Bolts and Nuts of Multi-Object Manipulation using Relational Reinforcement Learning

Achraf Azize

achraf.azize@polytechnique.edu

Ecole Polytechnique - InstaDeep

September 9, 2021

Abstract

Reinforcement Learning (RL) has been able to solve hard problems such as playing Atari games or solving the game of Go, with a unified approach. Yet scaling reinforcement learning algorithms to more complex robotic manipulation settings with sparse rewards, is still impractical, requiring a huge amount of data or human demonstrations. This paper is a step-by-step tutorial to train an agent to stack blocks into towers, generalizing over an unseen number of blocks and goal configurations, by utilizing the inductive biases of graph-based relational architectures. We propose two different methods to achieve this goal: curriculum learning and diversified batching. Starting from scratch, the learned policy stacks up to 4 blocks and exhibits zero-shot generalization by successfully stacking blocks into a previously unseen number of blocks and configurations (such as pyramids), without any further training.

Contents

1 Introduction

Reinforcement learning (RL) is a fundamental sequential decision-making problem for learning in uncertain environments. In each round, the learner observes the state of the environment and then selects an action. This action will provide them with a reward and take them to a new state via some unknown dynamics. The learner receives feedback in the form of trajectories (i.e., sequences of states, actions and rewards) which encode information about the unknown dynamics and objectives, and allow the agent to learn about the environment. The agent's aim is to select actions to maximize their total reward. To succeed in this problem, an agent needs to trade-off exploration to gather information about the environment (reward and dynamics) and exploitation of available information to maximize the cumulative reward.

Recently, a wide range of successes in learning policies for sequential decision-making problems has been achieved using reinforcement learning combined with neural networks. This includes simulated environments, such as playing Atari games (Mnih et al. (2013)), and robotic tasks, such as helicopter control (Ng et al. (2004)), hitting a baseball (Peters & Schaal (2008)), screwing a cap onto a bottle (Levine et al. (2015)), and door opening (Chebotar et al. (2017)).

However, designing a reward function that not only reflects the task at hand but is also carefully shaped to guide policy optimization is a common challenge (Ng et al. (1999)), especially in robotics. Popov et al. (2017), for example, use a cost function made up of five relatively complex terms that must be carefully weighted in order to train a policy for stacking bricks on top of each other. Because it necessitates both RL expertise and domain-specific knowledge, cost engineering limits the applicability of RL in the real world. Furthermore, it is inapplicable in situations where we are unsure of what constitutes acceptable behaviour.

Another option is to give the agent sparse rewards, which could be given after the agent completes the overall task (i.e., terminal reward) or when the agent completes critical steps (i.e., step-wise rewards). Sparse rewards are easier to define, but on the other hand, make exploration and credit assignment much more difficult, because many tasks require the execution of a long sequence of actions. As a result, sparse reward training either fails completely or necessitates massive amounts of data.

Practical reinforcement learning systems have avoided the challenge of learning from sparse rewards by using human demonstrations or meticulous reward shaping. The development of data-efficient algorithms that can learn from sparse rewards will eliminate the need for painful reward design and demonstrations. Better optimization methods Hessel et al. (2018); Haarnoja et al. (2018b); Schulman et al. (2017), combining model-based and model-free learning Levine et al. (2016), hierarchical learning Sutton et al. (1999), and designing better exploration methods Schmidhuber (2010); Lopes et al. (2012); Oudeyer & Kaplan (2009); Pathak et al. (2017); Sukhbaatar et al. (2018) have all been used in the past to improve the learning efficiency of RL algorithms. The compositional task structure has been used in a few recent works to make an impact.

Transfer of knowledge by pre-training on a source task followed by finetuning on a target task has been very successful in reducing data requirements in supervised deep learning (Donahue et al. (2014); Agrawal et al. (2016); Ren et al. (2015)). However, learning from multiple tasks and transferring that knowledge to reduce data requirements for a new task remains an open challenge in the context of RL(Zhang et al. (2018); Cobbe et al. (2018); Justesen et al. (2018); Nichol et al. (2018)).

One solution is to pace the agent's learning, giving it a new task only after it has mastered the previous ones (i.e., curriculum learning Bengio et al. (2009b)). It turns out that curriculum learning is also difficult in the RL setting: consider solving the problem of stacking multiple blocks into a tower using a curriculum of stacking an increasing number of blocks. Let's suppose the agent has perfected stacking two blocks. The third block now introduced maintains the task structure while changing the distribution of the agent's input. Due to a lack of appropriate inductive biases to deal with changes in inputs, the agent treats the new data distribution as a new learning problem and is unable to effectively leverage its knowledge from previous tasks.

In this work on multi-object manipulation, we successfully trained a policy represented by an attention based graph neural network to overcome the challenges associated with curriculum learning. Our agent has been able to learn how to stack four or more blocks from scratch (see Figure 11). The curriculum is supplemented by the attention-based GNN, which provides the appropriate inductive bias for transferring knowledge between tasks with different numbers of objects. In addition, our agent is capable of placing blocks in different configurations, such as pyramids without any additional training (i.e., zero-shot generalization). The approach used makes no task-specific assumptions and is thus applicable to a wide range of tasks involving multiple object manipulation.

To overcome the manually crafted curriculum used in the method above, we propose a diversified batching strategy, in which the agent is presented with a batch of inputs from various tasks. The idea is to combine a number of small graphs that represent the inputs into a single large graph. This idea is motivated by the fact that optimizing for these new diversified batches would be close enough to the real input distribution. This method achieves good results up to stacking two blocks, but is less sample efficient and more computationally demanding.

2 Preliminaries

In this section, we recall the basics of reinforcement learning, state-of-the-art policy gradient methods and algorithms, goal-conditioned RL and curriculum learning.

2.1 Reinforcement Learning

We consider the standard reinforcement learning formalism consisting of an agent interacting with an environment. To simplify the exposition we assume that the environment is fully observable. An environment is described by a set of states $\mathscr{S}$, a set of actions $\mathscr{A}$, a distribution of initial states $p(s_0)$, a reward function $r : \mathscr{S} \times \mathscr{A} \to \mathbb{R}$, transition probabilities $p(s_{t+1}|s_t, a_t)$ and a discount factor $\gamma \in [0,1]$.

A deterministic policy is a mapping from states to actions: $\pi : \mathscr{S} \to \mathscr{A}$.A. Every episode starts with sampling an initial state s_0. At every timestep t, the agent produces an action based on the current state: $a_t = \pi(s_t)$. Then it gets the reward $r_t = r(s_t, a_t)$ and the environment's new state is sampled from the distribution $p(.|s_t, a_t)$. A discounted sum of future rewards is called a return: $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$. The agent's goal is to maximize its expected return $\mathbb{E}_{s_0}[R_0|s_0]$. The Q-function or action-value function is defined as $Q^\pi(s_t, a_t) = \mathbb{E}[R_t|s_t, a_t]$.

Let $\pi^\star$ denote an optimal policy i.e. any policy $\pi^\star$ s.t. $Q^{\pi^\star}(s,a) \geq Q^\pi(s,a)$ for every $s \in \mathscr{S}$, $a \in \mathscr{A}$ and any policy π. All optimal policies have the same Q-function which is called optimal Q-function and denoted $Q^\star$. It is easy to show that it satisfies the following equation called the Bellman equation:

$$Q^\star(s,a) = \mathbb{E}_{s' \sim p(\mathring{u}|s,a)} \left[r(s,a) + \gamma \max_{a' \in \mathscr{A}} Q^\star(s', a') \right] \tag{1}$$

2.2 Policy gradient methods

Policy gradient is an approach to solve the reinforcement learning problem. The goal of reinforcement learning is to find an optimal behavior strategy for the agent to obtain optimal rewards. The policy gradient methods target at modeling and optimizing the policy directly. The policy is usually modeled with a parameterized function with respect to θ, $\pi_\theta(a|s)$. The value of the reward (objective) function depends on this policy and then various algorithms can be applied to optimize θ for the best reward.
The objective function is defined as:

$$J(\theta) = \sum_{s \in \mathscr{S}} d^\pi(s) V^\pi(s) = \sum_{s \in \mathscr{S}} d^\pi(s) \sum_{a \in \mathscr{A}} \pi_\theta(a|s) Q^\pi(s,a) \tag{2}$$

where $d^\pi(s)$ is the stationary distribution of Markov chain for π_θ (on-policy state distribution under π).

It is natural to expect policy-based methods to be more useful in the continuous space, because there is an infinite number of actions and (or) states to estimate the values for and hence value-based approaches are way too expensive computationally in the continuous space. For example, in generalized policy iteration, the policy improvement step $\arg\max_{a \in \mathscr{A}} Q^\pi(s,a)$ requires a full scan of the action space, suffering from the curse of dimensionality.

Using gradient ascent, we can move θ toward the direction suggested by the gradient $\nabla_\theta J(\theta)$ to find the best θ for π_θ that produces the highest return.

Luckily, the policy gradient theorem provides a nice reformation of the derivative of the objective function, to not involve the derivative of the state distribution $d^\pi(.)$ and simplify the gradient computation a lot:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s) \tag{3}$$

$$\propto \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta\pi_\theta(a|s) \tag{4}$$

2.2.1 SAC

Soft Actor-Critic (SAC) (Haarnoja et al. (2018a)) includes the policy's entropy measure into the reward to encourage exploration: we expect to learn a policy that acts as randomly as possible while still being able to succeed at the task. It is an off-policy actor-critic model following the maximum entropy reinforcement learning framework.

SAC has three main components:

- Separate policy and value function networks in an actor-critic architecture;
- Entropy maximization for stability and exploration;
- An off-policy formulation that allows reuse of previously gathered data for efficiency.

The policy is trained with the goal of maximizing the expected return and the entropy at the same time:

$$J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_{\pi_\theta}}[r(s_t,a_t) + \alpha\mathcal{H}(\pi_\theta(.|s_t))] \tag{5}$$

where $\mathcal{H}(.)$ is the entropy measure: $\mathcal{H}(P) = \underset{x\sim P}{\mathbb{E}}[-\log P(x)]$ and α controls how important the entropy term is, known as temperature parameter. The entropy maximization leads to policies that can explore more and capture multiple modes of near-optimal strategies (i.e., if there exist multiple options that seem to be equally good, the policy should assign each with an equal probability to be chosen).

Soft Q-value and soft state value functions are changed to include the entropy bonuses, and the Bellman equations become:

$$Q(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim\rho_\pi(s)}[V(s_{t+1})] \tag{6}$$

$$\text{where } V(s_t) = \mathbb{E}_{a_t\sim\pi}[Q(s_t,a_t) - \alpha\log\pi(a_t|s_t)] \tag{7}$$

The soft state value function is trained to minimize the mean squared error:

$$J_V(\psi) = \mathbb{E}_{s_t\sim\mathcal{D}}[\frac{1}{2}\big(V_\psi(s_t) - \mathbb{E}[Q_w(s_t,a_t) - \log\pi_\theta(a_t|s_t)]\big)^2] \tag{8}$$

where $\mathscr{D}$ is the replay buffer.

The soft Q function is trained to minimize the soft Bellman residual:

$$J_Q(w) = \mathbb{E}_{(s_t,a_t)\sim\mathscr{D}}[\frac{1}{2}\big(Q_w(s_t,a_t) - (r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim\rho_\pi(s)}[V_{\bar{\psi}}(s_{t+1})])\big)^2] \tag{9}$$

where $\bar{\psi}$ is the target value function which is the exponential moving average, just like how the parameter of the target Q network is treated in DQN to stabilize the training.

