AIMS CDT Project Report: Towards One-Shot Learning From Demonstration via Reinforcement Learning

Mark N Finean¹, Luisa M Zintgraf², Kyriacos Shiarlis³, Vitaly Kurin⁴, Shimon Whiteson²

¹AIMS CDT, University of Oxford
²Department of Computer Science, University of Oxford
³Informatics Institute, University of Amsterdam
⁴Morpheus Labs

Abstract—We explore meta-learning algorithms and architectures for use in one-shot learning from demonstration via reinforcement learning. We provide evidence that REPTILE does not work effectively at meta-learning in reinforcement learning environments and present preliminary findings on the effectiveness of GRUs at ‘fast adaptation’ to tasks in reinforcement learning environments.

I. INTRODUCTION

Imagine you have a household robot. It can wash the dishes, wipe the windows and take out the rubbish. However, it can’t vacuum the floors. In this scenario it would be beneficial to be able to give the robot a demonstration of how to vacuum and it subsequently perform this task perfectly. This is an example of one-shot learning from demonstration. 

In this project, we investigate potential methods to achieve one-shot learning from demonstration in a reinforcement learning (RL) setting.

We start by taking inspiration from the human brain. Humans are remarkable learners and can adapt quickly to new tasks and environments, especially as a child. Given a new task, such as playing a new game or sport, humans can often perform well after very few attempts. In contrast, current state-of-the-art agents often take thousands of iterations and parameter updates. Silver et al.'s famous ‘AlphaGo’ utilised 4.9 million games of self-play to become an expert at the board game ‘Go’ [1]. Humans outperform in this respect because when learning to perform a new task, humans utilise a vast wealth of prior knowledge. In the case of playing tennis, a human may not have played the game before but they do know how to grip the racket, understand the dynamics of their own body, and know the Physics of hitting the ball etc. Artificial agents should also be able to do the same and leverage prior knowledge so that only a small number of new examples are required to perform well at a new task. However, in practice, artificial agents are often trained from scratch i.e. from a randomly initialised neural network model. The process of training an agent so that it can adapt quickly to new tasks is known as ‘meta-learning’.

In this project, we investigate various architectures and algorithms for meta-learning in the domain of RL problems. Our goal is to accomplish one-shot learning from expert demonstrations via RL. To achieve this, we aim to develop a method of training such that a trained model can quickly adapt to perform well at a new task, using as few new data points and as few iterations of policy updates as possible.

Learning from demonstration is not a novel concept introduced here; it has already been accomplished in a multitude of papers [2] [3] [4] [5] [6]. Despite tremendous successes in deep reinforcement learning, the algorithms typically require large amounts of data before reaching acceptable performance levels. This poses problems in many real-world tasks, where the agent must learn in a real environment and so previous research has focused on learning from demonstrations in a supervised learning setting, known as 'behavioural cloning'.

Behavioural cloning is very powerful and in our experience outperforms reinforcement learning methods in control examples. The problem however is that behavioural cloning doesn’t generalise well to new tasks outside the bounds of the training data. We aim to develop an imitation learning method that generalises better through the use of inverse reinforcement learning.

In traditional imitation learning there is no reward function and so we cant use RL directly to train an agent. However, in inverse reinforcement learning we can try to infer the reward function and thus re-frame the problem in an RL setting. There has been research into meta-learning in RL scenarios. 

For example, Finn et al. demonstrate success using their meta-algorithm, MAML, in classification, regression and RL environments [7]. However, to our knowledge meta-learning has not yet been used in application to imitation learning within an RL setting. This is the focus of this research project.

We will describe the related work for this project, explain the toy problems adapted and used in our experiments and discuss the experiments conducted and our findings. We will then present ideas to pursue in future research.

II. RELATED WORK

Generative Adversarial Imitation Learning (GAIL) presents an attractive framework to express learning from expert demonstrations as a reinforcement learning problem [8]. In GAIL, there are two competing agents; a ‘Generator’ (the policy) which chooses actions given observations about
the environments, and a ‘Discriminator’ which acts as a classifier between policy-generated demonstrations and those provided by an expert. The Generator is trained to perform like the expert would using expert demonstrations. Meanwhile, the Discriminator is trained to classify a trajectory (sequence of state-action pairs) as being generated by the expert or the Generator policy. In cases where the policy ‘fools’ the Discriminator into classifying it’s actions as the expert, the policy is positively rewarded.

For the policy to learn from these reward signals, and thus learn to provide expert-like actions given new demonstrations, we therefore require an effective meta-learning algorithm.

