Transformers for Reinforcement Learning

Omar Bahri
Department of Computer Science
Utah State University
Logan, UT, USA
omar.bahri@usu.edu

***Abstract*—The motivation of most classical reinforcement learning algorithms is to learn an optimal, single-step policy by capitalizing on the Markov property. However, by taking a step back and redefining the problem as sequence modeling, where the goal is to predict a sequence of actions based on a previous sequence of states-actions-rewards, it becomes very tempting to leverage state-of-the-art sequence-to-sequence models. Janner et al. [1] and Chen et al. [2] did exactly that by training a transformer architecture on offline benchmark datasets. Their results show the superiority of this new approach. In this project, I aim to 1) learn about transformers and the attention mechanism, 2) explore and understand the work done in [1] and [2], 3) generate my own offline dataset on a simple problem, in compliance with the D4RL benchmark [3] format, and 4) evaluate Chen et al.'s Decision Transformer performance on this dataset, with the motivation of verifying whether Decision Transformer can achieve better results using a dataset generated using a simple algorithm?**

I. Introduction

Reinforcement learning RL is one of the three machine learning paradigms, along with supervised and unsupervised learning. RL algorithms work in a goal-oriented manner, by breaking down a complex goal into smaller steps and working towards the optimal solution by rewarding each small step; i.e. correct decisions are reinforced via incentivisation. From the simple Q-learning algorithm to the more complex algorithms that make use of Deep Neural Networks (DNNs), passing by Monte-Carlo and Actor-Critic approaches, the motivation of classical RL algorithms is to learn an optimal, single-step policy by capitalizing on the Markov property. In 2021, [1] and [2] explored the RL problem under another angle; they considered it as a sequence generation problem that aims to produce a sequence of actions which will acheive a sequence of high rewards. The main motivation behind this redefinition is to be able to leverage state-of-the-art sequence-to-sequence models, and in particular, transformers. The results presented in both papers have concluded that, by slightly adapting offline reinforcement learning datasets and training a transformer architecture, the model achieves performances superior to state-of-the-art algorithms. In both papers, the D4RL benchmark datasets were used for the evaluation [3].

The initial motivation of this project was for me to learn about transformers and the attention mechanism and to explore their application to RL through the problem redifinition proposed in [1] and [2]. After completing this first milestone and experimenting with their open-source code, I decided to push a step further and attempt to answer the following question. Can Decision Transformer [2] achieve the same results using a dataset generated by simply applying non-complex RL algorithm to a simple environment. Therefore, I have considered the OpenAI Gym [4] Pendulum environment, and generated two datasets in the D4RL benchmark [3] format, one using a random agent and the other using the Soft Actor-Critic (SAC) algorithm [5], [6]. Then, I trained the Decision Transformer model and both datasets, and evaluated the results. The rest of this report is organized as follows. In Section II, I describe transformers and the attention mechanism, decision transformer, and SAC as presented in the original papers; in Section III, I describe the experimental setup and present the results; and in Section III, I conclude with a summary.

II. Methods

A. Transformers

1) The Original Transformer Architecture: After the success of RNNs and encoder-decoder architectures, the first transformer was proposed in [7] in 2017. In this section, we will present its description as it came in the orginal paper. The transformer architecture follows an encoder-decoder architecture, where the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $z = (z_1, ..., z_n)$. Given z, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step the model is auto-regressive [8], consuming the previously generated symbols as additional input when generating the next. The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.

a) Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position- wise fully connected feed-forward network. We employ a residual connection around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is $LayerNorm(x + Sublayer(x))$, where $Sublayer(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{model} = 512$.

