A. Details on ACL baselines

In this section, we give details about our implementations of ACL methods, as well as their hyperparameters tuning.

A.1. Implementation details

Random We use as baseline a random teacher, which samples tasks using a uniform distribution over the task space.

ADR OpenAI et al. (2019) introduced *Automatic Domain Randomization* (ADR), an ACL method relying on the idea of *Domain Randomization* (Tobin et al., 2017; Peng et al., 2018). Instead of sampling tasks over the whole task space, ADR starts from a distribution centered on a single example easy for the student and progressively grows the distribution according to the learning agent's performance. Using this mechanism, it increases the difficulty of the tasks proposed to the student while still sampling in previously seen regions in order to try reducing potential forgetting.

This sampling distribution P_ϕ is parameterized by $\phi \in \mathbb{R}^{2d}$ (with d the number of dimensions of the task space). For each dimension, a lower and upper boundary are set $\phi = \{\phi_i^L, \phi_i^H\}_{i=1}^d$ allowing to sample uniformly on each dimension using these boundaries and obtain a task λ:

$$P_\phi(\lambda) = \prod_{i=1}^{d} U(\phi_i^L, \phi_i^H)$$

At the beginning, ϕ is centered on a single example (i.e. $\phi_i^L = \phi_i^H \ \forall i$). Then, at each episode, 1) ADR starts by sampling a new task $\lambda \sim P_\phi$. Following this, 2) ADR chooses with a probability p_b whether to modify λ in order to explore the task space or not. It thus samples a value ϵ uniformly in $[0; 1]$ and checks whether $\epsilon < p_b$. If this is not the case, ADR simply sends λ to the environment.

Otherwise, 3) ADR selects uniformly one of the dimensions of the task space, which we will call j as an example. Following this, 4) one of the two boundaries ϕ_j^L or ϕ_j^H is selected (50% chances for each boundary). Finally, 5) ADR replaces the j-th value of λ by the selected boundary and sends λ to the environment.

Moreover, ADR keeps a buffer D_i^L and D_i^H for each dimension i in the task space. Every time ϵ is greater than p_b and a value of λ is replaced by one of the selected boundary, ADR stores the episodic reward obtained at the end of the episode in the buffer associated to the selected boundary (e.g. the episodic reward is stored in D_k^L if the k-th value of lambda was replaced by ϕ_k^L).

Every time one of the buffers' size reaches m, the average $\overline{p}$ of episodic reward stored is calculated. Then, $\overline{p}$ is compared to two thresholds t_L and t_H (being hyperparameters of ADR) in order to know whether the boundary associated to the buffer must be reduced or increased.

As an example, let's say that D_k^L's size reached m, meaning that ϕ_k^L is the associated dimension (i.e. a λ sampled got its k-th value replaced by ϕ_k^L m times). Its average episodic reward $\overline{p}$ is calculated. It is first compared to t_L and, if $\overline{p} < t_l$, ϕ_k^L is increased by Δ (as ϕ_k^L is a lower boundary, this means that the task space is reduced). Similarly, if $\overline{p} > t_l$, ϕ_k^L is decreased by Δ (expanding the task space).

If instead of D_k^L we take D_k^H, our task space has to be expanded or reduced in the same way: if $\overline{p} < t_L$ then ϕ_k^H is reduced by Δ (as it is now an upper boundary of the task space) and if $\overline{p} > t_H$ then ϕ_k^H is increased by Δ. Finally, note that whenever one buffer's size reaches m, it is then emptied.

As no implementation was provided by the authors, we propose here an implementation being as close as possible to the algorithms given in OpenAI et al. (2019).

RIAC Proposed in Baranes & Oudeyer (2009), Robust Intelligent Adaptive Curiosity is based on the recursive splitting of the task space in hyperboxes, called regions. One region is split in two whenever a pre-defined number max_s of sampled tasks originate from the region. The split value is chosen such that there is maximal Learning Progress (LP) difference between the two regions, while maintaining a size min_d (i.e. a ratio of the size of the whole task space) for each region. The number of possible split to attempt is parameterized by n. We reuse the implementation and the value of the hyperparameters not mentioned here from Portelas et al. (2019). RIAC does not require expert knowledge.

Covar-GMM Covar-GMM was proposed in Moulin-Frier et al. (2014). As for RIAC, it does not require expert knowledge and is based on learning progress. The core idea of Covar-GMM is to fit a Gaussian Mixture Model (of maximum size max_k) every n episodes on recently sampled tasks *concatenated with both a time dimension and a competence dimension*. The Gaussian from which to sample a new task is then chosen proportionally to its respective learning progress, defined as the positive correlation between time and competence. Additionally, in order to preserve exploration, Covar-GMM has a probability r_p of sampling a task uniformly random instead of using one of its Gaussians. We use the implementation and hyperparameters from Portelas et al. (2019) which uses Absolute Learning Progress (ALP) instead of LP.

Moreover, as aforementioned in section 5, we modified the implementation to make it use expert knowledge (i.e. an initial distribution) when provided. Hence, instead of uniformly sampling tasks over the whole task space during the bootstrap phase at the beginning of training, Covar-GMM samples tasks from an initial Gaussian distribution of tasks provided by the expert.

ALP-GMM ALP-GMM is an ACL algorithm inspired from Covar-GMM, proposed in Portelas et al. (2019). Instead of relying on time competence correlation, which only allows to compute ALP over a single GMM fit, it computes a per-task ALP from the entire history of sampled tasks using a knn-based approach similar to those proposed in Forestier et al. (2017). Recent tasks are periodically used to fit a GMM on recently sampled tasks *concatenated with their respective ALP value*. The Gaussian from which to sample is then selected based on its mean ALP dimension. ALP-GMM does not require expert knowledge and has the same hyperparameters as Covar-GMM. We reused the implementation and hyperparameters (except max_k, n and r_p) provided by Portelas et al. (2019).

Additionally, as for Covar-GMM, we added the possibility to ALP-GMM to bootstrap tasks for an initial Gaussian distribution if the latter is provided, instead of uniformly bootstrapping tasks.

Goal-GAN Another teacher algorithm we included in this benchmark is called GoalGAN, and relies on the idea of sampling goals (i.e. states to reach in the environment) where the agent performs neither too well nor to badly, called *Goals Of Intermediate Difficulty* (GOID). However, as this goal generation introduces a curriculum in the agent's learning, one can see the goal selection process as a task selection process. We will thus call them tasks instead of goals in the following description. For sampling, Florensa et al. (2018) proposed to use a modified version of a *Generative Adversarial Network* (GAN) (Goodfellow et al., 2014) where the generator network is used to generate tasks for the student given a random noise, and the discriminator is trained to classify whether these tasks are of "intermediate difficulty". To define such an "intermediate difficulty", GoalGAN uses a binary reward signal defining whether the student succeeded in the proposed task. As our environments return scalar rewards, this implies a function interpreter hand-designed by an expert (in our case we set a threshold on the scalar reward, as explained in appendix C). For each task sampled, the teacher proposes it multiple times ($n_{rollouts}$) to the student and then calculates the average of successes obtained (lying in $[0; 1]$). Using a lower threshold R_{min} and an upper threshold R_{max}, GoalGAN calculates if the average lies in this interval of tasks neither too easy (with an average of successes very high) nor too hard (with an average of successes very low). If this is the case, this task is labelled as 1 for the discriminator (0 otherwise). This new task is then stored in a buffer (except if it already exists in the buffer a task at an euclidean distance smaller than ϵ from our new task). Every time a task has to be sampled, in order to prevent the GAN from forgetting previously seen GOIDs, the algorithm has the probability p_{old} of uniformly sampling from the buffer instead of using the GAN. Finally, the GAN is trained using the tasks previously sampled every n episodes.

Note that, in order to help the GAN to generate tasks in a feasible subspace of the task space at the beginning of training, GoalGAN can also pretrain its GAN using trivial tasks. In the original paper, as tasks are states, authors proposed to use the student to interact with the environment for a few steps, and use collected states as achievable tasks. However, in our case, this is not possible. We thus chose to reuse the same trick as the one in (Klink et al., 2020), that uses an initial Gaussian distribution to sample tasks and label them as positives (i.e. tasks of intermediate difficulty) in order to pretrain the GAN with them. See appendix C for the way we designed this initial distribution.

We reused and wrapped the version[5] of GoalGAN implemented by Klink et al. (2020), which is a slightly modified implementation of the original one made by Florensa et al. (2018). Our generator network takes an input that has the same number of dimensions as our task space, and uses two layers of 256 neurons with ReLU activation (and TanH activation for the last layer). Our discriminator uses two layers of 128 neurons. For ϵ, we used a distance of 10% on each dimension of the task space. As per Florensa et al. (2018), we set R_{min} to 0.25 and R_{max} to 0.75. Finally, as in the implementation made by

[5] https://github.com/psclklnk/spdl

Klink et al. (2020), we set the amount of noise δ added to each goal sampled by the generator network as a proportion of the size of the task space.

Self-Paced Proposed by Klink et al. (2020), *Self-Paced Deep Reinforcement Learning* (SPDL) samples tasks from a distribution that progressively moves towards a target distribution. The intuition behind it can be seen as similar to the one behind ADR, as the idea is to start from an initial task space and progressively shift it towards a target space, while adapting the pace to the agent's performance. However here, all task distributions (initial, current and target) are Gaussian distributions. SPDL thus maintains a current task distribution from which it samples tasks and changes it over training. This distribution shift is seen as an optimization problem using a dual objective maximizing the agent's performance over the current task space, while minimizing the Kullback-Leibler (KL) divergence between the current task distribution and the target task distribution. This forces the task selection function to propose tasks where the agent performs well while progressively going towards the target task space.

Initially designed for non-episodic RL setups, SPDL, unlike all our other teachers, receives information at every step of the student in the environment. After an offset of n_{OFFSET} first steps, and then every n_{STEP} steps, the algorithm estimates the expected return for the task sampled $\text{E}_{p(c)}[J(\pi, c)]$ using the value estimator function of the current student (with $p(c)$ the current task distribution, π the current policy of the student, and $J(\pi, c)$ the expected return for the task c with policy π).

With this, SPDL updates its current sampling distribution in order to maximize the following objective w.r.t. the current task distribution $p(c)$:

$$\max_{p(c)} \text{E}_{p(c)}[J(\pi, c)]$$

Additionally, a penalty term is added to this objective function, such that the KL divergence between $p(c)$ and the target distribution $\mu(c)$ is minimized. This penalty term is controlled by an α parameter automatically adjusted. This parameter is first set to 0 for K_α optimization steps and is then adjusted in order to maintain a constant proportion ζ between the KL divergence penalty and the expected reward term (see Klink et al. (2020) for more details on the way α is calculated). This optimization step is made such that the shift of distribution is not bigger than ϵ (i.e. $s.t. D_{\text{KL}}(p(c)||q(c)) \leq \epsilon$ with a shift from $p(c)$ to $q(c)$).

We reused the same implementation made by Klink et al. (2020) and wrapped it to our teacher architecture. However, as shown in section 5, using a Gaussian target distribution does not match with our Stump Tracks test set where tasks are uniformly sampled over the whole task space. In order to solve this issue, some adaptations to its architecture could be explored (e.g. using a truncated Gaussian as target distribution to get closer to a uniform distribution). While not provided yet in *TeachMyAgent*, we are currently working along with SPDL's authors on these modifications in order to show a fairer comparison of this promising method.

For the value estimators, we used the value network of both our PPO and SAC implementations (with the value network sharing its weights with the policy network for PPO). For the calculation of α, we chose to use the average reward, as in the experiments of Klink et al. (2020). We did not use the lower bound restriction on the standard deviation of the task distribution σ_{LB} proposed in Klink et al. (2020) as our target distributions were very large (see appendix C).

Setter-Solver Finally, the last ACL algorithm we implemented here is Setter-Solver (Racanière et al., 2020). In a very similar way to Goal-GAN, this method uses two neural networks: a *Judge* (replacing the discriminator) and a *Setter* (replacing the generator) outputting a task given a feasibility scalar in $[0; 1]$. During the training, the *Judge* is trained to output the right feasibility given a task sampled, and is used in the *Setter*'s losses to encourage the latter to sample tasks where the predicted feasibility was close to the real one. The *Setter* is also trained to sample tasks the student has succeeded (i.e. using a binary reward signal as Goal-GAN) while maximizing an entropy criterion encouraging it to sample diverse tasks.

