← Back to Articles

Train your first Decision Transformer

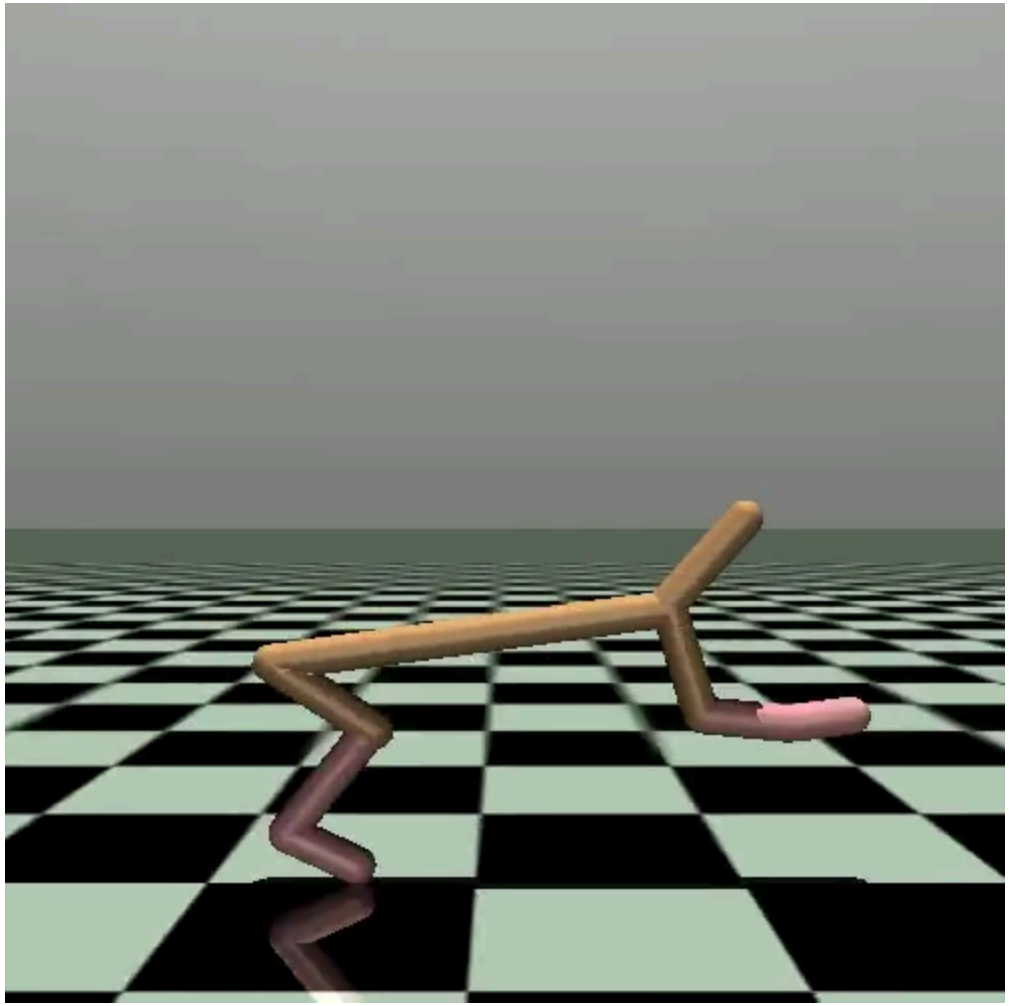
Published September 8, 2022

edbeeching
Edward Beeching

ThomasSimonini
Thomas Simonini

In a previous post, we announced the launch of Decision Transformers in the transformers library. This new technique of **using a Transformer as a Decision-making model** is getting increasingly popular.

So today, **you'll learn to train your first Offline Decision Transformer model from scratch to make a half-cheetah run.** We'll train it directly on a Google Colab that you can find here 👉 https://github.com/huggingface/blog/blob/main/notebooks/101_train-decision-transformers.ipynb



An "expert" Decision Transformers model, learned using offline RL in the Gym HalfCheetah environment.

Sounds exciting? Let's get started!

What are Decision Transformers?