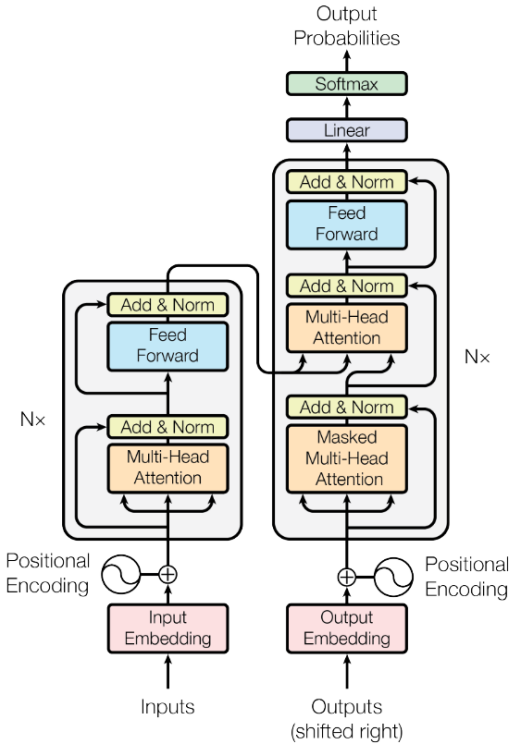

Fig. 1: The Original Transfomer Architecture [7]

b) Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

c) Attention: An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Scaled Dot-Product Attention: Vaswani et al. use what they call "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. They the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values. In practice, they compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. they compute the matrix of

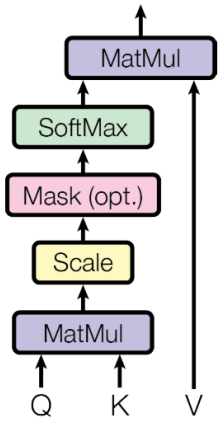

Fig. 2: Scaled Dot Product Attention [7]

outputs as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$

Multi-Head Attention: Instead of performing a single attention function with d_{model}-dimensional keys, values and queries, Vaswarani et al. found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 3. Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O \tag{2}$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in R^{hd_v \times d_{model}}$.

B. Decision Transformer

In [2], the authors proposed Decision Transformer, which models trajectories autoregressively with minimal modification to the transformer architecture, as summarized in Figure 4 and Algorithm 1. They used the GPT architecture [8], which modifies the transformer architecture with a causal self-attention mask to enable autoregressive generation, replacing the summation/softmax over the n tokens with only the previous tokens in the sequence ($j \in [1, i]$). The rest of this section includes a description of the model as presented in the original paper.

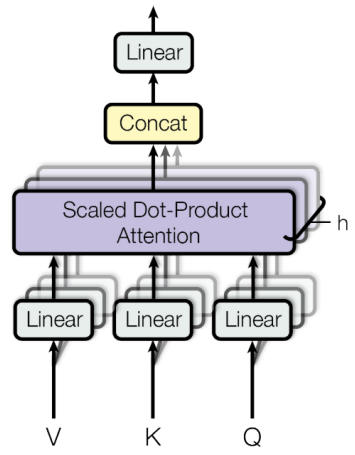

Fig. 3: Multi-Head Attention [7]

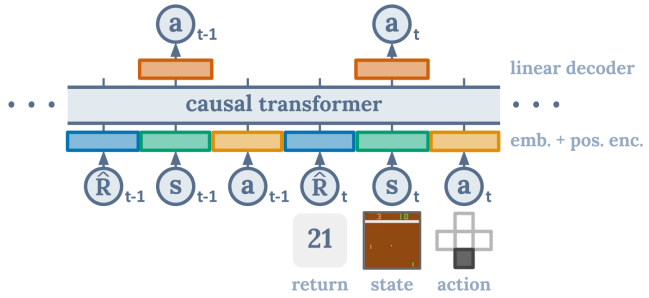

Fig. 4: Decision Transformer [2]

