IFT4055 Project

Mathieu D'Onofrio

May 8, 2024

1 Introduction

A challenge in reinforcement learning is the ability to generalize across different agent types, even if they vary in input and output shapes. Recent work demonstrates that it is possible to achieve this by learning a policy that allows for zero-shot generalization. Our project reexamines this work from a different perspective: that of a universal reward function capable of accommodating various kinds of agents.

In addition to the inherent benefits of learning general reward functions, such as a more streamlined process, there are significant applications in safe reinforcement learning. For example, transferring not only rewards but also constraints from a governing set of agents to a different type of agent can broaden the applicability of these methods.

2 Related Work

This project combines two principal areas of research: methods for inverse reinforcement learning (IRL) and shared policies.

2.1 Maximum-Likelihood Inverse Reinforcement Learning (ML-IRL)

IRL aims to infer the underlying reward functions from observed expert behaviors. Traditional IRL approaches often face significant computational bottlenecks due to their nested structure. This involves an inner loop that determines the optimal policy under a current reward function, and an outer loop that iteratively updates this reward function. These nested loops lead to quadratic time complexity.

To address this, we adopt Maximum-Likelihood Inverse Reinforcement Learning (ML-IRL) (Zeng et al. [2022]). This method benefits from a streamlined single-loop algorithm that mitigates the computational issues mentioned earlier. It also stands out for its simplicity and its capability to derive transferable reward functions. The foundational ML-IRL study explores learning a reward function using expert agent data, which is then applied to train a different agent. This method is limited by its application to agents of the same morphology, albeit with modified capabilities, such as reduced motor responsiveness.

ML-IRL proposes two methods for transferring reward functions:

- **Data Transfer**: This method involves running the ML-IRL algorithm with expert data derived from a different environment.

- **Reward Transfer**: This approach uses ML-IRL to learn the reward function in the appropriate expert environment, then applies this function to train a policy from scratch in a different environment.

Comparative performance assessments indicate that the data transfer method outperforms the reward transfer method, achieving approximately 69% and 58% of expert performance, respectively.

Algorithm 1 Maximum Likelihood Inverse Reinforcement Learning (ML-IRL)

```
1: Initialize reward parameter $\theta_0$ and policy $\pi_0$. Set the reward parameter's stepsize as $\alpha$.
2: for $k = 0, 1, \ldots, K-1$ do
3:     Policy Evaluation: Approximate the soft Q-function $Q_k(\cdot, \cdot)$ by $\hat{Q}_k(\cdot, \cdot)$.
4:     Policy Improvement: $\pi_{k+1}(a|s) \propto \exp\left(\hat{Q}_k(s, \cdot)\right), \forall s \in S, a \in A$.    (Lower-Level Update)
5:     Data Sampling I: Sample a trajectory $\tau_k$ from the dataset $D$.
6:     Data Sample II: Sample a trajectory $\tau_k^A = \{s_t, a_t\}_{t \geq 0}$ from the current policy $\pi_{k+1}$.
7:     Estimating Gradient: $g_k := h(\theta_k, \tau_k) - h(\theta_k, \tau_k^A)$ where $h(\theta, \tau) := \sum_{t=0}^{\infty} \gamma^t \nabla_\theta r(s_t, a_t; \theta)$.
8:     Reward Parameter Update: $\theta_{k+1} := \theta_k + \alpha g_k$.    (Upper-Level Update)
9: end for
```

ML-IRL offers considerable versatility in reward function design. It allows the reward function to be a function of either states and actions (State-Action) or solely states (State-Only). The research suggests that state-only rewards are more effective for predicting optimal policies across varying environmental dynamics or when learning new tasks. This effectiveness stems from the fact that preferences estimated solely based on states are less influenced by the specific dynamics in the expert's demonstration dataset. Additionally, direct imitation of the expert policy could lead to suboptimal outcomes when the learner and expert exhibit different transition dynamics.

2.2 Shared Modular Policies (SMP)

SMP (Shared Modular Policies) (Huang et al. [2020]) represents a cornerstone in the domain of shared policies. This method enhances the generalizability of policies across different agent morphologies by decomposing the policy into separate modules, each corresponding to an individual joint. These modules can then be reassembled in various configurations to suit different agent structures.

