Explainable Reinforcement Learning with Linear Model U-Trees

Fabrizio Micheloni
Technical University of Munich, Germany
fabrizio.micheloni@tum.de

Musfira Naqvi
Technical University of Munich, Germany
ga58jeq@mytum.de

Abstract

This paper discusses the development of the mimic model of the Deep Q network in order to enhance the interpretability of the networks sequence of decisions in the field of Reinforcement Learning (RL).
For this, we have used (Liu et al., 2019) work to create a mimic model which matches the predictions of a highly accurate model. The paper distills knowledge from a deep model to a mimic model with tree representation. This tree representation is called the Linear model U Tree (LMUT) as it has added a linear model to each leaf node. Our work reproduces the technique and adds a novel approach for evaluating the trained LMUTs. Furthermore we explore the issues related to performance and propose an architecture for scaling LMUT to more complex scenarios.

1 Introduction

As neural networks succeed in solving more complex tasks, another problem arises at the same time: It becomes difficult for the user to understand the model's inner structure. In other words, we are being confronted with a so-called "black-box-problem": The neural network will present us an output with a given input, however it's internal workings remain unkown to us. This makes it hard to retrace the model's predictions and decisions that eventually lead to it's final output. The "black-box-problem" also applies to Deep Reinforcement Learning (DRL).
In order to solve this problem one can apply mimic learning:
Mimic learning helps us to train an interpretable mimic model so that it's predictions will match that of a highly accurate model. Amongst other approaches, decision trees help to mimic the function of a deep neural net with complex structures. By passing input to a complex deep neural network we can then collect soft output labels. Afterwards, the decision tree can then be trained with the soft output as a supervisor in order to improve its own predictions.
Because of that we used the U-Tree as our next lead, as it is a reinforcement learning method that represents the Q function using a tree structure. (Liu et al., 2019) work provided us with the algorithm for the creation of a mimic model, which is also referred to as a Linear Model U-Tree (LMUT).
In this work, not only will we discuss how a LMUT is being created for a Deep Q network, but we also explain how the algorithm was improved in order to perform faster.

2 Technologies used

For the framework we have decided to use OpenAI Gym (Brockman et al., 2016) and Stable Baseline (Hill et al., 2018) as our framework as it is very intuitive and straightforward.

OpenGym AI is a toolkit that helps developers to run and train RL applications by offering interfaces for some well known RL task simulators (such as Mountain Car, Atari games...).

Stable Baselines is a set of implementations of Reinforcement Learning algorithm based on OpenAI Baselines which offers standardization (pre-trained models).

3 Algorithm Structure

In order to create the LMUT, following steps are required:

- Data Generation
- Data Gathering
- Node-Splitting

3.1 Data Generation

The active play setting helps with the creation of mimic data by letting a mature deep reinforcement learning model interact with an environment. Our active learner has three components: (q,f,I).

The first component q is query function that controls the active learner's interaction with the environment. This means it must consider the balance between exploration and exploitation. For this the ϵ-greedy schemes comes in handy as it refers to the probability of choosing to explore.

The second component f is the deep model that produces the q values: $f : (I,a) \rightarrow range(\hat{Q})$. For a given observation signal I on time step t, we select an action a (with the query function q). With the specification of a and I we can now compute our soft output Q values which was received with $f(I,a)$.

After performing action a, the environment provides a reward r and the next state observation I_{t+1}. We then record this transition (as we see in Figure 1).

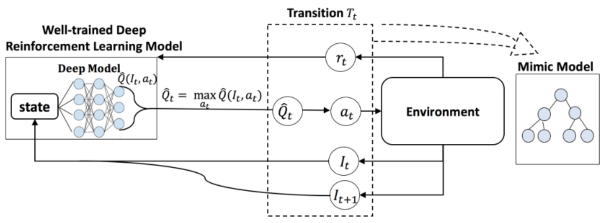


Figure 1: Active Play Setting

We then set I_{t+1} as the next starting observation signal and repeat the above steps until we have training data for the active learner to finish sufficient updates over the mimic model.

