A Review of the Decision Transformer Architecture:

Framing Reinforcement Learning as a Sequence Modeling Problem

Timo Imhof

***Abstract*— This article provides a review of the Decision Transformer, a new approach to framing reinforcement learning as a sequence modeling problem. For modeling trajectories as sequences, the representation is modified by adding the return-to-go token. The modeling architecture is a decoder-only variant of the Transformer architecture and uses masked self-attention for autoregressive action prediction. Results of the original paper show that the Decision Transformer provides a valid alternative in offline reinforcement learning tasks but leave space for future work in regards to the importance of the size of the context window and effect of the return-to-go token.**

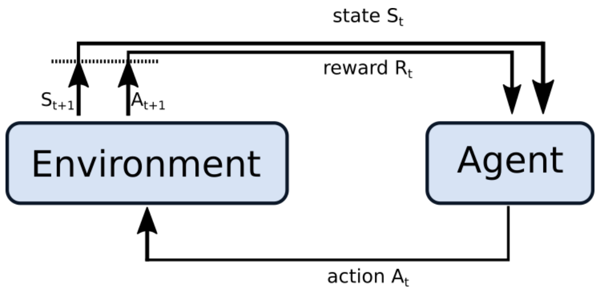

Fig. 1. The agent-environment cycle visualizes the main principles of reinforcement learning: observing states, choosing actions, and receiving rewards.

I. INTRODUCTION

Learning by interaction with an environment is a foundational idea underlying many theories of intelligence [15]. For humans, interaction plays a crucial role in learning and represents an important source of knowledge about the environment. A computational approach to learning from interaction is called Reinforcement Learning (RL) [15]. The field of RL focuses on the development of efficient methods that enable an agent to learn goal-directed behavior from interaction with an environment.

Formally, problems in RL are formulated with states, actions, and rewards (see Figure 1). At time step t, an agent situated in the state s_t takes an action a_t which transfers the agent to a successor state s_{t+1}. The decision of what action to take in which state is based on a policy, a set of rules that determines the interaction of the agent with the environment. A distinction is drawn between deterministic policies $a = \pi(s)$ and probabilistic policies $a \sim \pi(a|s)$, where a state is mapped to a probability distribution over actions instead of a single action.

For every choice of action, the agent receives a reward, which serves as a basis for evaluating the quality of its choice. Over the lifetime of an agent, the rewards of the individual timesteps are accumulated, and the cumulative reward is called return. The task of the agent is to change and develop its policy in a way that maximizes cumulative rewards (return) and, at each timestep, takes the optimal decision for each state, which results in optimal behavior.

The conventional algorithms tackling the given problem setting in RL are divided into two classes: model-free and model-based [7]. Both have different approaches: model-free methods directly learn policies based on evaluation of sampled trajectories, whereas model-based methods first build a model of a robot's state dynamics and subsequently use this model for policy improvement [7]. Although algorithms from both classes are successfully trained and used on different RL tasks, they also have their limitations and require additional modifications to ensure stability and convergence. Model-based approaches are strongly dependent on the quality of the learned model. Due to modeling errors, they are often brittle and need additional regulations and constraints to prevent model exploitation. Model-free strategies require real robot interactions to create data and therefore are more time-consuming [7].

A new approach, framing RL as a sequence modeling problem, is the Decision Transformer (DT). The DT is based on the Transformer architecture [16], a network architecture widely used in fields, such as Natural Language Processing, or Computer Vision, due to its scalability and ability to resolve long-range dependencies [5]. Transformers have shown successes in a variety of applications [4, 13], which resulted in several variants of the original framework due to their modularity. The DT is also a modified version of a Transformer and performs autoregressive action prediction over trajectories. Reward assignment is done directly via self-attention and therefore dispenses with components normally used in conventional RL, e.g., value functions or policy gradients [5].

This paper presents a review of the DT. First, we briefly introduce how sequence modeling is applied to RL. Next, we give an overview of the used Transformer architecture and provide details about the changes that have been made for the DT. After that, we cover the self-attention mechanism used in Transformers and how it is used for RL. Furthermore, we analyze the learning performance and other properties of DT based on the experimental results [5].

II. RELATED WORK

A similar approach using the Transformer architecture is the Trajectory Transformer [9]. The paper repurposes beam search for sequence generation by biasing trajectory samples according to cumulative reward. The flexibility of this approach is demonstrated across long-horizon dynamics prediction, imitation learning, goal-conditioned RL, and offline RL.

