Modular Adaptive Policy Selection for Multi-Task Imitation Learning through Task Division

Anonymous Author(s)

Abstract: Deep Imitation Learning requires many expert demonstrations, which can be hard to obtain, especially when many tasks are involved. However, different tasks often share similarities, so learning them jointly can greatly benefit them and alleviate the need for many demonstrations. But, joint multi-task learning often suffers from negative transfer, sharing information that should be task-specific. In this work, we introduce a method to perform multi-task imitation while allowing for task-specific features. This is done by using proto-policies as modules to divide the tasks into simple sub-behaviours that can be shared. The proto-policies operate in parallel and are adaptively chosen by a selector mechanism that is jointly trained with the modules. Experiments on different sets of tasks show that our method improves upon the accuracy of single agents and state-of-the-art meta-learning agents. We also demonstrate its ability to autonomously divide the tasks into both shared and task-specific sub-behaviours.

Keywords: Multi-task learning, imitation learning, routing networks

1 Introduction

In the field of imitation Learning (IL), an autonomous agent learns to perform tasks by mimicking demonstrations provided by an expert [1]. In control settings, there has been significant work tackling imitation of a single task [2, 3], as well as generalising around it [4, 5, 6] in recent years. However, learning multiple tasks separately is not always feasible or ideal, which is why the multi-task learning (MTL) paradigm [7] is used to combine the learning process of many tasks simultaneously. This combination can be challenging, because it not only needs to share commonalities between the tasks (positive transfer) but also avoid sharing unwanted dissimilarities that might hinder training (negative transfer) [8]. In our work, we aim to tackle this problem by introducing parallel modules that break down the shareable and non-shareable parts of tasks.

MTL can be particularly useful in deep IL dealing with control problems, since obtaining large expert datasets for robot execution can be challenging and time-consuming [9]. By allowing to share knowledge and datasets, it can not only improve performance, but also reduce the network size, compared to using a separate agent for each task [10]. This strategy also reduces the number of hyper-parameters requiring tuning during training, since all tasks are trained at once. MTL is also closely related to meta-learning. The objective of meta-learning in MTL and transfer learning (TL) is to find a common representation for all the tasks, so that agents can quickly converge to new tasks from this common point [11, 12, 13, 14]. However, this requires all tasks to be similar in the state space and similar in complexity, since it uses a common base policy and does not account for negative transfer. One way to tackle this is through adaptive networks, where the network architecture changes based on the task. This is achieved by having multiple optional layers – either in parallel [15, 16], sequential [17, 18] or both [19] – and a router that selects which of those to activate for each task. An example of such routing network can be seen in Figure 1b. This architecture can be particularly beneficial to image problems – especially given the many different types of layers that can be used as experts [16, 20]. However, control problems do not usually use very deep or convoluted architectures, so the benefits from such methods can be limited.

Submitted to the 5th Conference on Robot Learning (CoRL 2021). Do not distribute.
In our work, we aim to apply the benefits of adaptive architectures and mixture of experts in control problems. Simple policies have been shown to learn short, simple behaviours very well (e.g. reaching an object [21]) but struggle when the tasks become more complicated (e.g. hammering a nail [22, 23], performing acrobatics [24]). However, complicated tasks can usually be broken down to shorter, simpler sub-behaviours. Additionally, due to their simple nature, these behaviours are common amongst different tasks. As humans, we often can deduce how tasks can be broken down and the parts that can be shared thanks to our experience, but that can be difficult to model in agents [25, 26]. Also, similarities might not only be present in the task sequences, but in their general execution as well (e.g. performing a similar task in a different environment, or with a different tool). Our method aims to learn these simpler sub-behaviours by modelling them in the form of modular proto-policies.