3.2 Data Gathering

Initially we observed a continuous observation space. As one can see in Figure 2, each action within our action space has its own sub-tree.
With each split of the tree's node, the tree basically partitions the space into discrete portions. This can then be identified within the leaves. For example, leaf number 5 represents the discrete portion of observation f_1 between 0.3 and 0.8 excluded. At the same time it also shows the observation f_2 which includes every value until 0.6 (Figure 3).

Finally all the transition instances that were created during the interaction between the DQN and environment will be passed to their corresponding leaves in the tree. As can be seen in the example displayed in Figure 4, the transition instance (belonging to action 2) has the observation values $f_1 = 0.7$ and $f_2 = 0.5$. Therefore we navigate this transition through the LMUT by applying distinctions for every node until a leaf is reached.

3.3 Node Splitting

After the insertion of thousands of transition in the right position within the tree, we can now begin with the node splitting phase. It is important to recall, that each leaf has an integrated linear model. This linear model will be updated with the help of the Stochastic Gradient Descent. If one leaf achieved sufficient improvement, we determine a new split and the resulting leaves will then be added to the current partitioning cell.

For every node that is supposed to be split (in Figure 5 it is leaf l_2), first all of the distinctions are determined with which we can split our leaf. With

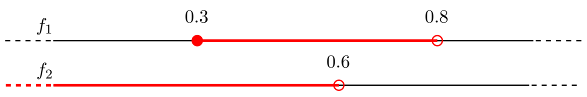


Figure 3: Feature Space of Leaf 5

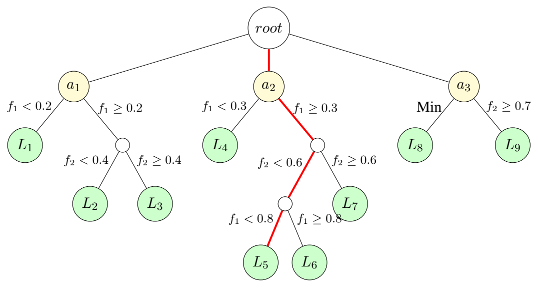


Figure 4: Passing transitions down to a leaf node

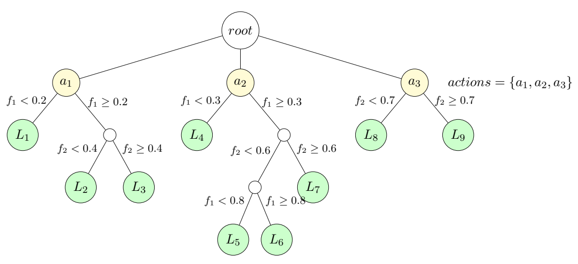


Figure 2: Data Gathering Phase

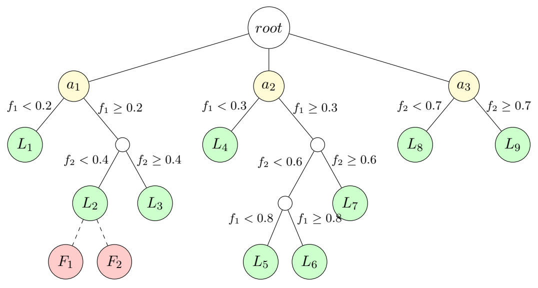
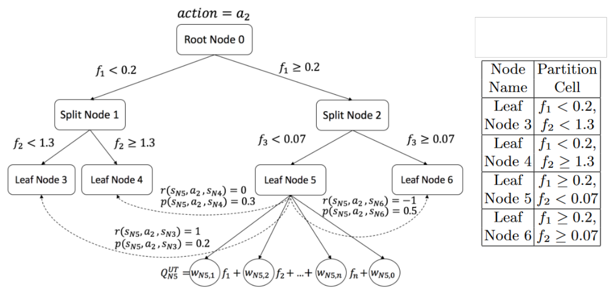


Figure 5: Splitting of node L_2 into two fringe nodes

	Feature	Influence
MountainCar	Position	4512.497
	Velocity	7574.625
CartPole	Cart Position	1.69
	Cart Velocity	70.367
	Pole Angle	8.796
	Pole Velocity	225.267

Table 1: Overall results for feature importance for each game's features

the help of the splitting criterion we can finally chose the best distinction with which we can split node int two fringe nodes.