III. Reinforcement Learning as a Sequence Modelling Problem

In this section, we focus on the formulation for RL as a sequence modeling problem and describe how the regular trajectory representation is modified for sequence modeling with a Transformer architecture.

In sequence modeling, a trajectory is seen as a list of tokens, $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, ..., s_T, a_T, r_T)$. Three tokens (s_t, a_t, r_t) correspond to one timestep, therefore a trajectory with k timesteps is represented by $3k$ tokens. The DT trains on rollouts of fully completed trajectories and should be able to generate actions based on future desired returns. Because the DT is an autoregressive model, it can only attend over past tokens and past rewards, which do not reflect the future desired returns. Therefore the trajectory representation is adjusted and the reward tokens are replaced with a return-to-go token $\hat{R}_t = \sum_{t'=t}^{T} r_{t'}$. Modifying a trajectory with the return-to-go leads to the following representation $\tau^{\mathrm{DT}} = (\hat{R}_0, s_0, a_0, \hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, ..., \hat{R}_T, s_T, a_T)$. In every timestep, when executing an action, the achieved reward is subtracted from the return-to-go token and the decremenation process is repeated until episode termination. The trajectory representation as sequence should enable learning the internal structure of the sequence and discovering meaningful patterns, to be able to conditionally generate actions at test time.

IV. Decision Transformer

In this section, we provide an overview of the Decision Transformer. First, we present the architectural details and modifications due to the RL setting. Further, we explain the self-attention mechanism used for the next token prediction and show the training process with trajectories.

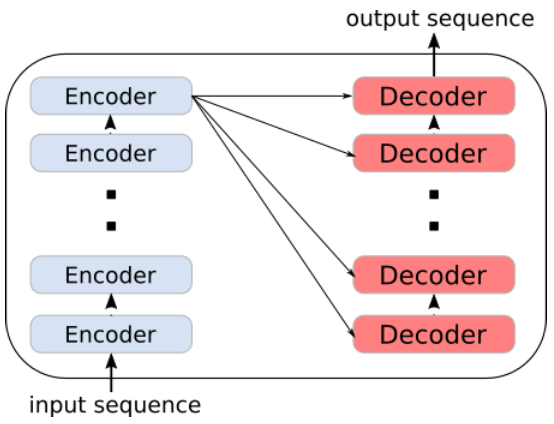

Fig. 2. Encoder-Decoder structure of a Transformer. An input sequence is passed through a series of encoders sequentially, after which the final encoder output is passed in parallel to a stack of decoders.

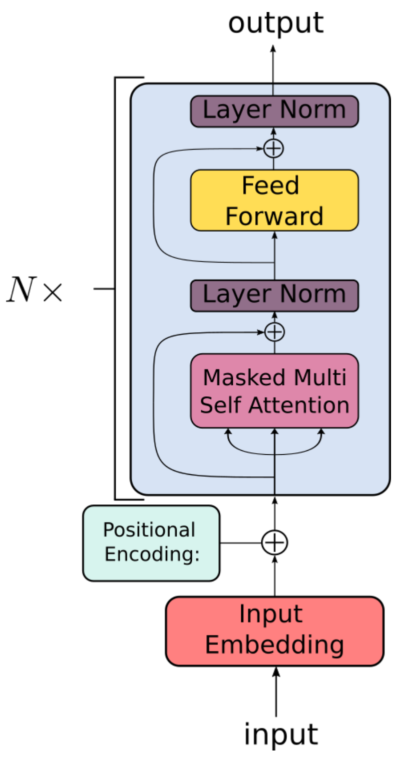

Fig. 3. GPT architecture used for the Decision Transformer. After an input sequence is passed through the input embeddings and positional encodings it is processed by N stacked identical decoders.

A. *Architecture Foundations of the Decision Transformer*

The Transformer was first presented as a neural network architecture for efficient modeling of sequential data [16]. Since then Transformers have significantly influenced progress in machine learning fields such as Natural Language Processing [6, 17] and Computer Vision [18, 8]. The original architecture consists of stacked encoder and decoder blocks (see Figure 2), but thanks to its popularity, several variations of the architecture have emerged [10, 14, 1].