For the implementation, Racanière et al. (2020) provided code to help reproducibility that implements both the *Setter* and *Judge*, but did not include neither losses nor optimization functions. Therefore, we provide here our own implementation of the full Setter-Solver algorithm trying to be as close as possible to the paper's details. We reused the code provided for the two neural networks and modified it to add losses, optimizers, and some modifications to better integrate it to our architecture. We kept the tricks added in the code provided by authors that uses a non-zero uniform function to sample the feasibility and a clipped sigmoid in the *Setter*'s output. Concerning the generator network, we kept the hyperparameters of the paper (i.e. a RNVP (Dinh et al., 2017) with three blocks of three layers) except the size of hidden layers n_{HIDDEN} that

we optimized. We also reused the three layers of 64 neurons architecture for the *Judge* as per the paper. Note that we used an Adam optimizer with a learning rate of $3 \cdot 10^{-4}$ for both the *Setter* and the *Judge*, while this was not precised for the *Judge* in Racanière et al. (2020). We optimized the upper bound δ of the uniformly sampled noise that is added to succeeded tasks in the validity *Setter*'s loss, as well as the update frequency n.

We did not use the conditioned version of the *Setter* or *Judge*. Indeed, first we generate the task before obtaining the first observation in our case as opposed to Racanière et al. (2020), and also because the first observation of an embodiment is always the same as both our environments have a startpad (see appendix B). Finally, we did not use the additional target distribution (called *desired goal distribution* in the original paper) loss that use a Wassertein discriminator (Arjovsky et al., 2017) to predict whether a task predicted belongs to the target distribution. Indeed, as shown in Racanière et al. (2020), using the targeted version of Setter-Solver offers more sample efficiency but leads to similar final results. Moreover, in our case, a target distribution is known only in the *High expert knowledge* setup of the challenge-specific experiments, in addition of having this part not implemented at all in the code provided by authors. We thus leave this upgrade to future work.

A.2. Hyperparameters tuning

In order to tune the different ACL methods to our experiments, we chose to perform a grid-search using our Stump Tracks environment with its original task space. As the Parkour is partly extended from it, in addition of the challenge-specific experiments, this environment offered us an appropriate setup. Each point sampled in the grid-search was trained for 7 million steps (instead of the 20 millions used in our experiments) with 16 seeds in order to reduce the (already high) computational cost. At the end of training, we calculated the percentage of mastered tasks on test set for each seed. The combination of hyperparameters having the best average over its seeds was chosen as the configuration for the benchmark.

In order to make the grid-search as fair as possible between the different ACL methods, given that the number of hyperparameters differs from one method to another, we sampled the same number of points for each teacher: 70 (± 10). The hyperparameters to tune for each teacher, as well as their values, were chosen following the recommendations given by their original paper.

Moreover, we chose to tune the teachers in what we call their "original" expert knowledge version (i.e. they have access to the same amount of prior knowledge as the one they used in their paper). Hence, teachers requiring expert knowledge use our high expert knowledge setup, and algorithms such as ALP-GMM use no expert knowledge.

Table 2 shows the values we tested for each hyperparameter and the combinations that obtained the best result.

Table 2. Hyperparameters tuning of the ACL methods.

ACL METHOD	HYPERPARAMETER	POSSIBLE VALUES	BEST VALUE
ADR	t_L	$[0, 50]$	0
ADR	t_H	$[180, 230, 280]$	180
ADR	p_b	$[0.3, 0.5, 0.7]$	0.7
ADR	m	$[10, 20]$	10
ADR	Δ	$[0.05, 0.1]$	0.1
RIAC	max_s	$[50, 150, 250, 350]$	150
RIAC	n	$[25, 50, 75, 100]$	75
RIAC	min_d	$[0.0677, 0.1, 0.1677, 0.2]$	0.1
COVAR-GMM	n	$[50, 150, 250, 350]$	150
COVAR-GMM	max_k	$[5, 10, 15, 20]$	15
COVAR-GMM	r_p	$[0.05, 0.1, 0.2, 0.3]$	0.1
ALP-GMM	n	$[50, 150, 250, 350]$	150
ALP-GMM	max_k	$[5, 10, 15, 20]$	10
ALP-GMM	r_p	$[0.05, 0.1, 0.2, 0.3]$	0.05
GOALGAN	δ	$[0.01, 0.05, 0.1]$	0.01
GOALGAN	n	$[100, 200, 300]$	100
GOALGAN	p_{old}	$[0.1, 0.2, 03]$	0.2
GOALGAN	$n_{rollouts}$	$[2, 5, 10]$	2
SPDL	n_{OFFSET}	$[100000, 200000]$	200000
SPDL	n_{STEP}	$[50000, 100000]$	100000
SPDL	K_α	$[0, 5, 10]$	0
SPDL	ζ	$[0.05, 0.25, 0.5]$	0.05
SPDL	ϵ	$[0.1, 0.8]$	0.8
SETTER-SOLVER	n	$[50, 100, 200, 300]$	100
SETTER-SOLVER	δ	$[0.005, 0.01, 0.05, 0.1]$	0.05
SETTER-SOLVER	n_{HIDDEN}	$[64, 128, 256, 512]$	128

B. Environment details

In this section, we give details about our two environments, their PCG algorithm, as well as some analysis about their task space. Note that our two environments follow the OpenAI Gym's interface and provide after each step, in addition of usual information (observation, reward, and whether the episode terminated), a binary value set to 1 if the cumulative reward of the episode reached 230. Additionally, we provide extra information and videos of our environments and embodiments, as well as policies learned at at `http://developmentalsystems.org/TeachMyAgent/`.

B.1. Stump Tracks

We present the Stump Tracks environment, an extended version of the environment introduced by Portelas et al. (2019). We only use two of the initially introduced dimensions of the procedural generation of task: stumps' height μ_s and spacing Δ_ss. As in Portelas et al. (2019), μ_s is used as the mean of a Gaussian distribution with standard deviation 0.1. Each stump has thus its height sampled from this Gaussian distribution and is placed at distance Δ_s from the previous one. We bound differently this task space depending on the experiment we perform, as explained in appendix C.

We kept the same observation space with 10 values indicating distance of the next object detected by lidars, head angle and velocity (linear and angular), as well as information from the embodiment (angle and speed of joints and also whether the lower limbs have contact with the ground). For information concerning the embodiment, the size of observation depends on the embodiment, as the number of joints varies (see below in B.3). We also kept the action space controlling joints with a torque.

B.2. Parkour

We introduce the Parkour, a Box2D parkour track inspired from the Stump Tracks and the environment introduced in Wang et al. (2020). It features different milieu in a complex task space.

B.2.1. Procedural generation

CPPN-encoded terrain First, similarly to the Stump Tracks, our Parkour features a ground (that has the same length as the one in Stump Tracks) where the agent starts at the leftmost side and has to reach the rightmost side. However, this ground is no longer flat and rather, as in Wang et al. (2020), generated using a function outputted by a neural network called CPPN (Stanley, 2007). This network takes in input a x position and outputs the associated y position of the ground. Using this, one can slide the CPPN over the possible x positions of the track in order to obtain the terrain. This method has the advantage of being able to easily generate non-linear and very diverse terrains as shown in Wang et al. (2020), while being light and fast to use as this only needs inference from the network. While CPPNs are usually used in an evolutionary setup where the architecture and weights are mutated, we chose here to rather initialize an arbitrary architecture and random weights and keep them fixed. For this architecture, we chose to use a four layers feedforward neural network with 64 units per layer and an alternation of TanH and Softplus activations (except for the output head which uses a linear activation) inspired from Ha (2016). Weights were sampled from a Gaussian distribution with mean 0 and standard deviation of 1. In addition of its x input, we added to our network three inputs that are set before generating the terrain as parameters controlling the generation. This vector θ of size 3 acts in a similar way as noise vector does in GANs for instance. Its size was chosen such that it allows to analyse the generation space and maintain the overall task space's number of dimensions quite small. As for the parameters in Stump Tracks, we bounded the space of values an ACL method could sample in θ. For this, we provide three hand-designed setups (easy, medium and hard) differing from the size of the resulting task space and the amount of feasible tasks in it (see appendix C.4).

Moreover, in addition of the y output of the ground, we added another output head y_c in order to create a ceiling in our tracks. As in Stump Tracks, the terrain starts with a flat startpad region (with a fixed distance between the ground and the ceiling) where the agent appears. Once $Y = (y_i)_{i \in X}$ and $Y_c = (y_{c_i})_{i \in X}$ generated by the CPPN, with X all the possible x positions in the track, we align them to their respective startpad:

$$y_i = y_i + startpad_g - y_0 \ \forall i \in Y$$

$$y_{c_i} = y_{c_i} + startpad_c - y_{c_0} \ \forall i \in Y_c$$

with $startpad_g$, $startpad_c$ being respectively the y position of the ground startpad and ceiling startpad, and y_0, y_{c_0} respectively the first y position of the ground and the ceiling outputted by our CPPN.

Using this non-linear generator (i.e. CPPN) allows us to have an input space where the difficulty landscape of the task space is rugged. Indeed, in addition of generating two non-linear functions for our ground and ceiling, the two latter can cross each other, creating unfeasible tasks (see figure 6). Additionally, our CPPN also makes the definition of prior knowledge over the input space more complex, as shown in figure 5.

Finally, as shown in figure 6, we smoothed the values of Y and Y_c by a parameter δ ($= 10$ in the training distribution) in order to make the roughness of the terrains adapted to our embodiments.

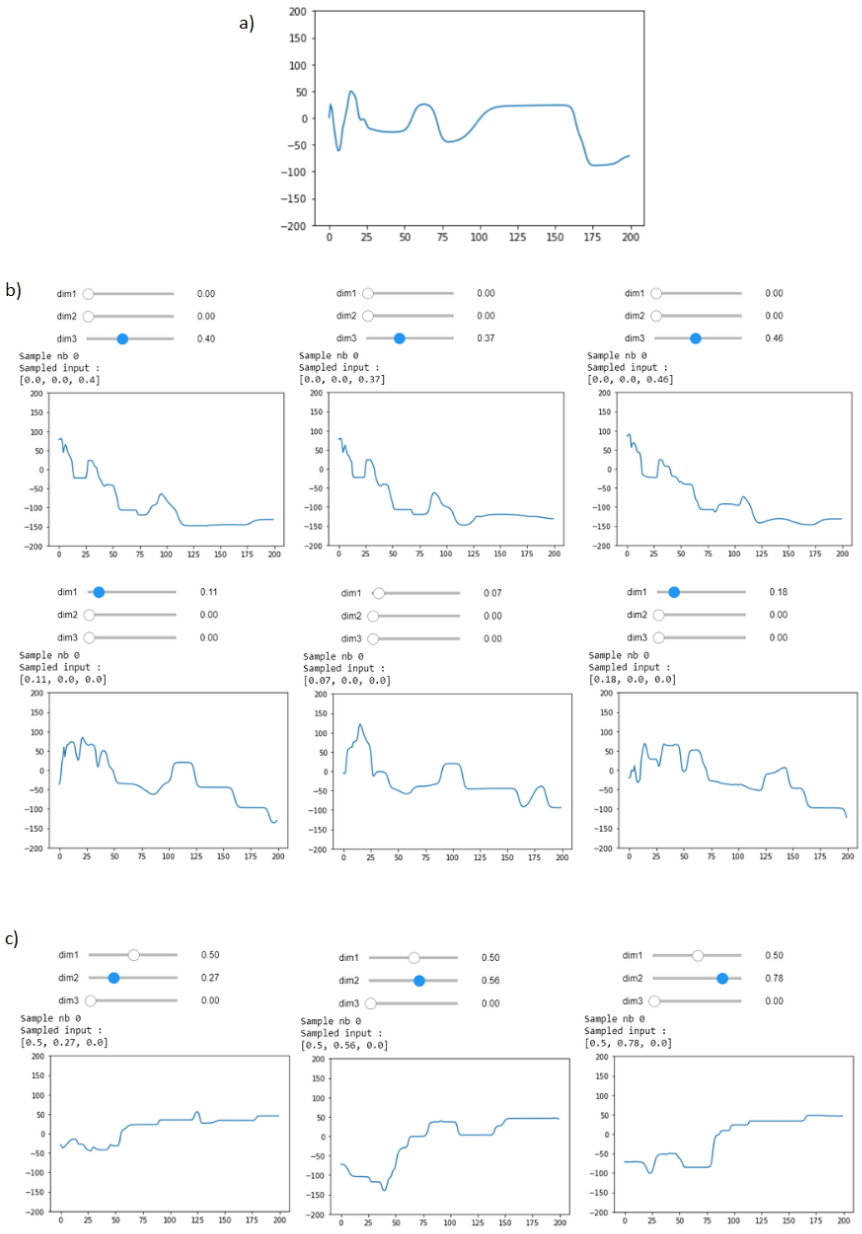

Figure 5. Overview of the input space of θ. First, in a) one can see the function generated when all the values of the input vector are set to zero. Secondly, in b) we can see that small changes over the space lead to similar functions and that big changes lead to very different results, showing that local similarity is maintained over the task space. Finally, c) shows how the difficulty landscape of θ can be rugged, as moving along the second dimension leads to terrains having a very different difficulty level.

