A study on the role of memory in visual navigation

Bimal Parakkal & Chuqi Wang
CS 231N Spring 2017

0 Abstract

The goal of this project is to expand on the recent work done by Yuke et.al on visual navigation using deep siamese actor-critic network to achieve better performance and sample efficiency. We aim to accomplish this by incorporating experience playback into the base model. The data for training the agent is derived using AI2-THOR framework which generates high quality 3D scenes and enables collection of large number of visual observations for actions and reactions in different environments. We will pitch our newly proposed architecture against standard RL models and original deep siamese [14] actor-critic network model and see if it discovers similar or shorter paths using less training samples without sacrificing generality.

1 Introduction

Visual navigation is an important part of mapless navigation. Traditional navigation techniques uses computer vision to acquire the target and obstacle avoidance to plan path accordingly. Visual navigation has the advantage that we use more contextual information from the environment as well as being able to have better semantic understanding of the objective [15].

One of the most recent work in this domain is the Yuke et al's target driven visual navigation model from which this project is inspired. The model was able to address two aspects of Deep RL which has received less attention in the past. The issues being
(1) lack of generalization capability to new goals, and (2) data inefficiency.
The model determines its optimal navigation policy using a feedforward network without memory. In other words, the agent's actions are only determined based on the current observation (last four frames). The author have noted however that memory can be useful for many scenarios, e.g., when the agent gets stuck in a corner, a network with memory can remember the previous trajectories and help the agent escape the corner

The goal of this project is to study the network architectures that can supplement the Yuke's model [5] with a state memory and study if the agent is still able to address the two issues which it was originally able to address. To impart memory to the model, we plan on trying to incorporate experience replay in our sampling using the algorithm introduced by Ziyu et al [8].

2 Related Work

There is an inherent difficulty in using Deep learning models for RL because Non-linear function approximators (Q-Network in the case of RL) have low learning stability on account of

(a) Correlated data input which constitute agent state
(b) The distribution of data keeps changing.

Many of the recent work to integrate deep learning into RL which try to address these issues have been met with quite a bit of success. [1] demonstrated use of experience

replay in Deep Q Networks to circumvent instability issues while during training and successfully train agents that were able to match or exceed human performance while playing atari games. The deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games.

Other proven ways that address training instability issues of Deep RL models involves using Parallel actor learners. [4] shows that Parallel actor-learners have a stabilizing effect on training allowing all four methods to successfully train neural network controllers. The papers investigates asynchronous variants of four standard reinforcement learning algorithms and best performing method, an asynchronous variant of actor-critic(A3C), surpassed the then current state-of-the-art on the Atari domain while training for half the time on a single multi-core CPU instead of a GPU.

[5] extends the use of A3C for visual navigation domain with necessary architectural adaptations.
The authors claim that navigational decisions demand an understanding of the relative spatial positions between the current locations and the target locations, as well as a holistic sense of scene layout. They propose a new deep siamese actor-critic network to capture such intuitions. Our goal is extend the work done by authors by investigating whether incorporating state memory into the architecture will improve the navigational performance. [6] and [7] two recent works on incorporating memory states into Deep RL

[6] investigates the effects of adding recurrency to a Deep Q-Network (DQN) by replacing the first post-convolutional fully-connected layer with a recurrent LSTM.The resulting Deep Recurrent Q-Network (DRQN), although capable of seeing only a single frame at each timestep, successfully integrates information through time and replicates DQN's performance on standard Atari games and partially observed equivalents featuring flickering game screens.

[7] proposes architectures consisting of convolutional networks for extracting high-level features from images, a memory that retains a recent history of observations, and a context vector used both for memory retrieval and (in part for) action-value estimation. The context vector construction method yields three architectural variants Memory Q-Network (MQN), Recurrent Memory Q-Network (RMQN), and Feedback Recurrent Memory Q-Network (FRMQN).

[8] The method addresses some major shortcomings in deep Q learning [1] or on-policy A3C [4] by introducing importance sampled experience replay to actor-critic model to improve sample efficiency of reinforcement learning algorithms in solving Atari games. It introduced useful techniques such as truncated importance sampling with bias correction technique.

3 Methods

Our model will build upon the deep Siamese actor-critic network introduced by Yuke et al[5] as well as the experience replay techniques introduced by Ziyu et al [8]. The deep Siamese network is composed of two parts, one generic actor critic layer and

another scene-specific layer (figure 1). The generic layer takes four consecutive frames from observation and target respectively and feed them into ResNet to generate 8192-d feature representation of proximity of the agent to the target. This is then projected down to 512d and fused together to be fed into scene specific layers, which captures scene specific characteristics. The output from this layer will ultimately decide the policy(which way for robot to turn).

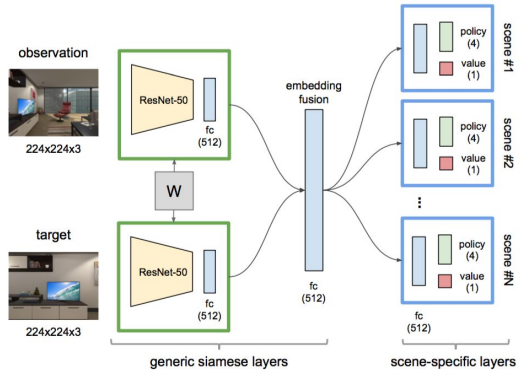

Figure 1. Deep Siamese Actor-critic model[5]

Adding memory[19] to this network could help it converge faster. Because in reinforcement learning we take a long sequence of steps, the training data could be closely correlated and breaks IID assumption. Memory replay can help avoid this problem and smooth transitions. During training, experiences <s, a, r, s'>, where s, s' are current and next state, a is action, r is reward, are stored and then sampled according to some distribution later. Besides smoothing transitions, a arguably more important function of memory is sample efficiency. Reinforcement learning requires a lot of samples to train. Since obtaining a sample involves an agent interacting with the environment (e.g. a simulator), the interaction could be expensive. Experience replay reuses past interactions to learn so that we could sample less and thereby reduce training time.

