Decision Transformers

John Tan Chong Min

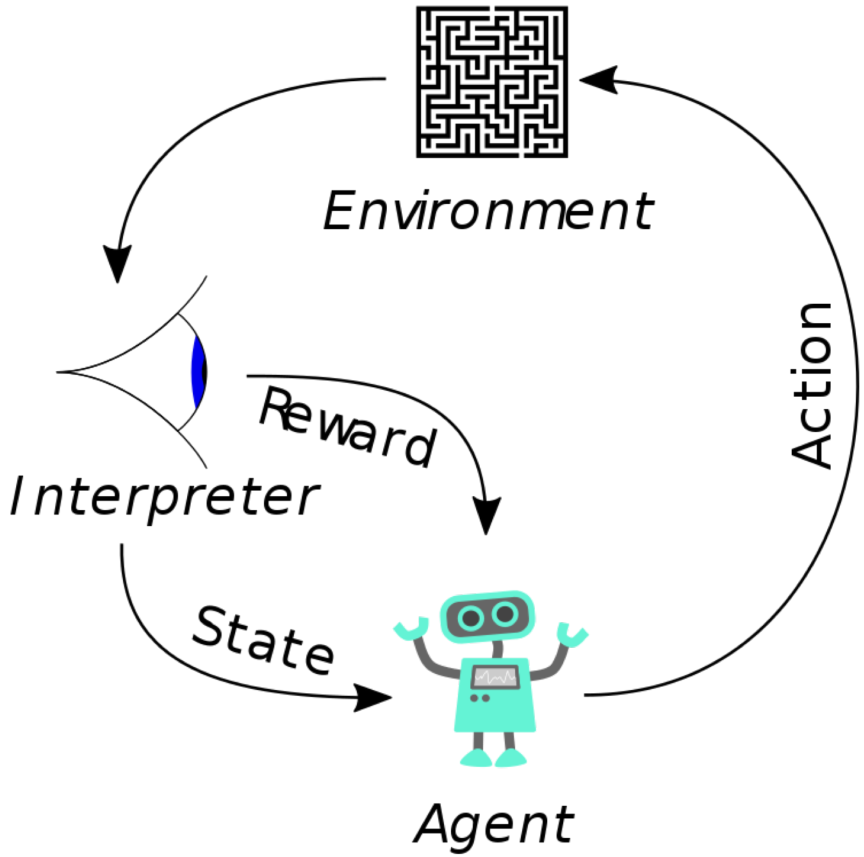
Background

- Reinforcement Learning (RL) uses Markov Decision Processes (MDP) to model environment
- Typically one-step (Bellman) updates are done
- What if we could just use Transformers instead to predict future action?