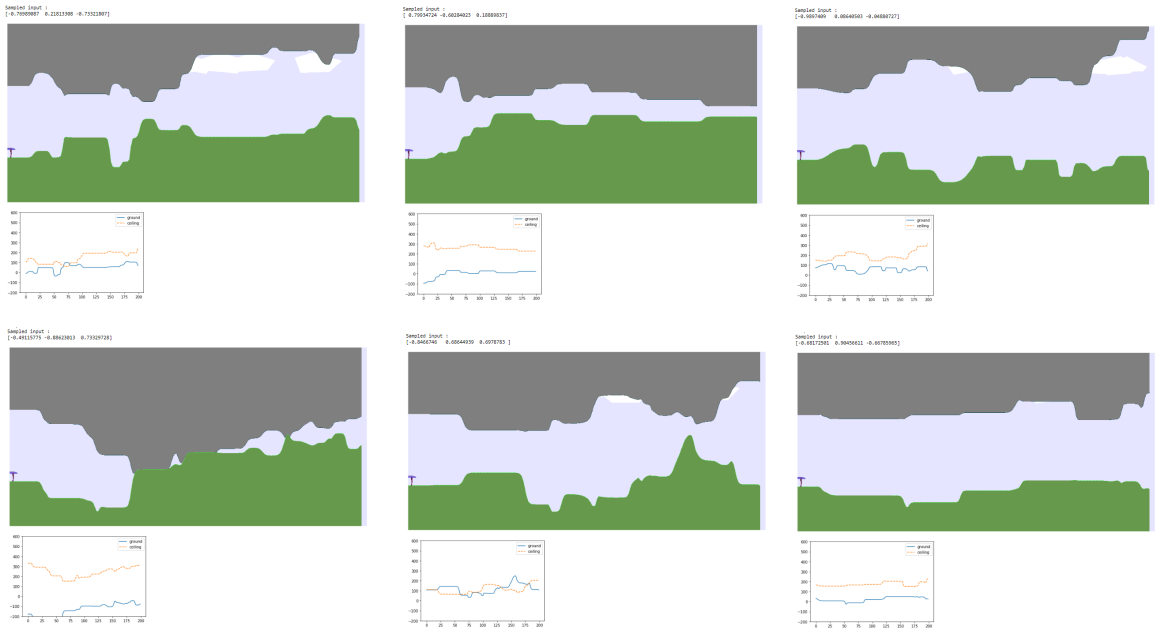

Figure 6. Here are some examples of generated tasks in the Parkour environment. While most of them seem too hard for a classic bipedal walker, the bottom left task is a good example of an unfeasible task, no matter which embodiment is used.

Creepers Once the terrain generated, we add what we call "creepers". Similarly to the stumps, we create objects at distance Δ_c from one another and of height sampled using a Gaussian distribution of mean μ_c and standard deviation 0.1 (the width can also be controlled but was fixed to 0.25 in our experiments). However, creepers are not obstacles for agents as stumps but rather graspable objects that embodiments can go through. Moreover, even though not used in our experiments, we provide the possibility to make creepers more realistic by dividing every creeper in multiple rectangles of height at most 1 linked with a rotating joint. As shown on our website , this creates creepers on which the climbers can swing.

Water Finally, we added a last dimension to our task space controlling the "water" level. Water is simulated using a rectangle object that the agent can go through and in which physics change (see below). This rectangle's width equals the terrain's width and its height is controlled by a parameter $\tau \in [0; 1]$ with 0 being an arbitrary lower limit the ground can reach and 1 the highest point of the current ceiling (generated by the CPPN for the current task).

B.2.2. PHYSICS

As previously mentioned, we introduced creepers and water along with new physics. First, in order to make our creepers graspable by the agents, we added sensors to the end of limb of certain embodiments (see section B.3 below). Every time one of these sensors enters in contact with a creeper, we look at the action in the action space of the agent that is associated to this sensor. If its value is greater than 0, we create a rotational joint between the sensor and the creeper at the contact point. As long as this action is greater than 0, the joint remains. As soon as the action goes negative or equals 0, we delete the joint (releasing the agent's limb from the creeper) and start watching again for contact. Note that, in order to better see whether the agent grasps a creeper, we color its sensors in red when a joint exists and in yellow otherwise (see our website). Additionally, in order to help the learning agent, we also make the ceiling graspable.

Secondly, concerning the water, we simulated a buoyancy force (inspired from Campbell (2013)) when an object enters water given its density compared to the water's density (set to 1). In addition, we implemented a "drag" and a "lift" force that simulate the resistance applied when an object moves in water and slows down the movement. Finally, we added a "push" force applied to an object having an angular velocity. This force simulates the fact that applying a torque to a rotational joint makes the object attached to the joint "push" the water and move (i.e. have a linear force applied). With these forces, we were able to simulate in a simplified way some physics of water, which resulted in very natural policies learned from our agents (see our website).

Finally, we simulated the fact that each embodiment is suited for one (or several) milieu, creating types of agents. Indeed, we first consider swimming agents that die (i.e. the actions sent by the DRL student to the environment no longer have effects on the motors of the embodiment) after spending more than 600 consecutive steps outside water. On the contrary, the two other types named climbers and walkers cannot survive underwater more than 600 consecutive steps. Both walkers and swimmers are allowed to have collisions with their body (including their head in the Parkour), whereas climbers are not allowed to touch the ground with any part of their body. Note that, while walkers appear with their legs touching the ground, swimmers appear a bit above the ground and climbers appear with all of their sensors attached to the ceiling (see figure 1).

All of these physics introduce the fact that an ACL teacher has to propose tasks in the right milieu for the current embodiment (i.e. mostly underwater for swimmers so that they do not die, with creepers and a ceiling high enough for climbers so that they do not touch the ground or die in water and with no water for walker so that they do not drown) in order to make the student learn.

B.2.3. Observation and action space

As in Stump Tracks, the agent is rewarded for moving forward and penalized for torque usage. An episode lasts 2000 steps unless the agent reaches the end of the track before or if a part of its body touches the ground if the embodiment is a climber. We also reused the 10 lidars per agent that were used in the Stump Tracks with all the lidars starting from the center of the head of the morphology. However, we modified them such that three configurations of covering exist (see the three tasks shown in figure 1):

- $90°$ from below the agent to ahead of it (used by walkers, as in Stump Tracks)
- $180°$ from below the agent to above it (used by swimmers)
- $90°$ from ahead of the agent to above it (used by climbers)

Moreover, in addition of the distance to the next object detected by each lidar, we added an information concerning the type of object detected by the lidar (-1 if water, 1 if creeper, 0 otherwise) such that the agent knows whether the object detected is an obstacle or can be passed through. Note also that once their origin point overlaps an object (e.g. water), lidars no longer detect it. Hence the lidars of an agent underwater no longer detect water (which would have made lidars useless as they would have only detected water). Therefore, in order to inform the DRL student whether the embodiment is underwater, we added an observation that is set to 1 if the agent's head is under the water level and 0 otherwise. Similarly, we added a binary observation telling whether the agent is dead or not (i.e. the actions we send to its motors no longer have impact). In addition, we kept the same information concerning the agent's head as in Stump Tracks (angle, linear velocity and angular velocity) as well as observations for each motor (angle and speed of joint as well as contact information for some of the attached limb). Finally, we added two binary observations per sensor (if the agent has sensors) telling whether the sensor has contact with a graspable surface and whether it is already attached with a joint. Without considering the information about motors and sensors which depend on the morphology, all of the information listed above create an observation vector of size 26. Note that, additionally, we provide the information to the teacher at each step whether the cumulative reward of the episode has reached 230 for the users using a binary reward.

Finally, for the action space, we kept the same behaviour as the one used in Stump Tracks (i.e. each agent has motors which are controlled through a torque value in $[-1; 1]$). Moreover, we added an action in $[-1; 1]$ per sensor for climbers to say whether this sensor must grasp (if it has contact with a graspable surface) or release.

B.3. Morphologies

We included in our benchmark the classic bipedal walker as well as its two modified versions introduced in Portelas et al. (2019): the short bipedal and the quadrupedal. For these three agents, we kept in their implementation the additional penalty for having an angle different than zero on their head, which was already in Portelas et al. (2019). Additionally, we created new walkers such as the spider or the millipede shown in figure 1. See our repository and website for the exhaustive list of embodiments we provide.

We introduce another type of morphologies: climbers. We propose two agents: a chimpanzee-like embodiment, as well as its simplified version without legs (reducing the action space to simplify the learning task). These two agents have two arms with two sensors at their extremity allowing them to grasp creepers or the ceiling.

Both walkers and climbers have a density of 1 on their legs and arms, and a density of 5 on their body and head, making them simply "sink" in water.

Finally, we created swimming morphologies with each of their body part having the same density as the water, making them in a zero-gravity setting when fully underwater. We propose a fish-like embodiment (see figure 1) with a fin and a tale that can wave its body to move (as well as moving its fin).

Note that we also included an amphibious bipedal walker allowed to survive both underwater and outside water. This gave interesting swimming policies as shown on our website (http://developmentalsystems.org/TeachMyAgent/).

C. Experimental details

In this section, we give details about the setups of our experiments.

C.1. DRL Students

We used the 0.1.1 version of OpenAI Spinningup's implementation of SAC that uses Tensorflow, as in Portelas et al. (2019). We modified it such that a teacher could set a task at each reset of the environment. We also kept the same hyperparameters as the ones used in Portelas et al. (2019):

- A two layers feedforward network with 400/300 units per hidden layer (ReLU activation) for both the value and policy network (using TanH activation on the output layer for the latter)
- An entropy coefficient of 0.005
- A learning rate of 0.001
- A mini-batch update every 10 steps using 1000 randomly sampled experiences from a buffer of size 2 millions

For PPO, we used OpenAI Baselines' (Tensorflow) implementation. We used the same two layers neural network as in SAC for the policy and value networks (which share weights). We modified the runner sampling trajectories from the environment in order to use a single synchronous runner instead of multiple asynchronous ones. We used the environment's wrappers proposed in the OpeanAI Baselines' implementation to clip the actions and normalize the observations and rewards. We added a test environment (as well as a teacher that sets tasks) to test the agent's performance every 500000 steps (as done with SAC). We also normalize the observations and rewards in this test environment using the same running average as the one used in the training environment, so that agent does not receive different information from both environments. We send to the teacher and monitor the original values of reward and observation sent by the environment instead of normalized ones. We set the λ factor of the Generalized Advantage Estimator to 0.95, the clipping parameter ϵ to 0.2 and the gradient clipping parameter to 0.5. Finally, we tuned the following hyperparameters using a grid-search on Stump Tracks for 10 millions steps with stumps' height and spacing respectively in $[0; 3]$ and $[0; 6]$:

- Size of experiences sampled between two updates: 2000
- Number of epochs per update: 5
- Learning rate: 0.0003
- Batch size: 1000
- Value function coefficient in loss: 0.5
- Entropy coefficient in loss: 0.0

Note that for both our DRL students, we used $\gamma = 0.99$.