The primary contribution of our work is a system that performs MTL by breaking down the tasks into sub-behaviours. To achieve this, we use multiple modules that work in parallel independently. Each of these modules is a proto-policy with the capacity to learn a simple behaviour. We also introduce a way to apply adaptive methods, similar to routing, to control problems. This is done by using a selector mechanism that decides how important each proto-policy is for a particular task, given the current state of the environment. The fact the modules learn only based on the current state and not the task means they learn general behaviours of the state-action space. The selector is the one that forces the proto-policies to learn different behaviours, which are either shared between tasks or used exclusively by one task. This way it achieves positive transfer without promoting negative transfer. The number of proto-policies depends on the number of common or different sub-behaviours present in the tasks, meaning a set of similar tasks can be represented with significantly fewer resources than using a separate agent for each task (Figure 1a). A pipeline of our system is shown in Figure 1c. Our system is evaluated on 3 different sets of tasks that exhibit similarities in 3 distinct ways. One has commonalities in its sequence (i.e. same actor performs different tasks), one in its environment (i.e. different actors perform in the same environment) and one in its actions (i.e. same actor performs in different environments). Our system shows better performance compared to a single agent baseline, as well as state-of-the-art meta-learning. Our method also offers deeper understanding of the learning process by visualising the role of each proto-policy separately. Qualitative results of this investigation can be seen in Section 4.2.

2 Related Work

Deep imitation learning has improved massively over the years [27, 2, 28, 29]. Whilst behavioural cloning (BC) [27] achieved quick convergence, it required a large dataset and had trouble generalising to unseen states. To alleviate this, [2] introduced Generative Adversarial Imitation Learning (GAIL). Although it generalised better, especially with limited datasets, it required interactions with the environment during training. Since then, various works have sought to improve generalisation [4, 5, 6], and even achieve one-shot learning [28, 29]. Similarly to our work, [6] sought to not
only imitate, but also segment skills. It achieved this by learning a multi-modal policy, but unlike
ours, it did not take negative transfer into account. Since it was based on GAIL, it also required
environmental interaction during training.

Meta-learning is closely connected to multi-task learning in control, since it can learn a general
representation around a set of tasks. In the field of imitation, one such work is Model Agnostic Meta
Learning (MAML), presented in [11] and optimised in [30], where it learnt a common policy for a
number of tasks. It then showed this representation point quickly converges to new similar tasks.
Whereas this work tried to find one common point between tasks, our work aims to break down
tasks into common and uncommon representations. Therefore, it did not accommodate for negative
transfer, like ours does. It also required a separate network for every single task. More recently,
[12] improved multi-task imitation by utilising the agent’s own experience as it trained. Inspired by
one-shot learning [29], it encoded the demonstrations into a task embedding and then evaluated the
policy roll-outs based on this embedding. If the roll-outs’ behaviour was similar to the experts, they
were added to the embedding. However, this technique required interaction with the environment
during training, which can be demanding in control.

Multi-task learning in control has also been achieved using both imitation and reinforcement learn-
ing (RL) [31, 32, 33], which uses a reward function rather than expert demonstrations [34].Contrary
to meta-learning, [10] tackled MTL in IL and RL by defining the policy as a function of not only
the state, but also the task. However, conditioning the entire feature pipeline on the task, as well
as using a single fixed policy, can make it hard to adapt to many different tasks. This is why we
use an adaptive approach that does not condition the feature on the task directly. This can also be
seen in [35], which designed a multi-headed MTL system that conditioned only the final features
on every task separately. Another similarity with our system is that it used many parallel streams of
features which are combined at high level. However, all the streams are needed to execute a task,
which can be problematic as the number of tasks and streams increases.

