Competitive RL agents with early exit networks GUT - Greater Use of Time

1st Trym Bø
Department of Informatics
University of Oslo
Oslo, Norway
tryb@ifi.uio.no

1st Evan Jåsund Kassab
Department of Informatics
University of Oslo
Oslo, Norway
evanjk@ifi.uio.no

1st Herman Pareli Nordaunet
Department of Informatics
University of Oslo
Oslo, Norway
hermanno@ifi.uio.no

2nd Ulysse Teller Masao Côté-Allard
Department of Informatics
University of Oslo
Oslo, Norway
ulysseca@ifi.uio.no

3rd Frank Veenstra
Department of Informatics
University of Oslo
Oslo, Norway
frankvee@ifi.uio.no

***Abstract*—When designing a neural network we have to decide how deep the network should be. The more layers, usually yields better accuracy, but often at the cost of a much larger computational time. Through our model, we seek to take inspiration from human biology and let Reinforcement Learning (RL) agents learn to adjust their computation time based on the given task. To enable this, we propose the network GUT - Greater Use of Time, that implements an early exit network to emulate humans ability to adapt to the current task and context.**

***Index Terms*—Deep Reinforcement Learning, Adaptive Agent, Early-Exit Neural Networks**

I. Introduction

State-of-the-art deep neural networks has proven to be able to solve complex tasks with super human accuracy [1] [10] [11]. Combining the incredible powers of deep neural networks with with the principles of reinforcement learning has also given great results [2]. However the trend of ever growing networks require a huge amount of computational power. As the usage of deep learning grows, so does the demand for networks to run fast on small, cheap devices like watches, IoT-devices and mobile phones, without loosing the state-of-the-art accuracy. Drawing inspiration from the concepts brought up in *Thinking Fast and Slow* by Daniel Kahneman [3], we introduce GUT - Greater Use of Time, a neural network which intends to reduce over-computation on easier tasks, while still achieving the same degree of accuracy as the full network.

To achieve this, GUT is an early exit neural network. When it is used in reinforcement learning agents, the goal is that it will teach the agents to adapt to the problems they are facing, while not using more computational power than necessary, by exiting the network when it is confident enough in its prediction. This approach is meant to benefit the agent in several ways: (i) when an agent is faced with a simple or known task it has high confidence of completing correctly after a few layers, it will exit the network and execute the action calculated, and saving valuable computation time. (ii) When an agent is faced with a difficult or less known task, the agent's GUT-network will go deeper into the network until the confidence is high enough. This is continued until the full network has been used, and the final action has been calculated. This will give the agents facing difficult tasks the same accuracy as with a full network. (iii) Normal strategies for reducing the computational cost of deep neural networks are optimizing the design of the network by reducing the amount of convolutions, nodes or edges, or by using different training strategies. These methods are often difficult and time consuming to develop. The GUT-net instead uses early exit, a method that is easy to apply on deep neural networks independent of network architecture, size and complexity. And it can run in tandem with other optimization attempts.

The agents with the GUT-network exists in a multi-agent environment and are faced with different tasks. The environment is highly competitive due to agents being able to interact with both objects and other agents. Therefor implementing different levels of complexity in terms of fast and slow thinking will explore advantages and disadvantages in the models dependent on the hyper parameters of the environment, and the composition of agents interacting with it. Several different types of agents in the same environment will also help make concrete evaluations on how they perform, and if the complexity of the agent is preferable.

II. Related work

There have been a lot of creative work on reducing computational cost for deep neural networks. Othman et al. [4] have experimented on offloading complex computations to nearby fiber channel servers through WiFi or cellular communication. A well tested method is completion reduction of the network it selves by structured sparsity in neural networks to remove irrelevant nodes, edges or convolutions from the network [7]

[8] [9]. This can achieve a reduction in computation while maintaining the accuracy of the network.

Work on reducing computational cost has also been done on reinforcement agents. Watters et al. [5] used curiosity driven exploration to increase robustness against over computing of task irrelevant perturbations. This lead to a great reduction in computational requirement to achieve the same result as other state of the art reinforcement learning methods.

The topic of adaptive early exit networks have also been experimented on a lot in recent times. Bolukbasi et al. [6] makes an early exit network by combining three state of the art image recognition networks of different complexity. The network passes the task on to a more expensive network if it is not confident enough with the answer. This approach solved the ImageNet image recognition challenge with a 280% speed up at 1% top-5 accuracy loss. Scardapane et al. [12] also highlights the benefits of early exit networks and their natural way into the next generation of neural networks. They also explores routing and offloading tasks to more expensive networks either local or remote as methods of early exit.

III. Methods

A. *GUT - Adaptive computational usage*

In this paper, we introduce the algorithm GUT: Greater Use of Time. The main motivation of GUT is the ability to create an adaptive agent that can choose a suited level of computational cost and time for predicting the task correctly. In simple terms we want to either think fast and exit the network early, or go deeper into the network and exit after more computing. The adaptive early exit network GUT is implemented using a predefined amount of exit branches after specific layers. The architecture of such a early exit network can be seen in Figure 1. The first step of GUT is to train the full network with reinforcement learning based on the agent's interaction with the environment. Then we need to train the early exit policy at each early exit branch. We start by training the latter of the exit branches, and work our way up in the network, until all the exit branches has a trained exit policy. In the exit policy the user has some control of how confident the exit policy has have before making a premature prediction. This threshold should be high enough to not loose too much of the accuracy that the full network is able to achieve.