C.2. General experimental setup

We call an experiment the repetition, using different seeds, of the training of a DRL student for 20 millions steps using tasks chosen at every reset of the environment by a selected ACL teacher. The seed is used to initialize the state of random generators used in the teacher, DRL student and environment. We provide to the teacher the bounds (i.e. a min and max value for each dimension) of the task space before starting the experiment. The DRL student then interacts with the environment and asks the ACL teacher to set the task (i.e. a vector controlling the procedural generation) at every reset of the environment. Once the episode ended, the teacher receives either the cumulative reward or a binary reward (set to 1 if the episodic reward is grater than 230) for GoalGAN and Setter-Solver. Teachers like SPDL can additionally access to the information sent by the environment at every step, allowing non-episodic ACL methods to run in our testbed.

Every 500000 steps of the DRL student in the environment, we test its performance on 100 predefined tasks (that we call test set). We monitor the episodic reward obtained on each of these tasks. We also monitor the average episodic

reward obtained on the tasks seen by the student during the last 500000 steps. We ask the teacher to sample 100 tasks every 250000 steps of the DRL student and store these tasks to monitor the evolution of the generated curriculum (see at `http://developmentalsystems.org/TeachMyAgent/`). For this sampling, we use the non-exploratory part of our teachers (e.g. ALP-GMM always samples from its GMM or ADR never sets one value to one of its bounds) and do not append these monitoring tasks to the buffers used by some teachers to avoid perturbing the teacher's process.

In our experiments we were able to run 8 seeds in parallel on a single Nvidia Tesla V100 GPU. In this setup, evaluating one ACL method requires approximately (based on ALP-GMM's wall-clocktime):

- 4608 gpu hours for all skill-specific experiments with 32 seeds.
- 168 gpu hours for the 48 seeded Parkour experiment.

Running both experiments would require 4776 gpu hours, or 48 hours on 100 Nvidia Tesla V100 GPUs. Users with smaller compute budgets could reduce the number of seeds (e.g. divide by 3) without strong statistical repercussions.

C.3. Stump Tracks variants

We used the Stump Tracks environment to create our challenge-specific comparison of the different ACL methods. We leveraged its two dimensional task space (stumps' height and spacing) to create experiments highlighting each of the 6 challenges listed in section 1. Each experiment used 32 seeds.

C.3.1. Test sets

We used the same test set in all our experiments on Stump Tracks in order to have common test setup to compare and analyse the performance of our different ACL methods. This test set is the same as the one used in Portelas et al. (2019) with 100 tasks evenly distributed over a task space with $\mu_s \in [0; 3]$ and $\Delta_s \in [0; 6]$.

C.3.2. Experiments

In the following paragraphs, we detail the setup of each of our experiments used in the challenge-specific comparison.

Expert knowledge setups We allow three different amounts of prior knowledge about the task to our ACL teachers:

- *No expert knowledge*
- *Low expert knowledge*
- *High expert knowledge*

First, in the *No expert knowledge* setup, no prior knowledge concerning the task is accessible. Hence, no reward mastery range (ADR, GOoalGAN and Setter-Solver) is given. Additionally, no prior knowledge concerning the task space like regions containing trivial tasks for the agent (e.g. for ADR or SPDL's initial distribution) or subspace containing the test tasks (e.g. for SPDL's target distribution) are known. However, we still provide these two distribution using the following method:

- *Initial distribution*: we sample the mean $\mu_{INITIAL}$ of a Gaussian distribution uniformly random over the task space. We choose the variance of each dimension such that the standard deviation over this dimension equals 10% of the range of the dimension (as done when expert knowledge is accessible).
- *Target distribution*: we provide a Gaussian distribution whose mean is set to the center of each dimension and standard deviation to one fourth of the range of each dimension (leading to more than 95% of the samples that lie between the min and max of each dimension). This choice of target distribution was made to get closer to our true test distribution (uniform over the whole task space), while maintaining most of the sampled tasks inside our bounds. However, it is clear that this target distribution is not close enough to our test distribution to make SPDL proposing a good curriculum and lead to an agent learning an efficient policy to perform well in our test set. As mentioned in section 5 and appendix A, using a Gaussian target distribution is not suited to our setup and would require modifications to make the target distribution match our true test distribution.

Hence in this setup, only ALP-GMM, RIAC, Covar-GMM and SPDL (even though its initial and target distribution do not give insightful prior knowledge) can run.

In the *Low expert knowledge* setup, we give access to reward mastery range. Therefore, GoalGAN, Setter-Solver and ADR can now enter in the comparison. The initial distribution is still randomly sampled as explained above. It is used by GoalGAN to pretrain its GAN at the beginning of the training process, but also by ADR which starts with a single example being $\mu_{INITIAL}$.

Finally, for the *High expert knowledge* setup, we give access to the information about regions of the task space. While the standard deviation of the initial distribution is still calculated in the same way (i.e. 10% of the range of each dimension), we set $\mu_{INITIAL}$ to $[0; 6]$, with the values being respectively μ_s and Δ_s. Hence, ADR now uses the task $[0; 6]$ as its initial task and GoalGAN pretrains its GAN with this distribution containing trivial tasks for the walking agent (as stumps are very small with a large spacing between them). SPDL also uses this new initial distribution, but keeps the same target distribution as we could not provide any distribution matching our real test distribution (i.e. uniform). Note that, as mentioned in appendix A, ALP-GMM and Covar-GMM use this initial distribution in their bootstrap phase in this setup.

Mostly unfeasible task space In this experiment, we use SAC with a classic bipedal walker. We consider stumps with height greater than 3 impossible to pass for a classic bipedal walker. Hence, in order to make most of the tasks in the task space unfeasible, we use in this experiment $\mu_s \in [0; 9]$ (and do not change $\Delta_s \in [0; 6]$) such that almost 80% of the tasks are unfeasible.

Mostly trivial task space Similarly, we use in this experiment $\mu_s \in [-3; 3]$ (the Stump Tracks environments clips the negative values with $\mu_s = \max(0, \mu_s)$). Hence 50% of the tasks in the task space will result in a Gaussian distribution used to generate stumps' height with mean 0. We also use SAC with a classic bipedal walker.

Forgetting students We simulate the catastrophic forgetting behaviour by resetting all the variables of the computational graph of our DRL student (SAC here) as well as its buffers every 7 millions steps (hence twice in a training of 20 millions steps). All variables (e.g. weights, optimizer's variables...) are reinitialized the same way they were before starting the training and the experience buffer used by SAC is emptied. Note that we also use the classic bipedal walker as embodiment and did not modify the initial task space ($\mu_s \in [0; 3]$ and $\Delta_s \in [0; 6]$).

Rugged difficulty landscape In order to create a rugged difficulty landscape over our task space, we cut it into 4 regions of same size and shuffle them (see algorithm 1). The teacher then samples tasks in the new task space using interpolation (see algorithm 2) which is now a discontinuous task space introducing peaks and cliffs in difficulty landscape. While the cut of regions is always the same, the shuffling process is seeded at each experiments.

Algorithm 1 Cutting and shuffling of the task space.

Input: Number of dimensions $\mathcal{D}$, bounds $(min_i)_{i\in[\mathcal{D}]}$ and $(max_i)_{i\in[\mathcal{D}]}$, number of cuts k
for $d \in [\mathcal{D}]$ **do**
  Initialise arrays $\mathcal{O}_d, \mathcal{S}_d$
  $size \leftarrow |max_d - min_d|/k$
  for $j \in [k]$ **do**
    Store pair $(min_d + j * size, min_d + (j + 1) * size)$ in $\mathcal{O}_d$ and $\mathcal{S}_d$
  end for
  Shuffle order of pairs in $\mathcal{S}_d$
end for

Algorithm 2 Interpolate sampled task in the shuffled task space.

Input: Number of dimensions $\mathcal{D}$, task vector $\mathcal{T}$, number of cuts k
Initialise the vector $\mathcal{I}$ of size $\mathcal{D}$
for $d \in [\mathcal{D}]$ **do**
  for $j \in [k]$ **do**
    Get pair o_j in $\mathcal{O}_d$
    Initialize l with the first element of o_j
    Initialize h with the second element of o_j
    if $l \leq \mathcal{T}_d \leq h$ **then**
      Get pair s_j in $\mathcal{S}_d$
      Get β as the interpolation of $\mathcal{T}_d$ from the interval o_j to the interval s_j
      Set $\mathcal{I}_d = \beta$
      End the loop
    end if
  end for
end for
return $\mathcal{I}$

Robustness to diverse students Finally, in order to highlight the robustness of an ACL teacher to diverse students, we perform 4 experiments (each with 32 seeds) and then aggregate results. We use the initial task space of Stump Tracks but use both PPO and SAC and two different embodiments:the short bipedal walker and the spider. Each embodiment is used both with PPO and SAC (hence 2 experiments per embodiment and thus a total of 4 experiments). We then aggregate the 128 seeds into a single experiment result.

C.4. Parkour experiments

We perform a single experiment in the Parkour environment using 48 seeds. Among these seeds, 16 use a classic bipedal walker, 16 a chimpanzee and 16 a fish embodiment. We set the bounds of the task space to the following:

- CPPN's input vector $\theta \in [-0.35, 0.05] \times [0.6, 1.0] \times [-0.1, 0.3]$
- Creepers' height $\mu_c \in [0; 4]$
- Creepers' spacing $\Delta_c \in [0; 5]$
- Water level $\tau \in [0; 1]$

Note that the above CPPN's input space is considered as our medium one. We also provide the easy space ($\theta \in [-0.25, -0.05] \times [0.8, 1.0] \times [0.0, 0.2]$) as well as the hard one ($\theta \in [-1, 1] \times [-1, 1] \times [-1, 1]$). Both the easy and medium spaces were designed from our hard task space. Their boundaries were searched such that the task space contains feasible tasks while maintaining diverse terrains. They differ in their ratio between feasible and unfeasible tasks.

C.4.1. Test sets

Unlike in the Stump Track experiments, we needed in the Parkour environment different test sets as our three embodiments (i.e. bipedal walker, chimpanzee, fish) are not meant to act and live in the same milieu (e.g. swimmers do not survive in tasks not containing water). Therefore, creating a test set composed of tasks uniformly sampled would not allow to assess the performance of the current embodiment. Hence, we made for the Parkour three different test sets, each constituted of 100 tasks. As the task space previously defined is composed of mostly unfeasible tasks for any embodiment, we hand-designed each of the three test sets with the aim of showcasing the abilities of each morphology type, as well as showing the ability of the learned policy to generalize. Each test set has 60 tasks that belong to the training task space and 40 out-of-distribution tasks (using tasks outside the medium CPPN's input space as well as smoothing values different than 10 for the δ parameter). They also share the same distribution between easy (1/3), medium (1/3) and hard (1/3) tasks. We chose each task such that it seems possible given the physical capacities of our embodiments. See figure 7 for some examples of the test tasks.

