CS598 Project Report
TensorFlow Implementation of 'Learning to Navigate in Complex Environments'

Tanmay Gangwani
Department of Computer Science
UIUC
gangwan2@illinois.edu

Andrey Zaytsev
Department of Computer Science
UIUC
zaytsev2@illinois.edu

1 Introduction

1.1 Motivation

Navigation in 3D environments is a vital problem in the field of Artificial Intelligence. Often in the real world problems, the environment is not only complex but also dynamically changing. For instance, in the closely related problem of indoor navigation, there are some static objects such as walls and furniture along with some objects that change their position and/or shape, such as humans or pets.

Solving the problem of navigation in dynamic environments has important applications that reach far beyond that of theoretical significance. Those range from small household robots navigating through apartments to drones and quadcopters performing search and rescue missions. In addition, the problem of autonomous driving is tightly coupled with that of navigation in a dynamically changing environment – the car needs not only to arrive to the destination but also avoid collisions with other moving objects along the way.

The problem of autonomous navigation in various environments has been studied before through multiple approaches. Some of them include looking at the problem from the perspective of conventional robotics methods. In such approach, the task is formulated as Simultaneous Localization and Mapping (SLAM), which has an explicit focus on position inferencing [3]. A different approach is provided by Mirowsk et al. [6] by framing navigation as a reinforcement learning problem. In this project, we attempt to reproduce some of the results from this paper. Our implementation, which we refer to as **DeepNav**, is written in TensorFlow and is available on Github[1].

1.2 Background

In the general reinforcement learning setting, an agent interacts with an environment according to a policy, collecting rewards at each time-step. Both the environment and the policy could be deterministic or stochastic. The agent wishes to learn a policy π^* which maximizes the expected sum of rewards from the initial state. More formally, the objective function to maximize is:

$$J(\pi_\theta) = E_{s_t \sim E, a_t \sim \pi}[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)] \quad (1)$$

where the states s_t are sampled from the environment E using some underlying state transition matrix $p(s_{t+1}|s_t, a_t)$; the actions a_t are sampled according the policy π, which is paramterized by θ; r represents the reward at each time step and $\gamma \in [0, 1]$ is the discount factor.

Policy gradient methods are a class of reinforcement learning algorithms that find the optimal policy π^* by calculating the gradient of the objective function (Equation 1) with respect to the policy parameters θ, and taking a small step in the direction of the gradient (Stochastic Gradient Descent). Policy gradients methods can be model-based [8; 4] or model-free [5; 7]. Among recent model-free approaches, the Asynchronous Advantage Actor-Critic (A3C) algorithm [7] has been found to achieve very good results on Atari games as well as continuous motor control tasks. The architecture in

[1]https://github.com/tgangwani/GA3C-DeepNavigation

DeepNav builds on top of the A3C architecture. Actor-Critic algorithms employ two networks, an actor and a critic, which may share some parameters. Given the observation state, the actor network calculates the distribution over the action space, whereas the critic network calculates the state-value (or action-value) function. To reduce the variance in the sampled policy gradients, the actor uses estimated advantage in the loss function formulation. The critic is trained using TD error. The *asynchronous* in A3C is due to the fact that there are independent copies of agents on different CPUs, each interacting with their own instance of the environment, updating the shared global model in an asynchronous manner.

1.3 A3C for Navigation

The application of A3C to navigation presents a few challenges. First of all, the rewards are usually very sparse. Since most often, the problem of navigation consists of arriving at a specified location, the reward is only given there. Due to the sparsity of rewards, model-free policy gradient algorithms such as A3C suffer from poor sample efficiency. It usually takes a large number of roll-outs before rewards can be credited to past actions, and a reasonable value function and policy begins to emerge. In addition, the environment is dynamic, which means the agent has to learn to combine the static knowledge about background such as walls and doors with temporally-correlated visual observations of various moving objects. Doing this involves combining long-term and short-term memory in a non-trivial manner.

To alleviate some of those issues, and also to enable the agent to learn representations that are specific to navigation, the authors in [6] introduce two auxiliary tasks that are learnt jointly with the optimal policy. The two additional tasks are depth prediction and loop closure prediction. Depth prediction involves predicting a depth map based on a single input image, whereas loop closure prediction outputs the probability that the current location was previously visited by the agent within the last few time-steps. This helps to localize the agent in the environment.