$action = a_2$
Root Node 0
$f_1 < 0.2$
$f_1 \geq 0.2$
Split Node 1
Split Node 2
$f_2 < 1.3$
$f_2 \geq 1.3$
$f_3 < 0.07$
$f_3 \geq 0.07$
Leaf Node 3
Leaf Node 4
Leaf Node 5
Leaf Node 6
$r(s_{N5}, a_2, s_{N4}) = 0$
$p(s_{N5}, a_2, s_{N4}) = 0.3$
$r(s_{N5}, a_2, s_{N6}) = -1$
$p(s_{N5}, a_2, s_{N6}) = 0.5$
$r(s_{N5}, a_2, s_{N3}) = 1$
$p(s_{N5}, a_2, s_{N3}) = 0.2$
$Q_{N5}^{UT} = w_{N5,1} f_1 + w_{N5,2} f_2 + ... + w_{N5,n} f_n + w_{N5,0}$

Node Name	Partition Cell
Leaf Node 3	$f_1 < 0.2$, $f_2 < 1.3$
Leaf Node 4	$f_1 < 0.2$, $f_2 \geq 1.3$
Leaf Node 5	$f_1 \geq 0.2$, $f_2 < 0.07$
Leaf Node 6	$f_1 \geq 0.2$, $f_2 \geq 0.07$

Figure 6: Linear Model In Our Leaves

As we can see in Figure 6, the q value of node 5 is being calculated with the linear combination of the feature weights of the model. With this, we can explain the prediction with the feature weights of the model.

4 Feature Influence

It is important to mention, that in LMUT we use feature vales as splitting thresholds to form partition cells for input signals. The question is: How are we supposed to compute the influence of these splitting features? During the node-splitting phase, the intermediate nodes and the leaves store the q values - which were retrieved from the transitions. Afterwards we can simply calculate the influence of the splitting feature with the help of Formula 1:

$$Inf_f^N = (1+\frac{|w_{Nf}|^2}{\sum_{j=1}^{J} |w_{Nj}|^2})(var_N - \sum_{c=1}^{C} \frac{Num_c}{\sum_{i=1}^{C} Num_i} var_c) \quad (1)$$

In order to then finally calculate the final influence of a splitting feature, we can simply sum over all the feature influences for every node N.
In table 1 we can see the results for the feature importance values.

5 Model Evaluation

The whole point of training a LMUT is to mimic as precisely as possible the original DQN. If we can guarantee that the mimic model would take more or less the same actions of its counterpart, or, in other words, predict the same q-value in a specific partition, then we can expect that explanations are reliable. Unfortunately it is not that simple to find such guarantee. In fact, giving a definition of performing "well" is already a challenge. Let's consider the games the tried to mimic: how can we say that in Mountain Car the LMUT agent is playing well?

The first approach we tried was plotting the predicted q-values both the DQN the LMUT. The assumption we had was that the DQN would have given an upper bound for the LMUT values. That is exactly what we achieved. Unfortunately this does not give us any hint about the question we introduced a few lines ago: is the LMUT performing well? The second approach was more subjective and involved the manual study of the model by watching them playing. At this point we had the confirmation that LMUTs were actually trained correctly for the respective game. But again this didn't give us any objective number that could help us to quantify the performance of the LMUT.