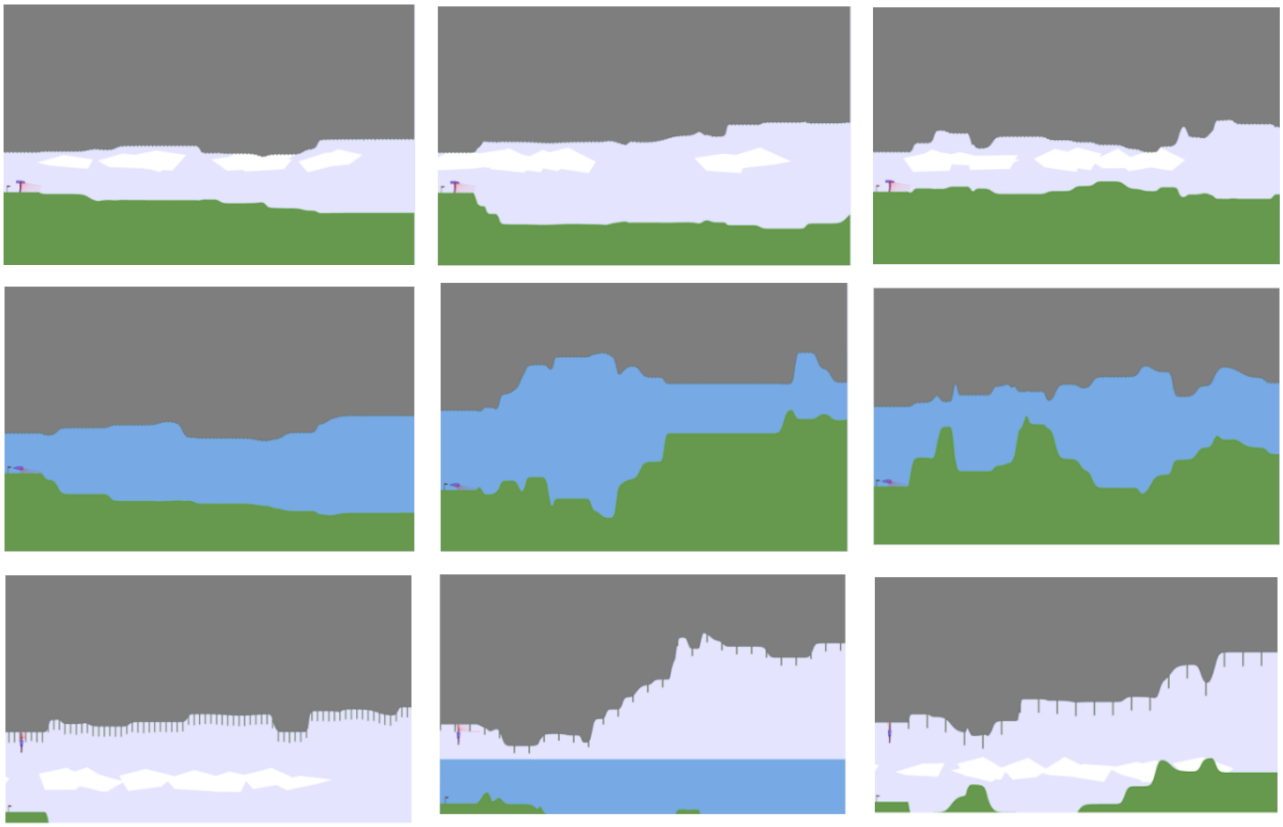

Figure 7. We show some examples of the tasks belonging to our Parkour's test sets. First line shows tasks from the walkers' test set, second line from the swimmers' one and finally last line for climbers.

D. Additional results

In this section, we provide additional results on experiments presented in section 5 as well as case studies. As mentioned in appendix C, we monitor both the episodic reward on each of the test tasks and the average episodic reward on training tasks every 500000 steps for each seed. We use the episodic reward on test tasks to calculate our percentage of "mastered" tasks metric, which calculates the percentage of tasks on which the agent obtained an episodic reward greater than 230. Additionally, we compare two algorithms in an experiment using Welch's t-test between their population of seeds.

D.1. Original Stump Tracks

We trained our SAC student for 20 millions steps on the original Stump Track task space (i.e. $\mu_s \in [0; 3]$ and $\Delta_s \in [0; 6]$) with each teacher and each expert knowledge setup. We used the best performance of each prior knowledge configuration as a baseline indication in figures 1 and 9. As in our challenge-specific experiments, we used 32 seeds as well as the same test set of 100 evenly distributed tasks. Results can be found in table 3.

Table 3. Percentage of mastered tasks after 20 millions steps on the original Stump Tracks challenge (i.e. $\mu_s \in [0; 3]$ and $\Delta_s \in [0; 6]$). Results shown are averages over 32 seeds along with the standard deviation. We highlight the best results in bold, which then acted as an upper baseline indication in the challenge-specific comparisons.

ALGORITHM	NO EK	LOW EK	HIGH EK
ADR	-	24.1 (± 20.8)	43.4 (± 7.2)
ALP-GMM	**52.1** (± 5.9)	**47.1** (± 13.9)	49.3 (± 5.9)
COVAR-GMM	43.0 (± 9.1)	40.25 (± 16.5)	45.2(± 10.1)
GOALGAN	-	29.9 (± 26.2)	**51.9** (± 7.3)
RIAC	40.5 (± 8.4)	39.6 ((± 11.2)	42.2 (± 5.4)
SPDL	20.8 (± 19.4)	18.5 (± 20.8)	34.0 (± 10.6)
SETTER-SOLVER	25.3 (± 10.7)	36.6(± 10.2)	37.4 (± 9.8)

D.2. Challenge-specific comparison

D.2.1. OVERALL RESULTS

We here show the performance after 20 millions steps of each ACL teacher on each challenge. Results are gathered in tables 4, 5 and 6, as well as in figure 8 where we show the results of Welch's t-test between all methods on every challenge.

Table 4. Percentage of mastered tasks after 20 millions steps with **no** prior knowledge in each challenge. Results shown are averages over all seeds along with the standard deviation. We highlight the best results in bold.

ALGORITHM	MOSTLY UNF.	MOSTLY TRIV.	FORGETTING STUD.	RUGGED DIF.	DIVERSE STUD.
RANDOM	18.0 (± 10.5)	22.2 (± 15.2)	27.8 (± 14.6)	30.3 (± 7.7)	22.3 (± 11.5)
ALP-GMM	**42.8** (± 6.6)	**43.7** (± 6.0)	**42.1** (± 6.9)	**42.5** (± 4.8)	31.5 (± 9.2)
COVAR-GMM	39.0 (± 9.9)	32.7 (± 16.0)	31.3 (± 16.2)	39.4 (± 7.4)	**32.3** (± 10.6)
RIAC	22.1 (± 14.5)	20.0 (± 10.9)	36.8 (± 6.9)	36.4 (± 7.9)	25.9 (± 11.3)
SPDL	6.4 (± 10.2)	15.3 (± 9.9)	10.4 (± 12.9)	19.3 (± 16.2)	8.9 (± 14.4)

Table 5. Percentage of mastered tasks after 20 millions steps with **low** prior knowledge in each challenge. Results shown are averages over all seeds along with the standard deviation. We highlight the best results in bold.

ALGORITHM	MOSTLY UNF.	MOSTLY TRIV.	FORGETTING STUD.	RUGGED DIF.	DIVERSE STUD.
RANDOM	18.0 (± 10.1)	18.0 (± 7.1)	27.8 (± 14.6)	30.3 (± 7.7)	22.3 (± 11.5)
ADR	7.8 (± 17.9)	22.2 (± 15.2)	21.2 (± 21.2)	17.0 (± 19.6)	15.6 (± 19.1)
ALP-GMM	**43.5** (± 13.0)	**43.0** (± 9.0)	**41.6** (± 12.5)	**44.2** (± 7.1)	31.3 (± 9.4)
COVAR-GMM	31.2 (± 16.8)	42.0 (± 8.4)	31.5 (± 18.4)	34.3 (± 10.7)	**32.1** (± 9.6)
GOALGAN	12.7 (± 16.2)	38.4 (± 16.1)	9.3 (± 15.8)	34.7 (± 19.1)	16.2 (± 17.5)
RIAC	20.5 (± 14.0)	21.3 (± 8.8)	34.3 (± 12.5)	38.3 (± 11.3)	26.0 (± 11.7)
SPDL	6.7 (± 10.2)	17.9 (± 12.2)	10.6 (± 12.2)	18.1 (± 15.8)	9.2 (± 14.2)
SETTER-SOLVER	25.3 (± 10.7)	35.5 (± 8.9)	33.9 (± 12.5)	31.6 (± 11.3)	25.4 (± 9.0)

Table 6. Percentage of mastered tasks after 20 millions steps with **high** prior knowledge in each challenge. Results shown are averages over all seeds along with the standard deviation. We highlight the best results in bold.

ALGORITHM	MOSTLY UNF.	MOSTLY TRIV.	FORGETTING STUD.	RUGGED DIF.	DIVERSE STUD.
RANDOM	18.0 (± 10.1)	18.0 (± 7.1)	27.8 (± 14.6)	30.3 (± 7.7)	22.3 (± 11.5)
ADR	45.3 (± 6.7)	32.5 (± 6.2)	39.8 (± 10.8)	17 (± 20.9)	32.3 (± 9.7)
ALP-GMM	**48.4** (± 11.2)	44.3 (± 14.2)	**43.0** (± 9.0)	**42.5** (± 7.3)	29.8 (± 8.8)
COVAR-GMM	38.2 (± 11.9)	39.6 (± 10.3)	39.5 (± 12.5)	41.3 (± 7.0)	**32.6** (± 10.2)
GOALGAN	39.7 (± 10.1)	**45.6** (± 13.5)	23.4 (± 19.7)	41.2 (± 12.6)	27.5 (± 9.4)
RIAC	25.2 (± 12.3)	22.1 (± 11.1)	37.7 (± 12.5)	37.7 (± 8.8)	25.8 (± 11.7)
SPDL	19.1 (± 12.5)	22.9 (± 6.9)	12.9 (± 11.2)	31.0 (± 11.2)	15.4 (± 15.1)
SETTER-SOLVER	28.2 (± 9.7)	33.7 (± 10.8)	37.4 (± 8.7)	34.7 (± 8.1)	24.0 (± 9.8)

Figure 8. Performance of the different teachers at the end of training in every experiment of our challenge-specific comparison. We plot as bars the average percentage of mastered tasks for each ACL method. Additionally, we compare in every experiment all possible couples of teacher methods using Welch's t-test and annotate the significantly different ($p < 0.05$) ones.

D.2.2. CASE STUDY: SAMPLE EFFICIENCY

In this section, we take a look at the sample efficiency of the different ACL methods using their performance after only 5 millions steps. We reuse the same radar chart as in section 5 in figure 9.

Looking at results, one can see the impact of ACL in the mostly unfeasible challenge, as some methods (e.g. ALP-GMM or ADR with high expert knowledge) already reach twice the performance of random after only 5 million steps. This highlights how leveraging a curriculum adapted to the student's capabilities is key when most tasks are unfeasible. On the opposite, when the task space is easier (as in the mostly trivial challenge), Random samples more tasks suited for the current student's abilities and the impact of Curriculum Learning is diminished.

Having the difficulty landscape rugged makes the search for learnable and adapted subspaces harder. Figure 9 shows that only 5 millions steps is not enough, even for teachers like ALP-GMM or Covar-GMM theoretically more suited for rugged difficulty landscapes, to explore and leverage regions with high learning progress.

Finally, one can see the strong impact of a well set initial distribution of tasks in the beginning of learning. Indeed, both ADR and GoalGAN already almost reach their final performance (i.e. the one they reached after 20 millions steps shown in figure 1) after 5 millions steps in the *High expert knowledge* setup, as they know where to focus and do not need exploration to find feasible subspaces. Similarly, adding expert knowledge to ALP-GMM increases its performance compared to the no and low expert knowledge setups, helping it focus the bootstrapping process on a feasible region. Leveraging this initial task distribution, GoalGAN obtains the best results in 3/5 challenges after 5 millions steps with high expert knowledge. This shows, in addition of the results from section 5, that GoalGAN is a very competitive method, especially when it has access to high expert knowledge.

Figure 9. **Performance of ACL methods measured after 5 millions steps.** Results are presented as an order of magnitude of the performance of Random. Performance is defined as the average percentage of mastered test tasks over all 32 seeds. We also provide the same indications (✚) of the best performance (measured after 5 millions steps) on the original Stump Tracks experiment as in figure 1. Finally, we indicate on each axis which method performed significantly better than Random ($p < 0.05$) using colored stars matching each method's color (e.g. ★ for Covar-GMM, ★ for ADR). *EK: Expert Knowledge.*

D.2.3. Case study: On the difficulty of GoalGAN and SPDL to adapt the curriculum to forgetting students

As mentioned in section 5, both GoalGAN and SPDL struggled on the forgetting student challenge, no matter the amount of expert knowledge. In order to better understand their behaviour in this challenge, we plot in figure 10 both the evolution of their percentage of mastered tasks and their average training return. We also add ALP-GMM and ADR (two students that performed well in this challenge) as baselines for comparison. While ADR and ALP-GMM make the student quickly recover from a reset (i.e. the percentage of mastered tasks quickly reaches the performance it had before the reset) and then carry on improving, both GoalGAN and SPDL suffer from resets and do not manage to recover, leading to a poor final performance.

This phenomenon could be explained by multiple factors. First, in the case of SPDL, even though the algorithm tries to shift its sampling distribution such that it maximizes the student's performance, the optimization methods also has to minimize the distance to the target distribution, which is a Gaussian spanning over the entire task space. However, resetting the student's policy requires the ACL method to revert back to the initial simple task distribution that it proposed at the beginning of training. Such a reverse process might not easily be achievable by SPDL, which optimization procedure progressively shifts its sampling distribution towards the target one.