We believe that learning in this manner will result in more general policies than those obtained by behavioural cloning, and ones that can adapt faster to new tasks.

One approach to meta-learning in RL environments is addressed in the field of ‘hierarchical reinforcement learning’. Hierarchical RL focuses on learning primitive sub-tasks which can be then be combined to accomplish a multitude of tasks. In the previous example of learning to play tennis, this might correspond to learning the separate sub-tasks, such as gripping the racket, moving your arms to an intended position, hand-eye coordination etc. As presented by Stadie et al., hierarchical RL tends to focus more on “defining specific architectures that should lead to hierarchical behaviour, whereas meta-learning instead attempts to directly optimise for these behaviours” [14]. We will not discuss hierarchical RL further in this report.

In Schulaman et al. (2018), they analyse algorithms for learning “a parameter initialisation that can be fine-tuned quickly on a new task, using only first-order derivatives for the meta-learning updates” [9]. They further present their meta-algorithm, REPTILE. REPTILE works by repeatedly sampling a group of tasks and performing $k$ steps of stochastic gradient descent (SGD) on each of them independently. The initial parameters of the model/policy are then updated towards the final parameters learned on each task. In the case of $k = 1$, this corresponds to “joint training in which SGD is performed on the mixture of all tasks. The REPTILE algorithm is an attractive meta-learning algorithm for us to explore using because it is relatively simple to implement.

In Abbeel et al. (2016), a stacked Long Short-Term Memory (LSTM) is exposed to multiple episodes from many different Markov Decision Processes (MDPs) [10]. Back-propagation then occurs through the entire temporal span of the LSTM to perform a policy gradient update. The intuition in using a Recurrent Neural Network (RNN) variant is that the memory weights will update so that the RNN will learn a more effective/faster RL algorithm that takes into account recent experience or demonstrations of a task.

III. PRELIMINARIES

We would like to apply meta-learning to variety of different learning problems and so we introduce some generic notation.

We consider a general discounted MDP with a finite-horizon which can be represented by

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \rho_0, \gamma, H).$$

The elements of $\mathcal{M}$ have the following definitions: $\mathcal{S}$ is the set of states, $\mathcal{A}$ the set of actions, $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow R^+$ is a transition probability distribution, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow R$ a reward function, $\rho_0 : \mathcal{S} \rightarrow R^+$ an initial state distribution, $\gamma \in [0, 1]$ a discount factor, and $H$ the horizon.

We can formally describe each task, $\tau$, by

$$\tau = (\mathcal{L}(x_1, a_1, \ldots, x_{H}, a_H), q(x_1), q(x_{t+1}|x_t, a_t), H),$$

where $\mathcal{L}$ is a loss function, $q(x_1)$ is a distribution over initial observations, $q(x_{t+1}|x_t, a_t)$ is a transition distribution, and $H$ specifies the episode length, such that $x \in \mathcal{S}$, $a \in \mathcal{A}$, $q \in \mathcal{T}$.

We use the loss $\mathcal{L} \rightarrow R$ for each roll-out as task-specific feedback to optimise our policy, $\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$, where $\theta$ are the parameters of our model.

Each episode $\tau$, with horizon $H$, consists of a sequence $(s_0, a_0, r_0, \ldots, s_{H}, a_H, r_H)$ of state, action, and corresponding reward at each time-step $t$. A roll-out consists of a number of episodes.

We define the discounted episodic return of $\tau$ as

$$R_\tau = \sum_{t=0}^H \gamma^t r_t,$$

which depends on the initial state distribution $\rho_0$, the agent’s policy $\pi_\theta$, and the transition distribution $T$.

The expected episodic return given by the agent’s policy $\pi_\theta$ is $\mathbb{E}_{\tau}[R_\tau]$.

For a given task, the goal is to solve for the $\theta^*$ which achieves the highest expected episodic return. In one-shot meta-learning from demonstration, we look to solve for a $\theta_{meta}^*$ such that $\theta_{meta}^*$ can be reached within one meta-update from $\theta_{meta}^*$ for all $\tau$.

IV. EXPERIMENTS

A. Toy Problem Environments

We look at two different toy problems in this project, both of which used OpenAI Gym and/or Mujoco environments [11] [12].

1) Fetch-Reach Robotic Arm: We used the Mujoco ‘Fetch-Reach’ environment in which we can execute ‘actions’ that move a robotic arm in 3D space. The aim is to train the arm to move as quickly as possible to a given goal position. The setup is shown in Figure 1.