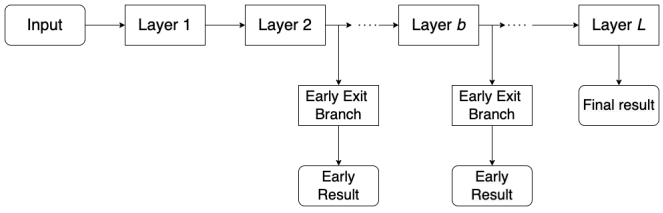

Fig. 1. Schematic illustration of an Early Exit architecture.

B. *The test environment*

The environment used to visualise, train and evaluate our proposed algorithm is called Food Collector [13], assembled by MLAgents. MLAgents is an open-source package built up by 3D worlds, designed to serve as templates for new environments or to test new ML algorithms. MLagents uses Unity to render the 3D graphics. In the Food Collector environment we use in this paper the goal for the agents is to interact with as many green food spheres as possible while avoiding red spheres. There are two types of rewards given to the agents in the environment:

$$R = \begin{cases} 1 & \text{Interact with } \textit{green} \text{ sphere} \\ -1 & \text{Interact with } \textit{red} \text{ sphere} \end{cases} \quad (1)$$

Each agent have the same behavior parameters which consists of the (i) actions and the (ii) Vector Observation space. There are 4 types of actions, 3 continuous actions and 1 discrete action. The continuous actions describe the movement of an agent, they can either move forward, to the side or rotate. The discrete action is used to control a laser each agent has. If an agent is hit by a laser, it freezes for a predefined amount of time.

The Vector Observation space is a set of 53 vectors representing the velocity of the agents, whether the agent has been hit by a laser and is frozen and/or is shooting its laser as well as a grid based perception of the objects around the agents forward direction.

Further down the road, we might also seek interest in using the (iii) Visual Observations that provides a First-Person camera of the agent as well as vector flag representing the frozen state of the agent.

C. *Python Low Level API*

To establish a communication between Python and Unity, MLagents' own Python Low Level API is used. This allows us to interact directly with the Unity Environment. By using this, we are able to retrieve data from the agent such as observations and rewards. From here we can pass the observation of the agent into the network which will give the agent an action. After an action is completed, a new cycle begins. This loop will continue long as the environment is active.

D. *Comparing networks*

We want to emulate how agents with different levels of complexity stack up against each other. We do this by having four different types of agents. We have these four levels of complexity: (i) fast thinking (ii) slow thinking (iii) hybrid (iv) random. The agents in the environment can have any of these levels of complexity. An agent with the fast thinking policy will always exit at a predefined early exit branch, the slow thinking policy will always exit at a latter exit branch, the hybrid will use the GUT-network where it can exit at any exit branch. There are also an agent with a network that exits at a random exit branch. By having all these different agents in the environment we can compare and exploit the benefits and disadvantages of the fast, slow and hybrid thinking. This way we can plot performance, and compare the different agents to see if the theory of a hybrid thinking is beneficial in our environment.

REFERENCES

[1] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016).

[2] Berner, Christopher, et al. "Dota 2 with large scale deep reinforcement learning." arXiv preprint arXiv:1912.06680 (2019).

[3] Kahneman, Daniel, 1934- author. Thinking, Fast and Slow. New York :Farrar, Straus and Giroux, 2011.

[4] Othman, Mazliza, Sajjad Ahmad Madani, and Samee Ullah Khan. "A survey of mobile cloud computing application models." IEEE communications surveys & tutorials 16.1 (2013): 393-413.

[5] Watters, Nicholas, et al. "Cobra: Data-efficient model-based rl through unsupervised object discovery and curiosity-driven exploration." arXiv preprint arXiv:1905.09275 (2019).

[6] Bolukbasi, Tolga, et al. "Adaptive neural networks for efficient inference." International Conference on Machine Learning. PMLR, 2017.

[7] Srivastava, Gaurav, et al. "Joint optimization of quantization and structured sparsity for compressed deep neural networks." ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2019.

[8] Wen, Wei, et al. "Learning structured sparsity in deep neural networks." Advances in neural information processing systems 29 (2016).

[9] Xie, Guotian, et al. "Interleaved structured sparse convolutional neural networks." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.

[10] Vinyals, Oriol, et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning." Nature 575.7782 (2019): 350-354.

[11] Chu, Tianshu, et al. "Multi-agent deep reinforcement learning for large-scale traffic signal control." IEEE Transactions on Intelligent Transportation Systems 21.3 (2019): 1086-1095.

[12] Scardapane, Simone, et al. "Why should we add early exits to neural networks?." Cognitive Computation 12.5 (2020): 954-966.

[13] Unity-Technologies (2022). ml-agents. https://github.com/Unity-Technologies/ml-agents