The foundation for the DT is the Generative Pre-trained Transformer (GPT) model [12], an n-layer decoder-only Transformer (see Figure 3). The GPT was presented for pre-training a language model on unlabeled text corpora, leveraging linguistic information to learn a general representation, used further for task-specific fine-tuning. N-layer stands for n stacked identical decoders. Each decoder has two sublayers; first, a masked multi-head self-attention mechanism, and second, a position-wise fully connected feed-forward network. For further information, consult the original papers [12, 16].

Due to the self-attention mask, the GPT model generates tokens autoregressively, which means that at time step t, the self-attention mask limits the transformer to attend over tokens with a time-index of $t-1$ or smaller and not use the whole sequence. When seeing a trajectory as a sequence of tokens, there are three tokens at each timestep with different token types: return-to-go, state, and action. The GPT architecture is designed for one token type and one token at one timestep, so for the DT, the input embeddings and the positional encodings of the GPT architecture are modified. To obtain token embeddings, a linear layer is learned for each token type, projecting raw inputs to the embedding dimensions [5]. For the positional encoding, an additional linear layer is learned for each time step. Since in RL one timestep responds to three tokens (R_t, s_t, a_t), the DT differs to the standard positional encoding of the GPT, and learns a new linear layer for every three tokens.

B. Self-Attention Mechanism

Self-attention is an attention mechanism relating different positions of a single sequence to compute a representation of the sequence [16]. The computational process is a reweighing of the mathematical token representation with a key, query, value concept to add information about individual token dependencies and therefore produce a context richer mathematical substitution of the sequence. To get an intuition, consider the example ”Bank of the river”. Depending on the context, the word ”bank” can have several meanings. The goal is, by using self-attention, to mathematically leverage the interdependencies of the individual words and add context to indicate that ”Bank” refers to ”river” and therefore is not a bank in the park to sit on. Figure 3 visualizes one self-attention function. The vectors (v_1, v_2, v_3, v_4) represent the individual tokens of ”Bank of the river”. To capture the dependencies of all tokens concerning the token ”river” (v_4) we use v_4 as the query. The keys are the original vectors (v_1, v_2, v_3, v_4) and the normalized dot product between keys and query results in the individual weights $(w_{41}, w_{42}, w_{43}, w_{44})$ for the values. The values are also the original vectors (v_1, v_2, v_3, v_4) and are multiplied by the weights, which results in a context richer representation of each token in relation to the token ”river”.

A distinction is made between additive and multiplicative attention (dot product), but due to the speed and space-efficiency in Transformers, dot product attention is used [16].

With multi-head attention instead of performing a single attention function, the Transformer performs multiple attention functions in parallel. The inputs are linearly projected versions of the keys, queries, and values. Using several functions enables the model to jointly attend to information from different representation subspaces at different positions, which permits for a result richer in context. The output values of the individual functions are concatenated and projected, resulting in the final values.

Self-attention is limited to a context window of size k, allowing the Transformer to only attend over k tokens.

C. Training Process with the Decision Transformer

For training, the DT uses sampled minibatches of sequence length N from a dataset of offline trajectories. With a context window of size w, the DT attends over w earlier tokens for predicting the next action token a_t at the timestep t.

The loss between the newly generated token ${a_t}^{\text{new}}$ and the original token a_t from the sequence is measured using the mean squared error, and the parameters of the DT are optimized via gradient descent. Executing the generated token a_t results in feedback in the form of a new state s_{t+1} and return-to-go token $R_{t+1} = R_t - r(s_t, a_t)$. The new tokens a_t, s_{t+1}, and R_{t+1} are appended to the sequence and the process is repeated until episode termination. For a visual representation, see Figure 5.