Adaptive networks have also been used in MTL by adapting their architecture based on the task
they are executing. In routing networks, [15] used parallel streams of layers and a router to select
which layers to use or skip. While their router was conditioned on each task and input, like our
selector, it was applied on every layer sequentially. Our system, on the other hand, trains high
level features in parallel and selects them near the end of the pipeline. Additionally, their router
was trained using RL, which can be hard to converge, whereas our selector works more like a
regulariser that preserves the objectives of MTL and is optimised with gradient descent. Similarly
to [15], [17] also connected their modules sequentially with no task-division, but updated their
router with gradient descent. Using gradient descent to train the router was also performed in [16],
which used a number of different types of layers and a router selected the best one based on the
task. This is similar to the mixture of experts presented in [20]. In their work, various different
types of layers were selected using a gate and were then linearly combined. Whereas our work
also combines features from multiple streams, its main difference from these two works is that it
does not use modules that are structurally different, instead it aims to help them learn different high
level representations. Using modules and BC to break down a task was also the objective of [36].
But while they managed to divide one task, our work aims to extend upon it and cover multiple
tasks. The work that mostly relates to us is [19], where they used modules both sequentially and
in parallel, with a gating mechanism deciding their contribution. Similarly to our objective, they
wanted the system to learn sub-categories of images but, unlike us, they had prior knowledge of the
different sub-categories, and that is hard to obtain in sequential control data.

3 Proposed Method

Background. The system is modelled after a Markov Decision Process (MDP). MDPs are expressed
with a set of states \( S = \{s_1, s_2, \ldots \} \), a set of actions \( A = \{a_1, a_2, \ldots \} \), the probability \( P(s'|s, a) \)
that action \( a \) will lead to state \( s' \) and the reward \( R(s, a) \) for taking action \( a \). In control settings the
states are the environment and the actions are the robot. In RL, the reward needs to be defined. In
IL, it might be available to the experts, but is unknown to the imitation algorithms and needs to be
inferred. The goal of imitation is to learn a policy \( \pi(a|s) \) i.e. a probability distribution that produces
actions similar to the experts. A demonstration dataset \( T \) comprises a number of trajectories \( T =
\{\tau_1, \tau_2, \ldots \} \) and each \( \tau = \{(s_1, a_1), (s_2, a_2), \ldots \} \) is a sequence of state-action pairs.
Algorithm 1: Modular Adaptive Policy Selection

Input: Set of expert trajectories $T_k$ for each task $k \in [1, K]$, initial module proto-policies $\theta_{0,i}, i \in [1,M]$, distribution policy $\phi_0$ and selector weights $w_0$.

1. while not done do
   // sample expert batches from all tasks
   $Q = \{\}$
   for $k = 1, \ldots, K$ do
     $Q_k \subset T_k$
     $Q = Q \cup Q_k$
   end
   // Update selector $G$ based on equations (2), (3) and (9)
   $w \leftarrow w - \alpha \nabla_w L_{total}(Q, w, \theta_1, \ldots, \theta_M, \phi)$
   // Update $\pi(\phi)$ and modules $\mu_i(\theta_i)$ with regards to the updated selector
   for $i = 1, \ldots, M$ do
     $\theta_i \leftarrow \theta_i - \beta \nabla_{\theta_i} L_{total}(Q, w, \theta_i, \phi)$
   end
   $\phi \leftarrow \phi - \gamma \nabla_{\phi} L_{total}(Q, \theta_1, \ldots, \theta_M, \phi)$
end

3.1 Modular Adaptive Policy Selection (MAPS)

Our framework, depicted in Figure 1c, consists of $M$ proto-policy modules $\mu_i(v|s)$ that operate in parallel. Similarly to single-task imitation networks, each module $\mu_i$ receives $s$, the current state of the environment, but outputs a number of features $v_i$ instead of a distribution. The selection of the modules is performed by a different, selector network $G(s, k) \in \mathbb{R}^M$ that outputs $M$ task-relevant scores, one for each module. After the scores are applied to the proto-policy features $v$, they are finally inserted to a probability distribution layer $\pi$ that transforms them into a distribution of actions $a$, like a policy network. Therefore, the final actions can be expressed as:

$$a = \pi(\phi(s, k \mu_1, \ldots, \mu_M(s)).$$

This architecture allows for learning primitives independently from specific tasks. This way, a module does not know which - or if any - tasks use it and how much. This leads to modules that learn commonalities between tasks, but also modules that learn features exclusively tailored to one task. Therefore, the system is able to perform positive transfer, while avoiding negative transfer.