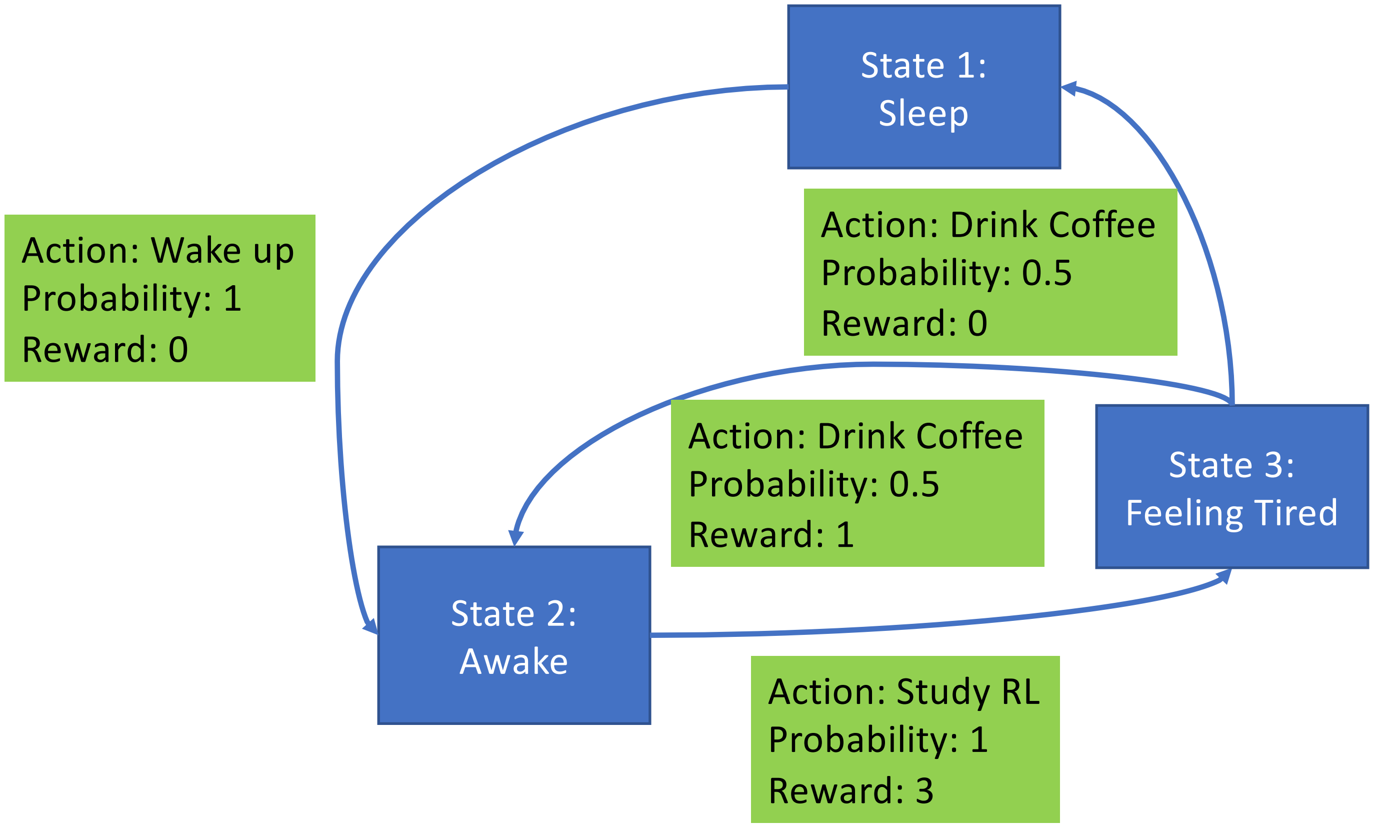
Reinforcement Learning (RL)

- For a Reinforcement Learning (RL) agent, we have
  - Environment $\mathcal{E}$
  - State s $\in S$
  - Action a $\in A$
  - Reward r $\in R$
  - Transition Function $P(\cdot\,|s,a)$
  - Discount factor γ

- Interaction with environment can be modelled as Markov Decision Process (MDP)



Markov Decision Process (MDP)



MDP states are memory-free.

Knowing the state itself is sufficient to do planning, no need for past state histories.

Question: What if your game/environment requires a history of past states?

Bellman Updates

- Based off Temporal Difference (TD) Learning

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \Big| s, a \right]$$

What if this one-step update is inefficient?

What if we can credit assign over sequences instead?