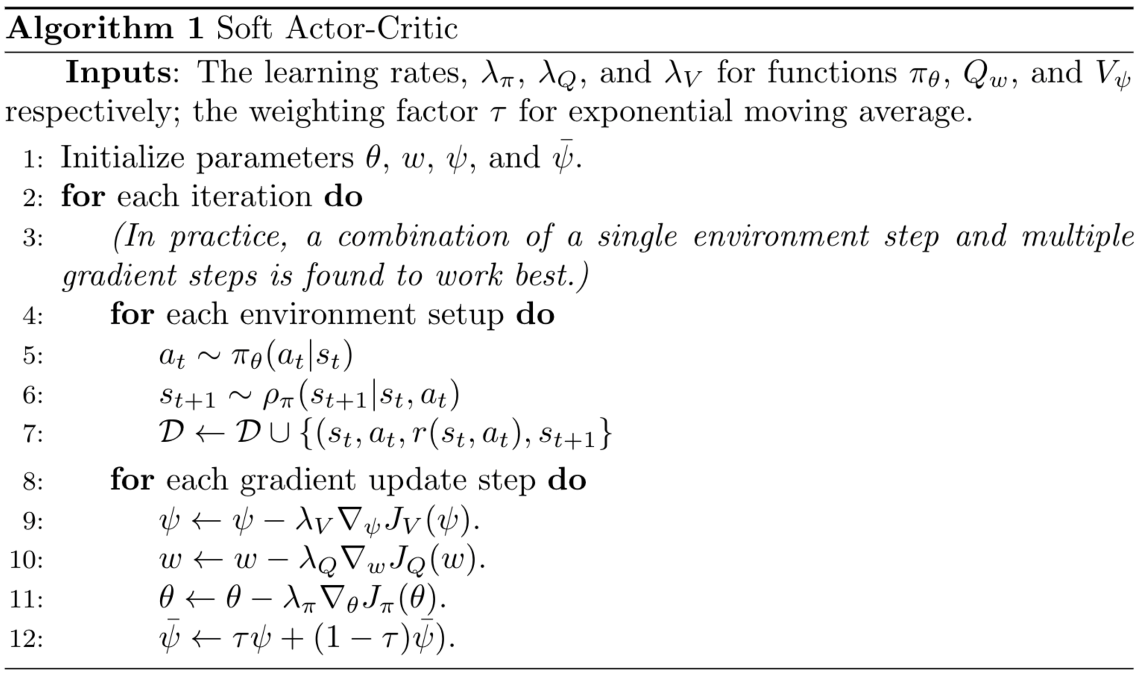
SAC updates the policy to minimize the KL-divergence:

$$\pi_{\text{new}} = \arg\min_{\pi'\in\Pi} D_{\text{KL}}\Big(\pi'(.|s_t)\,\|\,\frac{\exp(Q^{\pi_{\text{old}}}(s_t,.))}{Z^{\pi_{\text{old}}}(s_t)}\Big) \tag{10}$$

Unfortunately it is difficult to adjust the temperature parameter α, because the entropy can vary unpredictably both across tasks and during training as the policy becomes better. An improvement on SAC adjusts the temperature automatically by minimizing the loss function:

$$J(\alpha) = \mathbb{E}_{a_t\sim\pi_t}[-\alpha\log\pi_t(a_t \mid s_t) - \alpha\mathscr{H}_0] \tag{11}$$

The soft actor-critic algorithm is straightforward:



Algorithm 1 Soft Actor-Critic

Inputs: The learning rates, λ_π, λ_Q, and λ_V for functions π_θ, Q_w, and V_ψ respectively; the weighting factor τ for exponential moving average.

```
1:  Initialize parameters $\theta$, $w$, $\psi$, and $\bar{\psi}$.
2:  for each iteration do
3:      (In practice, a combination of a single environment step and multiple
        gradient steps is found to work best.)
4:      for each environment setup do
5:          $a_t \sim \pi_\theta(a_t|s_t)$
6:          $s_{t+1} \sim \rho_\pi(s_{t+1}|s_t, a_t)$
7:          $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1}\}$
8:      for each gradient update step do
9:          $\psi \leftarrow \psi - \lambda_V \nabla_\psi J_V(\psi)$.
10:         $w \leftarrow w - \lambda_Q \nabla_w J_Q(w)$.
11:         $\theta \leftarrow \theta - \lambda_\pi \nabla_\theta J_\pi(\theta)$.
12:         $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$.
```

Figure 1: The soft actor-critic algorithm. (Image source: original paper Haarnoja et al. (2018a))

2.3 Goal-conditioned RL

We are interested in training agents which learn to achieve multiple different goals. For solving multiple tasks, it is necessary to modify the learning problem by providing a task description as input. Goal conditioned policies are expressed as $a_t = \pi(s_t, s_g)$, where s_g represents the goal state. The learning problem is expressed as:

$$\max_{\pi} \mathbb{E}_{s_g \sim \rho(s_g), a \sim \pi, s \sim \mathcal{T}} [\sum_{i=t}^{T} \gamma^{(t-i)} r(s_t, a_t, s_g)] \tag{12}$$

where the goal s_g is sampled from a goal distribution $\rho(s_g)$.

2.4 HER: Hindsight Experience Replay

The concept behind Hindsight Experience Replay (HER) (Andrychowicz et al. (2017)) is simple: after experiencing an episode, we save every transition in the replay buffer, not only with the episode's original goal but also with a subset of other goals. Because the agent's actions are influenced by the goal pursued but not by the dynamics of the environment, we can replay each trajectory with any goal if we use an off-policy RL algorithm like SAC.

The set of additional goals used for replay is one decision that must be made in order to use HER. In the most basic version of HER, each trajectory is replayed with the goal that is met in the episode's final state.

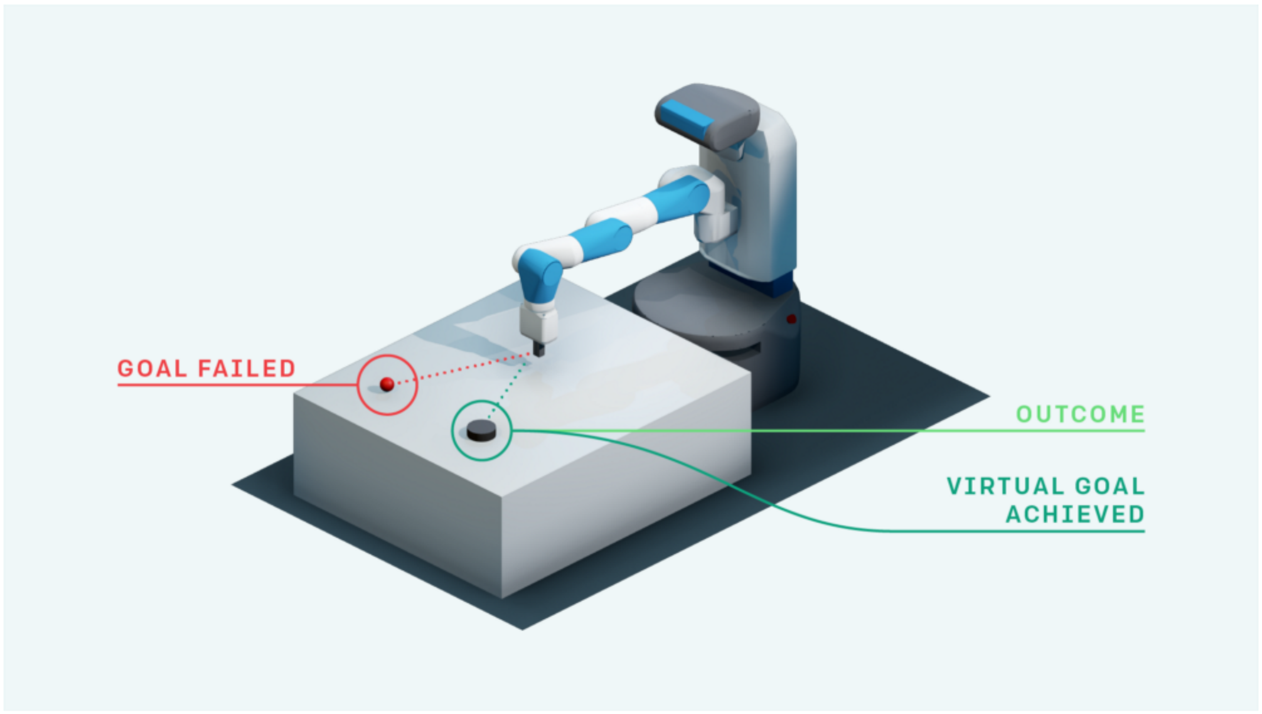


Figure 2: OpenAI Gym - HER: Robot arm must slide the puck to goal position. Image Source: OpenAI Website

The goals used for replay naturally shift from ones that are simple to achieve even by a random agent, to more difficult ones, which can be seen as a form of implicit curriculum. On the other hand, HER does not necessitate any control over the distribution of initial environment states, unlike explicit curriculum.

2.5 Curriculum Learning

Curriculum learning addresses the impact of data sampling strategies on learning, assuming that proper task sampling will result in more sample efficient learning and avoidance of local minima Elman (1993).
Prior research has shown that ordering tasks based on heuristic difficulty measures can be effective (Bengio et al. (2009a); Zaremba & Sutskever (2014)).
Automatic discovery of curricula based on learning progress Graves et al. (2017), adversarial self-play Sukhbaatar et al. (2018); Held et al. (2017), or backtracking Florensa et al. (2017) has been studied. So far, these methods have not yielded curricula capable of automatically discovering tasks of the complexity we consider. In this work, we will manually design a curriculum based on some assumptions presented in Section 5.3.

3 Relational Reinforcement Learning using inductive architecture biases

Many approaches in machine learning which have a capacity for relational reasoning use a "relational inductive bias". While not having a precise formal definition, we use this term to refer generally to inductive biases which impose "constraints" on relationships and interactions among entities in a learning process. Creative new machine learning architectures have rapidly proliferated in recent years, with practitioners often following a design pattern of composing elementary building blocks to form more complex, deep computational hierarchies and graphs. Building blocks such as “fully connected” layers are stacked into “multilayer perceptrons” (MLPs), “convolutional layers” are stacked into “convolutional neural networks” (CNNs), and a standard recipe for an image processing network is, generally, some variety of CNN composed with a MLP. This composition of layers provides a particular type of relational inductive bias that of hierarchical processing, in which computations are performed in stages, typically resulting in increasingly long range interactions among information in the input signal. The building blocks themselves also carry various relational inductive biases.

3.1 Relational reasoning and inductive bias

Structure is defined as the result of putting together a set of known building blocks. This composition (i.e., the arrangement of the elements) is captured by "structured representations," and "structured computations" operate on the elements and their composition as a whole. Using rules for how entities and relations can be composed, relational reasoning involves manipulating structured representations of entities and relations.

Learning is the process of gaining useful knowledge through observation and interaction with the environment. It entails sifting through a set of solutions in search of one that will provide a better explanation of the data or result in higher rewards. However, in many cases, there are several equally good options. Mitchell (2002) defined an inductive bias as the ability of a learning algorithm to prioritize one solution (or interpretation) over another regardless of the observed data.

In a Bayesian model, inductive biases are typically expressed through the prior distribution's choice and parameterization (Griffiths et al. (2010)). In other cases, an inductive bias could be added as a regularization term to avoid overfitting (Rumelhart et al. (1986)), or it could be encoded in the algorithm's architecture. The bias-variance tradeoff can be used to explain inductive biases, which often trade flexibility for increased sample complexity (Geman et al. (1992)). Inductive biases, in theory, should help find solutions that generalize in a desirable way while also improving the search for solutions without significantly lowering performance. Mismatched inductive biases, on the other hand, can lead to sub-optimal performance by introducing too strong constraints.