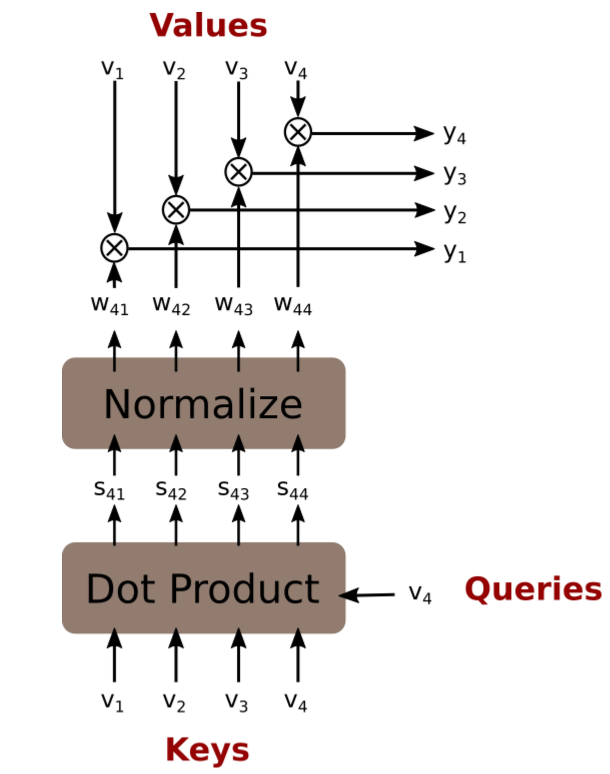

Fig. 4. Single self-attention operation. The keys, queries and values concept outputs a weighted version of the input with more context added.

V. Evaluation of the Decision Transformer in Offline Reinforcement Learning

In this section, we give an overview of the performance of the DT. We first provide definitions for the terms offline reinforcement learning, Markov Decision Process, and Temporal Difference learning. Afterward, we present the learning performance of the DT and further properties based on the experimental results [5].

A. Offline Reinforcement Learning

In offline reinforcement learning, only a limited amount of data stands available. The given dataset consists of trajectory

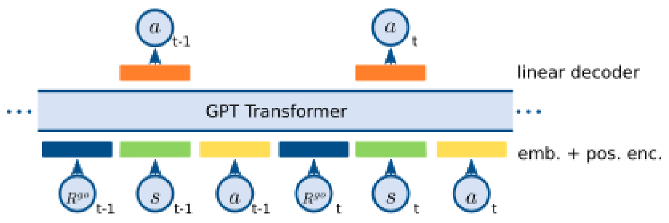

Fig. 5. Next-token generation of the Decision Transformer. At timestep t, for predicting a_t, the DT attends over w preceding tokens, including s_t.

rollouts of arbitrary policies and the goal is to learn a good policy based on these trajectories. An offline RL setting is harder because an agent cannot collect additional feedback (data) by exploring the environment, as it is possible in online reinforcement learning.

B. Markov Decision Process

A Markov decision process (MDP) is a mathematical model of decision-making under uncertainty defined by

- state space S,
- action space A,
- transition dynamics $p(s_{t+1}|s_t, a_t)$,
- reward function $r(s, a)$,
- initial state distribution $\mu_0(s)$,

constituting the tuple (S, A, p, r, μ_0). The MDP leverages the Markov property. The conditional probability distribution of future states of the process only depends upon the present state and not the past. In the context of RL the Markov property means that the state s_{t+1} only depends upon the state s_t and the action a_t resulting from the policy, $p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ...) = p(s_{t+1}|s_t, a_t)$ [11].

C. Temporal Difference Learning

Temporal Difference Learning (TD-Learning) algorithms update a value function based on a one-step lookahead prediction. The prediction value is computed with a one-step prediction error, called temporal difference error or TD-error. Algorithmic Description of TD-Learning with discount factor γ and step-size parameter α:

1) Observe transition (s_t, a_t, r_t, s_{t+1})
2) Compute TD error $\delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t)$
3) Update V-function $V_{t+1}(s_t) = V_t(s_t) + \alpha\delta_t$

Instead of using the V-function, TD-Learning can also be done for the Q-function. For a detailed explanation see [15]. A state-of-the-art model-free TD-learning algorithm is Conservative Q-Learning (CQL) [5], which was also used as a comparison to the performance of the DT.

D. Test Setting

The DT was evaluated alongside model-free offline RL and imitation learning algorithms, all trained in a MDP in an offline RL setting. The main points of comparison are CQL for TD-learning and Behavior Cloning (BC) for imitation learning. The performance was measured in different discrete tasks from Atari [2] and continuous control tasks from OpenAIGym [3]. In Atari, the DT performs comparably to CQL on 3 out of 4 games and better than BC in 3 out of 4 games. Whereas in the OpenAiGym tasks, the DT outperforms both CQL and BC in nearly all tasks. For more detailed results, see the original tables [5].

E. Further Experiments

The authors documented further experiments, which are summarized in this subsection.