Recap: Transformers

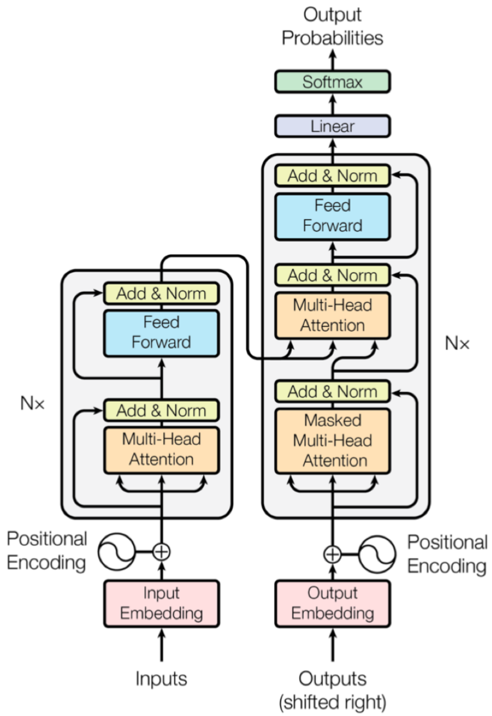


Figure 1: The Transformer - model architecture.

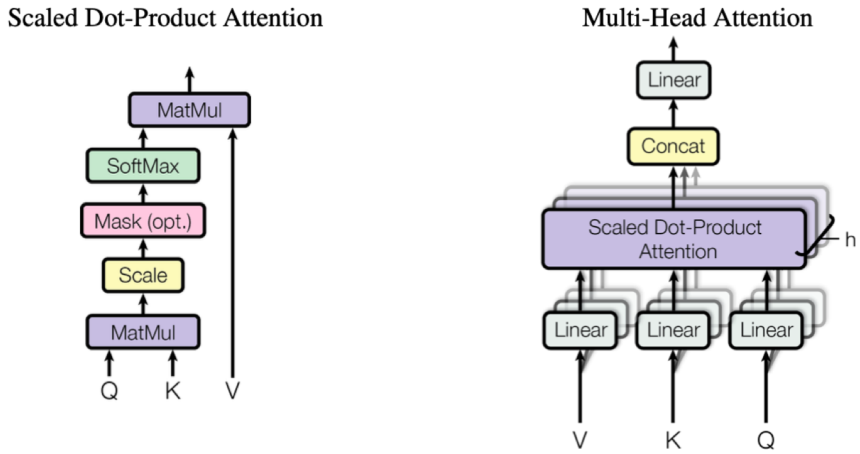
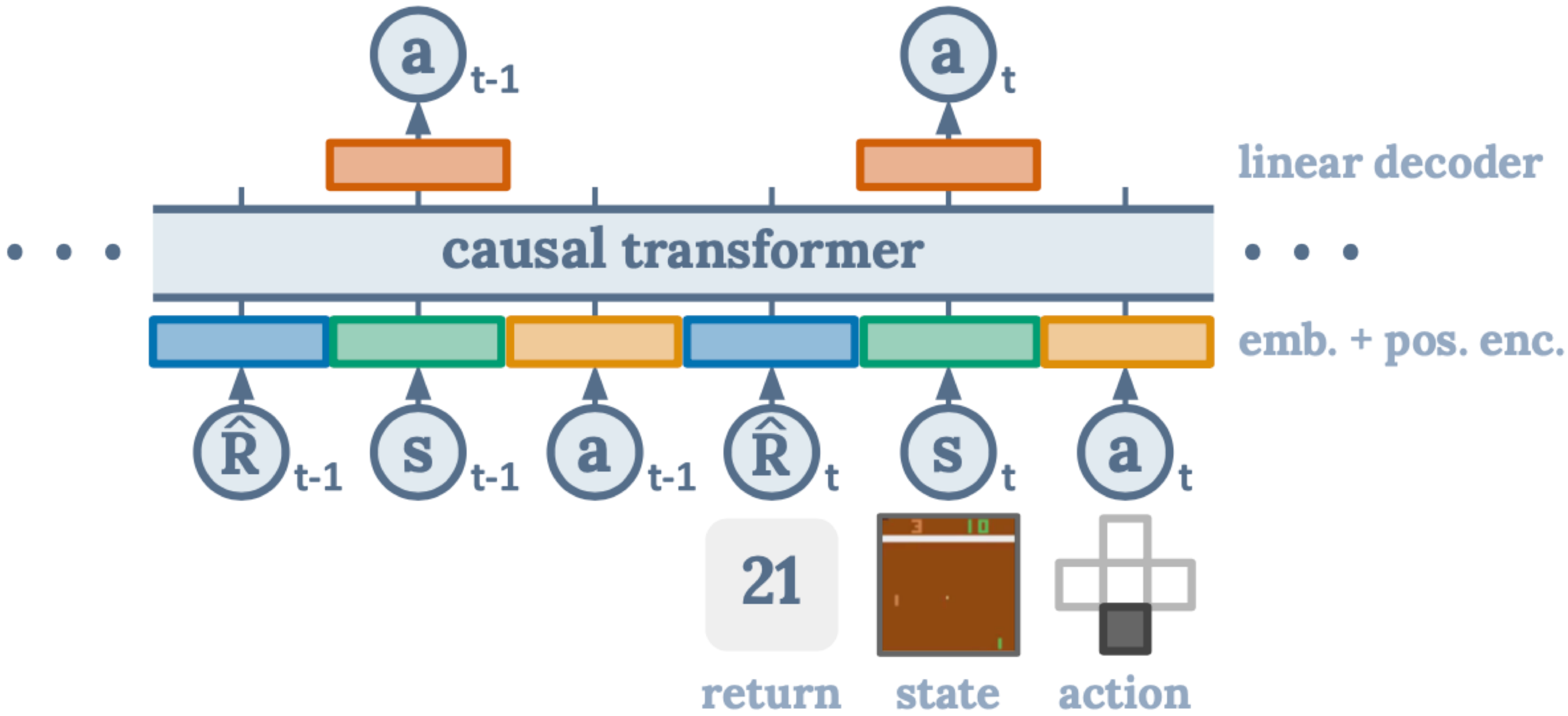