The two parts of the system, the selector and the proto-policy modules, have very distinct roles. The modules learn separate parts of the task space (be it for many tasks or one) and the selector learns which modules should be used by a task and when. These two objectives can be seen in the final loss of the system:

$$L_{total} = L_{imitate} + L_{selector}.$$  (2)

The term $L_{imitate}$ is the one that learns the tasks by performing the correct actions. The term $L_{selector}$ is a regulariser that ensures positive transfer is performed, while avoiding negative transfer.

For the rest of the paper we will be using behavioural cloning (BC) as the imitation technique for $L_{imitate}$, but any other IL technique (e.g. GAIL) can also be used. Since BC minimises the difference between expert and policy actions, the imitation loss can be written as:

$$L_{imitate} = L_{BC} = \|a - a_E\|^2_2 = \|\pi(\phi | s, k \mu_1, \ldots, \mu_M)| a_E\|^2_2,$$  (3)

where $(s, a_E)$ the state-action pairs of the expert trajectories $T_E$. As evident in equation (3), the partial derivatives with respect to $\theta_i$ (modules), $w$ (selector) and $\phi$ (distribution layer) are inter-connected. Therefore, training $\theta$ and $w$ jointly allows both of the them to benefit from each other. A description of the joint learning process is presented in Algorithm 1.

3.2 Module Selection

Routing networks have often been used in adaptive networks to determine the architecture used at any given point [15, 17]. However, these are usually applied in a sequential fashion, deciding if a
layer will be used or skipped based on the task and the features of the previous layer. In our case, the selector is applied once for all the modules, and its decision is about how all the modules will be combined. This is done by producing a significance score that is then multiplied with the features of the corresponding module. The final score \( g \) of selector \( G \) is normalised using softmax:

\[
g(s, k) = \text{softmax}(G(s, k)).
\]  

(4)

The selector is responsible for deciding which modules to activate, based on the current task \( k \) and state \( s \). Ideally, this decision would be based on which tasks, or parts of tasks, are similar – and should share modules – and which of them are too different and should have dedicated modules for themselves. Unfortunately, such information is not easily obtainable, or even available. To overcome this, our selection mechanism focuses more on the general objective of mutual learning, rather than specific common characteristics of each task. This objective was broken down into four terms: sharing, exploration, sparsity and smoothness.

**Sharing.** The objective of this term is to make different tasks share the same modules for the same input state \( s \). Therefore, it represents the positive transfer part of MTL. The concept of sharing has been previously presented in [17], where the router encouraged the tasks to share initial layers. We define our term in a similar way, only we have no preference in certain modules and we avoid using \( L1 \) loss, due to it being non-differentiable. Therefore, the sharing loss is the mean \( L2 \) difference between the scores of all the possible pairs of \( K \) tasks, given a specific state \( s \) and can be written as

\[
L_{\text{sharing}} = \frac{K!}{2!(K-2)!} \sum_{k_1=1}^{K-1} \sum_{k_2=k_1+1}^{K} \| g(s, k_1) - g(s, k_2) \|^2_2.
\]  

(5)

**Exploration.** The objective of this term is to make all modules learn something and avoid mode collapse to one or two modules. This incentivises the system to break the tasks down to multiple – shareable or not – elements. The exploration loss chosen for this term is similar to the importance loss presented in [16] and [20], where the coefficient of variation (\( \text{cv} \)) of a batch is minimised. This ensures that in a batch with samples from all tasks, all modules will be used. Therefore, this loss is per batch, and for \( T_k \) as the trajectories of task \( k \), it can be described as

\[
L_{\text{exploration}} = \text{cv}^2 \left( \sum_k \sum_{s \in T_k} g(s, k) \right).
\]  