Each module operates as a simple neural network that communicates with its neighboring modules. This local message passing mechanism is pivotal for assembling new policies adaptable to various morphologies.

Best performance is achieved when each limb is controlled by two modules: one for bottom-up message passing and another for top-down message passing. The bottom-up modules gather the state (Table 1) of their respective limb and incoming messages from connected limbs, then relay an output message to the neighboring modules. Conversely, the top-down modules receive messages from both the connected limbs and the associated bottom-up module, subsequently generating the action commands and forwarding messages to the connected limbs.

2.3 Amorpheus

One of the challenges with SMP is that their message-passing mechanism is restricted to only adjacent limbs, which can limit the model's flexibility. Additionally, the architecture employed by

Table 1: State Data for Each Limb

Data Type	**Description**
Position	The current spatial position of the limb.
Orientation	The current orientation of the limb in space.
dPosition / dt	The rate of change of position, representing velocity.
dOrientation / dt	The rate of change of orientation, representing angular velocity.
Type	The specific type or category of the limb.
Angle	The current angle between the limb and its adjacent joints.
Joint Range	The range of motion allowed in the limb's joints.

SMP is relatively uncommon. Amorpheus (Kurin et al. [2020]) addresses these issues by encoding the morphology of an agent as a sequence of limb embeddings and utilizing an unmasked transformer to model interactions between limbs. This effectively reimagines message passing as attention within a transformer block, allowing for interactions beyond just neighboring limbs and leveraging a widely recognized architecture.

Amorpheus builds upon the SMP codebase, thereby utilizing the same state vector for each limb as SMP (Table 1). The process begins with a linear projection of this state vector (encoder), which transforms it into an embedding of appropriate dimensionality. These embeddings are then processed through transformer encoder blocks, where message passing occurs via the attention mechanism. After this, the embeddings undergo a final linear projection (decoder) to generate the per-limb actions.

Amorpheus has demonstrated superior performance over SMP across various environments, underscoring the effectiveness of transformer-based architectures in outperforming the modular policies originally proposed by SMP.

2.4 AnyMorph

AnyMorph (Trabucco et al. [2022]) introduces a more adaptable approach to utilizing transformer-based models. Unlike SMP, which is restricted to agents with graph-like structures due to its reliance on neighboring limb interactions, Amorpheus overcomes this limitation. There is no concept of 'neighbors' in Amorpheus, enabling more flexibility. However, Amorpheus still necessitates that sensors and actuators be categorized into specific limbs, limiting the types of reinforcement learning problems it can address. This constraint essentially confines the architecture to agents composed of homogeneous parts, such as well-defined limbs.

AnyMorph seeks to overcome these alignment constraints by assigning an embedding to each individual sensor and motor, thus eliminating the need for predefined limb categorizations. The initial step involves tokenizing sensors and motors, with tokens represented as integers. A robust tokenization method would ensure that different morphologies share semantically meaningful, overlapping tokens. AnyMorph currently employs a straightforward tokenization approach, assigning tokens sequentially to each sensor and motor. For sub-morphologies, if a sensor or motor is absent, the corresponding token is simply omitted. This method is acknowledged as naive, with the intention of exploring more sophisticated tokenization strategies in future research.

With AnyMorph, the architecture becomes more complex due to the lack of direct sensor-to-motor mappings. Consequently, the sequence modeling challenge evolves into a sequence-to-sequence problem where the output sequence may differ in length from the input sequence. AnyMorph leverages the full transformer architecture to address this. Initially, sensor tokens are embedded using a lookup table, similar to word embeddings in language models. The sensor states are then encoded using sinusoidal functions akin to positional encoding in language models. Both the token and state embeddings are combined via a linear projection to produce the final embedding. These sensor embeddings are then fed into the transformer encoder. For generating actions, the corresponding tokens are encoded using lookup tables and processed through the transformer decoder. Finally, the output action embeddings undergo a linear projection to derive the final actions.