1) Decision Transformer in Comparison to Behavior Cloning: To evaluate characteristics of the DT concerning BC, Percentile Behavior Cloning (%BC) was introduced as a new learning method. %BC is defined by performing BC on only the X best percent of all trajectories available in the dataset (e.g. 10%BC, use the best 10% of available trajectories for behavior cloning). Tested were different percentages $(10\%, 25\%, 40\%, 100\%)$ in a low and high data setting. When data is plentiful 10%BC achieves the best performance, closely followed by the DT. In contrast, when using low amounts of data %BC performs better with increasing percentage, but still is outperformed by the DT.

2) Specifying the Return-To-Go Token at Test Time: At test time the DT is handed the first return-to-go token R_0, indicating the desirable return to reach in the task. In this experiment for different tasks, R_0 was chosen in a high variation, to test how well the DT can model the distribution of returns. The results show a high correlation between desired target return and true observed return.

3) Using a Longer Context Length: The context length K specifies how many tokens the DT can attend for predicting the next token. In this experiment a DT with context length $K = 30$ (or $K = 50$ for Pong) and another DT with $K = 1$ were compared on the same Atari games, resulting in superior performance when using a larger context length.

4) Performing Effective Long-Term Credit Assignment: The Key-to-Door environment task consists of 3 phases:

1) agent is placed in a room with a key
2) agent is placed in an empty room
3) agent is placed in a room with a door

where an agent first receives reward when reaching the door, but only if it picked up the key in the first phase. To complete the task successfully, reward must be propagated through the whole episode. The authors compared the DT, CQL, BC, and %BC with using 1K and 10K random trajectories for training (%BC was only trained on the successful trajectories). While DT and %BC performed equally well, CQL and normal BC both performed poorly.

5) DT as Accurate Critic in Sparse Reward Settings: Here the DT was modified to output return tokens in addition to action tokens on the Key-to-Door environment. The additional return tokens indicate the reward probability throughout the task. According to the authors, the DT constantly updates the reward probability based on events. When looking at a particular successful episode the attention weights from all time steps are concentrated on key events, such as picking up the key and reaching the door.

6) Performance in Sparse Reward Settings: In this experiment, the DT was compared to CQL, BC, and %BC in a delayed return version of the D4RL benchmarks in the OpenAIGym task Hopper. All reward earned along the trajectory is received only at the final timestep. The DT achieves slightly worse results than in the original dense reward benchmarks, while CQL performs poorly.

VI. Discussion

In this section, we discuss the results reported for the experiments and present questions arising from them.

There is an inconsistency in the results presented in [5] in Table 2 and Table 3. The Reacher environment with medium dataset scores 48.9 in BC and 58.4 in 100%BC. The results for BC and also %BC are originally conducted by the paper [5] and not taken from other sources. What stands out is the performance loss for 100%BC in HalfCheetah with the Medium-Replay Dataset. The other percentages perform nearly equal, ranging from 40.8 to 41.1, whereas 100%BC reaches a score of 4.3. We suspect the 4.3 could be a typing error, but if not, it would be interesting to know why there is such a huge performance gap between 40%BC and 100%BC.

Furthermore, the results from %BC give some interesting insights on the effectiveness of BC concerning the quality and quantity of the data used for training. In general, one would assume that 100%BC (normal BC) performs best since it sees all the data and therefore generalizes better to unseen situations, which would result in more stable policies. For settings with low amounts of data, this assumption is reasonable as shown in table 4 in [5]. Performing %BC on a too-small dataset deteriorates the performance, becoming worse as the percentage of seen trajectories decreases. But when data is plentiful, on average 10%BC achieves the best results. We suggest, that above a certain threshold of available data, quality is more important than quantity for learning a good policy with BC. However, this is only the case after a minimum data size is exceeded and generalization becomes possible based on the quality subset. The DT somewhat fits the assumption. The paper assumes that in high data regimes, the DT can focus on a subset of trajectories after seeing all data and therefore focus on the quality. But when the amount of available data is low the DT outperforms %BC because it can see and use all the data and therefore achieve better generalization.

In the experiments in [5], the size of the context window is adjusted on a per-task basis. For example, in the key-to-door environment, the context window is of the size of the episode length. The paper evaluated that a longer context length is beneficial for performance. But it would be interesting to further investigate, how the efficiency of the DT decreases with an increasingly smaller context window. Maybe additional experiments in this direction could provide insight into how huge the performance loss would be if the context window can't cover sufficient events for generating a reliable next token and if the DT could be modified to mitigate such a problem.