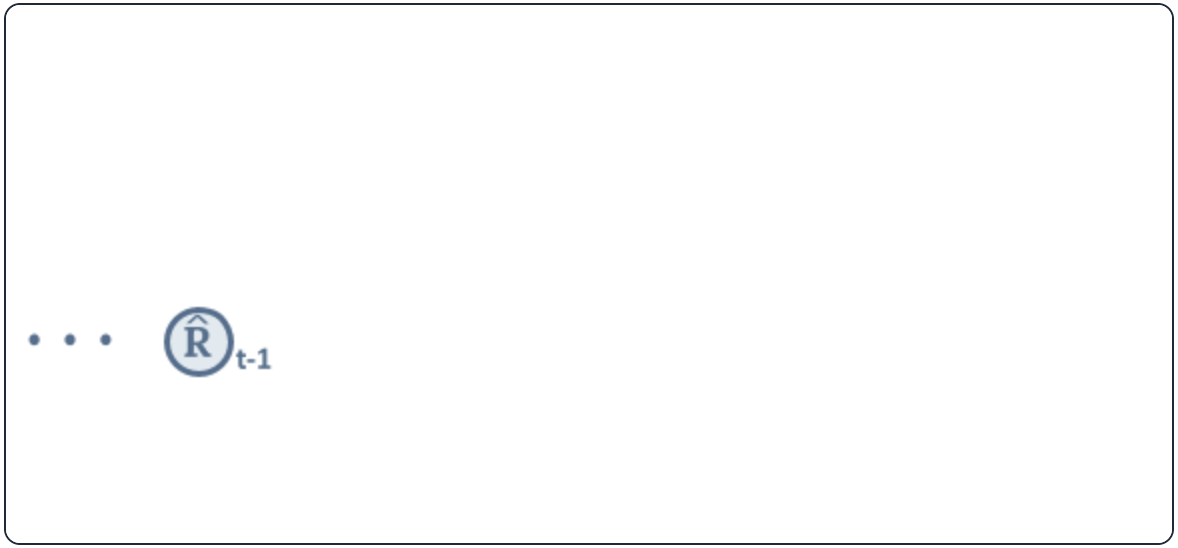
The Decision Transformer model was introduced by "Decision Transformer: Reinforcement Learning via Sequence Modeling" by Chen L. et al. It abstracts Reinforcement Learning as a **conditional-sequence modeling problem**.

The main idea is that instead of training a policy using RL methods, such as fitting a value function that will tell us what action to take to maximize the return (cumulative reward), **we use a sequence modeling algorithm (Transformer)** that, given the desired return, past states, and actions, will generate future actions to achieve this desired return. It's an autoregressive model conditioned on the desired return, past states, and actions to generate future actions that achieve the desired return.

This is a complete shift in the Reinforcement Learning paradigm since we use generative trajectory modeling (modeling the joint distribution of the sequence of states, actions, and rewards) to replace conventional RL algorithms. It means that in Decision Transformers, we don't maximize the return but rather generate a series of future actions that achieve the desired return.

The process goes this way:

1. We feed **the last K timesteps** into the Decision Transformer with three inputs:
    - Return-to-go
    - State
    - Action
2. **The tokens are embedded** either with a linear layer if the state is a vector or a CNN encoder if it's frames.
3. **The inputs are processed by a GPT-2 model**, which predicts future actions via autoregressive modeling.



Decision Transformer architecture. States, actions, and returns are fed into modality-specific linear embeddings, and a positional episodic timestep encoding is added. Tokens are fed into a GPT architecture which predicts actions autoregressively using a causal self-attention mask. Figure from [1].

There are different types of Decision Transformers, but today, we're going to train an offline Decision Transformer, meaning that we only use data collected from other agents or human demonstrations. **The agent does not interact with the environment**. If you want to know more about the difference between offline and online reinforcement learning, check this article.