Algorithm 1 Decision Transformer Pseudocode (for continuous actions)

```
# R, s, a, t: returns-to-go, states, actions, or timesteps
# transformer: transformer with causal masking (GPT)
# embed_s, embed_a, embed_R: linear embedding layers
# embed_t: learned episode positional embedding
# pred_a: linear action prediction layer

# main model
def DecisionTransformer(R, s, a, t):
    # compute embeddings for tokens
    pos_embedding = embed_t(t)  # per-timestep (note: not per-token)
    s_embedding = embed_s(s) + pos_embedding
    a_embedding = embed_a(a) + pos_embedding
    R_embedding = embed_R(R) + pos_embedding

    # interleave tokens as (R_1, s_1, a_1, ..., R_K, s_K)
    input_embeds = stack(R_embedding, s_embedding, a_embedding)

    # use transformer to get hidden states
    hidden_states = transformer(input_embeds=input_embeds)

    # select hidden states for action prediction tokens
    a_hidden = unstack(hidden_states).actions

    # predict action
    return pred_a(a_hidden)

# training loop
for (R, s, a, t) in dataloader:  # dims: (batch_size, K, dim)
    a_preds = DecisionTransformer(R, s, a, t)
    loss = mean((a_preds - a)**2)  # L2 loss for continuous actions
    optimizer.zero_grad(); loss.backward(); optimizer.step()

# evaluation loop
target_return = 1  # for instance, expert-level return
R, s, a, t, done = [target_return], [env.reset()], [], [1], False
while not done:  # autoregressive generation/sampling
    # sample next action
    action = DecisionTransformer(R, s, a, t)[-1]  # for cts actions
    new_s, r, done, _ = env.step(action)

    # append new tokens to sequence
    R = R + [R[-1] - r]  # decrement returns-to-go with reward
    s, a, t = s + [new_s], a + [action], t + [len(R)]
    R, s, a, t = R[-K:], ...  # only keep context length of K
```

1) Trajectory Representation: The key desiderata in the authors' choice of trajectory representation are that it should enable transformers to learn meaningful patterns and should be able to conditionally generate actions at test time. It is nontrivial to model rewards since we would like the model to generate actions based on future desired returns, rather than past rewards. As a result, instead of feeding the rewards directly, the model is fed with the returns-to-go $\hat{R}_t = \sum_{t'=t}^{T} r_{t'}$. This leads to the following trajectory representation which is amenable to autoregressive training and generation:

$$\tau = (\hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, ..., \hat{R}_T, s_T, a_T) \tag{3}$$

At test time, the desired performance (e.g. 1 for success or 0 for failure) can be specified, as well as the environment starting state, as the conditioning information to initiate generation. After executing the generated action for the current state, the target return is decremented by the achieved reward and repeat until episode termination.

2) Architecture: The last K timesteps are fed into Decision Transformer, for a total of $3K$ tokens (one for each modality: return-to-go, state, or action). To obtain token embeddings, a linear layer is learned for each modality, which projects raw inputs to the embedding dimension, followed by layer normalization. For environments with visual inputs, the state is fed into a convolutional encoder instead of a linear layer. Additionally, an embedding for each timestep is learned and added to each token – note this is different than the standard positional embedding used by transformers, as one timestep corresponds to three tokens. The tokens are then processed by a GPT [8] model, which predicts future action tokens via autoregressive modeling.

3) Training: Given a dataset of offline trajectories, minibatches of sequence length K are sampled from the dataset. The prediction head corresponding to the input token s_t is trained to predict a_t – either with cross-entropy loss for discrete actions or mean-squared error for continuous actions – and the losses for each timestep are averaged.

C. Soft Actor-Critic

The Soft Actor-Critic (SAC) [9] is a RL algorithm that works for continuous action environments. In this section, I will describe the algorithm as presented in the original paper and in [10]. The biggest feature of SAC is that it uses a modified RL objective function. Instead of only seeking to maximize the lifetime rewards, SAC seeks to also maximize the entropy of the policy. The high entropy is maximized to explicitly encourage exploration, to encourage the policy to assign equal probabilities to actions that have same or nearly equal Q-values, and also to ensure that it does not collapse into repeatedly selecting a particular action that could exploit some inconsistency in the approximated Q function. Therefore, SAC overcomes the brittleness problem by encouraging the policy network to explore and not assign a very high probability to any one part of the range of actions. The objective function is thus augmented with the expected entropy of the policy as follows, where $\mathcal{H}$ is the entropy and α determines the relative im-portance of the entropy term against the reward.