AnyMorph achieves superior performance compared to Amorpheus, while offering a more flexible design.

2.5 MetaMorph

MetaMorph (Gupta et al. [2022]) emerged concurrently with AnyMorph and similarly builds upon the foundations of Amorpheus. While it largely adopts the same architecture as Amorpheus, MetaMorph distinguishes itself by incorporating learned position embeddings. These embeddings are derived from the tokens assigned during a depth-first traversal of the agent's kinematic tree, enhancing the model's ability to interpret the structural nuances of different morphologies.

Unlike previous works, MetaMorph utilizes a unique testing ground known as the UNIMAL environment, which features 100 evolved morphologies. This diverse environment allows MetaMorph to demonstrate its robust ability to generalize across a wide array of morphological configurations.

3 Methods

We propose a method for extending Maximum-Likelihood Inverse Reinforcement Learning (ML-IRL) across multiple environments, distinguishing between those with and without expert guidance. Additionally, we introduce a state-only reward function architecture inspired by the AnyMorph framework. We also evaluate two other variations of our proposed architecture.

3.1 Environments

The environments employed in this project are modified versions of the cheetah environments from prior shared modular policies research. This includes 15 distinct cheetah morphologies.

These environments have been updated to be compatible with the latest version of Gymnasium. To streamline our approach and align more closely with the original MuJoCo environments, we eliminated the per-limb feature engineering typically required in limb-aligned studies. Consequently, in the full cheetah morphology scenario, the observations directly mirror those found in the standard HalfCheetah environment of Gymnasium.

3.2 Expert Trajectories

To generate expert trajectories for each environment, we trained a soft actor-critic (SAC) model equipped with a straightforward two-layer MLP architecture. We then collected 10 trajectories from each environment, incorporating a modest level of exploration noise.

For validation purposes, we used the results from AnyMorph as a benchmark to ensure that our expert-generated trajectories achieve desirable returns. This serves as a sanity check for the efficacy of our trained experts.

3.3 Extending ML-IRL for shared reward functions

When adapting ML-IRL for multiple morphologies, we employ a strategy where we alternate between updating each policy, taking turns to update the reward function. This method allows each morphology-specific policy to refine its behavior iteratively.

Algorithm 2 Multiple Maximum-Likelihood Inverse Reinforcement Learning (Multi ML-IRL)

```
 1: Initialize reward parameters $\theta_{i,0}$ for each expert $i$. Initialize policies $\pi_{i,0}$ for $i = 0, 1, \ldots, E+N-1$.
    Set the stepsize $\alpha$.
 2: for $k = 0, 1, \ldots, \mathrm{K} - 1$ do
 3:   for $i = 0, 1, \ldots, E - 1$ do
 4:     Policy Evaluation: Approximate soft Q-function $Q_{i,k}(\cdot,\cdot)$ by $\hat{Q}_{i,k}(\cdot,\cdot)$
 5:     Policy Improvement: $\pi_{i,k+1}(a|s) \propto \exp\left(\hat{Q}_{i,k}(s,\cdot)\right), \forall s \in S, a \in A$.
 6:     Data Sampling I: Sample a trajectory $\tau_{i,k}$ from the dataset $D_i$.
 7:     Data Sample II: Sample a trajectory $\tau_{i,k}^A = \{s_t, a_t\}_{t\geq 0}$ from the current policy $\pi_{i,k+1}$.
 8:     Estimating Gradient: $g_{i,k} := h(\theta_{i,k}, \tau_{i,k}) - h(\theta_{i,k}, \tau_{i,k}^A)$, $h(\theta, \tau) := \sum_{t=0}^{\infty} \gamma^t \nabla_\theta r(s_t; \theta)$.
 9:     Reward Parameter Update: $\theta_{i,k+1} := \theta_{i,k} + \alpha g_{i,k}$.
10:   end for
11:   for $i = E, E+1, \ldots, E+N-1$ do
12:     Policy Evaluation: Approximate soft Q-function $Q_{i,k}(\cdot,\cdot)$ by $\hat{Q}_{i,k}(\cdot,\cdot)$.
13:     Policy Improvement: $\pi_{i,k+1}(a|s) \propto \exp\left(\hat{Q}_{i,k}(s,\cdot)\right), \forall s \in S, a \in A$.
14:   end for
15: end for
```