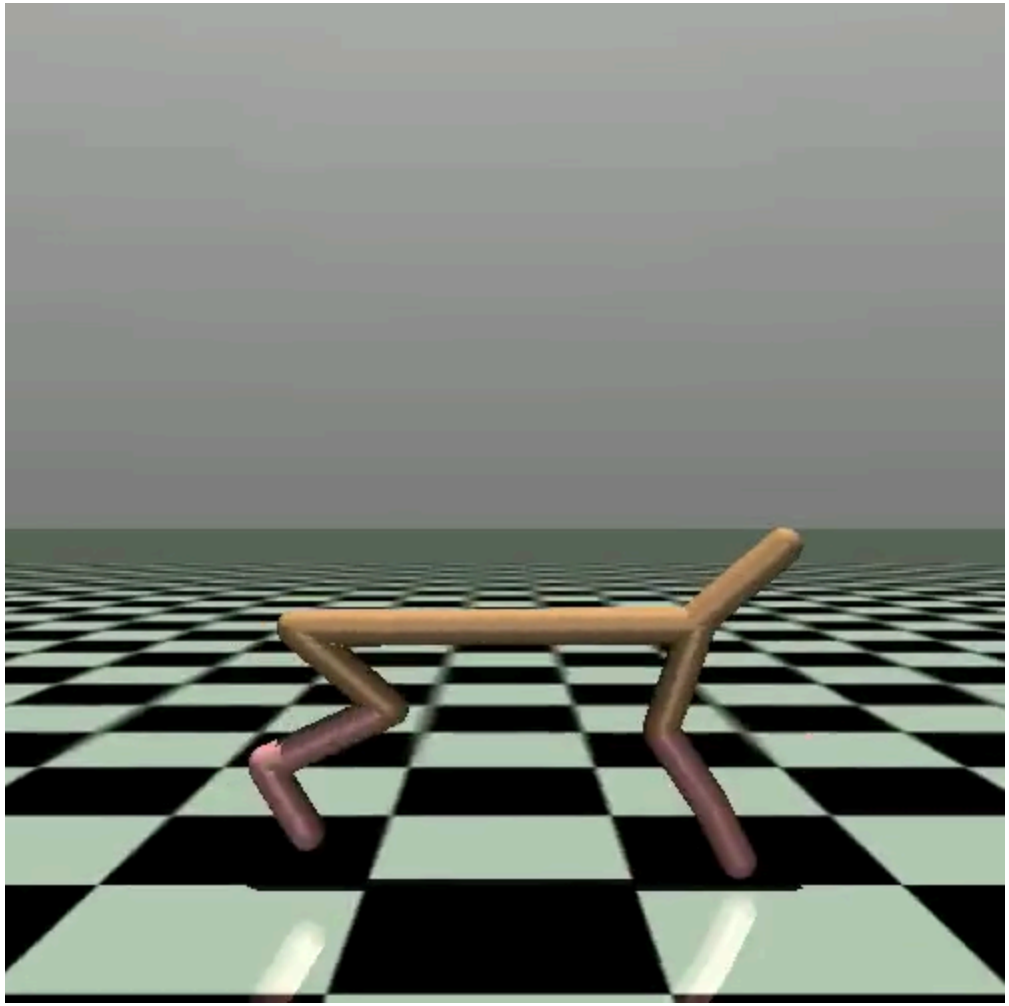
Now that we understand the theory behind Offline Decision Transformers, **let's see how we're going to train one in practice.**

Training Decision Transformers

In the previous post, we demonstrated how to use a transformers Decision Transformer model and load pretrained weights from the 🤗 hub.

In this part we will use 🤗 Trainer and a custom Data Collator to train a Decision Transformer model from scratch, using an Offline RL Dataset hosted on the 🤗 hub. You can find code for this tutorial in this Colab notebook.

We will be performing offline RL to learn the following behavior in the mujoco halfcheetah environment.



An "expert" Decision Transformers model, learned using offline RL in the Gym HalfCheetah environment.

Loading the dataset and building the Custom Data Collator

We host a number of Offline RL Datasets on the hub. Today we will be training with the halfcheetah “expert” dataset, hosted here on hub.

First we need to import the `load_dataset` function from the 🤗 datasets package and download the dataset to our machine.

```
from datasets import load_dataset
dataset = load_dataset("edbeeching/decision_transformer_gym_replay", "halfcheetah-expe
```

While most datasets on the hub are ready to use out of the box, sometimes we wish to perform some additional processing or modification of the dataset. In this case we wish to match the author's implementation, that is we need to:

- Normalize each feature by subtracting the mean and dividing by the standard deviation.
- Pre-compute discounted returns for each trajectory.
- Scale the rewards and returns by a factor of 1000.
- Augment the dataset sampling distribution so it takes into account the length of the expert agent's trajectories.

In order to perform this dataset preprocessing, we will use a custom 🤗 Data Collator.