3.2 Relational inductive biases in standard deep learning building blocks

3.2.1 Fully connected layers

A fully connected layer is perhaps the most common building block (Van Der Malsburg (1986)). Each element, or "unit," of the output vector is the dot product of a weight vector, an added bias term, and finally a non-linearity such as a rectified linear unit (ReLU). As a result, the entities are the network's units, the relationships are all-to-all (all units in layer j are connected to all units in layer j+1), and the weights and biases define the rules. In a fully connected layer, the implicit relational inductive bias is thus very weak: all input units can interact to determine the value of any output unit, independently across outputs.

3.2.2 Convolutional layers

A convolutional layer is another common building block (LeCun et al. (1989)). It's done by convolutioning an input vector or tensor with a kernel of the same rank, adding a bias term, and then applying point-wise non-linearity. Individual units (or grid elements, such as pixels) are still present, but the relationships are more sparse. The differences between a fully connected layer and a convolutional layer impose some significant inductive biases in relational reasoning: Translation invariance and locality.

Locality reflects that the arguments to the relational rule are those entities in close proximity with one another in the input signal's coordinate space. Translation invariance reflects reuse of the same rule across localities in the input. These biases are very effective for processing natural image data because there is high covariance within local neighborhoods, which diminishes with distance, and because the statistics are mostly stationary across an image.

3.2.3 Recurrent layers

A recurrent layer (Elman (1990)) is a third common building block that is implemented in a series of steps. The inputs and hidden states at each processing step can be viewed as entities, and the

Markov dependence of one step's hidden state on the previous hidden state and the current input as a the relations. To update the hidden state, the rule for combining the entities takes a step's inputs and hidden state as arguments. The rule is repeated at each step, reflecting the relational inductive bias of temporal invariance (which is similar to the translational invariance of a CNN in space). Because of their Markovian structure, RNNs have a bias for locality in the sequence.

3.3 DeepSet: permutation invariance

In some machine learning tasks such as estimation of population statistics, anomaly detection or multiple instance learning, the order of the inputs is not important. The architecture used in these tasks should take in consideration this inductive bias, by being permutation invariant. Another requirement generally associated with the latter is being able to process inputs of any size. Problems that satisfy these two conditions are called set-input problems, since the requirements stem from the definition of a set. Classical neural-net based models do not satisfy easily these requirements: classical feed-forward neural networks violate both requirements, and RNNs are sensitive to input order.

DeepSet (Zaheer et al. (2017)) is a simple model satisfying both aforementioned requirements, and more importantly, is proven to be a universal approximator for any set function. In this model, each element in a set is first independently fed into a feed-forward neural network that takes fixed-size inputs. Resulting feature-space embeddings are then aggregated using a pooling operation (mean, sum, max or similar). The final output is obtained by further non-linear processing of the aggregated embedding.

Theorem 1. *A function $f : 2^{\mathfrak{X}} \to \mathcal{Y}$ operating on a set X is a valid set function i.e., **invariant** to the permutation of instances in X: for any permutation π : $f(\{x_1,\dots,x_M\}) = f(\{x_{\pi(1)},\dots,x_{\pi(M)}\})$, iff it can be decomposed in the form:*

$$\rho\left(\sum_{x\in X}\phi(x)\right) \tag{13}$$

, for suitable transformations ϕ and ρ.

Proof. See supplementary material. □

We can deconstruct 13 into two parts: an encoder ϕ which independently acts on each element of a set of items, and a decoder $\rho(\sum)$ which aggregates these encoded features and produces our desired output. Most network architectures for set-structured data follow this encoder-decoder structure.

Even though this set pooling approach is theoretically attractive, it remains unclear whether we can approximate complex mappings well using only instance-based feature extractors and simple pooling operations. Since every element in a set is processed independently in a set pooling operation, some information regarding interactions between elements has to be necessarily discarded. This can make some problems unnecessarily difficult to solve. (For example, learning the function that maps a set to its max can be very hard, when using the sum pooling operation.)

3.4 Beyond DeepSets: ReNN and TransformerSets

3.4.1 ReNN

In order to overcome the limitations of DeepSets, many methods and models were proposed. For instance, to encode pairwise-interactions, we could use features from all pairwise combinations of objects act as input to the same MLP, which is exactly the ReNN Raposo et al. (2017a) architecture with Object pair prior. However, this strategy scales exponentially with the number of interactions.

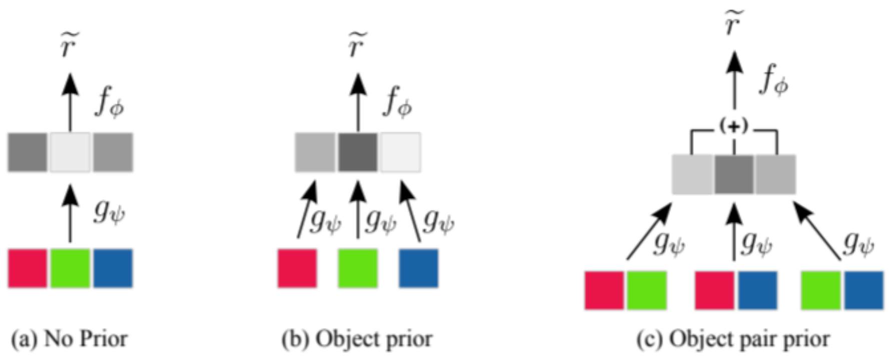


Figure 3: ReNN architecture - Image Source : original paper Raposo et al. (2017a)

An other architecture that deals with the limitations of DeepSets is the Set Transformer, with two novel design choices:

- Using a self-attention mechanism to process every element in the input set, which allows to naturally encode pairwise or higher-order interactions between elements in the set.
- Using a self-attention mechanism to aggregate features, which is especially beneficial when the problem requires multiple outputs which depend on each other

3.4.2 Attention Mechanism

An attention function $\mathrm{Att}(Q,K,V)$ is a function that maps queries $Q \in \mathbb{R}^{n\times d_q}$ to outputs using n_v key-value pairs $K \in \mathbb{R}^{n_v\times d_q}$, $V \in \mathbb{R}^{n_v\times d_v}$.

$$\mathrm{Att}(Q,K,V;\omega) = \omega\left(QK^\top\right)V. \tag{14}$$

The pairwise dot product $QK^\top \in \mathbb{R}^{n\times n_v}$ measures how similar each pair of query and key vectors is, with weights computed with an activation function ω. The output $\omega(QK^\top)V$ is a weighted sum of V where a value gets more weight if its corresponding key has larger dot product with the query.

Multi-head attention is an extension of the previous attention scheme. Instead of computing a single attention function, this method first projects Q,K,V onto h different d_q^M, d_q^M, d_v^M-dimensional vectors, respectively. An attention function $(\mathrm{Att}(\cdot;\omega_j))$ is applied to each of these h projections.

The output is a linear transformation of the concatenation of all attention outputs:

$$\text{Multihead}(Q, K, V; \lambda, \omega) = \text{concat}(O_1, \cdots, O_h) W^O, \tag{15}$$

$$\text{where } \; O_j = \text{Att}(Q W_j^Q, K W_j^K, V W_j^V; \omega_j) \tag{16}$$

Note that Multihead$(\cdot,\cdot,\cdot;\lambda)$ has learnable parameters $\lambda = \{W_j^Q, W_j^K, W_j^V\}_{j=1}^h$, where $W_j^Q, W_j^K \in \mathbb{R}^{d_q \times d_q^M}$, $W_j^V \in \mathbb{R}^{d_v \times d_v^M}$, $W^O \in \mathbb{R}^{h d_v^M \times d}$.

3.4.3 Building Blocks

Given matrices $X, Y \in \mathbb{R}^{n \times d}$ which represent two sets of d-dimensional vectors, we define the Multihead Attention Block (MAB) with parameters ω as follows:

$$\text{MAB}(X, Y) = \text{LayerNorm}(H + \text{rFF}(H)), \tag{17}$$

$$\text{where } \; H = \text{LayerNorm}(X + \text{Multihead}(X, Y, Y; \omega)) \tag{18}$$

rFF is any row-wise feedforward layer (i.e., it processes each instance independently and identically), and LayerNorm is layer normalization.

Using the MAB, we define the Set Attention Block (SAB) as

$$\text{SAB}(X) := \text{MAB}(X, X).$$

A potential problem with using SABs for set-structured data is the quadratic time complexity $\mathcal{O}(n^2)$, which may be too expensive for large sets ($n \gg 1$). The *Induced Set Attention Block* (ISAB) bypasses this problem. Along with the set $X \in \mathbb{R}^{n \times d}$, additionally define m d-dimensional vectors $I \in \mathbb{R}^{m \times d}$ called *inducing points*. Inducing points I are part of the ISAB itself, and they are *trainable parameters* that are trained along with other parameters of the network.

An ISAB with m inducing points I is defined as:

$$\text{ISAB}_m(X) = \text{MAB}(X, H) \in \mathbb{R}^{n \times d}, \tag{19}$$

$$\text{where } \; H = \text{MAB}(I, X) \in \mathbb{R}^{m \times d}. \tag{20}$$

3.4.4 The overall Set Transformer architecture

Using the ingredients explained above, a set transformer (Raposo et al. (2017b)) consists of an encoder and a decoder as follows:
The Encoder : $X \mapsto Z \in \mathbb{R}^{n \times d}$ is a stack of SABs or ISABs, for example:

$$\text{Encoder}(X) = \text{SAB}(\text{SAB}(X)) \tag{21}$$

$$\text{Encoder}(X) = \text{ISAB}_m(\text{ISAB}_m(X)). \tag{22}$$

We point out again that the time complexity for ℓ stacks of SABs and ISABs are $\mathcal{O}(\ell n^2)$ and $\mathcal{O}(\ell n m)$, respectively.

After the encoder transforms data $X \in \mathbb{R}^{n \times d_x}$ into features $Z \in \mathbb{R}^{n \times d}$, the decoder aggregates them into a single or a set of vectors which is fed into a feed-forward network to get final outputs.

A common aggregation scheme in permutation invariant networks is a dimension-wise average or maximum of the feature vectors. Set Transformers aggregate features by applying multihead attention on a learnable set of k seed vectors $S \in \mathbb{R}^{k \times d}$. Let $Z \in \mathbb{R}^{n \times d}$ be the set of features constructed from an encoder. *Pooling by Multihead Attention* (PMA) with k seed vectors is defined as

$$\mathrm{PMA}_k(Z) = \mathrm{MAB}(S, \mathrm{rFF}(Z)). \tag{23}$$

Note that the output of PMA_k is a set of k items. We use one seed vector ($k = 1$) in most cases. To further model the interactions among the k outputs, we apply an SAB afterwards:

$$\mathrm{Decoder} = \mathrm{SAB}(\mathrm{PMA}_k(Z)). \tag{24}$$

3.4.5 Analysis

Since the blocks used to construct the encoder (i.e., SAB, ISAB) are permutation invariant, the mapping of the encoder $X \rightarrow Z$ is permutation invariant as well. Combined with the fact that the PMA in the decoder is a permutation invariant transformation, we have the following:

Proposition 1. *The Set Transformer is permutation invariant.*

Being able to approximate any function is a desirable property, especially for black-box models such as deep neural networks. Building on previous results about the universal approximation of permutation invariant functions, we have:

Proposition 2. *The Set Transformer is a universal approximator of permutation invariant functions.*

3.5 Graph Networks

While the standard deep learning toolkit includes methods with different types of relational inductive biases, there is no “default” deep learning component that works with any relational structure. Models with explicit representations of entities and relations, as well as learning algorithms that find rules for computing their interactions and ways to ground them in data, are required. Importantly, the world’s entities (such as objects and agents) do not have a natural order; rather, orderings are defined by the properties of their relationships. Invariance to ordering except in the face of relations is a property that a deep learning component should ideally reflect. Graphs, in general, are a representation that can support any (pairwise) relational structure, and computations over graphs can provide a stronger relational inductive bias than convolutional and recurrent layers.