The results indicate that the DT provides a valid alternative to conventional approaches in the offline RL setting, but it is not apparent how it fits among other offline RL algorithms. Based on the correlation in scores with %BC the DT could be seen as an advanced model in imitation learning. It also predicts based on seen data, but can more effectively utilize prior experience in low data settings and therefore outperform BC as well as %BC. However, it is unclear what role return-to-go plays in the effectiveness of DT. Because Behavior Cloning is reward agnostic [5], further evaluation with reward-dependent algorithms could provide more insights into this.

VII. Conclusion

We reviewed the Decision Transformer, which generates future action tokens based on preceding states, action, and return-to-go tokens. The approach proposes a method for reinforcement learning via sequence modeling. We provided additional explanations and detailed descriptions to make the topic more accessible for readers without a background in reinforcement learning and sequence modeling. Results of the original paper show that the Decision Transformer provides a valid alternative in offline reinforcement learning tasks and therefore lays a foundation for future work for combining ideas in sequence modeling and reinforcement learning.

Acknowledgment

We especially thank Boris Belousov for his guidance, assistance in research, and general support. This research was initiated and supported by the Intelligent Autonomous Systems group of the TU-Darmstadt within the scope of the seminar "Humanoid Robotics" by Oleg Arenz.

References

[1] Anurag Arnab et al. "ViViT: A Video Vision Transformer". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2021, pp. 6836–6846.

[2] Marc G Bellemare et al. "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[3] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[4] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[5] Lili Chen et al. "Decision Transformer: Reinforcement Learning via Sequence Modeling". In: *CoRR* abs/2106.01345 (2021). arXiv: 2106.01345. URL: https://arxiv.org/abs/2106.01345.

[6] Anton Chernyavskiy, Dmitry Ilvovsky, and Preslav Nakov. "Transformers: "The End of History" for NLP?" In: *CoRR* abs/2105.00813 (2021). arXiv: 2105.00813. URL: https://arxiv.org/abs/2105.00813.

[7] Marc P. Deisenroth, Gerhard Neumann, and Jan Peters. *A Survey on Policy Search for Robotics*. Ed. by H. Christensen and R. Siegwart. Vol. 2. Foundations and Trends in Robotics 1–2. ISBN: 978-1-60198-702-0. NOW Publishers, 2013, pp. 1–142. DOI: http://dx.doi.org/10.1561/2300000021.

[8] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *CoRR* abs/2010.11929 (2020). arXiv: `2010.11929`. URL: `https://arxiv.org/abs/2010.11929`.

[9] Michael Janner, Qiyang Li, and Sergey Levine. “Offline Reinforcement Learning as One Big Sequence Modeling Problem”. In: *Advances in neural information processing systems* 34 (2021).

[10] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. “Reformer: The Efficient Transformer”. In: *CoRR* abs/2001.04451 (2020). arXiv: `2001.04451`. URL: `https://arxiv.org/abs/2001.04451`.

[11] Martin L. Puterman. “Markov Decision Processes: Discrete Stochastic Dynamic Programming”. In: *Wiley Series in Probability and Statistics*. 1994.

[12] Alec Radford et al. “Improving language understanding by generative pre-training”. In: ().

[13] Aditya Ramesh et al. “Zero-Shot Text-to-Image Generation”. In: *CoRR* abs/2102.12092 (2021). arXiv: `2102.12092`. URL: `https://arxiv.org/abs/2102.12092`.

[14] Leo Sampaio Ferraz Ribeiro et al. “Sketchformer: Transformer-Based Representation for Sketched Structure”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[16] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: `1706.03762`. URL: `http://arxiv.org/abs/1706.03762`.

[17] Thomas Wolf et al. “Transformers: State-of-the-Art Natural Language Processing”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. DOI: `10.18653/v1/2020.emnlp-demos.6`. URL: `https://aclanthology.org/2020.emnlp-demos.6`.

[18] Bichen Wu et al. “Visual Transformers: Token-based Image Representation and Processing for Computer Vision”. In: *CoRR* abs/2006.03677 (2020). arXiv: `2006.03677`. URL: `https://arxiv.org/abs/2006.03677`.