Now let's get started on the Custom Data Collator for Offline Reinforcement Learning.

```
@dataclass
class DecisionTransformerGymDataCollator:
    return_tensors: str = "pt"
    max_len: int = 20 #subsets of the episode we use for training
```

```
    state_dim: int = 17  # size of state space
    act_dim: int = 6  # size of action space
    max_ep_len: int = 1000 # max episode length in the dataset
    scale: float = 1000.0  # normalization of rewards/returns
    state_mean: np.array = None  # to store state means
    state_std: np.array = None  # to store state stds
    p_sample: np.array = None  # a distribution to take account trajectory lengths
    n_traj: int = 0 # to store the number of trajectories in the dataset

    def __init__(self, dataset) -> None:
        self.act_dim = len(dataset[0]["actions"][0])
        self.state_dim = len(dataset[0]["observations"][0])
        self.dataset = dataset
        # calculate dataset stats for normalization of states
        states = []
        traj_lens = []
        for obs in dataset["observations"]:
            states.extend(obs)
            traj_lens.append(len(obs))
        self.n_traj = len(traj_lens)
        states = np.vstack(states)
        self.state_mean, self.state_std = np.mean(states, axis=0), np.std(states, axis

        traj_lens = np.array(traj_lens)
        self.p_sample = traj_lens / sum(traj_lens)

    def _discount_cumsum(self, x, gamma):
        discount_cumsum = np.zeros_like(x)
        discount_cumsum[-1] = x[-1]
```

```
        for t in reversed(range(x.shape[0] - 1)):
            discount_cumsum[t] = x[t] + gamma * discount_cumsum[t + 1]
        return discount_cumsum

    def __call__(self, features):
        batch_size = len(features)
        # this is a bit of a hack to be able to sample of a non-uniform distribution
        batch_inds = np.random.choice(
            np.arange(self.n_traj),
            size=batch_size,
            replace=True,
            p=self.p_sample,  # reweights so we sample according to timesteps
        )
        # a batch of dataset features
        s, a, r, d, rtg, timesteps, mask = [], [], [], [], [], [], []

        for ind in batch_inds:
            # for feature in features:
            feature = self.dataset[int(ind)]
            si = random.randint(0, len(feature["rewards"]) - 1)

            # get sequences from dataset
            s.append(np.array(feature["observations"][si : si + self.max_len]).reshape
            a.append(np.array(feature["actions"][si : si + self.max_len]).reshape(1, -
            r.append(np.array(feature["rewards"][si : si + self.max_len]).reshape(1, -

            d.append(np.array(feature["dones"][si : si + self.max_len]).reshape(1, -1)
            timesteps.append(np.arange(si, si + s[-1].shape[1]).reshape(1, -1))
            timesteps[-1][timesteps[-1] >= self.max_ep_len] = self.max_ep_len - 1  # p
```

```
            rtg.append(
                self._discount_cumsum(np.array(feature["rewards"][si:]), gamma=1.0)[
                    : s[-1].shape[1]   # TODO check the +1 removed here
                ].reshape(1, -1, 1)
            )
            if rtg[-1].shape[1] < s[-1].shape[1]:
                print("if true")
                rtg[-1] = np.concatenate([rtg[-1], np.zeros((1, 1, 1))], axis=1)

            # padding and state + reward normalization
            tlen = s[-1].shape[1]
            s[-1] = np.concatenate([np.zeros((1, self.max_len - tlen, self.state_dim))
            s[-1] = (s[-1] - self.state_mean) / self.state_std
            a[-1] = np.concatenate(
                [np.ones((1, self.max_len - tlen, self.act_dim)) * -10.0, a[-1]],
                axis=1,
            )
            r[-1] = np.concatenate([np.zeros((1, self.max_len - tlen, 1)), r[-1]], axi
            d[-1] = np.concatenate([np.ones((1, self.max_len - tlen)) * 2, d[-1]], axi
            rtg[-1] = np.concatenate([np.zeros((1, self.max_len - tlen, 1)), rtg[-1]],
            timesteps[-1] = np.concatenate([np.zeros((1, self.max_len - tlen)), timest
            mask.append(np.concatenate([np.zeros((1, self.max_len - tlen)), np.ones((1

        s = torch.from_numpy(np.concatenate(s, axis=0)).float()
        a = torch.from_numpy(np.concatenate(a, axis=0)).float()
        r = torch.from_numpy(np.concatenate(r, axis=0)).float()
        d = torch.from_numpy(np.concatenate(d, axis=0))
        rtg = torch.from_numpy(np.concatenate(rtg, axis=0)).float()
        timesteps = torch.from_numpy(np.concatenate(timesteps, axis=0)).long()
```

```
        mask = torch.from_numpy(np.concatenate(mask, axis=0)).float()

        return {
            "states": s,
            "actions": a,
            "rewards": r,
            "returns_to_go": rtg,
            "timesteps": timesteps,
            "attention_mask": mask,
        }
```