3.5.1 Background

Under the umbrella of "graph neural networks" (Gori et al. (2005)), neural networks that operate on graphs and structure their computations accordingly have been developed and explored extensively for more than a decade, but they have grown rapidly in scope and popularity in recent years.

In supervised, semi-supervised, unsupervised, and reinforcement learning settings, models from the graph neural network family have been investigated in a wide range of problem domains. They've performed well in tasks with a complex relational structure, such as visual scene comprehension or learning the dynamics of physical systems (Battaglia et al. (2016)). Many traditional computer science problems involving reasoning about discrete entities and structure, such as combinatorial optimization (Bello et al. (2016)), boolean satisfiability (Selsam et al. (2019)), program representation and verification (Allamanis et al. (2018)) and performing inference in graphical models (Yoon et al. (2018)) have also been explored with graph neural networks.

3.5.2 Definition of "graph"

A *graph* is defined as a 3-tuple $G = (\mathbf{u}, V, E)$.The $\mathbf{u}$ is a global attribute; for example, $\mathbf{u}$ might represent the gravitational field. The $V = \{\mathbf{v}_i\}_{i=1:N^v}$ is the set of nodes (of cardinality N^v), where each $\mathbf{v}_i$ is a node's attribute. The $E = \{(\mathbf{e}_k, r_k, s_k)\}_{k=1:N^e}$ is the set of edges (of cardinality N^e), where each $\mathbf{e}_k$ is the edge's attribute, r_k is the index of the receiver node, and s_k is the index of the sender node.

3.5.3 GN block

A GN block contains three "update" functions, ϕ, and three "aggregation" functions, ρ,

$$\begin{aligned} \mathbf{e}'_k &= \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right) & \bar{\mathbf{e}}'_i &= \rho^{e\to v}\left(E'_i\right) \\ \mathbf{v}'_i &= \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right) & \bar{\mathbf{e}}' &= \rho^{e\to u}\left(E'\right) \\ \mathbf{u}' &= \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right) & \bar{\mathbf{v}}' &= \rho^{v\to u}\left(V'\right) \end{aligned} \tag{25}$$

where $E'_i = \left\{\left(\mathbf{e}'_k, r_k, s_k\right)\right\}_{r_k=i,\, k=1:N^e}$, $V' = \left\{\mathbf{v}'_i\right\}_{i=1:N^v}$, and $E' = \bigcup_i E'_i = \left\{\left(\mathbf{e}'_k, r_k, s_k\right)\right\}_{k=1:N^e}$.

The ϕ^e is mapped across all edges to compute per-edge updates, the ϕ^v is mapped across all nodes to compute per-node updates, and the ϕ^u is applied once as the global update. The ρ functions each take a set as input, and reduce it to a single element which represents the aggregated information. Crucially, the ρ functions must be invariant to permutations of their inputs, and should take variable numbers of arguments (e.g., element-wise summation, mean, maximum, etc.).

When a graph, G, is provided as input to a GN block, the computations proceed from the edge, to the node, to the global level.

1. ϕ^e is applied per edge, with arguments $(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$, and returns $\mathbf{e}'_k$. The set of resulting per-edge outputs for each node, i, is, $E'_i = \left\{\left(\mathbf{e}'_k, r_k, s_k\right)\right\}_{r_k=i,\, k=1:N^e}$. And $E' = \bigcup_i E'_i = \left\{\left(\mathbf{e}'_k, r_k, s_k\right)\right\}_{k=1:N^e}$ is the set of all per-edge outputs.
2. $\rho^{e\to v}$ is applied to E'_i, and aggregates the edge updates for edges that project to vertex i, into $\bar{\mathbf{e}}'_i$, which will be used in the next step's node update.
3. ϕ^v is applied to each node i, to compute an updated node attribute, $\mathbf{v}'_i$. The set of resulting per-node outputs is, $V' = \left\{\mathbf{v}'_i\right\}_{i=1:N^v}$.

4. $\rho^{e\to u}$ is applied to E', and aggregates all edge updates, into $\bar{\mathbf{e}}'$, which will then be used in the next step's global update.
5. $\rho^{v\to u}$ is applied to V', and aggregates all node updates, into $\bar{\mathbf{v}}'$, which will then be used in the next step's global update.
6. ϕ^u is applied once per graph, and computes an update for the global attribute, $\mathbf{u}'$.

Though we present this sequence of steps here, the order is not strictly enforced: it is possible to reverse the update functions to proceed from global, to per-node, to per-edge updates, for example.

3.5.4 Invariant and equivariant functions on graphs

We start with some definitions. For a vector $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{F}^n$ and a permutation $\sigma \in S_n$ we define:

$$\sigma \star \mathbf{x} = (x_{\sigma^{-1}(1)}, \dots, x_{\sigma^{-1}(n)})$$

where $\mathbb{F}$ could be $\mathbb{R}$, $\mathbb{R}^d$, etc.

- a function $f : \mathbb{F}^n \to \mathbb{F}$ is **invariant** if for all $\mathbf{x}$ and $\sigma \in S_n$ we have $f(\sigma \star \mathbf{x}) = f(\mathbf{x})$
- a function $f : \mathbb{F}^n \to \mathbb{F}^n$ is **equivariant** if for all $\mathbf{x}$ and $\sigma \in S_n$ we have $f(\sigma \star \mathbf{x}) = \sigma \star f(\mathbf{x})$

We already characterised invariant function in Theorem 1, we do the same for equivariant functions:

Theorem 2. *A function $f : \mathbb{F}^n \to \mathbb{F}^n$ is **equivariant** iff it can be decomposed in the form:*

$$f_j(x) = \rho(x_j, \sum_{i=1}^{n} \phi(x_i)) \tag{26}$$

for suitable transformations $\phi : \mathbb{F}^n \to \mathbb{F}^n$ and $\rho : \mathbb{F} \times \mathbb{F}^n \to \mathbb{F}$, with $f(x) = (f_1(x), \dots, f_n(x))$

Proof. See supplementary material. □

3.5.5 Message Passing Layer

A message passing layer is a function applied in parallel to all nodes (with the same function) producing a mapping from node features at layer l: $\mathbf{h}^l = (h_1^l, \dots, h_n^l)$ to node features at layer $l+1$: $\mathbf{h}^{l+1} = F(\mathbf{h}^l)$ with $F : \mathbb{F}^n \to \mathbb{F}^n$ clearly an equivariant function: permuting its input should permute its output.

From results of Theorem 2, every message passing layer should be of the form 26, more precisely, we have:

$$\mathbf{h}_j^{l+1} = \rho(h_j^l, \sum_{i\sim j} \phi(h_i^l)) \tag{27}$$

where the sum is only taken from the neighbours of j.

By varying the functions ρ and ϕ, you get: vanilla GCN, GraphSage, Graph Attention Network, MoNet, Gated Graph ConvNet, Graph Isomorphism Networks...

3.5.6 Graph Attention Network

The graph attention network layer (Veličković et al. (2018)) is a special case of the message passing layer, where the node features at layer $l+1$ are a weighted mean of the node features at layer l, where the weights are calculated using an attention mechanism:

$$\mathbf{h}_i^{l+1} = \sigma(\alpha_{i,i}^l \Theta \mathbf{h}_i^l + \sum_{j\sim i} \alpha_{i,j}^l \Theta \mathbf{h}_j^l), \tag{28}$$

$$\text{where } \alpha_{i,j}^l = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top [\Theta \mathbf{h}_i^l \parallel \Theta \mathbf{h}_j^l]\right)\right)}{\sum_{k\sim i} \exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top [\Theta \mathbf{h}_i^l \parallel \Theta \mathbf{h}_k^l]\right)\right)}. \tag{29}$$

and $\parallel$ is the concatenation operation, a and Θ are learnable parameters, and σ a non-linearity.

To stabilize the learning process of self-attention, extending the mechanism to employ multi-head attention is beneficial: K independent attention mechanisms execute the transformation of Equation 26, and then their features are concatenated, resulting K times the features.

This formulation of GAT layers suffers from the static attention problem: since the linear layers in the standard GAT are applied right after each other, the ranking of attended nodes is unconditioned on the query node. The GATv2 operator from the “How Attentive are Graph Attention Networks?” paper Brody et al. (2021), fixes this problem by changing the order of the operations:

$$\mathbf{h}_i^{l+1} = \sigma(\alpha_{i,i}^l \Theta \mathbf{h}_i^l + \sum_{j\sim i} \alpha_{i,j}^l \Theta \mathbf{h}_j^l), \tag{30}$$

$$\text{where } \alpha_{i,j}^l = \frac{\exp\left(\mathbf{a}^\top \text{LeakyReLU}\left(\Theta [\mathbf{h}_i^l \parallel \mathbf{h}_j^l]\right)\right)}{\sum_{k\sim i} \exp\left(\mathbf{a}^\top \text{LeakyReLU}\left(\Theta [\mathbf{h}_i^l \parallel \mathbf{h}_k^l]\right)\right)}. \tag{31}$$

4 Problem formulation and experimental setup

4.1 Environment

The simulated robotic environment consists of a 7-DoF Fetch robot arm equipped with a two-fingered parallel jaw gripper, based on OpenAI's FetchPickAndPlace (Plappert et al. (2018)). The robot is simulated using the MuJoCo (Todorov et al. (2012)) physics engine and is tasked to manipulate blocks kept on a table. Each block is a cube with sides of 5cm.

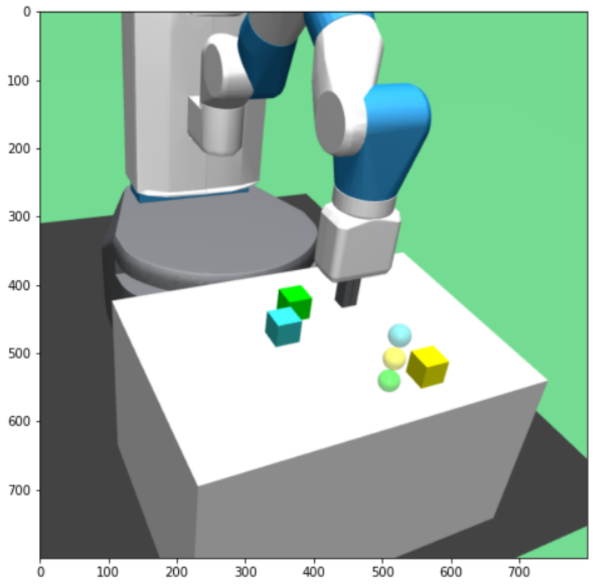


Figure 4: Example of the environment: 3 Blocks, Single Tower Case, True Stack

4.1.1 Observations

The agent observes a scene consisting of a gripper and N different blocks (cubes).