For our transfer learning approach, we categorize our environments into two groups: one with environments that possess expert trajectories and one without. When cycling through the environments, if an environment lacks an expert trajectory, we omit the reward function update for that particular policy's trajectories. This approach deviates from the data transfer method outlined in the original ML-IRL paper, where a gradient update on the reward function using expert data from a different morphology is typically performed. Thus, our method (Algo 2) serves as a hybrid between traditional reward transfer and data transfer techniques.

3.4 Deriving a reward function architecture from AnyMorph

To adapt the AnyMorph policy architecture for deriving a reward function, we need to address two significant differences. Firstly, unlike the output of a variable-length sequence of actions in the original AnyMorph architecture, the reward function architecture outputs a single scalar value. Secondly, since we have access to the action vector, this information can be incorporated into our action embeddings in the same way as the states.

For the action embeddings, we employ the same methodology as used for our state embeddings. To transform our model from a sequence-to-sequence format to a sequence-to-reward format, we implement a pooling operation over the sequence. This is followed by a linear projection to reduce the dimensionality to a single scalar value, representing the reward. This adjustment allows the model (Fig 1) to efficiently process and output a reward based on the aggregated information from the entire sequence.

For our project, we prefer to utilize state-only reward functions, as they are expected to enhance generalization, as previously discussed. Consequently, we have decided to omit the decoder component of the transformer, effectively transitioning to an architecture that solely consists of a transformer encoder. This modification not only simplifies the architecture (Fig 2) significantly but also reduces the number of parameters. This is particularly beneficial for reward functions, which are typically compact and require fewer parameters to function effectively.