The last method we studied partially answers the question. The approach is slightly different: instead of just comparing the final LMUT and the DQN, why don't we save intermediate checkpoints of the LMUT during training and try to understand if it became better or worse? Before explaining the method for doing so, let's introduce the concept of Relative Absolute Error (RAE). RAE explains how much difference there is between the q-values of the two models, as shown in Formula 2.

$$RAE_n = \sum_{t \in n} \frac{|q_{dqn}(t) - q_{lmut}(t)|}{\sum_{i=1}^{n}(count(t \in n))} \quad (2)$$

The absolute value of the subtraction is normalized by the sum of transitions up to game i. If we apply RAE to only one generation we suffer from the same problem encountered with the first solution, but if we apply it to many generations we have a hint of whether during the training the LMUT was actually improving. Ideally, we need a generation whose RAE is a line as close as possible to the x axis. An example of how such comparison if made is provided in Figure 7 (not based on actual data since we didn't didn't keep checkpoints during training). In blue we see the performances of our model. If a model has a worse performance, then we expect it to be an upper bound of the blue line. The red line is an example of such scenario. On the other hand, when a generation has better performances, then we expect it to be a lower bound: the green line shows a generation that is indeed better and reflects this very last scenario.

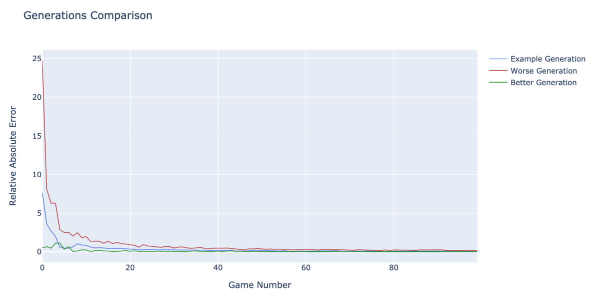


Figure 7: RAE comparison for three different generations of the same game

6 Training Optimization

Even though LMUT is a powerful technique for mimicking a complex DQN model, the training phase has some performance issues that make it hardly applicable to many Reinforcement Learning scenarios. In fact, the more the action space and observation space become wider, the more RAM, CPU and GPU are needed since more transitions need to be generated and handled. Furthermore, the more the tree grows in height, the more nodes should be trained with gradient descent. If we also consider that in the node-splitting phase we have to inspect several candidates per leaf (the fringes), the performance issues become clearly an overwhelming problem that might invalidate the whole technique. Here we describe some techniques for boosting performances at instance-wise level (single machine) and an architecture for scaling the algorithm in a distributed fashion (multiple machines).

6.1 Instance-Wise Optimization

If we consider instance-wise optimization we can mostly focus on CPU and GPU since the memory needed for the training cannot be reduced that much. Let's discuss what each improvement individually.

6.1.1 Node skipping

Here the idea behind is very simple: just skip the node splitting phase for nodes with less than k (new hyperparameter) nodes. This was particularly effective because only after only a few iterations the tree starts having tens of leaves that contain a handful of transitions. Clearly, this is not problematic by itself, however, at every iteration the algorithm updates the weights of each node and computes the Splitting Criterion for each fringe (which becomes very expensive). Hence, we mitigated this problem by introducing another hyperparameter that is used as a guard in front of each node: if there aren't enough transitions, skip the node. A positive side effect is that it helps preventing overfitting, acting as a regularization tool.

6.1.2 Lazy Nodes

We also introduced a new concept, namely the Lazy Node (LN). A LN is a node that after being inspected has not been splitted. Then, we rely on the assumption that a node that has not been splitted will not have many chances to be splitted in the next iteration, since it takes many iterations for gathering a meaningful number of transitions that can change the result of the Splitting Criterion. Hence, We wait s (new hyperparameter) iterations before checking it again. Similarly to the previous technique, also LNs work as regularization mechanism for avoiding overfitting.

6.1.3 Parallelization of the node splitting phase and data generation phase

For achieving parallelization we introduced a Thread Pool Executor that exploits multiple cores of the processor for generating games at the same time (generation phase). Similarly, the weights update and Splitting Criterion computation are parallelized. More specifically, several nodes are inspected at the same time (node splitting phase).