2 Approach

In [6], the authors modify A3C for use in maze-navigation. A few sample mazes are shown in Figure 1. The maze inputs are obtained from DeepMind Lab [2], which is an open-source 3D platform for studying the behavior of RL agents in complicated, visual game settings. A rich Python API affords easy communication with the environment. The agent uses the API to receive observations (RGB images, with optional depth information), agent velocity information (translational and rotational) and rewards – either on reaching destination coordinates, or 'fruits' along the way.

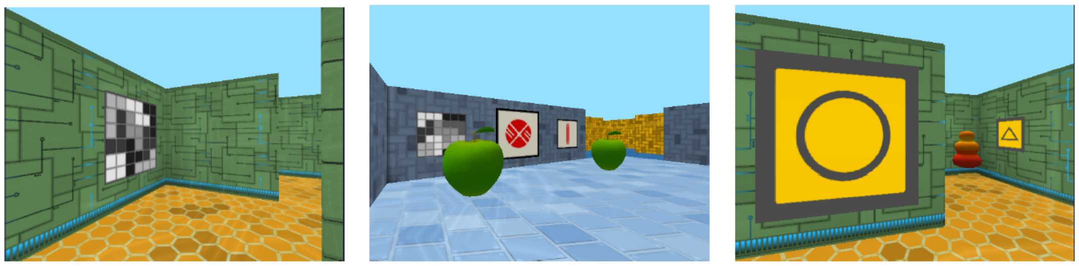
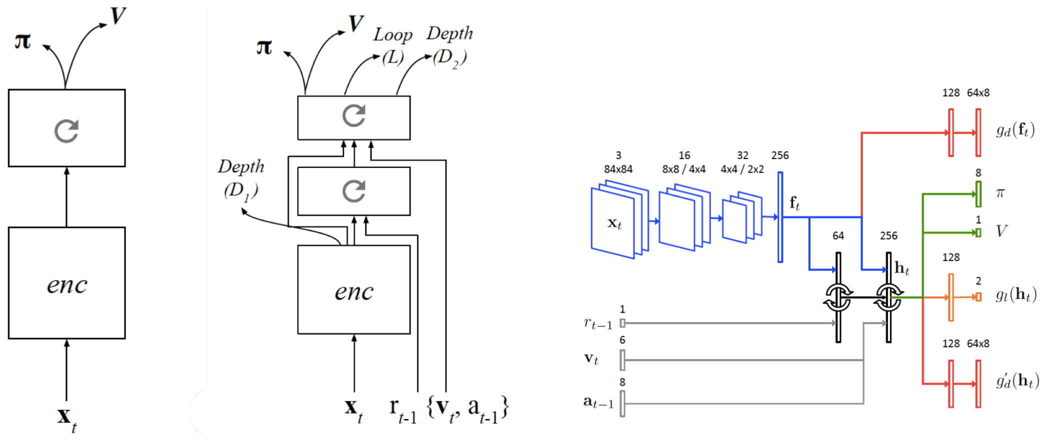


Figure 1: Snapshots for mazes used for navigation in [6].

2.1 Architecture

Figure 2a shows the base A3C architecture introduced in [7]. Images are fed as input x_t to layers of convolution, the output of which is given to an LSTM. Two loss heads are attached to the LSTM output – the actor head π, which outputs a distribution over the next action, and a critic head V, which outputs the value function for x_t.

Figure 2b shows the modifications made to the A3C architecture. Firstly, it uses two LSTMs stacked on top of each other which allows for learning more complex recurrence patterns, as required for navigation. Secondly, the neural network is fed with extra inputs. These come in the form of the reward at previous time step r_{t-1}, the agent velocity v_t obtained directly from the environment, and



(a) Baseline (b) Modified (c) Detailed

Figure 2: Various architecture designs. Figures taken from [6]. (a) Baseline A3C from previous work with convolution layers and single LSTM. (b) Modified A3C with convolution layers, stacked LSTM, extra inputs and auxiliary losses. (c) Modified A3C with details on the size of each layer.

the action taken at the previous time step a_{t-1}. r_{t-1} is given as input to the first LSTM layer. v_t and a_{t-1} are passed to the second LSTM layer along with the outputs of the first LSTM layer and the convolution layers. Thirdly, the parameters are trained using additional losses as described next.