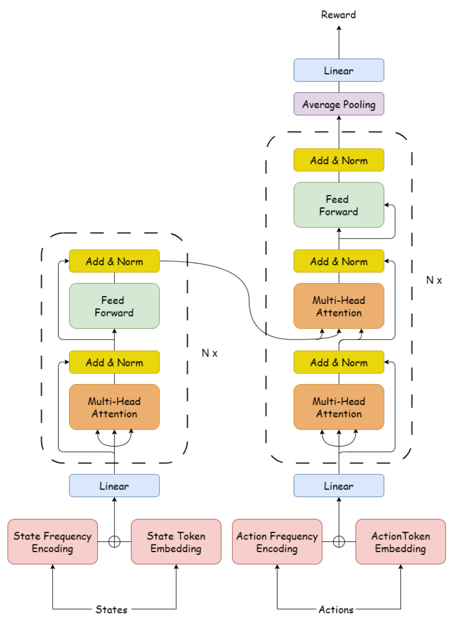

Figure 1: State-Action Reward Architecture

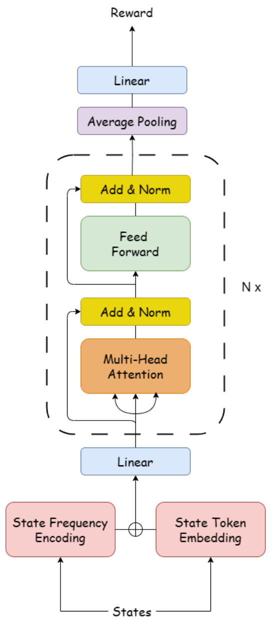

Figure 2: State-Only Reward Architectures

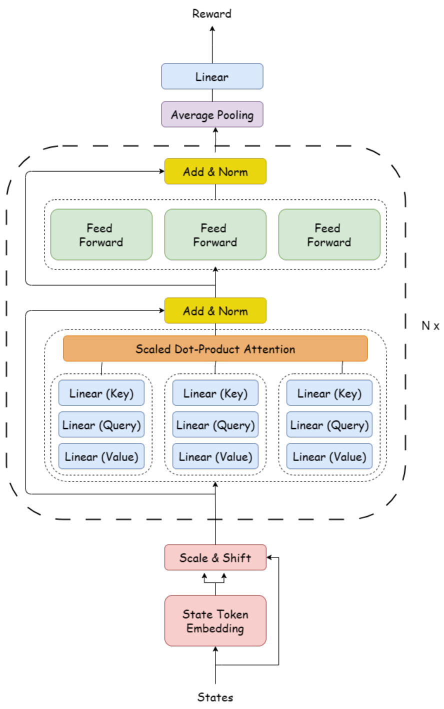

Figure 3: Heterogeneous Reward Architecture

Additionally, we have developed an alternative architecture in which we replace the sinusoidal frequency encoding with a simple scale-and-shift operation. This modification is designed to test the necessity and effectiveness of frequency encoding in our model. By comparing the two, we aim to determine whether the traditional frequency encoding approach provides significant benefits or if a simpler method can achieve comparable results. This is not the objective of our project, but something to try.

3.5 The case for heterogeneous architectures

While exploring alternate architectures falls outside the main scope of our project, it could be an fun and interesting area to explore.

Sensors in our model may vary widely, including measurements in meters, radians, meters per second, radians per second, and sometimes even categorical data. In the standard architecture, all these diverse sensor types are processed into a common embedding using the same weights. However, we hypothesize that assigning each sensor token its own set of weights might be more effective.

To explore this, we propose testing a heterogeneous transformer architecture (Fig 3). In this model, each token—representing a different sensor type—would have its own dedicated set of weights, keys, queries, values, and feed-forward networks. The sensors would still interact through the standard attention mechanism, but their representations would not be shared across different tokens. This approach aims to bring the transformer architecture closer to the concept used in shared modular policies, where each limb had its own dedicated neural network.

We also propose regularizing the ML-IRL model by applying variance regularization to the pooling layer, penalizing a larger variance. This strategy is designed to compensate for the increase in parameters. The underlying idea is to encourage each sensor to contribute towards learning an approximation of the actual reward, ensuring that every element within the system actively participates in the prediction process. This regularization aims to enhance the robustness and accuracy of the reward predictions across different sensor inputs.

4 Experiments & Results

For our experiments, we begin by learning individual MLP-based reward functions for each morphology, which serve as baselines. While these baseline performances, fine-tuned to specific experts, are not necessarily expected to be matched, they provide a valuable reference for evaluating the effectiveness of our shared reward function. In reporting results, we calculate the average returns for the set of cheetah morphologies under study and present these as a percentage of the average returns achieved by experts.