$$J(\pi) = \Sigma_{t=0}^{T} \mathbf{E}_{(s_t,a_t)} \sim \rho_\pi[r(s_t,a_t) + \alpha\mathcal{H}(\pi(.|s_t))] \quad (4)$$

SAC makes use of three networks: a state value function V parameterized by Ψ, a soft Q-function Q parameterized by θ, and a policy function π parameterized by ϕ. While there is no need in principle to have separate approximators for the V and Q functions which are related through the policy, the authors say that in practice having separate function approximators help in convergence. The Value network is trained by minimizing the error in Equation (5), the Q network by minimizing the error in Equation (6), and the policy network π by minimizing the error in Equation (8).

$$J_v(\psi) = \mathbb{E}_{s_t\sim D}[\frac{1}{2}(V_\psi(s_t) - \mathbb{E}_{a_t\sim\pi_\phi}[Q_\theta(s_t,a_t) - log\pi_\phi(a_t|s_t)])^2] \quad (5)$$

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim D}[\frac{1}{2}(Q_\theta(s_t,a_t) - \hat{Q}(s_t,a_t))^2] \quad (6)$$

where:

$$\hat{Q}(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\psi}}(s_t+1)] \quad (7)$$

$$J_{pi}(\phi) = \mathbb{E}_{s_t\sim D}[D_{KL}(\pi_\phi(.|s_t)||\frac{exp(Q_\theta(s_t,.))}{Z_\theta(s_t)})] \quad (8)$$

where D_{KL} is the Kullback-Leibler Divergence [11].

III. Datasets Generation

In order to answer the question of whether Decision Transformer is able to achieve a high performance on a simple RL environment, using a custom generated dataset, I decided to consider the OpenAI Gym Pendulum environment and generate my own D4RL [3] compliant datasets. The approach I followed is similar to the work of [3]. The datasets I generated are as follows.

- *pendulum-random*: generated using a random agent, 1,000,000 time steps.
- *pendulum-medium-replay*: generated for the replay buffer of a SAC agent, 1,000 time steps.
- *pendulum-medium*: generated by a trained SAC agent, 200,000 time steps.

IV. Results

I trained Decision Transformer on each of the three datasets: pendulum-random, pendulum-medium-replay, and pendulum medium. After hyper-parameter tuning, the results of the best model for each dataset are presented in this section. Table 1 shows the parameters that result in the best performance.

TABLE I: Best Parameters

Model	Batch Size	Embedding Dimension	Learning Rate	Weight Decay	Number of Epochs
random	64	128	10^{-4}	10^{-4}	10^6
medium-replay	64	32	10^{-2}	10^{-4}	10^6
medium	64	64	10^{-2}	10^{-4}	10^6

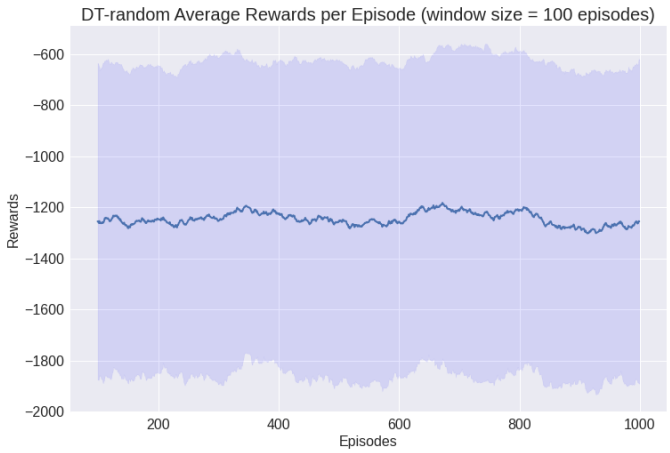

Fig. 5: Learning on pendulum-random

A. Random Agent Dataset

Training Decision Transformer on the pendulum-random dataset agent was not succssful. The learned policy produced results similar to a random agent. The learning curve is shown in Figure 5. While this behaviour is expected as the dataset does not contain any useful information, one of the main motivations for offline reinforcement learning is to be able to learn better policies from lower quality datasets, including randomly generated ones. This ensures that the model is not doing *imitation learning*, such as *behavior cloning*. In the original paper, Decision Transformer [2] is able to achieve significant learning from random datasets.