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

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

Taken from: Attention is all you need. Vaswani et al. (2017)

Decision Transformer Architecture



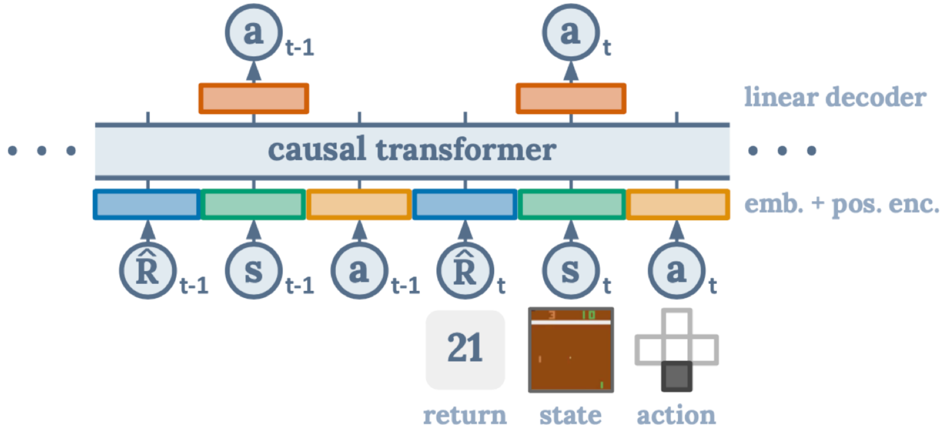
- Trains offline using expert trajectories
- Predicts the next best action given the sequence of *r*, *s*, *a* up till current timestep
- $R_t = \sum_{t'=t}^{T} r_{t'}$ is the rewards-to-go, which is the future cumulative sum of rewards
- Outputs a probability vector of actions