(6)

**Sparsity.** The objective of this term is to ensure each module learns a distinct part of a task. Using the sharing and exploration terms only, it is still possible for all the tasks to use all the modules uniformly. But that renders the selector useless and the modules indistinguishable. The sparsity loss counters this by making each task use only a small subset of modules. Additionally, the combination of sparsity and exploration enables a module to be exclusively used by one task, thus allowing task-specific learning and avoiding negative transfer. Regarding the loss, while [17] also introduced a sparsity regularisation in the form of log-likelihood, it proved quite unstable in our case. This was due to the logarithmic nature of the loss reaching \(-\infty\), which could not be easily balanced with the rest of the terms that minimise at 0. The Shannon entropy [37], on the other hand, proved to be a much more stable choice. However, due to the competing nature of this term with sharing and exploration, uniform distribution is still an acceptable local minimum for \( M > 1/e \). To avoid this, we shift the entropy function so that its maximum lands on the \( \frac{1}{M} \) uniform point. Hence, the final sparsity loss is

\[
L_{\text{sparsity}} = -\frac{1}{M} \sum_{i}^{M} g_i(s, k) \frac{1}{e^M} \log g_i(s, k).
\]  

(7)

**Smoothness.** The objective of this term is to provide temporal stability in the selection of modules for each task. This way it avoids erratic module changes between steps. To achieve this, we use the discrete derivative of \( g(s, k) \):

\[
L_{\text{smoothing}} = \| g(s_t, k) - g(s_{t-1}, k) \|^2_2,
\]  

(8)

where \( s_t \) and \( s_{t-1} \) the states at time steps \( t \) and \( t-1 \) in a trajectory \( T_k \).

The total loss of selector \( G \) comprises the 4 above-mentioned losses:

\[
L_{\text{selector}} = \lambda_1 L_{\text{sharing}} + \lambda_2 L_{\text{exploration}} + \lambda_3 L_{\text{sparsity}} + \lambda_4 L_{\text{smoothing}}.
\]  

(9)
It is worth noting that the individual losses compete with each other. Whereas any combination of sharing, exploration and sparsity in pairs makes the loss decrease, the combination of all three makes them compete. This is a manifestation of the opposing MTL objectives of positive transfer without negative transfer. Therefore, careful consideration of the $\lambda$ coefficients is required to achieve balance. For example, if we know there are dissimilar tasks, greater emphasis should be given to exploration (e.g. Gravity set in Section 4.1). On the other hand, if all the tasks are executed in a similar manner, a greater emphasis on sparsity and sharing might be more appropriate (e.g. Sawyer set in Section 4.1). The smoothing loss closely relates to sparsity and is necessary for modular consistency. The combination of sparsity and smoothness ensures that each task consistently uses the same few modules, rather than using a single but different module at every step.

4 Experimental Results

We evaluate the system under 3 different MTL settings covering environmental, actor and task variations. We investigate the success rate and stability of the system against a single agent per task baseline (Figure 1a) and MAML, which is a standard meta-learning approach for MTL and TL. We also evaluated the method qualitatively, by investigating the features each module learns and if the tasks can be broken down in segments, according to our original hypothesis.

Environmental variation set (HalfCheetah Gravity). This set of 5 tasks involves the same agent, performing the same task, only under different environments. Specifically, they all involve OpenAI’s [38] HalfCheetah agent running, only under different accelerations of gravity. For $g_r = 9.8 m/s^2$ as the default HalfCheetah’s gravity acceleration, the 5 different values used are $0.5g_r, 0.75g_r, g_r, 1.25g_r, \text{ and } 1.5g_r$. These environments were first introduced in [39]. To obtain the expert Gravity datasets, an RL agent was trained for each task using the reward function provided in [39] and then was used to generate 35 experts.