2) Hopper Environments: The second environment we used was to adapt the OpenAI ‘Hopper’ environment. In the classic Hopper reinforcement learning problem, we train a model that learns how to hop and gains greater reward as it travels farther distances. We believe an impressive demonstration of meta-learning in an RL setting would be to train a policy on a number of different Hopper environments
and for the policy to be able to quickly adapt to a new set of physical parameters, such as changing the length of the foot. We thus created an archive of adapted Hopper environments in which the physical parameters of the Hopper were varied. This is illustrated in the examples shown in Figure 2. The objective is then to learn a model that can quickly adapt to perform well at an unseen set of Hopper parameters.

V. TRAINING EXPERTS

To learn from demonstrations, we require an expert that can produce demonstrations. To train expert policies, we used policy gradients with Trust Region Policy Optimisation (TRPO), “an iterative procedure for optimising policies, with guaranteed monotonic improvement” [13].

In each TRPO update (epoch), we generate a series of roll-outs, using the policy to produce a mean action. At each time step, we execute an action chosen from a normal distribution about the action predicted by the policy, thus introducing some exploration. A reward is obtained for the resulting actions. In the case of Fetch-Reach, this reward is equal to the negative distance between it’s resultant position and the goal position. Maximising the reward thus corresponds to reaching the goal position in the minimum time.

Typically, we used a batch size of 1,000-10,000 for Fetch-Reach where a single episode consists of 50 time-steps after which the environment is reset (20-200 episodes per roll-out), and 25,000 for Hopper environment.

For our expert agent we used a Policy and Value network as a baseline, both using three fully connected hidden layers of size 64. We compared the performance when training on purely observation to that of including the goal position in the state. The results of using this architecture for the Fetch-Reach environment are shown in Figure 3. The goal position was reset to a random position at the start of each episode. We can see that by including the goal position in the state, the policy learns effectively. In contrast, without the goal
position the policy does not learn at all, further emphasising
the need for an effective meta-learning algorithm. We also
show that using a larger batch-size of 10k gives a less volatile
training profile and converges to a marginally improved
average return than that of a 1k batch size.

For a sample of 18 different Hopper environments, using
the same architecture as described above, we trained experts
for each task over the same number of iterations. We then
executed each policy on all 18 Hopper environments to
investigate how well these policies generalise. A resultant
heat-map is shown in Figure 4, illustrating that the experts
do not generalise well to different Hopper environments.

However, as discussed previously, the Physics in each of
these environments is the same and there are a lot of
similarities between the environments. Thus the experiment
provides motivation for finding an effective meta-learning
procedure to produce policies capable of quick adaptation to
the environment.

For a sample of 18 different Hopper environments, using
the same architecture as described above, we trained experts
for each task over the same number of iterations. We then
executed each policy on all 18 Hopper environments to
investigate how well these policies generalise. A resultant
heat-map is shown in Figure 4, illustrating that the experts
do not generalise well to different Hopper environments.

However, as discussed previously, the Physics in each of
these environments is the same and there are a lot of
similarities between the environments. Thus the experiment
provides motivation for finding an effective meta-learning
procedure to produce policies capable of quick adaptation to
the environment.

VI. REPTILE

Schulman et al. (2018) successfully applied REPTILE to
non-RL problems such as few-shot classification. However,
they note that application so far of REPTILE in an RL
setting has yielded negative results [9].

We investigated using REPTILE as our meta-learning
algorithm on the Hopper environments and trained for
200 meta-updates. The result of training on the Hopper
environments is shown in Figure 5. For comparison,
evaluation of trained experts in each of these environments
yields an average return typically around 2000, indicating
that the resulting policy obtained using REPTILE does not
learn an effective generalisation that can adapt to different
tasks.

Experiments conducted on the Fetch-Reach environment
have so far yielded similarly negative results with one
task often dominating the training. Analysis of REPTILE’s
failure in RL is a current research interest of ours. Under
the current implementation, we believe there are conceptual
flaws:

1) The value network should be directly linked to
the policy that it is used in conjunction with. It is
counter-intuitive that the gradient taken to update
the value network should effectively be an average
of gradients that are calculated using different policies.

2) Classical gradient descent methods and algorithms
such as MAML use true gradients which for non-zero
gradients should guarantee an improvement in the
loss which is being optimised. In contrast, REPTILE
updates in an aggregate gradient direction which does
not have these guarantees.