Decision Transformer Pseudocode

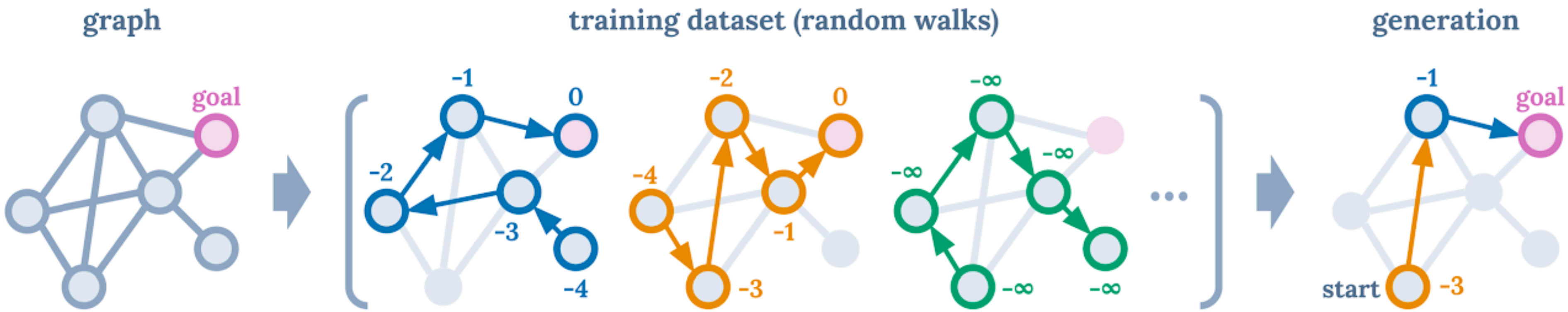
Algorithm 1 Decision Transformer Pseudocode (for continuous actions)

```
# 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
```



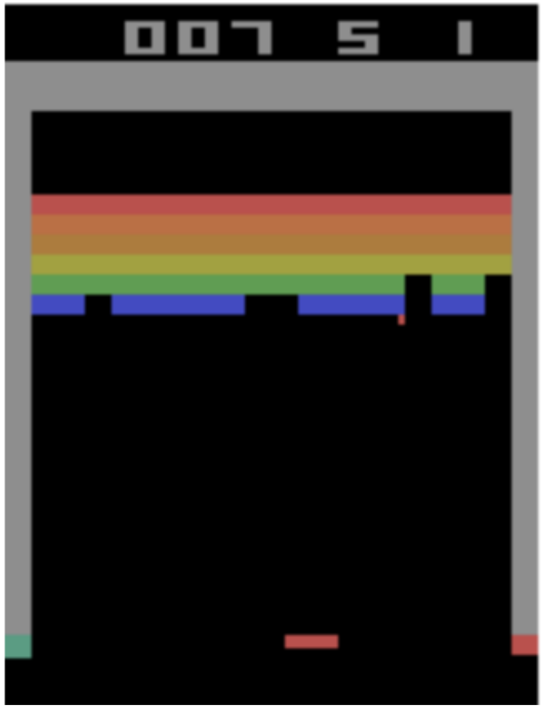
Intution: Mix and match best subsequences



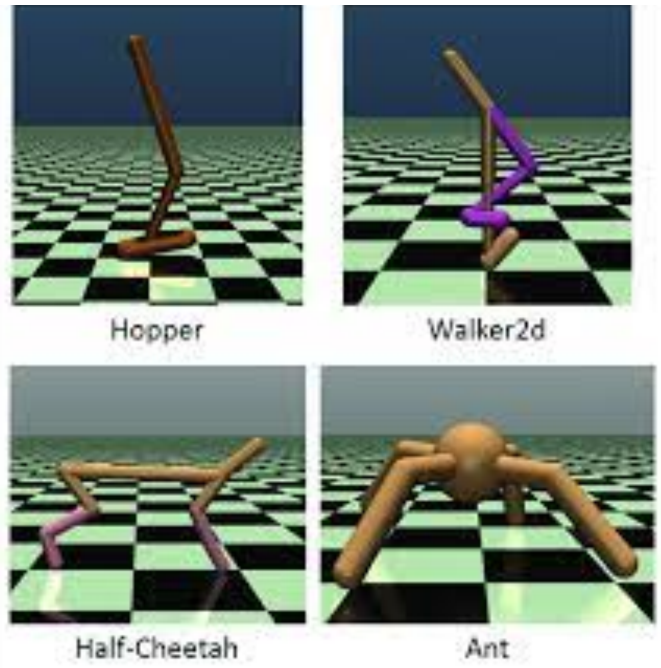
- From training dataset, attend to the best parts of each dataset for the task at hand to generate optimal sequence