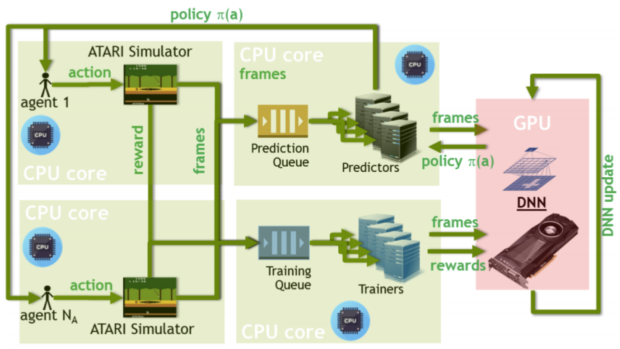
Two separate depth maps are created by adding small *depth networks*, D_1 and D_2, atop the convolution layers and the second LSTM layer, respectively. A depth network consists of a fully connected layer followed by softmax outputs. A *loop prediction network*, L, comprising of a fully connected layer followed by a sigmoid is attached to the second LSTM layer. Mis-classification losses from D_1, D_2 and L are added to the baseline losses from the actor and the critic. Depth prediction is shaped as a classification task rather than a regression task by discretizing the depth at each pixel into 8 bands [6]. Loop closure is a binary classification task.

Figure 2c shows the sizes of various layers in the modified architecture. The convolution layers are of the same dimension as in A3C [7]. The first and second LSTM layers have 64 and 256 cells, respectively. Our action space is 6 dimensional: {forward, backward, left, right, rotate-left, rotate-right}. We predict depth for 64 pixels in the center of the image. Since depth is discretised into 8 bands, we have 64 independent 8-dimensional softmax outputs from each of the depth networks. We omit the loop prediction network from our current implementation. We plan to include it in future work.

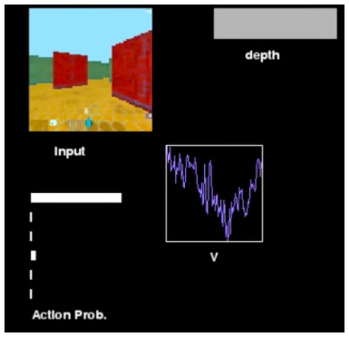
2.2 GPU-based A3C baseline

For our baseline, we chose the hybrid CPU-GPU implementation of the A3C algorithm by Babaeizadeh et al. [1]. The system architecture, referred to as GA3C, is depicted in Figure 3a. Multiple game agents update a shared global model which resides on the GPU. It uses task-specific CPU threads, which exchange messages using FIFO queues. More specifically, there are 3 different kinds of threads – the *agent threads*, each of which interacts with its own instance of the environment and collects experiences; the *predictor threads*, which perform inference on behalf of the agent threads by interacting with the model on the GPU; the *trainer threads*, which batch the experiences of the agent threads and send them to the GPU for gradient calculation. The agents interact with the predictors and the trainers using a prediction queue and a training queue, respectively.

The baseline code includes an entropy regularization term in the actor loss, $H(\pi(s_t, \theta))$. This prevents the collapse of the action distribution π to a particular value, thereby encouraging exploration during the training process. Training is performed using RMSProp. The training speed in GA3C is sensitive to the balance between the number of agents, predictors and trainers. The authors propose an annealing process to configure the system dynamically. For our results below, however, we fix the number of agents to 8, and have 2 predictor and trainer threads each.

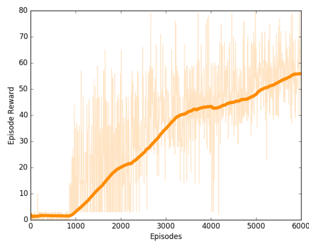


(a) GA3C system architecture

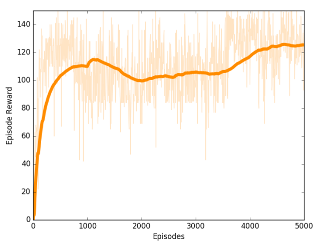


(b) Video screen-shot

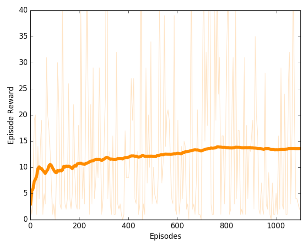
Figure 3: (a) Illustration of the system architecture from [1]. (b) Screen-shot of a live agent navigating a maze using DeepNav; see results section for description.



(a) Static Maze 1



(b) Stairway to Melon



(c) Static Maze 3

Figure 4: Performance of DeepNav on static mazes.