Actor variation set (HalfCheetah Modified). This set of 11 tasks involves performing the same task, under the same environment, only using a different actor. Specifically, they all involve OpenAI’s HalfCheetah agent running, only the HalfCheetah’s body parts are modified. The model has 5 different body parts, and each of them is enlarged or shrunk by 25% for every task. The body parts include head, torso, thigh, leg and foot. The original model along with the 5 enlarged and 5 shrunk models, make up the 11 tasks of this dataset. Like the Gravity tasks, the Modified environments were introduced in [39] and the 35 experts were obtained using an RL agent.

Task variation set (Sawyer). This set of 10 tasks involves using the same actor performing different tasks in similar environments. Specifically, the Sawyer robot gripper is used to perform different tasks and manipulate various objects. These tasks include reach, push, pick and place, press button, insert peg in hole, open door, open drawer, close drawer, open window, and close window. These tasks were first introduced in [21]. The expert datasets were obtained using a human expert that manually executed 20 instances of the tasks in the simulated environments. The selector $G$ and all the proto-policies $\mu_i$ used have the same structure and size, with 3 hidden layers of 128 features each. Both the modules and the selector are optimised using Adam [40] with a learning rate of $3 \times 10^{-4}$. The batch size for Gravity and Modified is 128, whereas Sawyer uses 32. Both BC and MAML use the same policy structure, optimiser and batch size as the modules. MAML also uses a meta-batch size of 160 for Gravity and Modified, and 40 for Sawyer. The hyperparameters used were $\lambda_1 = 1, \lambda_2 = 0.1, \lambda_3 = 0.5, \lambda_4 = 1$ for Sawyer and $\lambda_1 = 1, \lambda_2 = 0.5, \lambda_3 = 0.001, \lambda_4 = 1$ for Gravity and Modified because sparsity was overpowering the BC loss. All sets use $\lambda_{imitate} = 0.75, \lambda_{selector} = 0.25$ and the datasets use 70% for training and 30% for validation and tuning. The simulation environment used in this work is Mujoco Pro [41].

4.1 Quantitative Results

We evaluate the performance of our method by unrolling the policies on 100 test initial positions for each task and measuring its highest success rate over various different numbers of experts. We also compare the results with those from a single BC agent for each task, as well as an agents trained with MAML and then finetuned on each task (up to 10 updates), as described in [11]. Whereas MAML can be used in both supervised and RL problems, only its RL version was previously tested in control environments [11, 21].
Figure 2: Success of MAPS on Gravity (left), Sawyer (middle) and Modified (right) sets using 10
modules. It is compared to MAML, multi-task task-id conditioned and single agent BC.

Table 1: Mean success % of MAPS across all tasks for different number of modules M and experts.

<table>
<thead>
<tr>
<th>experts</th>
<th>M=3</th>
<th>M=5</th>
<th>M=10</th>
<th>M=40</th>
<th>M=20</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>21 ± 5</td>
<td>35 ± 2</td>
<td>47 ± 4</td>
<td>37 ± 13</td>
<td>31 ± 5</td>
</tr>
<tr>
<td>15</td>
<td>43 ± 2</td>
<td>64 ± 2</td>
<td>82 ± 3</td>
<td>61 ± 9</td>
<td>76 ± 4</td>
</tr>
<tr>
<td>25</td>
<td>58 ± 4</td>
<td>79 ± 4</td>
<td>93 ± 3</td>
<td>76 ± 12</td>
<td>80 ± 2</td>
</tr>
<tr>
<td>35</td>
<td>68 ± 3</td>
<td>81 ± 2</td>
<td>95 ± 2</td>
<td>79 ± 6</td>
<td>83 ± 2</td>
</tr>
</tbody>
</table>