That was a lot of code, the TLDR is that we defined a class that takes our dataset, performs the required preprocessing and will return us batches of **states**, **actions**, **rewards**, **returns**, **timesteps** and **masks.** These batches can be directly used to train a Decision Transformer model with a 🤗 transformers Trainer.

Training the Decision Transformer model with a 🤗 transformers Trainer.

In order to train the model with the 🤗 Trainer class, we first need to ensure the dictionary it returns contains a loss, in this case L-2 norm of the models action predictions and the targets. We achieve this by making a TrainableDT class, which inherits from the Decision Transformer model.

```
class TrainableDT(DecisionTransformerModel):
    def __init__(self, config):
        super().__init__(config)
```

```
    def forward(self, **kwargs):
        output = super().forward(**kwargs)
        # add the DT loss
        action_preds = output[1]
        action_targets = kwargs["actions"]
        attention_mask = kwargs["attention_mask"]
        act_dim = action_preds.shape[2]
        action_preds = action_preds.reshape(-1, act_dim)[attention_mask.reshape(-1) >
        action_targets = action_targets.reshape(-1, act_dim)[attention_mask.reshape(-1

        loss = torch.mean((action_preds - action_targets) ** 2)

        return {"loss": loss}

    def original_forward(self, **kwargs):
        return super().forward(**kwargs)
```

The transformers Trainer class required a number of arguments, defined in the TrainingArguments class. We use the same hyperparameters are in the authors original implementation, but train for fewer iterations. This takes around 40 minutes to train in a Colab notebook, so grab a coffee or read the 🤗 Annotated Diffusion blog post while you wait. The authors train for around 3 hours, so the results we get here will not be quite as good as theirs.

```
training_args = TrainingArguments(
    output_dir="output/",
    remove_unused_columns=False,
```

```
    num_train_epochs=120,
    per_device_train_batch_size=64,
    learning_rate=1e-4,
    weight_decay=1e-4,
    warmup_ratio=0.1,
    optim="adamw_torch",
    max_grad_norm=0.25,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset["train"],
    data_collator=collator,
)

trainer.train()
```

Now that we explained the theory behind Decision Transformer, the Trainer, and how to train it. **You're ready to train your first offline Decision Transformer model from scratch to make a half-cheetah run** 👉 https://github.com/huggingface/blog/blob/main/notebooks/101_train-decision-transformers.ipynb The Colab includes visualizations of the trained model, as well as how to save your model on the 🤗 hub.

Conclusion

This post has demonstrated how to train the Decision Transformer on an offline RL dataset, hosted on 🤗 datasets. We have used a 🤗 transformers Trainer and a custom data collator.

In addition to Decision Transformers, **we want to support more use cases and tools from the Deep Reinforcement Learning community**. Therefore, it would be great to hear your feedback on the Decision Transformer model, and more generally anything we can build with you that would be useful for RL. Feel free to reach out to us.

What's next?

In the coming weeks and months, **we plan on supporting other tools from the ecosystem**:

- Expanding our repository of Decision Transformer models with models trained or finetuned in an online setting [2]
- Integrating sample-factory version 2.0

The best way to keep in touch is to join our discord server to exchange with us and with the community.

References

[1] Chen, Lili, et al. "Decision transformer: Reinforcement learning via sequence modeling." *Advances in neural information processing systems* 34 (2021).

[2] Zheng, Qinqing and Zhang, Amy and Grover, Aditya “*Online Decision Transformer*” (arXiv preprint, 2022)

More Articles from our Blog

Putting RL back in RLHF

By vwxyzjn June 12, 2024 • △ 57

Jack of All Trades, Master of Some, a Multi-Purpose Transformer Agent

By qgallouedec April 22, 2024 • △ 78