6.1.4 Max depth stopping criterion

If a leaf reaches the given depth d (new hyperparameter) the algorithm stops inspecting it and it is not splitted again.

6.2 Distributed Training

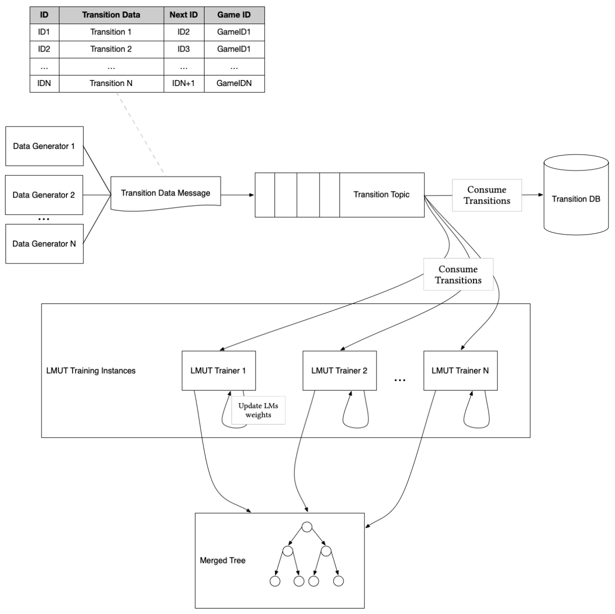
Even though the instance-wise optimizations showed in the last section are making the overall training faster, the algorithm presented in the paper can be scaled only vertically because of the highly coupled subroutines. Due to the rapidly increasing number of transitions generated, RAM can saturate very quickly, making LMUTs hard to use in more complex scenarios. The solution that we present here aims to overcome such limitations with a distributed and scalable architecture. Before we jump into the details, let's repeat how the classic LMUT training algorithm is structured. First we have the Data Generation phase in which transitions are drawn from the RL model that we want to mimic. Then comes the Data Gathering phase, where we insert the transitions into the leaves and update the values that turn the LMUT into an MDP/Agent. The last part is the Node Splitting phase, in which weights of the Linear Models are updated and nodes are splitted. Now we show how we rearranged the phases for making them fit into our custom distributed architecture.

6.2.1 Data Generation and Node Splitting

As far as the Data Generation and Node Splitting phase are concerned (Figure 8), the architecture follows an online setting: the data is generated and the same time we have weights update and node splitting. The first boost concerns the Data Generators. This becomes very simple because simulations can be executed in parallel and resulting transitions are published onto a topic. To each transition is also assigned a unique ID, the ID of the following transition and the ID of the game (simulation) it belongs to. The reason why we need those information will be made clear later. In parallel, other components of the architecture can consume transitions for making their job. The first consumer is a database that stores transitions for later use. A NoSql may be suitable here due to the non-transactional scenario.

Transitions are also consumed by many different LMUT Trainers which store a partial representation of the tree (Figure 9). Sadly, here we also have to introduce some redundancy: each Trainer consumes every single transition and discards those who do not fit into its partition space. Periodically the Trainers perform gradient descent on the leaves and try to split them.

This architecture solves completely the RAM problem, since when an instance is close to getting saturated we can just split it into two other instances. This implies that transitions get splitted too. Ideally, an instance split divides evenly the amount of transitions stored. This can be achieved by augmenting the tree data structure. Each node can keep a count of the number of transitions on



ID	Transition Data	Next ID	Game ID
ID1	Transition 1	ID2	GameID1
ID2	Transition 2	ID3	GameID1
...	...	...	...
IDN	Transition N	IDN+1	GameIDN

Figure 8: Data Generation and LMUT training scalable architecture

the left and on the right. That way, the best split can be found in $\mathcal{O}(d)$, where d is the depth of the partial representation of the tree.