Game Environments Used

Atari



MuJoCo



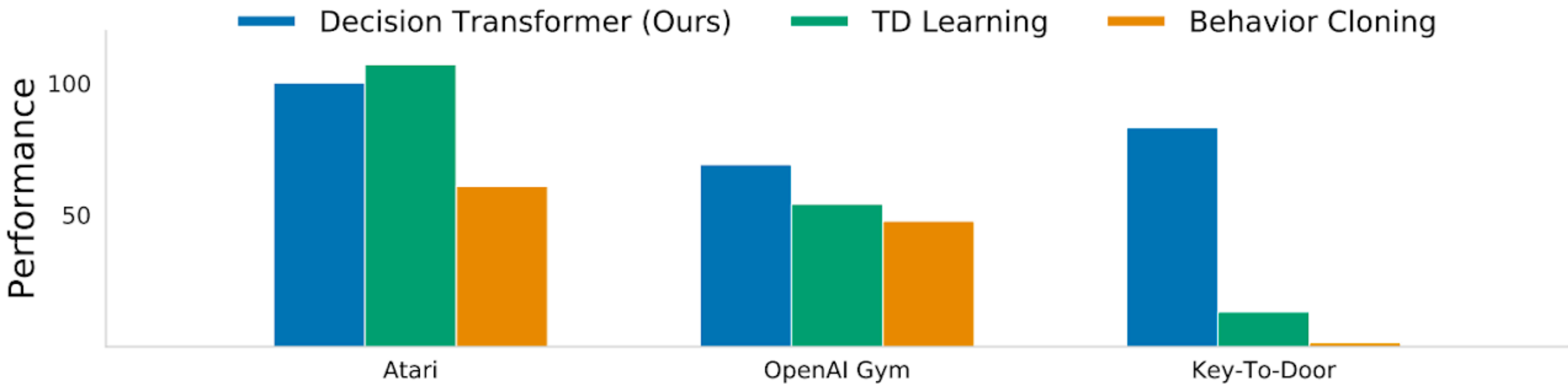
Key-to-Door

3 rooms.
1st room has key,
2nd room distractor apples,
3rd room door

Aim: Collect key
and unlock door

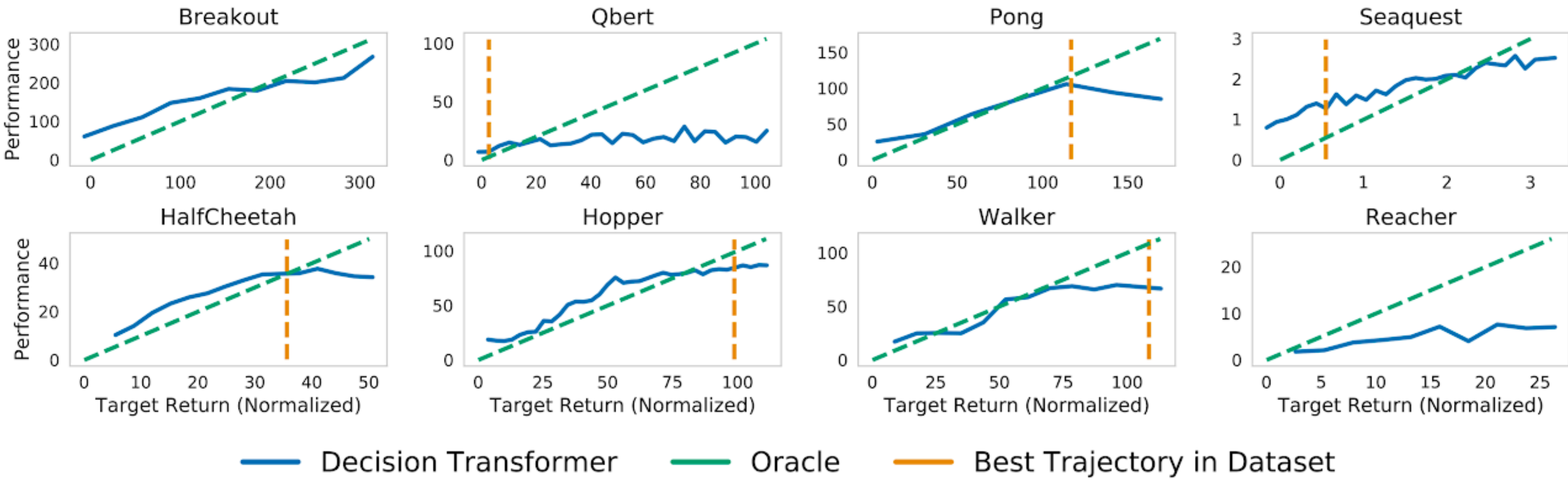
Partial observability

Performance