Concerning GoalGAN, the performance impact of student resets is most likely due to its use of a buffer of "Goals of Intermediate Difficulty", used to train the goal generator. Upon student reset, this buffer becomes partially obsolete, as the student is reinitialized, making it lose all learned walking gaits, i.e. everything must be learned again. This means the goal generator will propose tasks that are too complex for a student that is just starting to learn. Because GoalGAN cannot reset its buffer of "Goals of Intermediate Difficulty" (which would require knowledge over the student's internal state), it impairs its ability to quickly shift to simpler task subspaces.

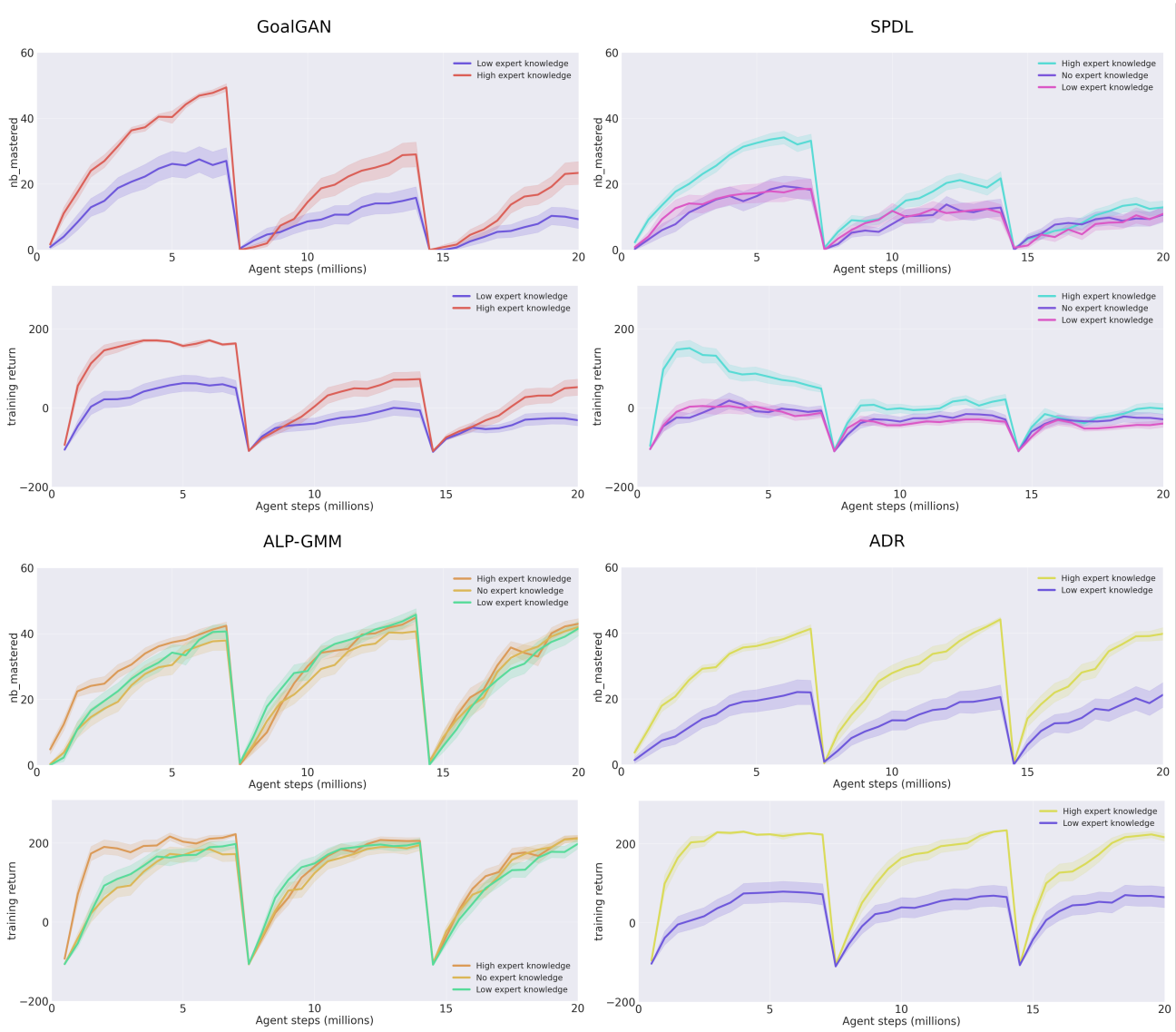

Figure 10. Percentage of mastered test tasks and average training return of GoalGAN, SPDL, ALP-GMM and ADR on the forgetting student challenge. Curves are averages over 32 seeds along with the standard error of the mean.

D.2.4. Case study: Impact of expert knowledge on ALP-GMM

As aforementioned, ALP-GMM is a method initially not requiring any expert knowledge. Moreover, it relies on an exploration (bootstrap) phase to fill its buffer, usually using uniform sampling over the task space. In *TeachMyAgent*, we provide an extended version of it where we added the possibility to use expert knowledge by bootstrapping from an initial distribution instead of a uniform distribution. In this case study, we take a look at the impact such a theoretical improvement

had on their performance. We focus on the mostly unfeasible and forgetting student challenges, as the first highlighted the most how prior knowledge can help an ACL method (helping it start in a feasible region) and the latter showed interesting results for this case study, in addition of being easy to analyse (as it only uses a bipedal walker on the original task space of the Stump Tracks). We gather these results in figure 11. Note that both the no and low expert knowledge setups are the same for ALP-GMM , meaning that any difference between their results is only due to variance in both the student's learning and ACL process.

When looking at these results, one can see that the high expert knowledge setup is significantly better than the two other setups at the beginning of the training in both challenges. These results can also be completed by our sample efficiency case study (see figure 9), showing that adding expert knowledge to ALP-GMM makes it more sample efficient. Then, as training advances, the difference becomes statistically insignificant ($p > 0.05$). Finally, while the final results given in tables 4, 5, and 6 show an improved percentage of mastered tasks in almost all challenges, with a notable difference (at least $+5$) on the mostly unfeasible challenge, results on the original Stump Tracks experiments (table 3) show better results with no expert knowledge. It is thus not clear whether adding this prior knowledge to ALP-GMM benefits the whole training instead of just the beginning. Note that similar behaviours were also obtained with Covar-GMM, even though they were not as significant as the ones of ALP-GMM.

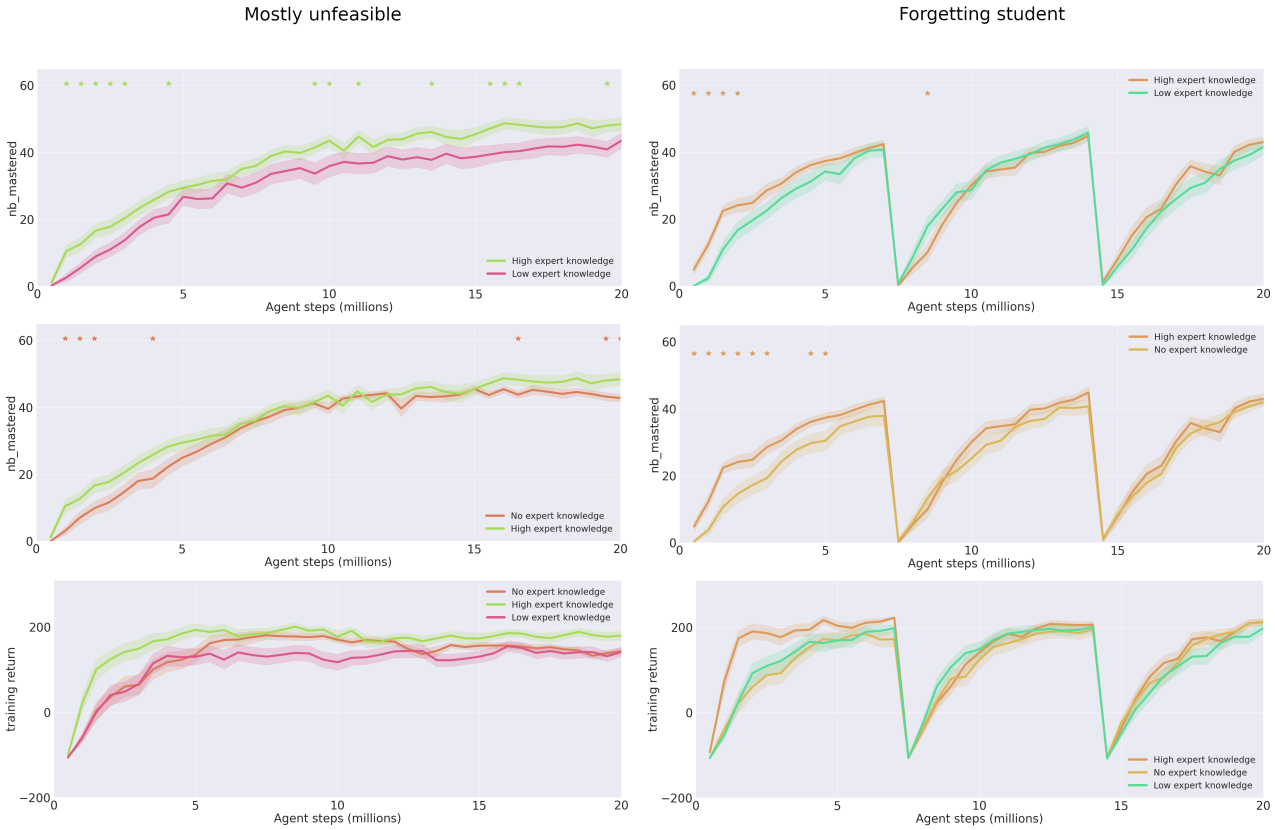

Figure 11. Percentage of mastered test tasks and average training return of ALP-GMM on both the mostly unfeasible and the forgetting student challenges. Curves are averages over 32 seeds along with the standard error of the mean. We compare the impact of high expert knowledge compared to the two other setups using Welch's t-test and highlight significant ($p < 0.05$) differences with stars.)

D.2.5. Case study: What ADR needs

ADR is a very efficient and light method, that, when all its expert knowledge requirements are satisfied, competes with the best teachers. However, in order to obtain such an efficient behaviour, ADR needs certain conditions that are implied by its

construction. First, as explained in appendix A, ADR starts its process using a single task, and makes the assumption that this latter is easy enough for the freshly initialized student. It then progressively grows its sampling distribution around this task if the student manages to "master" the proposed tasks. While this behaviour seems close to SPDL's, ADR does not have any target distribution to help it shift the distribution even if the student's performance are not good enough. Hence, ADR can get completely stuck if it is initialized on a task lying in a very hard region, whereas SPDL would still try to converge to the target distribution (even though the performance would not be as good as if its initial distribution was set in an easy subspace). Similarly, GoalGAN also uses an initial distribution at the beginning of the training which, as shown in the results, has a strong impact on the final performance. However, even without it, GoalGAN is still able to reach a decent performance in certain challenges (e.g. mostly trivial) unlike ADR. This observation can also be seen in the Parkour's experiments, where GoalGAN reaches the top 4 while ADR obtains the worst performance. In order to highlight this explanation, we provide the results of ADR in the mostly unfeasible and mostly trivial challenges in figure 12, in addition of the clear difference between expert knowledge setups showed by figure 1 and tables 4, 5, and 6. Using figure 12, one can see the clear and significant ($p < 0.05$) difference between the two expert knowledge setups.