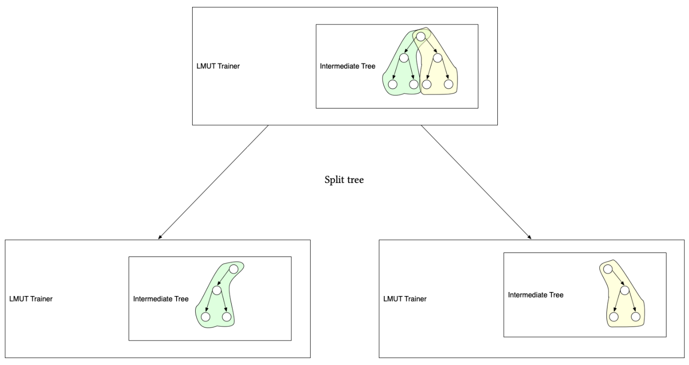


Figure 9: Example of how the tree is divided into partial representations which can be handled by different training instances

6.2.2 MDP/Agent Training

At this point we have a trained LMUT in which each leaf has weights that explain what features are most important according to each partitions of the input space. However, we can't use it as a mimic Agent nor as a MDP because we are missing some information, namely the rewards, the transition probabilities and the q-values. The process is shown in Figure 10.

As said before, all generated transitions were stored into a database. Now we want to make them available so that we can turn our trained LMUT into a MDP and mimic Agent that can replace the mimicked RL model. We can also generate more transitions since this does not change the final result. Now we make use of a queue instead of a topic, since we want to ensure that only one instance consumes each message. Clearly, differently from the previous section, there is no redundancy with respect to the consuming of messages. Messages are published as entire games, each containing the respective transitions (we can retrieve a game from the NoSql database by using its unique Game ID). This is necessary because in that phase we need to know the leaf to which a partition belongs at time t and also the leaf of the following transition at time step $t + 1$. The training is made very efficient because we can replicate the trained LMUT obtained in the previous section multiple times. Partial solution are updated until all available data is consumed. We end simply by merging all the partial solutions. During this phase we don't have memory issues, so the parallelization makes the process only faster.

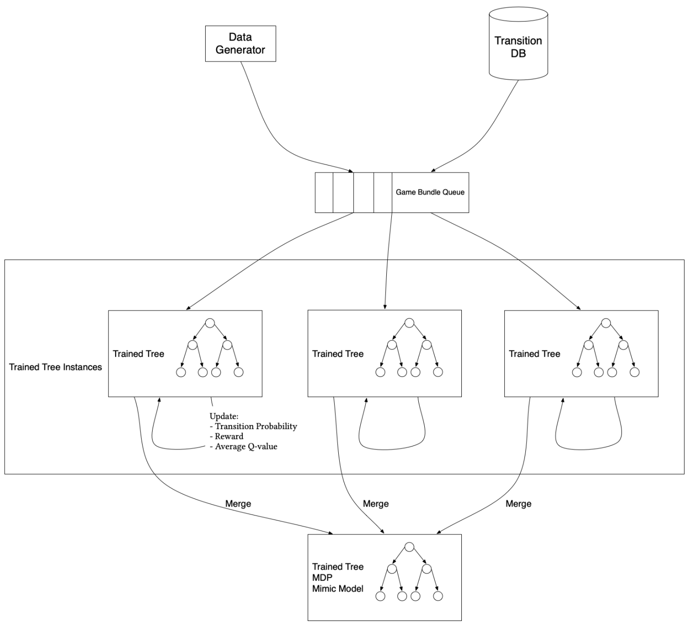


Figure 10: Distributed training of the MDP and mimic Agent

References

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. 2018. Stable baselines. `https://github.com/hill-a/stable-baselines`.

Guiliang Liu, Oliver Schulte, Wang Zhu, and Qingcan Li. 2019. Toward interpretable deep reinforcement learning with linear model u-trees. In *Machine Learning and Knowledge Discovery in Databases*, pages 414–429, Cham. Springer International Publishing.