B. SAC Training

As we can see in Figure 6, the SAC agent is able achieve good returns on the Pendulum problem starting from the 200^{th} iteration. I use the replay buffer of this model to create pendulum-medium-replay dataset, and the trained agent to produce the pendulum-medium dataset.

C. Medium Datasets

1) pendulum-medium-replay: The performance of the Decision Transformer model on the pendulum-medium-replay dataset did slightly improve over a random agent. However, it did not reach that of the SAC agent. Figure 7 shows its learning curve.

2) pendulum-medium: The performance of the Decision Transformer model on the pendulum-medium dataset was significantly better than a random agent. However, it still did not reach that of the SAC agent. Figure 8 shows its learning curve. As mentioned in Table 1, the model was trained for 10^6 iterations. However, it is only evaluated on 200 episodes

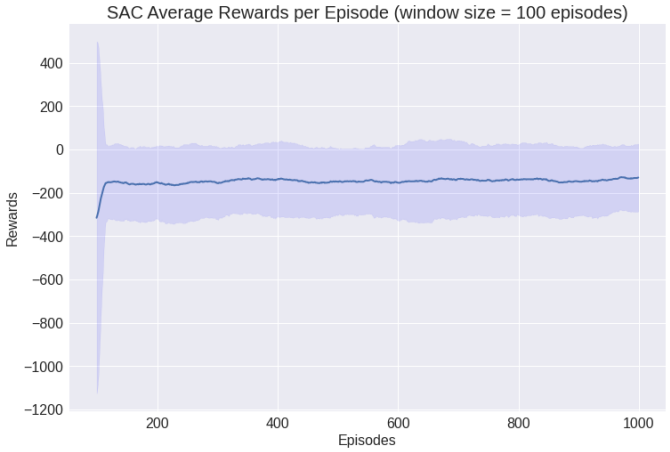

Fig. 6: SAC Learning

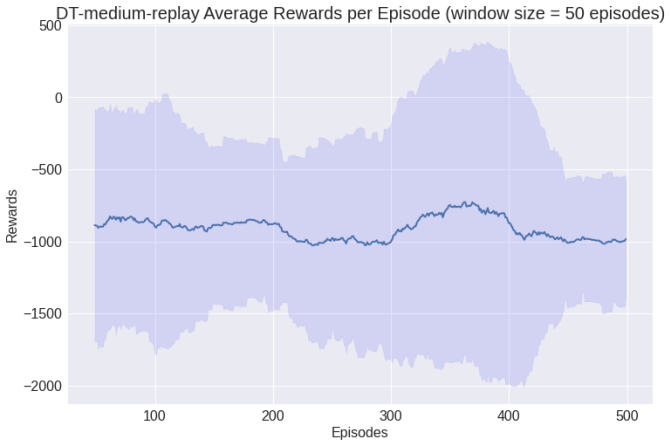

Fig. 7: Learning on pendulum-medium-replay

after each 10^5 iterations, and thus 1000 episodes in the figure. This also means that curve starts with a partiallay trained policy. The first rewards were in the [-1600,-1200] range.

D. Average Rewards Comparison

Table 2 displays the average rewards over 200 episodes for each model. We can see that the pendulum-random agent had a performance similar to that of a random agent. The pendulum-medium-replay did slightly better than random. And the pendulum-medium did significantly better than random, but could not match the performance of the SAC agent.