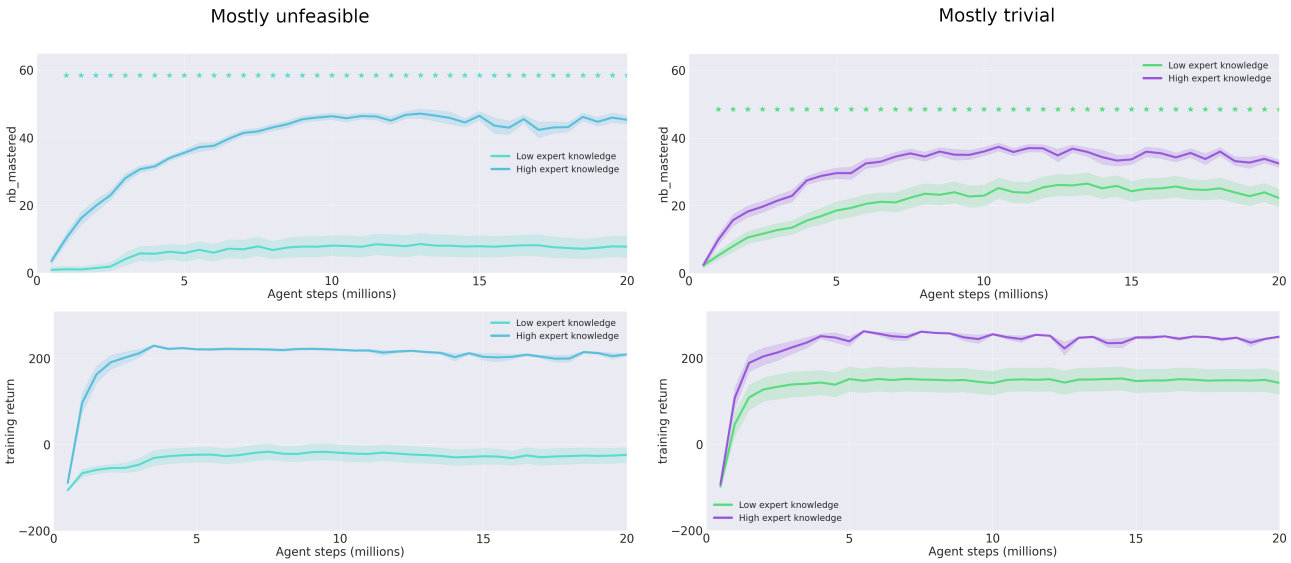

Figure 12. Percentage of mastered test tasks and average training return of ADR on both the mostly unfeasible and mostly trivial challenges. Curves are averages over 32 seeds along with the standard error of the mean. We compare the impact of high to low expert knowledge using Welch's t-test ($p < 0.05$) and highlight significant differences with stars.)

In addition of an initial task well set using prior knowledge about the task space, ADR needs a difficulty landscape not too rugged to be able to expand and reach regions with high learning progress for the student. Indeed, when looking at its algorithm, one can see that the sampling distribution grows in a certain direction only if the student is able to master the tasks proposed at the edge of the distribution on this direction. If it is not the case (i.e. if this region of the task space is too hard for the current student's capabilities), the sampling distribution will shrink. This simple mechanism makes the strong assumption that if the difficulty is too hard at one edge of the distribution, there is no need to go further, implicitly saying that the difficulty further is at least as hard as the one at the edge. While this works well in the vanilla task space of our Stump Tracks environment (our difficulty is clearly smooth and even monotonically increasing), any task space with a rugged difficulty landscape would make the problem harder for ADR. Indeed, as it reaches a valley in the difficulty landscape surrounded by hills of unfeasible (or too hard for the current student's abilities) tasks, ADR can get stuck. In order to highlight this behaviour, we use our rugged difficulty landscape challenge, where we created a discontinuous difficulty landscape where unfeasible regions can lie in the middle of the task space. Figure 13 shows how ADR is unable to solve this challenge, no matter the amount of expert knowledge it uses, leading to the worst performance of our benchmark (significantly worse than Random at $p < 0.05$). Note that this issue also happens in our Parkour experiments, as the difficulty of the task space is very rugged (see section 5).

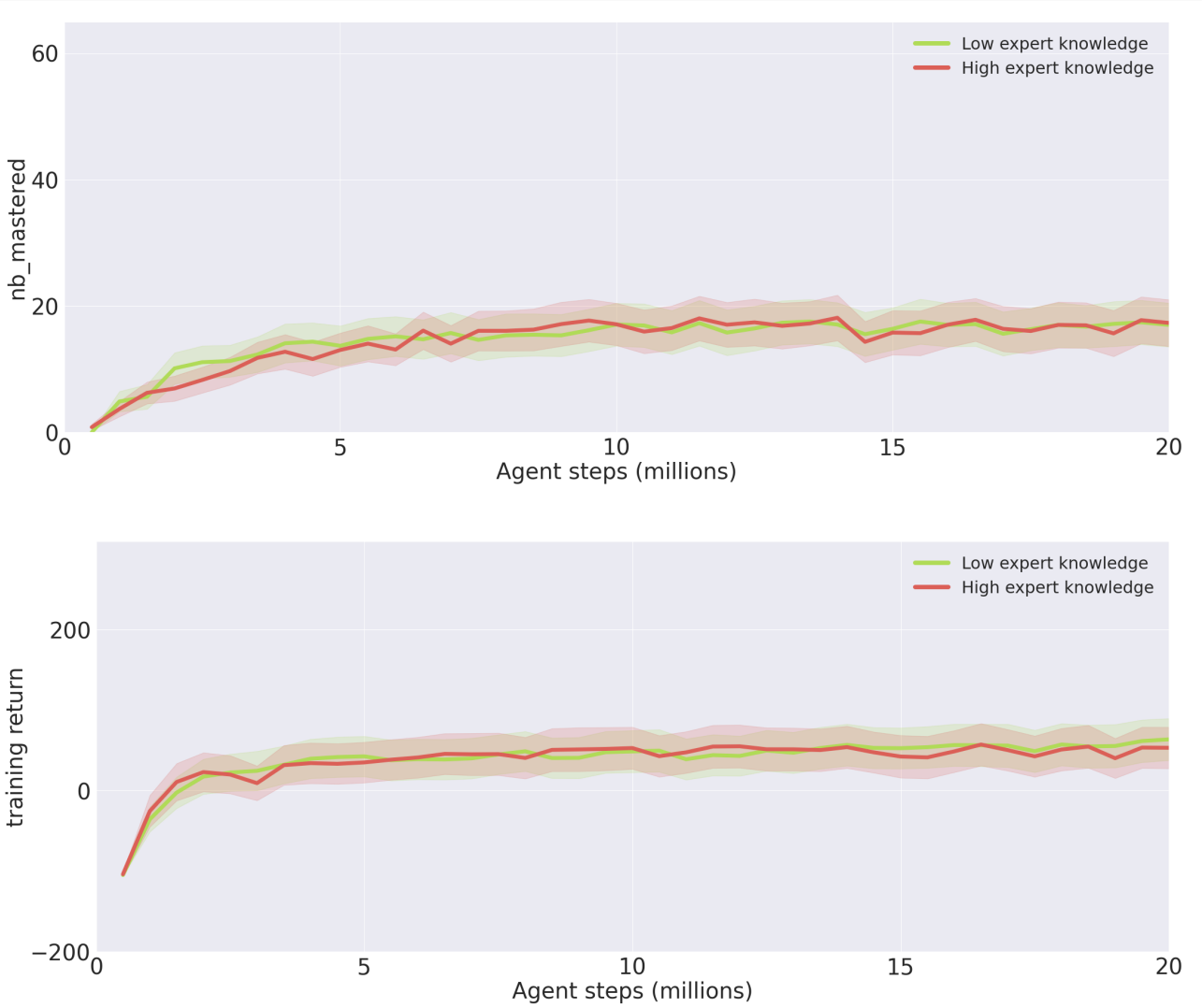

Figure 13. Percentage of mastered test tasks and average training return of ADR on rugged difficulty landscape challenge. Curves are averages over 32 seeds along with the standard error of the mean.

D.3. Parkour

D.3.1. Overall results

In this section, we present the performance of our teacher algorithms on the Parkour track experiments. We present the final results in table 7 as well as a comparison in figure 14 using Welch's t-test. We also provide insights concerning the obtained policies at `http://developmentalsystems.org/TeachMyAgent/`. When looking at the overall results, one can see that ALP-GMM is the only teacher performing significantly better than Random throughout training. Covar-GMM's performance are very close to ALP-GMM, as well as RIAC, which obtains very similar results to GoalGAN. While being very close to Random, Setter-Solver's results are not significantly different from ALP-GMM's results by the end of the training. Finally, while SPDL's behavior is very similar to Random, ADR reaches a plateau very soon and eventually obtains significantly worse results than the random teacher.

Table 7. Percentage of mastered tasks after 20 millions steps on the Parkour track. Results shown are averages over 16 seeds along with the standard deviation for each morphology as well as the aggregation of the 48 seeds in the overall column. We highlight the best results in bold.

ALGORITHM	BIPEDALWALKER	FISH	CLIMBER	OVERALL
RANDOM	27.25 (± 10.7)	23.6 (± 21.3)	0.0 (± 0.0)	16.9 (± 18.3)
ADR	14.7 (± 19.4)	5.3 (± 20.6)	0.0 (± 0.0)	6.7 (± 17.4)
ALP-GMM	**42.7** (± 11.2)	36.1 (± 28.5)	0.4 (± 1.2)	**26.4** (± 25.7)
COVAR-GMM	35.7 (± 15.9)	29.9 (± 27.9)	0.5 (± 1.9)	22.1 (± 24.2)
GOALGAN	25.4 (± 24.7)	34.7 (± 37.0)	0.8 (± 2.7)	20.3 (± 29.5)
RIAC	31.2 (± 8.2)	**37.4** (± 25.4)	0.4 (± 1.4)	23.0 (± 22.4)
SPDL	30.6 (± 22.8)	9.0 (± 24.2)	**1.0** (± 3.4)	13.5 (± 23.0)
SETTER-SOLVER	28.75 (± 20.7)	5.1 (± 7.6)	0.0 (± 0.0)	11.3 (± 17.9)

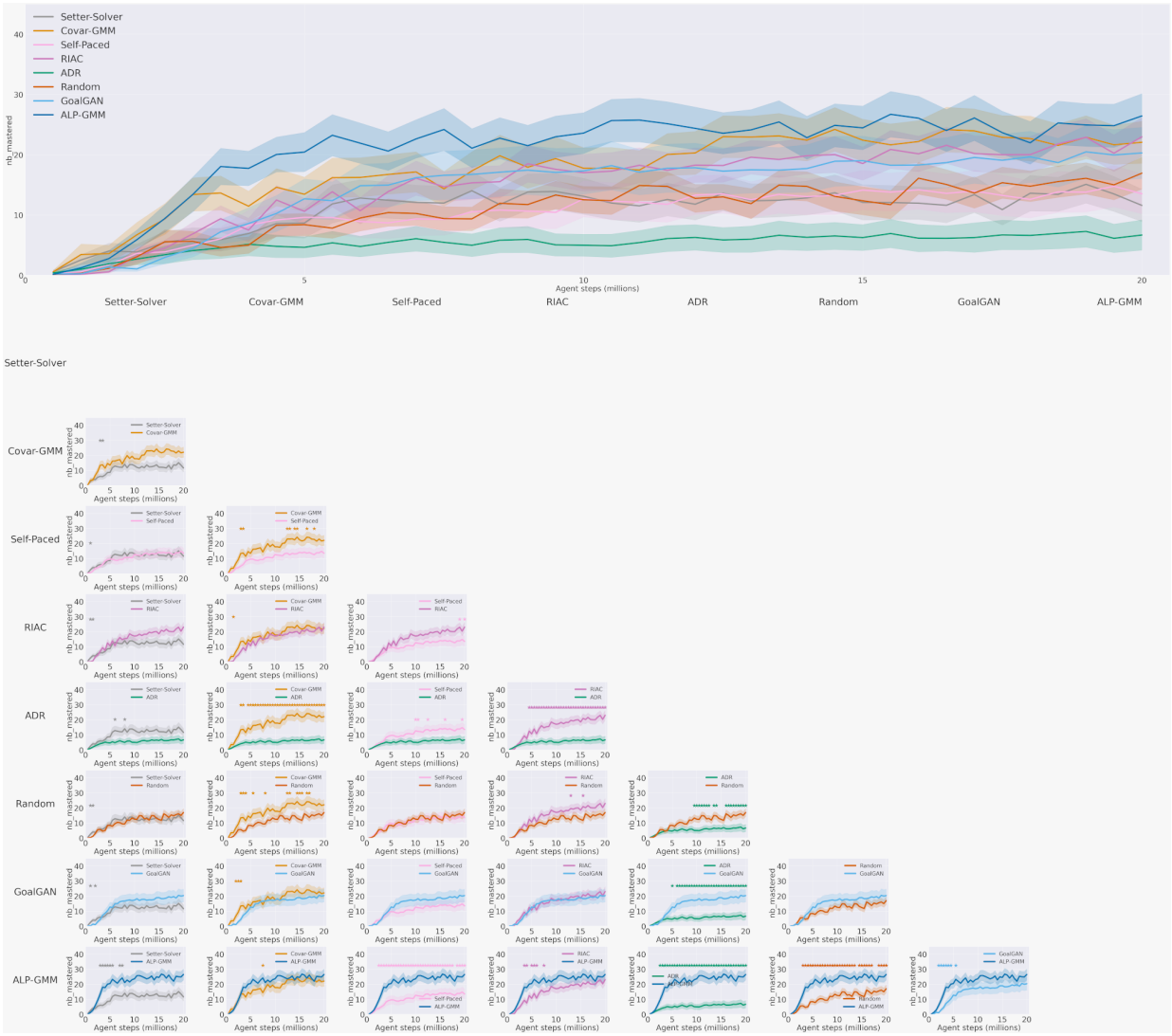

Figure 14. **Comparison of the ACL methods on the Parkour experiments**. Upper figure shows the average percentage of mastered tasks over the 48 seeds with the standard error of the mean. We then extract all the possible couples of methods and compare their two curves. At each time step (i.e. every 500000 steps), we use Welch's t-test to compare the two distributions of seeds. If a significant difference exists ($p < 0.05$), we add a star above the curves at this time step.