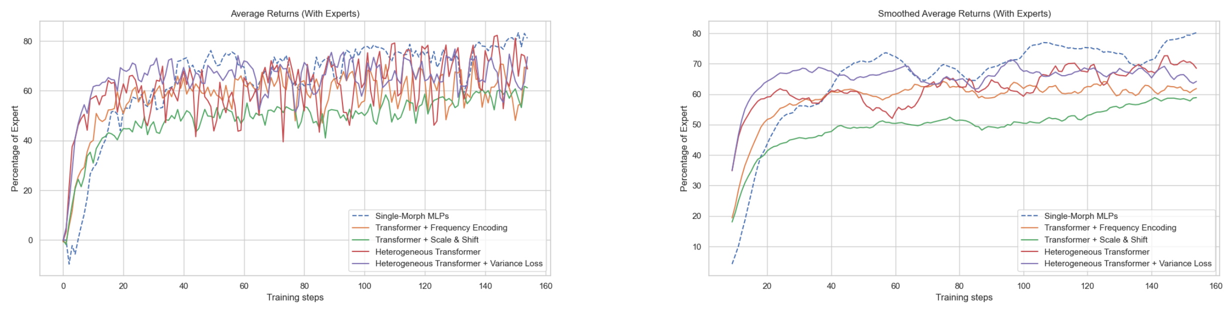

Figure 4: Cheetahs With Experts

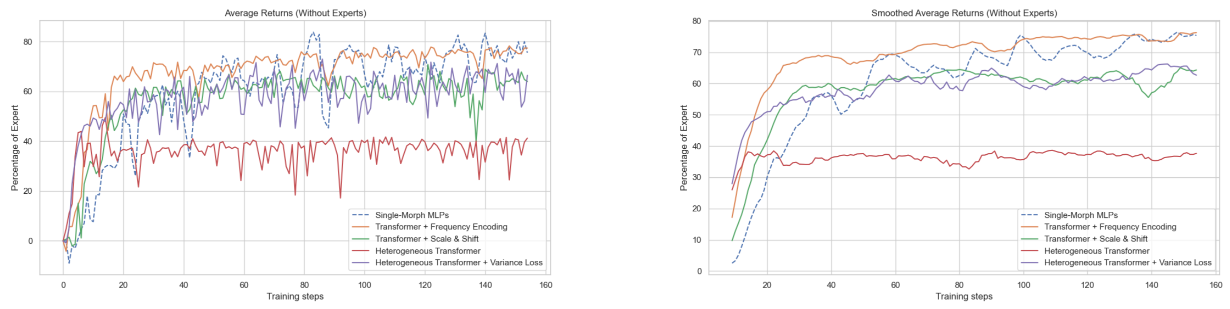

Figure 5: Cheetahs Without Experts

Our results are illustrated in two plots: the first (Fig 4) displays the average returns of the cheetah morphologies that had expert trajectories (12 morphologies), while the second (Fig 5) shows the average returns for the cheetahs without expert trajectories (3 morphologies), where reward transfer was implemented. Due to the constraints of time, we were only able to run these experiments once before preparing this report, leading to potentially unstable results. Therefore, we refrain from making definitive conclusions about the superiority of any architecture based on this single trial. Additionally, we provide smoothed versions of these plots to enhance their readability. Completing these experiments can require several days.

5 Material & Technologies Used

The project was written in Python and primarily utilized the following libraries:

- **PyTorch**: An open-source machine learning library.
- **Gymnasium**: A toolkit for developing and comparing reinforcement learning algorithms, providing a standard API for interacting with a wide range of environments.
- **Tianshou**: A high-performance reinforcement learning library built on top of PyTorch, designed to provide a clean and efficient implementation for RL researchers.
- **Hydra**: A framework for elegantly configuring complex applications, which simplifies the management of configurations and overrides from the command line.

The experiments were conducted on a computer located in Glen's lab.

6 New learned skills

This project served as my introduction to the field of reinforcement learning, where I explored advanced topics such as inverse reinforcement learning. It also provided an opportunity to understand the complexities of sharing models across various agent types, prompting me to engage with numerous niche articles on the subject. Additionally, I gained practical experience with transformer models in a unique context, further enhancing my understanding of these models and exploring many methods for creating embeddings. This experience not only deepened my knowledge of reinforcement learning and machine learning techniques but also honed my skills in applying these concepts to solve ongoing challenges.

7 Conclusion

In conclusion, this project demonstrates the feasibility of learning a common reward function for multiple morphologies. Moreover, it shows that such a reward function can effectively generalize to morphologies that were not seen during training.

References

Agrim Gupta, Linxi Fan, Surya Ganguli, and Li Fei-Fei. Metamorph: Learning universal controllers with transformers. *arXiv preprint arXiv:2203.11931*, 2022.

Wenlong Huang, Igor Mordatch, and Deepak Pathak. One policy to control them all: Shared modular policies for agent-agnostic control. In *International Conference on Machine Learning*, pages 4455–4464. PMLR, 2020.

Vitaly Kurin, Maximilian Igl, Tim Rocktäschel, Wendelin Boehmer, and Shimon Whiteson. My body is a cage: the role of morphology in graph-based incompatible control. *arXiv preprint arXiv:2010.01856*, 2020.

Brandon Trabucco, Mariano Phielipp, and Glen Berseth. Anymorph: Learning transferable polices by inferring agent morphology. In *International Conference on Machine Learning*, pages 21677–21691. PMLR, 2022.

Siliang Zeng, Mingyi Hong, and Alfredo Garcia. Structural estimation of markov decision processes in high-dimensional state space with finite-time guarantees. *arXiv preprint arXiv:2210.01282*, 2022.