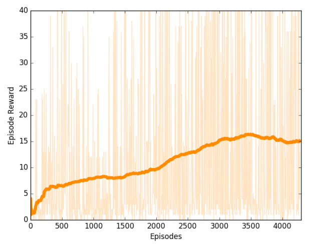
3 Results

We experiment with a subset of visually rich mazes from the DeepMind Lab environment [2]. In each setting, the agent is spawned at a random location with a random orientation, and is tasked with reaching a fixed destination, following which the agent gets re-spawned in another location. Each episode consists of a constant number of time-steps. This number if large enough so that the agent is able to reach the destination multiple times under the optimal policy. The agent collects a big reward on reaching the destination; small sparse rewards are also present in the environment in the form of fruits. We run on 6 different mazes. Following the names in DeepMind Lab, these are: `nav_maze_static_{01,03}.map`, `nav_maze_random_goal_{01,02,03}.map`, and `stairway_to_melon.map`.

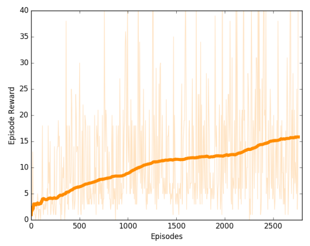
Figure 4 and Figure 5 plot the cumulative reward gathered by the agent in an episode during the training process for all the maze environments. We plot the total reward per episode (light orange) and the moving average over previous 1000 episodes (dark orange). Our average cumulative rewards per episode are not as high as those reported in the paper [6] for the following reasons. Firstly, due to limited resource availability, we run our agents for very short amount of time (1000-6000 episodes) and don't train till convergence. The learning curve for most of the mazes are on an upward trend when we terminate training. We observe individual episodes of high cumulative rewards, e.g. 40+ with Static Maze 3. Therefore, given more training time, we expect to see better results. Secondly, and perhaps more importantly, we fix the hyperparameters for all our experiments and don't perform a grid search for the optimal combination. Hyperparameter tuning can greatly impact the performance of the A3C algorithm; this is also noted by the authors of the original paper [7]. Thirdly, we omit the loop closure prediction loss from our implementation. Adding this could potentially lead to better results for some mazes, although [6] shows that it doesn't provide any boost.

Videos of agents trained with DeepNav are available for `nav_maze_static_01.map`[2] and `stairway_to_melon.map`[3]. Figure 3b shows a screen-shot. For an agent navigating the maze,

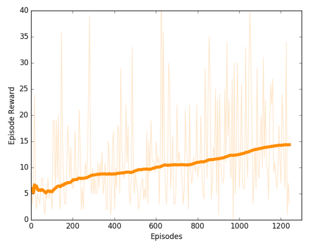
[2]https://www.youtube.com/watch?v=vyS0Z7wdHHs
[3]https://www.youtube.com/watch?v=0R5MGM7VPo4



(a) Random Maze 1



(b) Random Maze 2



(c) Random Maze 3

Figure 5: Performance of DeepNav on random mazes.

we demonstrate the action probabilities, the state-value function (V) and the depth network prediction (D_1) at each step of the way.

4 Conclusions

We train an agent to navigate in visually complex mazes using a model-free, policy gradient reinforcement learning algorithm. We follow the methodology of the Mirowski et al. [6] and modify the A3C architecture to incorporate extra inputs—velocity, previous action, previous reward—and auxiliary depth prediction loss. We use the maze environments provided by DeepMind Lab to evaluate our agents. Since the A3C algorithm is not very data-efficient, we were unable to train the agents to convergence with our limited resources. We believe that tuning the hyperparameters of the DeepNav architecture can help in faster convergence. Moreover, using the annealing process in GA3C to achieve a better balance between the number of agent, trainer and predictor threads can also lead to faster training.

References

[1] Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., and Kautz, J. Ga3c: Gpu-based a3c for deep reinforcement learning. *arXiv preprint arXiv:1611.06256* (2016).

[2] Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., et al. Deepmind lab. *arXiv preprint arXiv:1612.03801* (2016).

[3] Dissanayake, M. G., Newman, P., Clark, S., Durrant-Whyte, H. F., and Csorba, M. A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on robotics and automation 17*, 3 (2001), 229–241.

[4] Grondman, I., Vaandrager, M., Busoniu, L., Babuska, R., and Schuitema, E. Efficient model learning methods for actor–critic control. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 42*, 3 (2012), 591–602.

[5] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[6] Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., et al. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673* (2016).

[7] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning* (2016).

[8] Ng, A. Y., Kim, H. J., Jordan, M. I., Sastry, S., and Ballianda, S. Autonomous helicopter flight via reinforcement learning. In *NIPS* (2003), vol. 16.