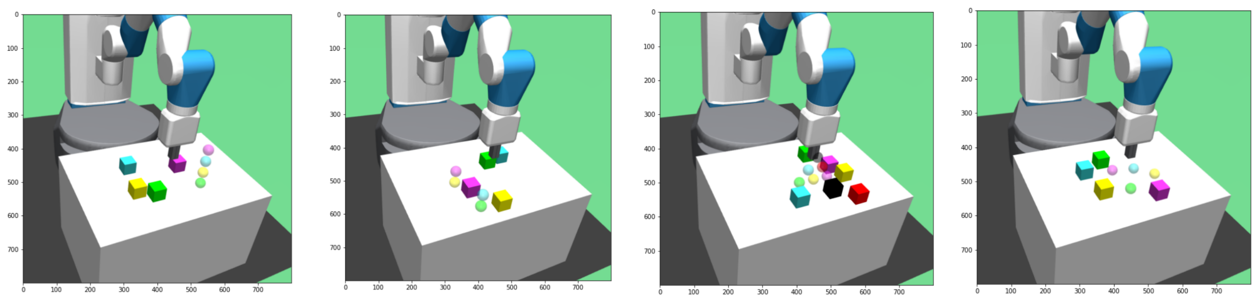
- The gripper is represented by X_{grip} and consists of the gripper's velocity and position.
- Each block is represented by a 15D vector: 3D position (x_p^i), 3D orientation expressed as Euler angles, 3D position relative to the gripper, 3D Cartesian velocity and 3D angular velocity, denoted by $X_{block_i}^{features}$
- Each goal of each block is represented by a 3D position vector $X_{block_i}^{goal}$

To have a representation of the scene that is permutation invariant (in order to be easily represented as a graph later), we choose to represent the overall input as: $\{X_i, \quad i \in \{1,..,N\}\}$ with $X_i = \{X_{grip}, X_{block_i}^{features}, X_{block_i}^{goal}\}$.
The maximum length of every episode is $50 * N$ steps, where N is the number of blocks.

4.1.2 Goals

Goals describe the desired 3D positions of the N blocks, with some fixed tolerance of ϵ. Different goal configurations are possible:

- **Single Tower**: A single point is uniformly sampled on the table to serve as the base of the tower. The goal positions of the other blocks correspond to translation along the z-axis from the base.
- **Multiple Tower**: Few points are sampled on the table to serve as the base location of multiple towers. Each block was randomly assigned to a tower to produce towers of approximately equal height.
- **Pyramid**: A uniformly sampled point on the table served as a corner point for pyramid configuration.
- **False Stack-Only**: The goal position of (N - 1) blocks were chosen randomly on the table, while the last block was chosen either on the table or in the air.



(a) Single Tower (b) Multiple Tower (c) Pyramid (d) False-Stack

Figure 5: Different goal configurations: goals are represented with spheres of same colors as the blocks

4.1.3 Action

The action space is 4-dimensional. Three dimensions specify the desired relative gripper position at the next time-step. We use MuJoCo constraints to move the gripper towards the desired position. The last dimension specifies the desired distance between the 2 fingers which are position controlled.

4.1.4 Reward

We use two types of reward:

- Sparse: The agent is only rewarded when all the objects are in their respective goal positions (with a tolerance). Each time-step where one of the goals is not achieved, the agent receives a -1 reward, if all the blocks are well in their goal positions, the agent receives a 0 reward. To sum up: $r = \min\left\{ -\mathbb{1}_{\left\| x_i^p - X_{block_i}^{goal} \geq \epsilon \right\|},\ i \in \{1, \ldots, N\} \right\}$

- Incremental: This is a step-wise reward where the agent is rewarded every time one of the blocks is well placed. Each time step where all the N goals are not achieved, the agent receives -N as a reward, if exactly one of the goals is realised, It receives -N+1, etc. To sum up: $r = -\sum_{i=1}^{N} \mathbb{1}_{\left\| x_i^p - X_{block_i}^{goal} \geq \epsilon \right\|}$

For both rewards, we take $\epsilon = 5\text{cm}$

4.1.5 State-goal initialisation

At the start of every episode, the initial block positions are randomly initialized on the table and the goal positions are sampled using a pre-determined distribution, depending on the goal configuration to be used. Refer to section 7.2.2 for modifications made to state environment depending on the experiment.

4.1.6 Multi-env wrapper

This is a special wrapper used to have a diversified batch: at each start of an episode, a number of block N is chosen randomly in $\{1, \ldots, N_{max}\}$.

To sum up, an environment is completely characterised by:

- the number of blocks N
- the goal configuration (case): SingeTower, MultiTower, Pyramid, etc
- the reward type: sparse, incremental

Here is an example for creating a gym environment:

```python
gym_env = gym.make(f"FetchBlockConstruction_{args.num_blocks}Blocks_{args.
    reward}Reward_DictstateObs_42Rendersize_{args.stack_only}Stackonly_{
    args.case}Case-v1")
```

Listing 1: gym environment creation example

4.2 Problem formulation

The goal is to train an agent to stack blocks. We want the agent to be:

- robust to the variation of the number of blocks N. This for example dismisses the use of some architectures like an MLP since the size of the first layer depends on N
- permutation invariant to the order of the blocks in the input representation
- generalise over an unseen number of blocks and goal configurations

5 Method

In this section, we present two simple methods for solving long-horizon, sparse reward tasks using reinforcement learning: curriculum learning where the tasks are presented in an increasingly complex order and diversified batching where all the different tasks are presented to the agent at once (in the same batch).
But before, a brief presentation of the internal repository used for this task and the architecture of the agents.

5.1 PARL

All the experiments were done using the PARL repository, an internal repository used at InstaDeep. For the sake of this report, only one important feature will be introduced: distributed agents.

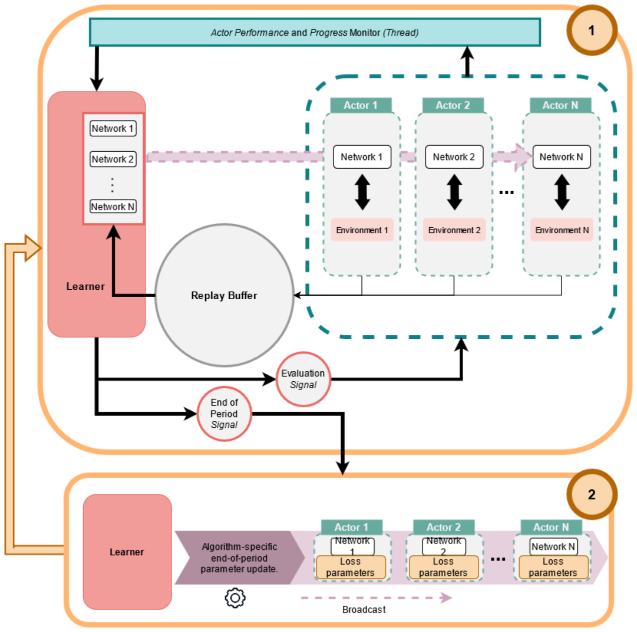


Figure 6: Distributed agents in the PARL repository

An agent must interact with the environment to generate its own training data. This motivates interacting with multiple instances of an environment (simulated or otherwise) in parallel to generate more experience to learn from.

In the PARL repository, it is done in an asynchronous way: in the illustrated example this distributed agent consists of a data storage process, a learner process, and one or more distributed actors, each with their own environment. In a nutshell, actors collect transitions from their respective environment, add it to the replay buffer, then the learner samples a batch from the replay buffer, updates its parameters and synchronizes them with the different actors.

By adopting rate limitation, a desired relative rate of learning to acting can be enforced, allowing the actor and learner processes to run unblocked, so long as they remain within some defined tolerance of the prescribed rate.

This distributed asynchronous scheme has two benefits: the learning process proceeds as quickly as possible regardless of the speed of data gathering and by making use of more actors in parallel the data generation process is accelerated. For a more in-depth read about the different technical implementation details (reverb tables, adders, syncers, rate limiters), the article from DeepMind introducing ACME (Hoffman et al. (2020) is a good start.

5.2 Agents

The RL agents are equipped with inductive biases of relational reasoning, in order to enable learning, by using the power of graph-based architectures. We use Soft-Actor Critic (Haarnoja et al. (2018a)) as our base learning algorithm, alongside goal re-labelling via hindsight experience replay (Andrychowicz et al. (2017)).

5.2.1 Architecture

Both the actor and critic in SAC are represented using a graph attention network (GATv2, introduced in Section 3.5.6). This GAT architecture is composed of three message passing rounds, with 64D linear layers each and 4 multi-head attention. To ease optimization, a normalization layer is added between the output of message passing round t and the input of message passing round t + 1. After the 3 message passing rounds, a self-attention pooling operation reduces the graph representation (which at this point is a N*64*4 dimensions vector) into one 64D vector, which then goes through a 3 layers MLP (Multi-Layer Perceptron) decoder with 64D layers each.

All the hyperparameters related to the GAT architecture are resumed in the table 2

5.2.2 SAC+HER agent

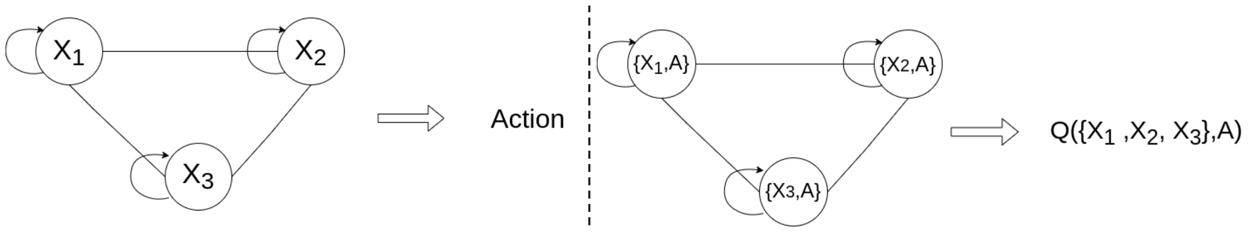
We use a goal modified implementation of SAC with Automatically Adjusted Temperature (automatic entropy tuning). As mentioned above, the actor and the critic are graph attention networks, which means that the input of these networks should be graphs.

The actor:

For the actor, the input is a graph representation of the state s, composed of N nodes (N the number of blocks), each node represented with its overall feature X_i (cf observations Section 4.1.1), and connected to all the other nodes (fully connected graph). The output of the actor is a 4D dimensional vector representing the action.

The critic:

For the critic, the input is normally a state and action tuple (s, a), which will also be represented with a fully connected graph of N nodes, with each node being represented with (X_i, a). The output is a 1D scalar representing the Q value of the state action tuple (s, a).



(a) The actor (b) The critic

Figure 7: SAC's actor and critic graph representation

Graph Implementation:

In order to implement graph attention networks, the pytorch-geometric package comes very handy. A graph is represented by its node features list and an edge index which is a graph connectivity in COO format with shape $[2, num\_edges]$. Please refer to the example in section 5.4 for examples of graphs and their representation.

HER relabelling

The actors, running in parallel processes as explained above, collect full episodes of transitions (1 episode = 50 * N timesteps, N the number of blocks) and add them to a replay buffer, also hosted in a different process. The goal relabelling strategy of HER happens when the learner samples a batch of transitions from the replay buffer: we replace the desired goal with the achieved goal and recalculate the corresponding rewards.

All the hyperparameters related to SAC+HER can be found in the table 3.

5.3 Curriculum

In order for the agent to be able to learn, tasks are presented in a meaningful order, with increasing complexity. To do so, we make the following two assumptions:

- the tasks get harder as the number of blocks N gets bigger
- stacking is the hardest task among the other goal configurations, while False Stack-Only is the easiest.

The first assumption is straightforward; having more blocks to deal with makes the task always harder, with a longer horizon and more goals to achieve.

For the second assumption, we argue that stacking perfectly blocks is the hardest task among the other goal configurations presented because the agent needs to: sequentially achieve each goal perfectly in order for the stack to stay at equilibrium, try not to destroy the current stack while trying to reach for the remaining blocks, placing each block with precision in top of the current stack, etc. On the other hand, with stack-only at False, the agent just have to put the blocks in different positions without dealing with the elements cited before.

The curriculum learning consists of repeating three phases:

- load the parameters of the previous round (of the actor, the critic and the target critic)
- train the agent in a new slightly harder environment until the success rate is higher than 0.9
- save the new parameters (of the actor, critic and target critic)

The curriculum is totally characterised by the list of gradually harder environments. Since an environment is characterised also by by N, the goal configuration and the reward, we will directly consider the curriculum as a sequence of these tuples. Other special case environments may be introduced when needed, and are presented in the results section.