V. Summary and Conclusion

Offline RL attempts to help RL achieve the breakthrough in terms of deployment in real world environments, by leveraging large datasets and following the steps of supervised learning. Moreover, by redefining RL as a sequence modeling probelem, it becomes possible to make use of elaborate architecture such as transformers. The resuls shown in [1] and [2] are very promising. However, I have not been able to achieve similar performance on custom datasets I generated on the Pendulum environment. While one of the models was able to perform significantly better than random, it did not even reach the the performance of the policy that generated the dataset. Imitation learning, such as behavior cleaning, is one of the main challenges facing offline reinforcement learning algorithms. In order to prove that a trained model is not only imitating the dataset, it should be able to learn from a random dataset. The fact that Decision Transformer was not able to do that on the pendulum-random dataset might suggest that applying such a complex model to a simple problem is not appropriate (overkill?). Regardless of the results, I am glad I chose this project. I now fully understand how sequence-to-sequence modeling, attention, transformers and self-attention work. In addition, I have been fairly introduced to the realm of offline RL.

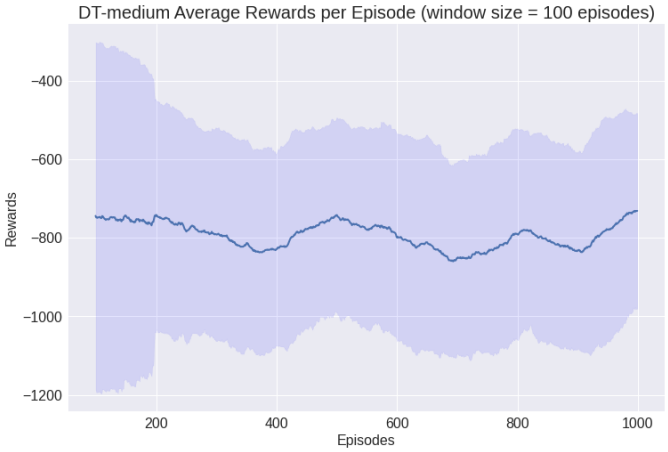

Fig. 8: Learning on pendulum-medium

TABLE II: Average Rewards

Model	Average Rewards (200 episodes)
random agent	-1228.31
SAC agent	**-138.44**
pendulum-random	-1235.27
pendulum-medium-replay	-1025.76
pendulum-medium	-607.84

References

[1] M. Janner, Q. Li, and S. Levine, "Reinforcement Learning as One Big Sequence Modeling Problem," 2021. [Online]. Available: http://arxiv.org/abs/2106.02039

[2] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, "Decision Transformer: Reinforcement Learning via Sequence Modeling," jun 2021. [Online]. Available: https://arxiv.org/abs/2106.01345v2

[3] J. Fu, A. Kumar, O. Nachum, G. Brain, G. T. Google Brain, and S. Levine, "D4RL: Datasets for Deep Data-Driven Reinforcement Learning," apr 2020. [Online]. Available: https://arxiv.org/abs/2004.07219v4

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[5] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor."

[6] Kushagra06, "kushagra06/SAC: Pytorch implementation of Soft Actor-Critic." [Online]. Available: https://github.com/kushagra06/SAC

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in Neural Information Processing Systems*, vol. 2017-Decem, pp. 5999–6009, jun 2017. [Online]. Available: https://arxiv.org/abs/1706.03762v5

[8] A. Graves, "Generating Sequences With Recurrent Neural Networks," aug 2013. [Online]. Available: https://arxiv.org/abs/1308.0850v5

[9] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor."

[10] V. V.Kumar, "Soft Actor-Critic Demystified. An intuitive explanation of the theory… — Towards Data Science." [Online]. Available: https://towardsdatascience.com/soft-actor-critic-demystified-b8427df61665

[11] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *https://doi.org/10.1214/aoms/1177729694*, vol. 22, no. 1, pp. 79–86, mar 1951. [Online]. Available: https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-22/issue-1/On-Information-and-Sufficiency/10.1214/aoms/1177729694.fullhttps://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-22/issue-1/On-Information-and-Su