert data and generates

[2] See appendix B for details on representation.
[3] See appendix B for explanations on the limitation.

structured examples for training the replay buffer. These examples are just outside the boundary of what the transformer knows as it requires minimal exploration, which we conjecture makes them more useful for training. We maintain a difficulty scheduler that tells us how much to shuffle the cube on the current iteration, $\rho(t)$, and a schedule on ϵ, $\epsilon(t)$. The algorithm is summarised in algorithm box 1.

Algorithm 1 Self-Imitating, Off-Policy, Online Decision Transformer

1: Initialize $\mathcal{D}$, the replay buffer, to the empty set.
2: **for** $t = 1, \ldots, K$ **do**
3: **for** $i = 1, \ldots,$ number of training examples generated **do**
4: Sample s_0 from cubes shuffled randomly for $\rho(t)$ steps.
5: Follow policy π'_θ to generate the trajectory:

$$\tau = (s_0^{(i)}, a_0^{(i)}, R_0^{(i)}, s_1^{(i)}, a_1^{(i)}, R_1^{(i)}, \ldots, s_N^{(i)}, a_N^{(i)}, R_N^{(i)}).$$

6: If $s_N^{(i)}$ is the solved state, add τ to $\mathcal{D}$.
7: **end for**
8: Train the transformer on the replay buffer, updating the parameters as follows:

$$\theta \leftarrow \min_\theta \mathbb{E}_{(\mathcal{F}_{M:N}, a_N) \sim \mathcal{D}} \left[-\log(\pi_\theta(a_N \mid \mathcal{F}_{M:N})) \right].$$

9: **end for**

Conjecture 1. *If, as $t \to \infty$, $\rho(t) \to \infty$ and $\epsilon(t) \to 0$, then we have that $\pi_\theta \to \pi^*$, where π^* is the optimal policy.*

4 Experiment

We borrowed the NanoGPT code from Karpathy [2021] as a template for RT. To implement DT and ODT, we modify the code in the HuggingFace API in Wolf et al. [2021]. For RT, we have compared various choices of the weight matrix for WCE loss, batch sizes (8), number of heads (4), number of block layers (4), training iterations (2000). For DT, we run a sensitivity analysis on the model size hyperparameters, adjusting the number of heads, layers, and size of the layers in the transformer and run tests to check that our choice of reward is optimal. For ODT, we test the performance on fast and slow settings of the hyperparameters, and investigate the performance afterwards. Please see appendix B for detailed descriptions of the experiment environment and hyperparameters.

5 Results

See table 1 for the models' performances.
It turns out that RT obtained the best results. It is worth noting that being able to restore shuffle length of 10 with success rate of 25% is quite impressive. For a naive estimate, there are about 18^{10} possible configurations. What is actually interesting is that we find that the inference time performance is always notably better when employing the attention mask. The only justification to this is that since we have multiple dependent blocks in the decoder, and the input sequence of a block is the output sequence of the preceding block, and so by removing the mask we are mutating the intermediate sequence between blocks which may cause the butterfly effect. Also, the importance of the use of the weighted cross entropy loss in this setting is experimentally verified. Without it, the model loses focus and is satisfied with low training loss since it successfully predict most/all state tokens correctly, which is not what we desired because we have added the transition function in the autoregressive part of the architecture and the model is only used for predicting the next action to restore the cube.
DT does very well, but surprisingly worse than the RT! We also observe that increasing model capacity in fact results in overfitting, even with the aid of increasing the dropout rate. Surprisingly, using r_{-1} in fact resulted in worse performance than r_0 reward. This exposes the following limitation of DT, that we need a precise knowledge of the final reward. With r_0, DT is allowed to solve the cube in any number of moves, unlike r_{-1}, which has to use a certain number to meet the initial returns-to-go.