Figure 2 shows the experimental results on all the tasks of the 3 sets. As a general rule, MAPS with
10 modules performs significantly better than the single agents and MAML. It should be noted that
MAML seems to struggle in some tasks, potentially due to them being too different and it being
unable to find a single common representation between them. The only case it did unquestionably
better is in the SmallFoot and HalfCheetah tasks in the Modified set (Figure 2 right). The fact these
two task follow a similar trend indicates MAML managed to converge to a point around these two,
giving them a significant boost. However, the same cannot be said for the rest of the tasks, which
again shows how MAML has trouble generalising to tasks that may be too different. MAPS, on the
other hand, shows a steady performance in all tasks and clearly surpasses the single agent baseline.

Table 1 shows ablation studies of MAPS with various numbers of modules. The only case where
a greater number of modules than tasks was beneficial was in the Gravity set, whereas the other
sets’ performance seems to plateau at $M = 10$. This practically shows how the number of modules
should not depend on the number of tasks, but rather the similarities between them. In the Sawyer
set, where we expect more similarities between the tasks, we see the success of $M = 5$ is comparable
to the one with 10 modules (albeit more unstable). Therefore, even with half the size of the single
agents’ resources, MAPS can still achieve better performance.
4.2 Qualitative Module Understanding and Task Breakdown

We also evaluate how well MAPS manages to break down task behaviour in the Sawyer set. Figure 3 (left) presents the occurrence of every module in each individual task. This shows the influence of the selector losses, as well as the common or task-specific nature of the modules.

The Figure 3 bar graph shows our initial hypothesis is correct and tasks are composed of shareable modules, task-specific modules, or a combination of the two. An example of this is the subset of window-open, drawer-open and drawer-close tasks. In this subset the tasks use both shareable and task-specific modules. The most similar tasks are window-open and drawer-close, which use the same modules $\mu_5$ and $\mu_9$. An example of applying these modules individually on each task can be seen in Figure 3 (right). The figure shows these two modules have learnt two distinct behaviours: $\mu_9$ (orange) has learnt to reach (and grab when possible) the handle, and $\mu_5$ (blue) has learnt to move forward and reach the target. It is very encouraging then that drawer-open also uses $\mu_9$ to perform the "grab the handle" sub-behaviour. However, its movement after that needs to be backwards, compared to the forward movement $\mu_5$ performs for the other two. Therefore, this movement is learnt in a different module $\mu_7$ (magenta), which is task-specific for drawer-open. This behaviour of drawer-open demonstrates how MAPS can perform positive transfer, while avoiding negative transfer. Finally, Figure 3 also shows how the combination of all the $\mu_i$ modules into a policy $\pi$ produces a successful trajectory. It should be noted that even though the sub-behaviour "grab the handle" is a natural movement for humans, it is there only because it was present in the expert trajectories performed by the human expert. Using only $\mu_5$ in the drawer-close task is enough to succeed, which is more likely what an RL agent would have learnt. However, MAPS is capable of also finding and modelling minute sub-behaviours that make the trajectories look more natural.

5 Conclusion and Future Work

We present a new method of imitation in multi-task control problems by breaking down the tasks into distinct sub-behaviours. To achieve this, we use an adaptive architecture that utilises a multitude of independent proto-policy modules and a selector that forces them to cooperate. Our experiments show that this method can be used on various different MTL settings and can not only improve the success rate, but also reduce the resources required, compared to using separate agents. We also show how the proto-policies can be evaluated individually by qualitatively analysing their learnt behaviour. This indicates that MAPS can also offer a better understanding of what is being learnt in every module, something that is not always easy to do in neural networks [42].

As a future extension of our work, we would like to see it used in not only MTL, but also transfer learning problems. The already-learnt modules should be useful to new tasks, especially in one-shot, or even zero-shot scenarios. Additionally, the fact we use BC for the experiments in this paper does not mean it cannot be of use to other methodologies as well. Therefore, we would also like to see MAPS used with other IL methods besides BC, like GAIL.
References