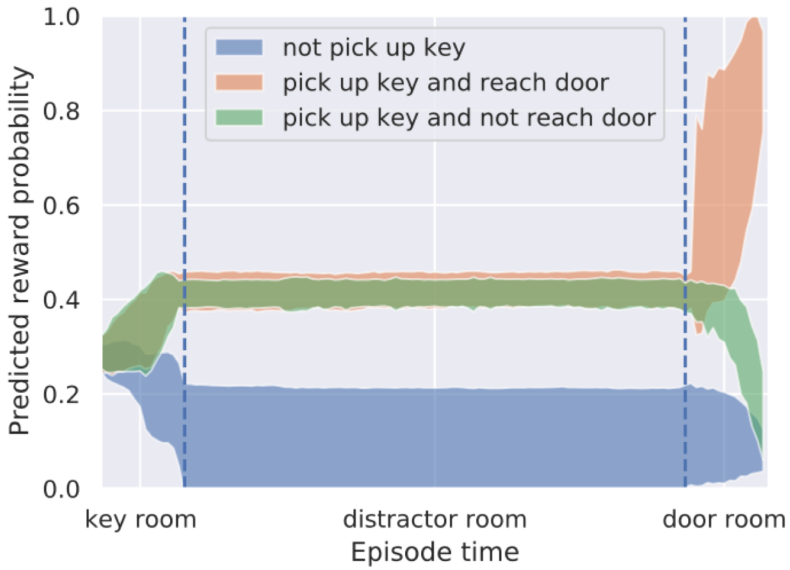
- Decision Transformer is better than TD Learning and Behavior Cloning in most cases
- TD Learning:
  - Conservative Q-Learning – regularized Q-functions whose expectation is lower bound of true value
- Behavior Cloning: Learn from top x% of experiences only