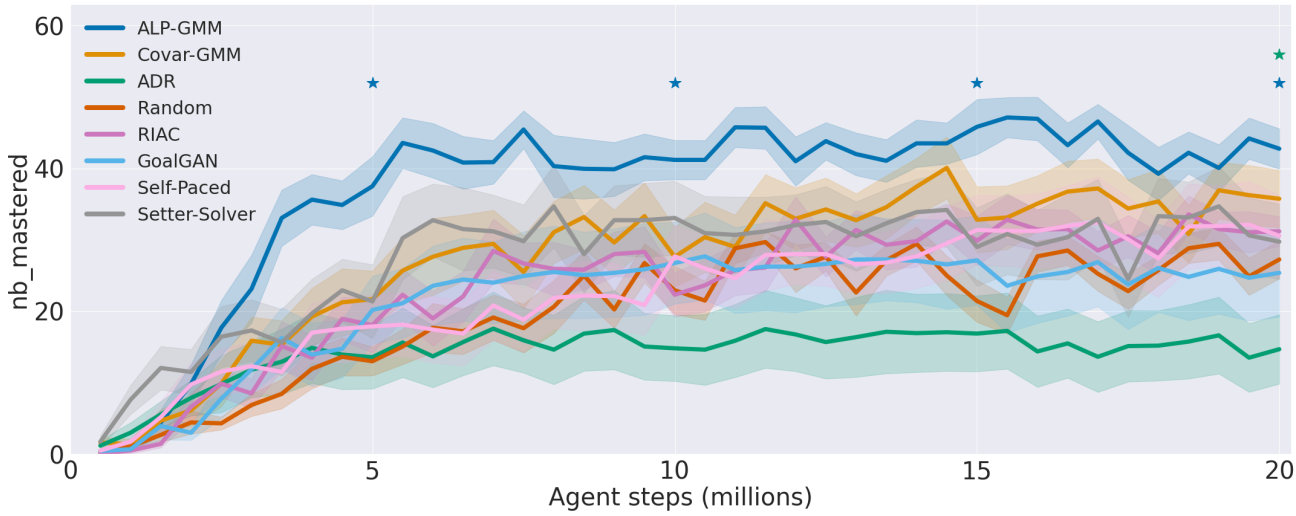

Figure 15. Average percentage of mastered tasks over 16 seeds using the **bipedal walker** along with the standard error of the mean. We calculate every 5 millions steps which method obtained statistically different ($p < 0.05$) results from Random and indicate it with a star.

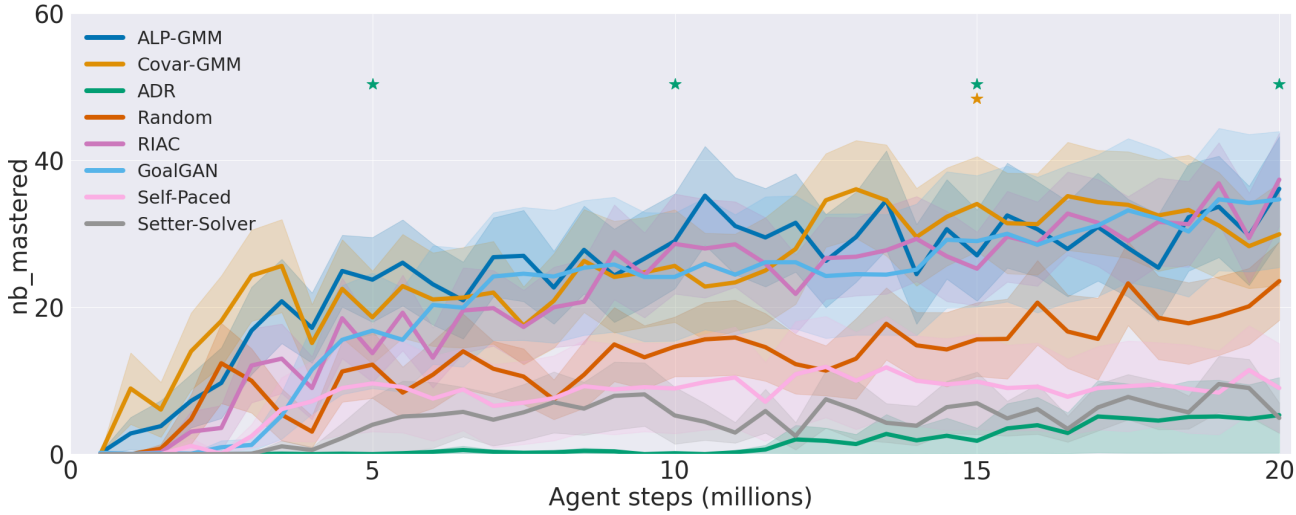

Figure 16. Average percentage of mastered tasks over 16 seeds using the **fish** along with the standard error of the mean. We calculate every 5 millions steps which method obtained statistically different ($p < 0.05$) results from Random and indicate it with a star.

D.3.2. Case study: Learning climbing locomotion

As shown in figures 4 and 17, none of the ACL methods implemented in *TeachMyAgent* managed to find a curriculum helping the student to learn an efficient climbing policy and master more than 1% of our test set. While learning climbing locomotion can arguably appear as a harder challenge compared to the swimming and walking locomotion, we present in this case study the results of an experiment using our easy CPPN's input space (see appendix C.4), as well as no water (i.e. the maximum level is set to 0.2, leading to no tasks with water). Using this, we show that simplifying the task space allows our Random teacher to master more than 6% our test set with its best seed reaching 30% at the end of learning in only 10 millions steps. In comparison, our results in the benchmark show a best performance of 1% of mastered tasks (SPDL) with its best seed reaching only 14% by the end of learning. As this simpler task space contains more feasible tasks, these results

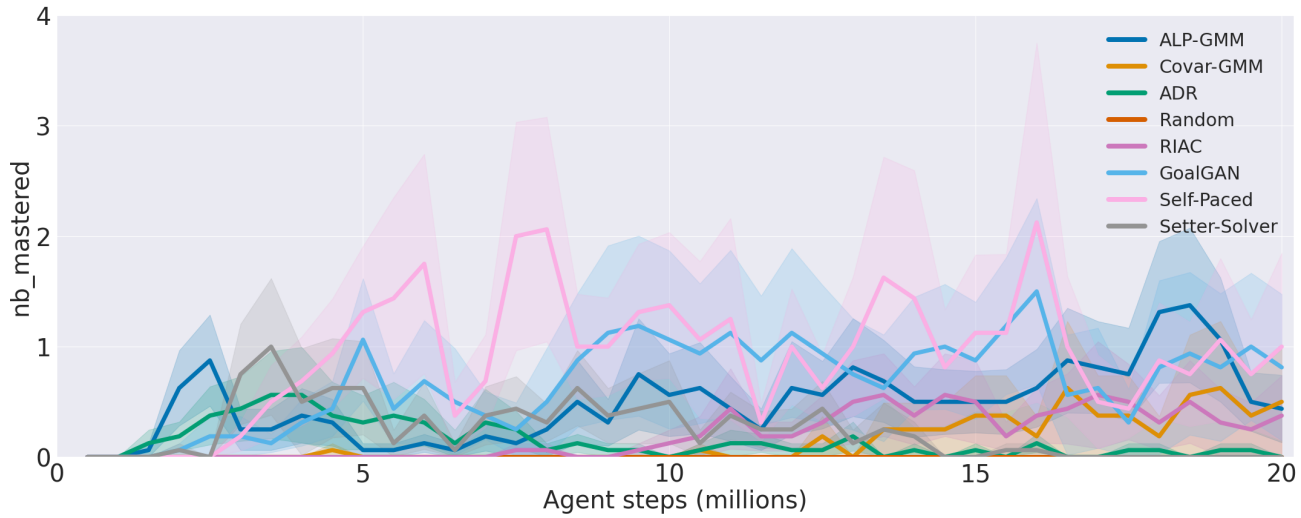

Figure 17. Average percentage of mastered tasks over 16 seeds using the **chimpanzee** along with standard error of the mean. We calculate every 5 millions steps which method obtained statistically different ($p < 0.05$) results from Random and indicate it with a star.

show that the poor performance obtained with the chimpanzee embodiment are due to the inability of the implemented ACL algorithms to find feasible subspaces for their student. This also hints possible better performance by future methods in this totally open challenge of *TeachMyAgent*. See figure 18 for the evolution of percentage of mastered tasks by the Random teacher in this simpler experiment.

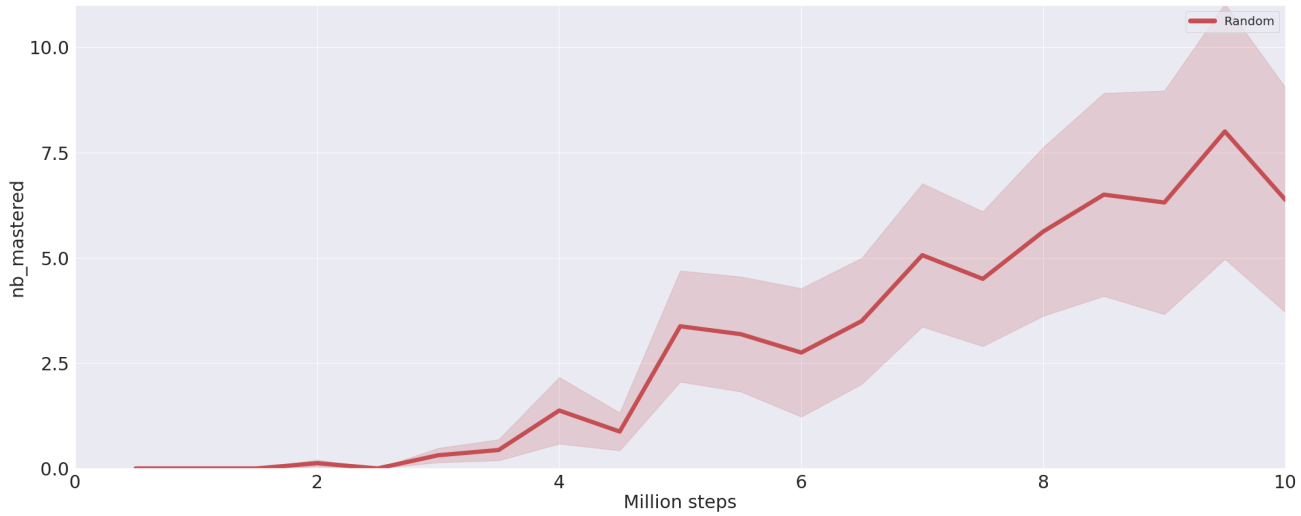

Figure 18. **Random teacher in the easy CPPN's input space with no water**. Average percentage of mastered tasks over 16 seeds using our chimpanzee embodiment along with the standard error of the mean.