5.4 Diversified batching

Rather than presenting tasks in an increasing complexity fashion, we propose to use a batching strategy where the agent is presented uniformly examples from a various number of blocks and goals distributions into the same batch of data sampled by the learner.

Batching is essential for allowing a deep learning model's training to scale to large amounts of data. Instead of processing each example individually, a batch groups a collection of examples into a unified representation that can be processed in parallel.

This procedure is typically accomplished in the image or language domain by rescaling or padding each example into a set of equally-sized shapes, and the examples are then grouped in an additional dimension. The batch size refers to the length of this dimension, which is equal to the number of examples grouped into the batch.

Because the input graphs come from examples with different number of blocks, they can hold any number of nodes or edges. As a result, the two approaches described above are either not feasible or will consume a lot of memory. We use a different approach to parallelization across a variety of examples: Adjacency matrices are stacked diagonally (creating a massive graph with multiple isolated subgraphs), and node and target features are simply concatenated in the node dimension, i.e.

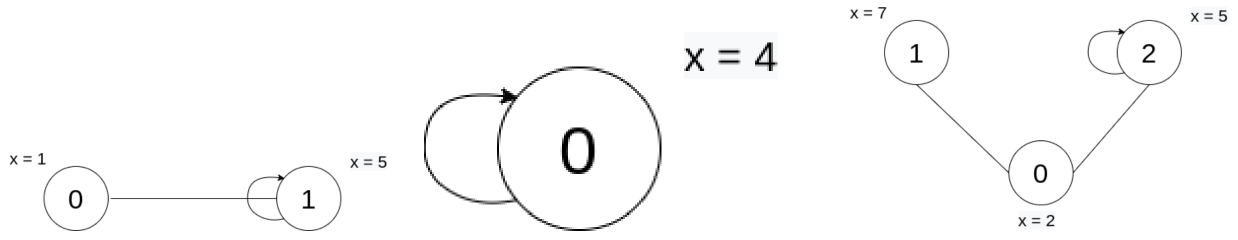
$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix}, \qquad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_n \end{bmatrix}. \tag{32}$$

This method has a number of significant advantages over other batching methods:

- Messages cannot be exchanged between two nodes that belong to different graphs, so GNN operators that rely on a message passing scheme do not need to be modified.
- There is no overhead in terms of computation or memory. This batching procedure works flawlessly without any padding of node or edge features. Adjacency matrices have no additional memory overhead because they are saved in a sparse manner with only non-zero entries, i.e., the edges.

We only need to keep a list of pointers for where the nodes associated with each graph begin and end in the batch.

Example: For this example, we will consider a batch composed of 3 graphs:



(a) First Graph (b) Second Graph (c) Third Graph

Figure 8: Example of graphs for explaining the batching strategy

Using the graph representation of pytorch-geometric:

The first graph: $X_1 = [1,5]$ and edge_index$_1 = \begin{matrix}[[0,1,1] \\ [1,0,1]]\end{matrix}$.

The second graph: $X_2 = [4]$ and edge_index$_2 = \begin{matrix}[[0] \\ [0]]\end{matrix}$.

The third graph: $X_3 = [2,7,6]$ and edge_index$_3 = \begin{matrix}[[0,0,1,2,2] \\ 1,2,0,0,2]]\end{matrix}$.

The Batch graph will be represented by: $X = [1,5,4,2,7,6]$ and edge_index $= \begin{matrix}[[0,1,1,2,3,3,4,5,5] \\ [1,0,1,2,4,5,3,3,5]]\end{matrix}$ and batch_assignment $= [0,0,1,2,2,2]$

Implementation details In order to implement the diversified batching approach, we use the multi-env wrapper, where every episode added to the reverb table represents a scene with different number of blocks. Since the length of an episode is a multiple of the number of blocks (N*50), epsiodes in the reverb table will have different lengths. To deal with varying size episode lenghts, the easiest way is to pad each episode with null transitions to have the same "max episode length". When the learner samples a batch of transitions from the batch of episodes in the reverb table, we unpadd by only selecting non null transitions.

6 Results

In this section, we will present the results of training an agent to stack blocks on the environment presented in section 4.1 using the methods explained in section 5 : we will use the sparse reward with a simple curriculum, show the emergent behaviour and local minima, propose a way to fix it, explore the zero-shot generalisation of the new agent and its failure mods and finally present the results of the diversified batching strategy.

6.1 First attempt: curriculum with the sparse reward

We train the SAC+HER agent with the graph attention network architecture presented on Section 5.2 in the environment of Section 4.1 with the sparse reward. We follow the following curriculum:

- First, the agent is tasked to select and place a single block at goal positions that have been uniformly and randomly assigned to the table or in the air: 1block_False_stack
- The robot then had to pick and place two blocks, with one block's goal position sampled on the table and the second block's goal position sampled using the process described above: 2blocks_False_stack.
- Starting with two blocks, the robot was tasked with stacking them in a single tower configuration. The robot was given N blocks to stack after it perfected stacking (N-1) blocks: Nblocks_True_stack.

N was sequentially increased from 2 to 4. The transition points in this curriculum were chosen when the success rate is higher than 0.9.

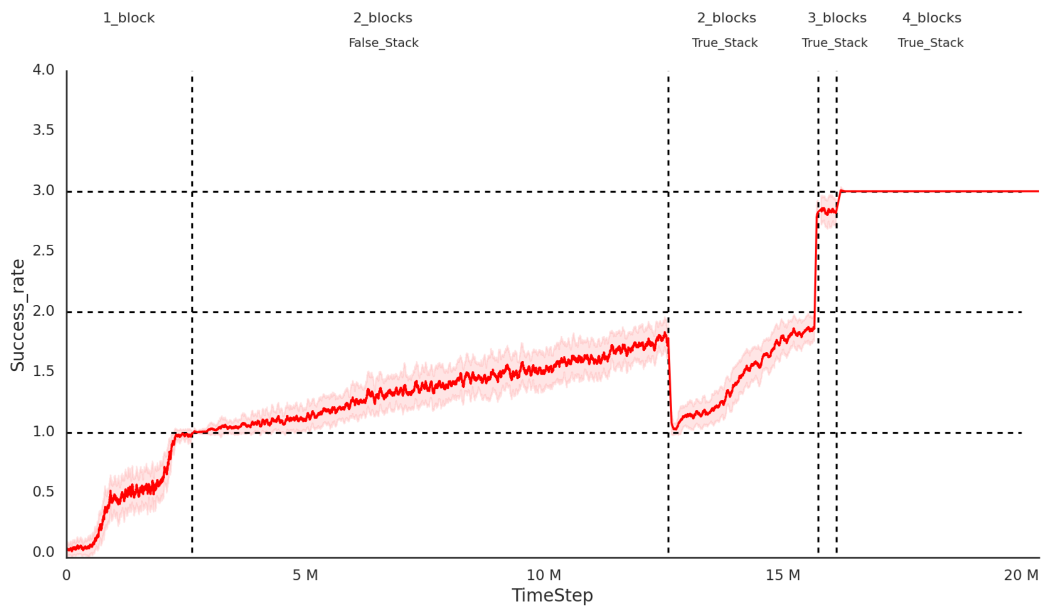


Figure 9: Result of the curriculum learning strategy with the sparse reward

The agent failed to stack 4 blocks, even when the transition from stacking 3 blocks to stacking 4 blocks was simplified by transitioning first to the 4blocks_False_Stack or even initialising the environment with stacks of 3 blocks already on position.

6.2 Emerging behaviour of the first attempt: local minima

The agent stacks perfectly 3 blocks but fails to stack 4 blocks. By looking at the emerging behaviour, we can see the reason why: the robot uses its gripper to hold the stack so it does not fall. In the 3 blocks stacking, the robot even uses its gripper to hold 2 blocks while putting them on the third block. This strategy clearly fails to transfer to 4 blocks, because the robot cannot put 3 blocks in its gripper.

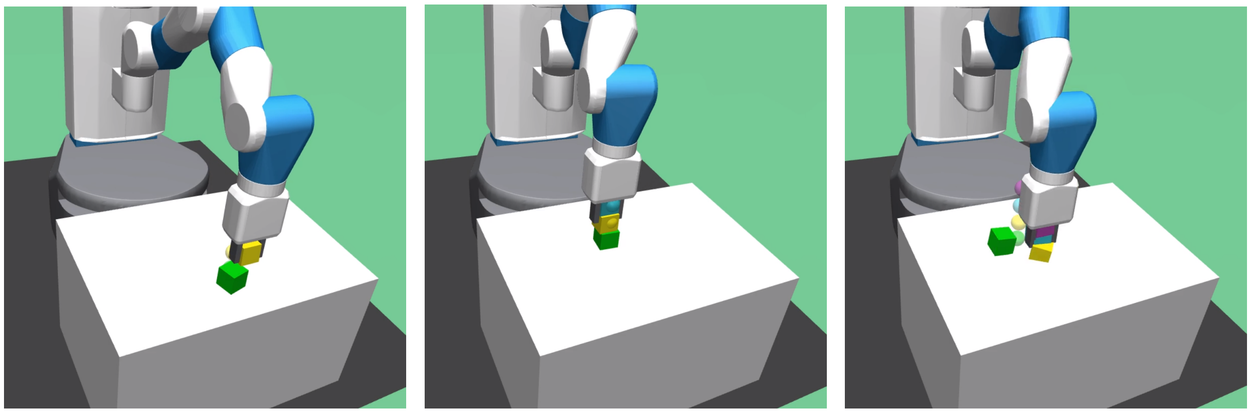

(a) 2 blocks (b) 3 blocks (c) 4 blocks

Figure 10: The emerging local minima behaviour: the agent holds the top blocks in its gripper

6.3 New reward and curriculum

To discourage the local minima emerging behavior behaviour, we added an additional term in the reward function to encourage the robot to move its hand away from the tower: this additional reward $\mathbb{1}_{gripp\_away}$ is only provided when the hand was at a distance greater than 2δ from the "fully stacked" tower. We also use the incremental reward, the overall reward is therefore given by: $r = -\sum_{i=1}^{N} \mathbb{1}_{\left\| x_i^p - X_{block_i}^{goal} \geq \epsilon \right\|} + \mathbb{1}_{gripp\_away}$

In order to fix the local minimum presented in the previous section, we had to restart the learning from the beginning. The curriculum presented before is no longer able to train the agent: we modify it by simplifying the transition from stacking N to stacking N+1 by adding a stage of training in $(N+1)$Blocks_False_Stack, where $N-1$ blocks were placed randomly on the table, and the last block is either placed on the table or in the air. This helped simply the transition and enable the learning for transition from N to $N+1$ stacking tasks, except for the case of 2 blocks where we even simplify the transition to 2Blocks_False_Stack by adding a stage we called Simple Case: one of the blocks is already placed in the right position while the position of the other block is chosen randomly on table or on the air. This added stage was needed since going from 1 block to 2 blocks seems to be the hardest, since it is the first time the agent experienced the presence of multiple objects.