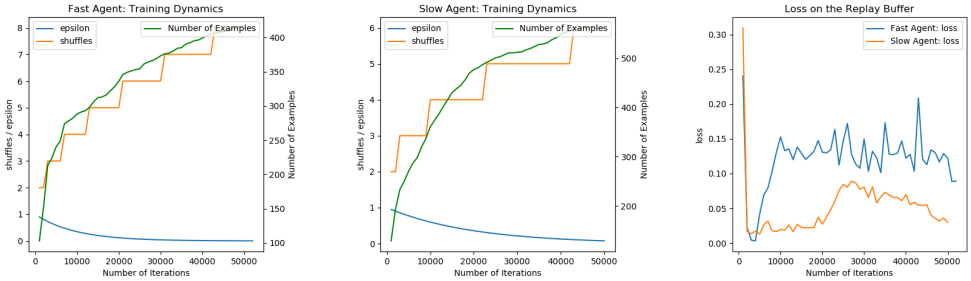

Figure 2: ODT results. The slow agent uses: $\rho(t) = \text{round}((t/1000)^{0.4})$, $\epsilon(t) = (0.95)^{t/1000}$ and the fast uses: $\rho(t) = \text{round}((t/1000)^{0.5})$, $\epsilon(t) = (0.9)^{t/1000}$

Interestingly, we see similar scaling laws as in Kaplan et al. [2020], even in this very different context in figure 1.

For the ODT, the hypothesis was that we should increase the difficulty and exploration slowly so the agent can still discover new trajectories through some exploration. Indeed, we see this in figure 2, where increasing the difficulty and exploration quickly results in less examples discovered. Interestingly, it also results in a larger loss in the replay buffer, suggesting that the data gathered is more noisy and the solutions discovered less useful.

However, the final results in table 1 are worse than expected. The problem with self-imitation is that there is a chicken and egg problem. To be able to solve an example, we need similar examples to be in the replay buffer. To add a similar example to the replay buffer, we need to solve it. The hope was that using exploration should lay the first egg, but it is clear from our experiments that more insights are needed.

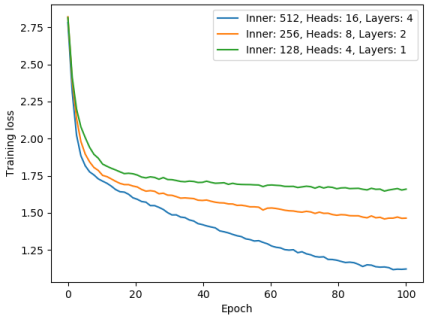

Figure 1: Scaling Laws for the Decision Transformer

Table 1: Number of solutions per 100 random shuffles, where the parentheses are (number of layers, number of heads, size of layers).

# of Shuffles	ODT with (512, 16, 4)	DT with r_{-1} (512, 16, 4)	DT with r_{-1} (256, 8, 2)	DT with r_{-1} (128, 4, 1)	DT with r_0 (256, 8, 2)	DT with r_0 (512, 16, 4)	Default RT
1	14	79	86	90	100	75	100
2	4	65	85	79	97	64	100
3	1	60	46	47	70	54	100
4	0	36	38	37	64	37	99
5	0	20	24	22	47	28	89
6	0	10	18	21	29	22	64
7	0	5	11	12	16	11	65
8	0	9	9	4	12	6	39
9	1	4	3	3	4	5	25
10	0	1	1	2	1	4	24

6 Conclusion

In summary, we have developed specific techniques for RT which generalizes to RL domain, investigated the influence of the reward function and model capacity for DT, and proposed a method to transform DT to online DT. Surprisingly, RT outperforms DT, which suggests that it is more effective to apply those techniques that we developed than using an off-the-shelf DT algorithm. Furthermore, we have spotted the fundamental flaw of DT via comparison of performances using different rewards, which is that we must specify a specific return-to-go to derive the desired action. Unfortunately, ODT requires further insight to be useful, but we have laid a foundation for future work.

References

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Andrej Karpathy. nanogpt: Minimalistic gpt implementation. `https://github.com/karpathy/nanoGPT`, 2021. Accessed: April 19, 2023.

Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. In *International Conference on Machine Learning*, pages 3878–3887. PMLR, 2018.

Keiran Paster, Sheila McIlraith, and Jimmy Ba. You can't count on luck: Why decision transformers fail in stochastic environments. *arXiv preprint arXiv:2205.15967*, 2022.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):4555–4576, 2021.