While we further investigate REPTILE, we looked towards
the use of RNNs to provide an effective meta-learner.

VII. INVESTIGATING USE OF RNNS

A. Motivation

Gated Recurrent Units (GRUs) are a variant of the RNN.
Studies suggest that GRU’s are ‘better than more traditional
recurrent units such as tanh units’ but also comparable to
the often used Long Short-Term Memory (LSTM) [15].

A GRU consists of two ‘gates’: a reset gate and an update
gate. These gates determine how new inputs are combined
with previous memory and what memory we retain. The
basic idea of using a gating mechanism is to learn long-term
dependencies between data points.

We believe that a GRU could act as an effective mechanism
for ‘identifying’ and embedding which task the policy
should be following, given a history of states, actions
and rewards within an environment. Figure 6 shows the
propagation of data through a GRU. At each time step, a hidden state is produced and output. This forms one of the inputs to the GRU at the subsequent time-step. Further, at meta-test time in our demonstration problem, we will provide the policy with a test demonstration from which it must determine what the task is. We believe that a GRU has the potential to process this demonstration and extract an embedding for the task. In the case of Fetch-Reach, the perfect embedding might be the goal position.

B. Implementation

We implemented a GRU as an embedding layer in the Policy and Value networks, subsequently feeding into the fully-connected layers. The existing code was thus modified to consist of:

- A value network with a GRU embedding, followed by three fully-connected layers.
- A policy network with a GRU embedding, followed by three fully-connected layers.
- Tanh() was used as the activation function in this architecture
- TRPO was used to update the value and policy networks

C. Encouraging Efficient Exploration

We hypothesised that by optimising a GRU policy with a zeroed hidden state at the start of each episode, we would force the GRU to learn how to efficiently explore within an episode; once its gets the ‘scent’ of improved reward, it locks on and quickly follows the trail to the goal position. We would thus expect to see that as the weights are updated in each epoch, the goal should be reached more quickly for the same task. We tested this by training on a single goal position and training on 10 different goal positions. We compared the GRU results to that of the baseline architecture. Results are shown in Figure 7 and demonstrate that the GRU policy significantly outperforms the baseline architecture, supporting our hypothesis. In this experiment, a sequence length of 50 was used for the GRU (equal to the episode length).

D. Varying Sequence Length

Inspired by the research conducted by Abbeel et al. in RL² [10], we investigated training with a sequence length equal to two episodes of the same task (100 steps) and propagating the hidden state from the first to the second, We believe that this should encourage learning a hidden state that is a good embedding for the task/goal. Thus, the second episode should receive an embedding of the task and perform better. Abbeel et al. use this method in navigating a 2D maze and often had success in the agent remembering the target’s location and “using it to act optimally in the second episode”. This concept is very appealing for our one-shot learning from demonstration ambition, because the meta-test demonstration would effectively be the first episode, and our imitation would constitute the second episode.

We acknowledge that this method may be likely to get stuck in local minima, being punished for exploration. However, what we desire is not necessarily to perform well in the first episode, but rather to output a perfect embedding to enable high performance in the second episode. We investigated the effect of masking the rewards in the first episode, intending to promote exploration. The results for training on a single task are shown in Figure 8. We see that contrary to our hypothesis, masking the loss of all but the last episode in each sequence yields a worse performance throughout training. We also observe that increasing the sequence length to incorporate multiple episodes slows the initial progress in training. The results for training on 10 different tasks are shown in Figure 9. The effects of masking are still generally negative, however using a sequence length equal to two episodes with the hidden state passed between them outperformed a sequence length equal to a single episode. This supports our hypothesis that experience in the first episode can be used to produce an embedding that enables better performance in episode two.

VIII. Conclusion

The results presented in this report indicate that a GRU mechanism can assist a policy to meta-learn in a reinforcement learning environment. In further research we aim...
to build on these findings to establish a reliable meta-learning algorithm. We will then implement this algorithm in conjunction with GAIL to investigate its ability to meta-learn from demonstrations.

Our research further highlights the weakness of REPTILE when applied to a reinforcement learning environment. We also aim to analyse and explain these shortcomings in our future work. We further present the toy problem challenge of meta-learning a policy that can quickly adapt to new hopper environments with different physical parameters.

Fig. 8: Effects of varying the sequence length and masking all but the final episode in a sequence when applied to a single task.

Fig. 9: Effects of varying the sequence length and masking all but the final episode in a sequence when applied to a 10 different tasks.

REFERENCES