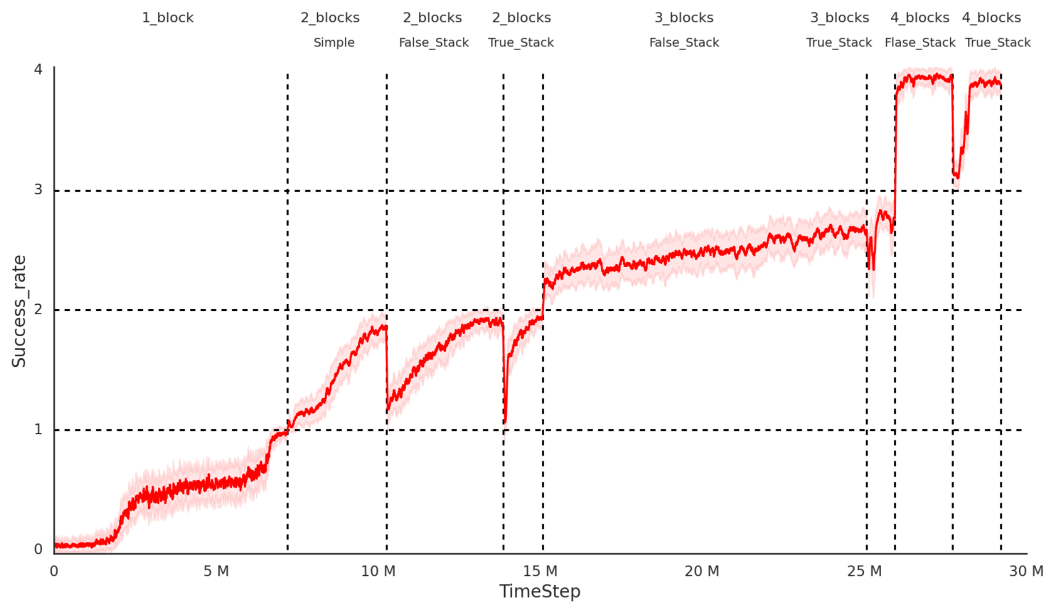


Figure 11: Result of the new reward with the new curriculum

6.4 Emerging behaviour and failure mods of the new agent

The accompanying videos (available at the link) show that the new agent automatically learns to perfectly stack up to 4 blocks and getting the gripper far from the finished stack. While learning this task, the agent learned to push other blocks to grasp a particular block, grasps two blocks at a time and places them one by one to save time and other complex behaviors. These strategies emerge automatically as a consequence of optimizing a sparse reward function.

On the other hand, our system's major failure modes are the following:

- **Oscillation**: The agent oscillates its end-effector without moving closer to the target. This frequently occurs when the target block is very close to the tower's base. Picking up the block in this scenario risks toppling the tower.
- **Insufficient time for recovery**: The agent is unable to stack all of the blocks within the episode's maximum time limit.
- **Blocks fall off during stacking**: the agent knocks one or more blocks off the table. The agent is unable to complete the task because the blocks are no longer on the table.
- **Blocks fall off after stacking**: the agent succeeds in stacking the tower, but the tower topples and the blocks fall off the table.

For stacking tasks, the most common failure mode is the "Blocks fall off after stacking".

6.5 Zero-shot generalization

It's ideal to learn policies that can be re-purposed for new and related tasks not seen in the training. If our architecture does indeed provide a good inductive bias, then different block configuration tasks should be solved with high accuracy. To demonstrate this, we evaluated the learned policy's performance on previously unseen block configurations without any fine-tuning (i.e. zero-shot generalization). Figure 3 summarizes the findings of this investigation.

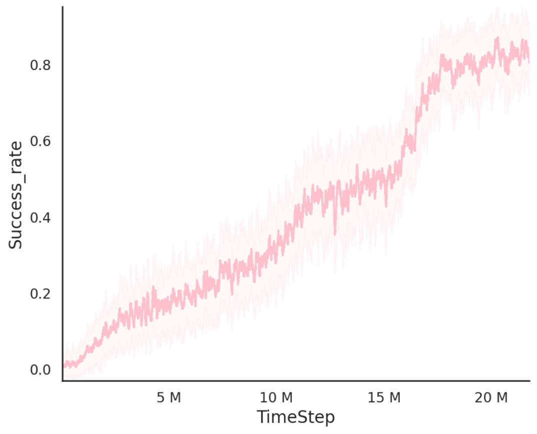
Table 1: Zero-shot generalization for an agent trained on stacking 3 blocks

3Blocks Stack-True	**4Blocks Stack-True**	**3Blocks MultipleTowers**	**3Blocks Pyramid**	**4Blocks Stack-False**	**4Blocks MultipleTowers**	**4Blocks Pyramid**	**5Blocks Stack-False**
1	0	0.905 (+- 0.03)	0.87(+- 0.05)	0.79 (+- 0.03)	0.63 (+-0.1)	0.59 (+-0.07)	0.47 (+-0.05)

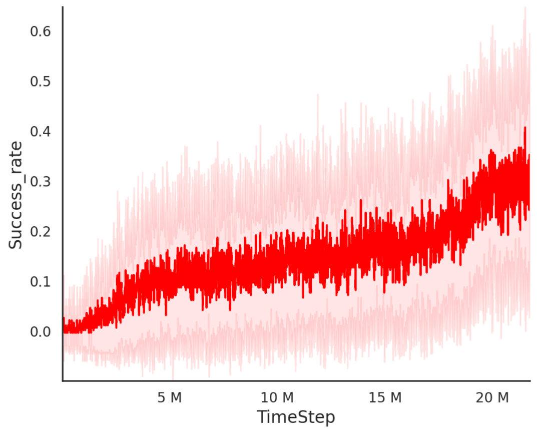
The agent trained on stacking blocks generalize very well to other goal configurations, even with higher number of blocks. This validates the hypothesis that stacking blocks is indeed the hardest task of the goal configurations explored.

6.6 Diversified Batching results

We use the diversified batching strategy explained in Section 5.4 with the multi-env wrapper of Section ? and the sparse reward. With this strategy, the agent learns to stack 2 blocks, however fails in stacking 3 blocks.



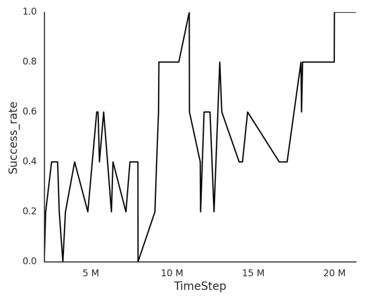
(a) 2 blocks



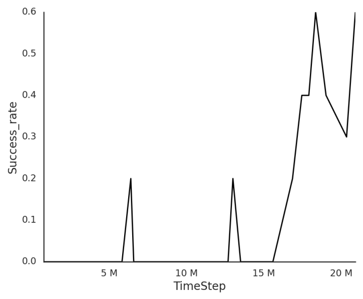
(b) 3 blocks

Figure 12: Results of diversified batching for 2 blocks and 3 blocks

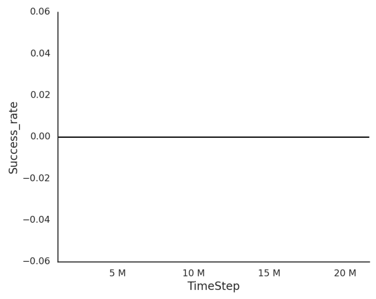
One interesting phenomena happens while the agent learns with the diversified strategy: we observe an implied curriculum, where the agent seems to succeed naturally in the easy tasks and transfers it to the harder one:



(a) 1 block



(b) 2 blocks



(c) 3 blocks

Figure 13: Implied curriculum while training with a diversified batching on 3 blocks

7 Discussion

7.1 Curriculum vs Batching

The importance of humanly crafted "stages" is shown in the results of the curriculum experiment. Skipping one of those stages resulted in the learning being disabled entirely. This, we believe, is related to the use of gradient-based methods (SAC): these methods need to succeed in the task from time to time in the exploratory phase, or they will not work. If the stage's difficulty rises dramatically, the agent experiences no reward and does not improve throughout the training. Another flaw in the curriculum is that every transition from one stage to the next is a new optimization problem, with the agent optimizing for a different goal and reward distribution. Having an agent with good inductive biases should make this new optimization problem easier, but new hyperparameters need to be fine-tuned everytime (especially the learning rate).

We experimented with a diversified batching strategy to eliminate the need for human-crafted curriculum. We believe that by randomly presenting the different tasks to the agent within the same batch, the agent will be able to directly optimize for an input, reward and goal distributions that are close to the "real" ones. The agent appears to be following an implied curriculum, completing tasks from easy to difficult. Based on the curriculum's results, it's possible that the diversified agent could have achieved similar results by adding more simplified transitional cases to the multi-env wrapper (stack false and simple case). However, this goes against our original goal of not using "stages" created by humans. In addition, the batching strategy appears to be less sample efficient than the curriculum and more computationally intensive (due to padding and batching graphs).

7.2 Important implementation details

Making a reinforcement learning experiment work in practise is generally a hard task. In addition to having different parts that should be working well independently and jointly, there are different small details that make a huge difference. While trying to have working experiments, we found some of these details that are worth mentioning:

7.2.1 Weight initialisation

The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network. If either occurs, loss gradients will either be too large or too small to flow backwards beneficially, and the network will take longer to converge, if it is even able to do so at all.

Another aspect of a good initialization is ensuring that the SAC agent has a Squashed-Gaussian actor with a zero mean and unit variance. This is critical for a successful SAC implementation.

Both elements are implemented in practice by using a Xavier initialization of the architecture's **every** weight matrix.

7.2.2 Environment initialisation

Having a "smart" environment initialization that helps the agent experience good reward signals is generally a good idea. This was accomplished in our case by: having blocks placed directly in their goal position, or in the gripper, or by having stacks that were already partially formed, with a small probability.

7.2.3 Architecture hyperparameters

Hyperparameters are crucial in reinforcement learning experiments. There are a few different ways to explore them. In our experiments, we discovered that two architecture-related hyperparameters were crucial in easing the optimization problem and even enabling learning in the first problems: the number of multi-head attention and layer normalization. The annex contains a list of all the other hyperparameters.

7.2.4 Saving and loading models in the curriculum

It was critical to save and load all details about the experiment in order to have a working curriculum: learning should resume without resetting any parameters, including the model's optimizers and SAC's entropy parameter (α). Saving and loading only the architecture's weights results in a non-working curriculum.

7.3 The other implementation

Another implementation is available at this link, which also tries to stack blocks using curriculum learning and graph-based architectures. However, when we tried to run it, we ran into a variety of issues and errors. After 1.4M timesteps and 5 days of training on a GCP VM (n2-standard-48, 48 vCPUs, 1 gpu), it achieved a 0.3 success rate on 1 block manipulation after multiple modifications. For comparaison, after 2M timesteps and less than an hour of training on the same machine, our implementation achieves a 1.0 success rate on 1 block manipulation. On the other hand, the script for executing the curriculum does not include all of the necessary information to fully replicate the results.

8 Conclusion

Using deep reinforcement learning and relational graph architecture, we have presented a framework for learning long-horizon, sparse reward tasks.

We proposed two methods to accomplish this: curriculum learning, which produced good results stacking up to four blocks and generalized with no fine tuning to other unseen block and goal configurations, but was sensitive to the curricula's human-crafted "stages." The diversified batch agent, on the other hand, only succeeded in stacking two blocks, and while it was less efficient and more computationally demanding than the curriculum agent, it did away with the need for manually designed transitions.

Automatically discovering curricula is an interesting area to explore for future research. The computation time in relational architectures is quadratic in the number of entities, which is also a source of concern. Scaling these methods to environments with much larger numbers of objects necessitates the development of computationally efficient methods. Finally, while the results presented are from state observation, we'd like to expand our system to include visual and other sensory observations in the future.

9 Acknowledgements

This work was completed while interning at InstaDeep. I'd like to express my gratitude to Thomas Pierrot, Arthur Flajolet and Nicolas Perrin, without whom none of this would have been possible.

Bibliography

Agrawal, P., Nair, A., Abbeel, P., Malik, J., and Levine, S. Learning to poke by poking: Experiential learning of intuitive physics. *NIPS*, 2016.