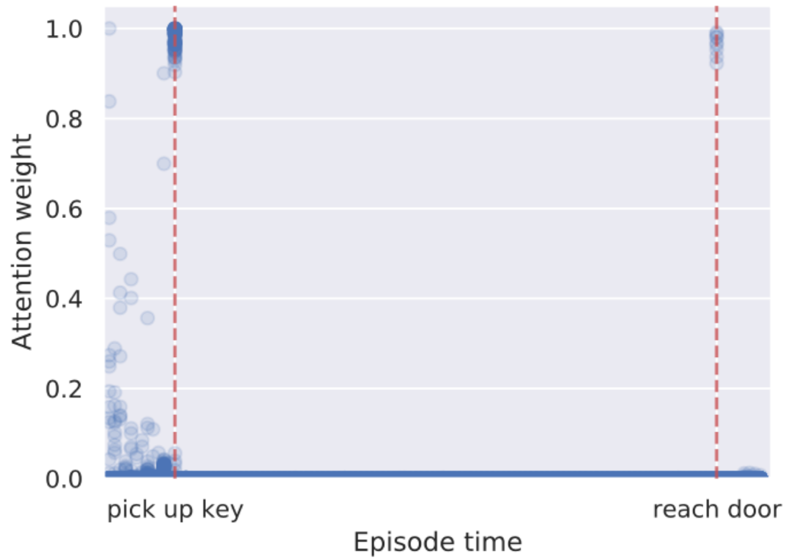
Results on Atari/MuJoCo Benchmarks



- Can only do as well as the expert Trajectory in dataset (except Sequest)

Key-to-Door Task

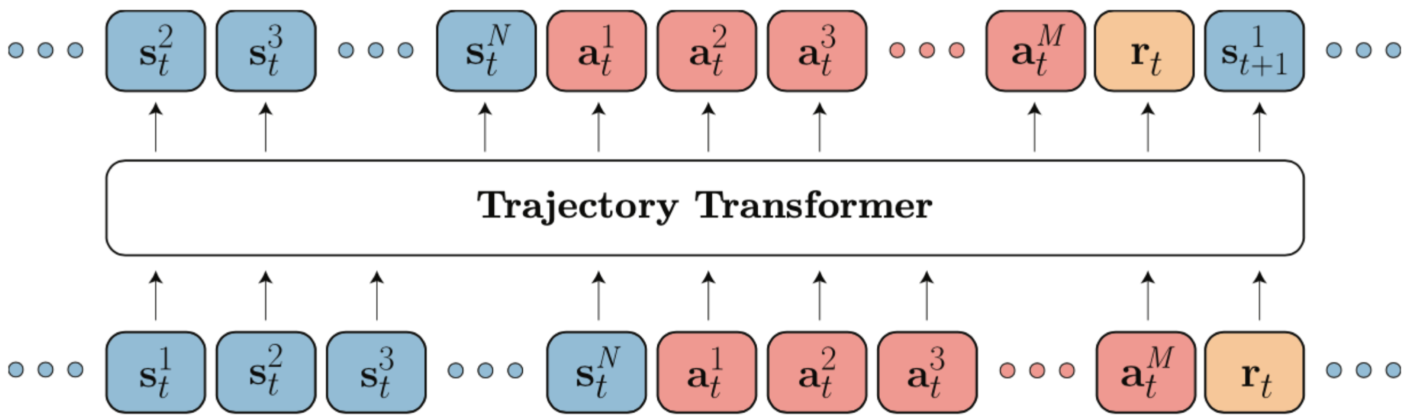




- Predicts an expected reward
- Attends to the important scenarios (pick up key, reach door), and not attend to distractor apples

Similar work: Trajectory Transformer

- Decision Transformer: Model-free prediction, only predicts actions
- Trajectory Transformer: Model-based prediction using state and reward prediction as well



$$\mathcal{L}(\tau) = \sum_{t=1}^{T}\Big(\sum_{i=1}^{N}\log P_\theta\big(s_t^i \mid \mathbf{s}_t^{<i}, \boldsymbol{\tau}_{<t}\big) + \sum_{j=1}^{M}\log P_\theta\big(a_t^j \mid \mathbf{a}_t^{<j}, \mathbf{s}_t, \boldsymbol{\tau}_{<t}\big) + \log P_\theta\big(r_t \mid \mathbf{a}_t, \mathbf{s}_t, \boldsymbol{\tau}_{<t}\big)\Big)$$

Questions to Ponder

- How to better calculate Rewards-to-go?
- Can we have a better way of framing this sequence problem?
- How to minimize number of offline experiences needed to learn?
- How to learn without needing expert trajectories?

Discussion