Thomas Wolf, Victor Sanh, Julien Chaumond, and Clement Delangue. Decision transformer. `https://huggingface.co/docs/transformers/model_doc/decision_transformer.html`, 2021. Accessed: April 19, 2023.

Omry Yadan. Hydra - a framework for elegantly configuring complex applications. Github, 2019. URL `https://github.com/facebookresearch/hydra`.

Shangtong Zhang and Richard S Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017.

Qinqing Zheng, Amy Zhang, and Aditya Grover. Online decision transformer. In *International Conference on Machine Learning*, pages 27042–27059. PMLR, 2022.

A Contributions

1. **Ted**: Built the regular transformer, including the special masking, weighted cross entropy, and training, performed the theoretical and experimental analysis, wrote up the transformer part of the paper, did the literature review, came up with the direction the project should go in.
2. **Mark**: Built the decision transformer and the self-imitating transformer, performed its theoretical and experimental analysis, wrote up the DT and ODT part of the paper, and helped write the rest of the paper.
3. **Shaul**: Provided the environment, utilities, foundation work like NanoGPT. Translated Decision Transformer code to the 15 puzzle and ran experiments which are in the appendix due to page limitations, and briefly wrote up that part of the paper.

B Experiment Details

Representation: To represent the state of the 3×3 Rubik's cube, instead of a naive color labelling which costs 54 tokens each state, we spend 26 tokens to represent all pieces (e.g. corner piece with 3 colors, edge piece with 2 colors, center piece with 1 color). This requires a larger state token space of size 78, but it is intuitively a good trade-off because the raw color representation significantly consumes the transformer's context length.

Architectural limitation of transformer: Ideally, we want the transformer to restrict its attention to tokens of action space rather than all tokens including tokens to represent state because we have the full access to the deterministic transition function. However, this is not possible without a significant redesign of the architecture. To inspect this limitation, we recall that during training time, the transformer masks the attention matrix for the model to predict a sequence s' where the position i'th slot of s' is predicted given $s[:i+1]$ where s is the input sequence and we used Python slicing syntax here. Then, we compare s' with the target sequence t using cross entropy loss. However, t contains both action and state tokens, and therefore we must allow the transformer to output all tokens, not just action tokens.

To run the experiments, we used the facebook framework Hydra by Yadan [2019] to be able to easily store past experiments and manage hyperparameters. All decision transformer and online decision transformer experiments where run on lambda labs using 1 Nvidia A10 GPU (24 GB PCIe) with 30 vCPUs, 200 GiB RAM, and 1.4 TiB SSD.

As an implementation trick, for all of the architectures above, instead of simply training the entire trajectory, we can break symmetry and increase the size of the training data by randomly choosing an index to start from, starting from the middle. That way, a data point of length N actually gives us N different data points.

For the decision transformer, another trick is to first normalize the state tokens, then shift and scale all future states by the same amount. This seems to result in faster training. Sadly, such a technique is hard to generalize to the online transformer as the training data is always changing, and starts off small.

For fairness, in all of these experiments, we will fix the following hyperparameters that were lightly tuned by hand: the learning rate is 10^{-3}, the batch size is 128, the size of the hidden state is 128, the number of training examples is 10000, the length of each example is 20 (so in particular, we have 200,000 possible trajectories to train on), and the number of epochs is 100. For the largest model, we use a decay rate of 10^{-4} and otherwise a decay rate of 10^{-8} (as opposed to dropout).

In our experiments with the online-DT, we set $K = 50$ and the number of training examples generated to 1000.

In building the table, we give the model $\min(10, 3n)$ chances to solve the cube, when the cube has been shuffled for n steps.

C 15 puzzle

We adapted the Decision transformer model to the 15 puzzle. For this the (lightly tuned) hyper parameters were a learning rate of 10^{-3}, batch size of 128, hidden state size of 128, 8 heads, 2 layers, 10000 training examples each of length 100, a decay rate of 10^{-8} and 10 epochs. After this it was able to solve 50% of puzzles generated by 5 random moves, 20% generated by 20 random moves and none of 40 or more random moves.