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs, 2018.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017. URL http://arxiv.org/abs/1707.01495.

Battaglia, P., Pascanu, R., Lai, M., Rezende, D. J., and kavukcuoglu, K. Interaction networks for learning about objects, relations and physics. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pp. 4509–4517, USA, 2016. Curran Associates Inc. ISBN 978-1-5108-3881-9. URL http://dl.acm.org/citation.cfm?id=3157382.3157601.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning, 2016.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pp. 41–48, New York, NY, USA, 2009a. ACM. ISBN 978-1-60558-516-1. doi: 10.1145/1553374.1553380. URL http://doi.acm.org/10.1145/1553374.1553380.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48. ACM, 2009b.

Brody, S., Alon, U., and Yahav, E. How attentive are graph attention networks?, 2021.

Chebotar, Y., Hausman, K., Zhang, M., Sukhatme, G., Schaal, S., and Levine, S. Combining model-based and model-free updates for trajectory-centric reinforcement learning. 03 2017.

Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*, 2018.

Donahue, J., Jia, Y., Vinyals, O., Hoffman, J., Zhang, N., Tzeng, E., and Darrell, T. Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pp. 647–655, 2014.

Elman, J. L. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. ISSN 0364-0213. doi: https://doi.org/10.1016/0364-0213(90)90002-E. URL https://www.sciencedirect.com/science/article/pii/036402139090002E.

Elman, J. L. Learning and development in neural networks: the importance of starting small. *Cognition*, 48(1):71–99, 1993. ISSN 0010-0277. doi: https://doi.org/10.1016/0010-0277(93)90058-4.

Florensa, C., Held, D., Wulfmeier, M., Zhang, M., and Abbeel, P. Reverse curriculum generation for reinforcement learning. *arXiv preprint arXiv:1707.05300*, 2017.

Geman, S., Bienenstock, E., and Doursat, R. Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58, 01 1992. doi: 10.1162/neco.1992.4.1.1.

Gori, M., Monfardini, G., and Scarselli, F. A new model for earning in raph domains. volume 2, pp. 729 – 734 vol. 2, 01 2005. ISBN 0-7803-9048-2. doi: 10.1109/IJCNN.2005.1555942.

Graves, A., Bellemare, M. G., Menick, J., Munos, R., and Kavukcuoglu, K. Automated curriculum learning for neural networks. *CoRR*, abs/1704.03003, 2017. URL http://arxiv.org/abs/1704.03003.

Griffiths, T., Chater, N., Kemp, C., Perfors, A., and Tenenbaum, J. Probabilistic models of cognition: Exploring representations and inductive biases. *Trends in cognitive sciences*, 14:357–64, 08 2010. doi: 10.1016/j.tics.2010.05.004.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018a. URL http://arxiv.org/abs/1801.01290.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018b. URL http://arxiv.org/abs/1801.01290.

Held, D., Geng, X., Florensa, C., and Abbeel, P. Automatic goal generation for reinforcement learning agents. *CoRR*, abs/1705.06366, 2017. URL http://arxiv.org/abs/1705.06366.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Hoffman, M., Shahriari, B., Aslanides, J., Barth-Maron, G., Behbahani, F., Norman, T., Abdolmaleki, A., Cassirer, A., Yang, F., Baumli, K., Henderson, S., Novikov, A., Colmenarejo, S. G., Cabi, S., Gülçehre, Ç., Paine, T. L., Cowie, A., Wang, Z., Piot, B., and de Freitas, N. Acme: A research framework for distributed reinforcement learning. *CoRR*, abs/2006.00979, 2020. URL https://arxiv.org/abs/2006.00979.

Justesen, N., Torrado, R. R., Bontrager, P., Khalifa, A., Togelius, J., and Risi, S. Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv preprint arXiv:1806.10729*, 2018.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4): 541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. End-to-end training of deep visuomotor policies. *CoRR*, abs/1504.00702, 2015. URL http://arxiv.org/abs/1504.00702.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. End-to-end training of deep visuomotor policies. *JMLR*, 2016.

Lopes, M., Lang, T., Toussaint, M., and Oudeyer, P.-Y. Exploration in model-based reinforcement learning by empirically estimating learning progress. In *NIPS*, 2012.

Mitchell, T. The need for biases in learning generalizations. 10 2002.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Ng, A., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. Inverted autonomous helicopter flight via reinforcement learning. *Proceedings of the International Symposium on Experimental Robotics*, 01 2004.

Ng, A. Y., Harada, D., and Russell, S. Policy invariance under reward transformations: Theory and application to reward shaping. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 278–287. Morgan Kaufmann, 1999.

Nichol, A., Achiam, J., and Schulman, J. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

Oudeyer, P.-Y. and Kaplan, F. What is intrinsic motivation? a typology of computational approaches. *Frontiers in neurorobotics*, 2009.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 16–17, 2017.

Peters, J. and Schaal, S. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21:682–, 06 2008. doi: 10.1016/j.neunet.2008.02.003.

Plappert, M., Andrychowicz, M., Ray, A., McGrew, B., Baker, B., Powell, G., Schneider, J., Tobin, J., Chociej, M., Welinder, P., Kumar, V., and Zaremba, W. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *CoRR*, abs/1802.09464, 2018. URL http://arxiv.org/abs/1802.09464.

Popov, I., Heess, N., Lillicrap, T. P., Hafner, R., Barth-Maron, G., Vecerík, M., Lampe, T., Tassa, Y., Erez, T., and Riedmiller, M. A. Data-efficient deep reinforcement learning for dexterous manipulation. *CoRR*, abs/1704.03073, 2017. URL http://arxiv.org/abs/1704.03073.

Raposo, D., Santoro, A., Barrett, D., Pascanu, R., Lillicrap, T., and Battaglia, P. Discovering objects and their relations from entangled scene representations, 2017a.

Raposo, D., Santoro, A., Barrett, D., Pascanu, R., Lillicrap, T., and Battaglia, P. Discovering objects and their relations from entangled scene representations, 2017b.

Ren, S., He, K., Girshick, R., and Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pp. 91–99, 2015.

Rumelhart, Mcclelland, J., and L, J. *Parallel distributed processing: explorations in the microstructure of cognition. Volume 1. Foundations.* 01 1986.

Schmidhuber, J. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, 2010.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a sat solver from single-bit supervision, 2019.

Sukhbaatar, S., Lin, Z., Kostrikov, I., Synnaeve, G., Szlam, A., and Fergus, R. Intrinsic motivation and automatic curricula via asymmetric self-play. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=SkT5Yg-RZ.

Sutton, R., Precup, D., and Singh, S. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

Van Der Malsburg, C. Frank rosenblatt: Principles of neurodynamics: Perceptrons and the theory of brain mechanisms. In Palm, G. and Aertsen, A. (eds.), *Brain Theory*, pp. 245–248, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg. ISBN 978-3-642-70911-1.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks, 2018.

Yoon, K., Liao, R., Xiong, Y., Zhang, L., Fetaya, E., Urtasun, R., Zemel, R. S., and Pitkow, X. Inference in probabilistic graphical models by graph neural networks. *CoRR*, abs/1803.07710, 2018. URL http://arxiv.org/abs/1803.07710.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Póczos, B., Salakhutdinov, R., and Smola, A. J. Deep sets. *CoRR*, abs/1703.06114, 2017. URL http://arxiv.org/abs/1703.06114.

Zaremba, W. and Sutskever, I. Learning to execute. *CoRR*, abs/1410.4615, 2014. URL http://arxiv.org/abs/1410.4615.

Zhang, C., Vinyals, O., Munos, R., and Bengio, S. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018.

A DeepSet Theorem (Proof of Theorems 1 and 2):

A.1 The invariant case

We give some remarks before providing the proof below. For the sake of simplicity, we consider here a fixed number of points n on the unit interval $[0,1]$;

$f:[0,1]\to\mathbb{R}$ a continuous invariant function. Lets find two continuous functions ρ and ϕ such that $f(x)=\rho(\sum(\phi(x_i))$

We start with a crucial result stating that: a set of n real points is characterized by the first n moments of its empirical measure. For $n=2$ it means that: we can recover the values of x_1 and x_2 from the quantities $p_1=x_1+x_2$ and $p_2=x_1^2+x_2^2$

Proposition 3. *Let* $\Phi:[0,1]^n_{\leq}\to\mathbb{R}^n$, *where* $[0,1]^n_{\leq}=\left\{\mathbf{x}\in[0,1]^n, x_1\leq x_2\leq\cdots\leq x_n\right\}$ *be defined by:*

$$\Phi(x_1,\ldots,x_n)=\left(\sum_i x_i,\sum_i x_i^2,\ldots,\sum_i x_i^n\right) \tag{33}$$

is injective and has a continuous inverse mapping.

The proof of this proposition follows from Newton's identities.

We are now ready to prove the theorem. Let $\phi:[0,1]\to\mathbb{R}^n$ be defined by $\phi(x)=(x,x^2,\ldots,x^n)$ and $\rho=f\circ\Phi^{-1}$. Note that $\rho:\mathrm{Im}(\Phi)\to\mathbb{R}$ and $\sum_i\phi(x_i)=\Phi(\mathbf{x}_{\leq})$ where $\mathbf{x}_{\leq}$x is the vector $\mathbf{x}$ with components sorted in non-decreasing order. Hence as soon as f is invariant, we have $f(\mathbf{x})=f(\mathbf{x}_{\leq})$. We only need to extend the function ρ from the domain $\mathrm{Im}(\Phi)$ to $\mathbb{R}^n$ in a continuous way. This can be done by considering the projection π on the compact $\mathrm{Im}(\Phi)$ and define $\rho(\mathbf{x})=f\circ\Phi^{-1}(\pi(\mathbf{x}))$.

A.2 The equivariant case

We show now that the equivariant case is not more difficult than the invariant case. Consider a permutation $\sigma\in\mathscr{S}_n$ such that $\sigma(1)=1$ so that $f(\sigma\star\mathbf{x})=\sigma\star f(\mathbf{x})$gives for the first component: $f_1(x_1,x_{\sigma(2)},\ldots,x_{\sigma(n)})=f_1(x_1,x_2,\ldots,x_n)$. For any x_1 the mapping $(x_2,\ldots,x_n)\mapsto f_1(x_1,x_2,\ldots,x_n)$ is invariant. Hence by the deepset theorem, we have $f_1(x_1,x_2,\ldots,x_n)=\rho\left(x_1,\sum_{i\neq1}\phi(x_i)\right)$

Now consider a permutation such that $\sigma(1)=k,\sigma(k)=1$ and $\sigma(i)=i$ for $i\neq1,k$i then we have $f_k(x_1,x_2,\ldots,x_n)=f_1(x_k,x_2\ldots,x_1,\ldots x_n)$ hence $f_k(x_1,x_2,\ldots,x_n)=\rho\left(x_k,\sum_{i\neq k}\phi(x_i)\right)$ and the equivariant case follows.

B Hyperparameters Tables

Table 2: GAT Architecture Hyperparameters.

Parameter	Setting
Embedding dimension	64
Number of message passing modules	3
Graph module weight sharing	False
Graph encoder layer dims	[64, 64, 64]
Multi-head Attention per message passing	[4, 4, 4]
Graph Attention Pooling	True
Decoder MLP hidden layers	[64, 64, 64]
Input normalization	None
Norm Layers	True
Activation function	Leaky ReLU
Activation function (attention)	tanh

Table 3: SAC+HER Hyperparameters

Parameter	Setting
Number of workers	13
Replay buffer max size	1e6
Optimizer	Adam
Learning rate	1e-3
Discount factor	.99
Batch size	2048
HER fraction of re-labelled goals	.8
SAC target entropy	4