We have an agent interacting with its environment over discrete time steps over a discrete set of actions. Using the we process the 8192-d features with 3 FC layers to project it down to 512 features and then pass the results through either a 512x4 policy layer to produce an action or 512x1 layer to estimate an value. Note our setup has only 4 possible discrete actions (forward, backward, turn left, turn right). We also define state action value Q given state and action only value V both given a policy π.

$$Q^{\pi}(x_t, a_t) = \mathbb{E}_{x_{t+1:\infty}, a_{t+1:\infty}}\left[R_t \middle| x_t, a_t\right]$$

$$V^{\pi}(x_t) = \mathbb{E}_{a_t}\left[Q^{\pi}(x_t, a_t) \middle| x_t\right]$$

We use the advantage function A = Q - V to measure how much better action is better than expected.

$$A^{\pi}(x_t, a_t) = Q^{\pi}(x_t, a_t) - V^{\pi}(x_t)$$

Actor here is the policy and the critic here is the Q function. To update the parameters of the differentiable policy π_θ, we use this gradient in table.1. We replaced advantage function with temporal difference residual (table.2). This yields a gradient update formula of table.3. We applied experience replay technique to A3C model to create the off-policy version of A3C.

1	$g = \mathbb{E}_{x_{0:\infty}, a_{0:\infty}}\left[\sum_{t\geq 0} A^{\pi}(x_t, a_t)\nabla_\theta \log \pi_\theta(a_t|x_t)\right].$
2	$r_t + \gamma V^{\pi}(x_{t+1}) - V^{\pi}(x_t)$
3	$\hat{g}^{\mathrm{a3c}} = \sum_{t\geq 0}\left(\left(\sum_{i=0}^{k-1}\gamma^i r_{t+i}\right) + \gamma^k V^{\pi}_{\theta_v}(x_{t+k}) - V^{\pi}_{\theta_v}(x_t)\right)\nabla_\theta \log \pi_\theta(a_t|x_t)$

4	$\widehat{g}^{\text{imp}} = \left(\prod_{t=0}^{k} \rho_t\right) \sum_{t=0}^{k} \left(\sum_{i=0}^{k} \gamma^i r_{t+i}\right) \nabla_\theta \log \pi_\theta(a_t \mid x_t)$
5	$g^{\text{marg}} = \mathbb{E}_{x_t \sim \beta, a_t \sim \mu} \left[\rho_t \nabla_\theta \log \pi_\theta(a_t \mid x_t) Q^\pi(x_t, a_t)\right]$
6	$Q^{\text{ret}}(x_t, a_t) = r_t + \gamma \bar{\rho}_{t+1} [Q^{\text{ret}}(x_{t+1}, a_{t+1}) - Q(x_{t+1}, a_{t+1})] + \gamma V(x_{t+1})$
7	$Q^{\text{ret}}(x_t, a_t) = r_t + \gamma \bar{\rho}_{t+1} [Q^{\text{ret}}(x_{t+1}, a_{t+1}) - Q(x_{t+1}, a_{t+1})] + \gamma V(x_{t+1})$ $= \mathbb{E}_{x_t}\left[\mathbb{E}_{a_t}[\bar{\rho}_t \nabla_\theta \log \pi_\theta(a_t \mid x_t) Q^\pi(x_t, a_t)] + \mathbb{E}_{a \sim \pi}\left(\left[\frac{\rho_t(a) - c}{\rho_t(a)}\right]_+ \nabla_\theta \log \pi_\theta(a \mid x_t) Q^\pi(x_t, a)\right)\right]$
8	$\widehat{g}^{\text{acer}} = \bar{\rho}_t \nabla_\theta \log \pi_\theta(a_t \mid x_t)[Q^{\text{ret}}(x_t, a_t) - V_{\theta_v}(x_t)]$ $+ \mathbb{E}_{a \sim \pi}\left(\left[\frac{\rho_t(a) - c}{\rho_t(a)}\right]_+ \nabla_\theta \log \pi_\theta(a \mid x_t)[Q_{\theta_v}(x_t, a) - V_{\theta_v}(x_t)]\right)$

Table of Formulas [8]

We switch between on-policy learning and off-policy learning. When doing on-policy learning, we sample directly from the environment. When doing off-policy, we sample k records from the experience memory. We used an accumulator to accumulate the gradient from the k samples and update the model parameters in batch. To maximize the effect of experience replay, we take weighted some of the gradience from each experience, favoring those experience that gives us greatest gain. The generic importance weighted policy gradient [16] is given by formula table.4. Note that because of the series product term of $\varrho_t = \pi(a_t|x_t)/\mu(a_t|x_t)$, the expression suffers from high variance. Variance could be reduced by truncating rho by replacing it with ϱ_bar = min(c, ϱ)[9]. This truncation introduces high bias, which could be addressed by approximating the gradient with a marginal value function. This is represented by the function in table.5. E is the expectation w.r.t. limiting distribution β with average policy μ [8], [18]. Q^π is estimated by lambda return. Although this formula reduces bias, it requires sensitive parameter choice. Lambda return method is enhanced by Retrace estimator [10]. The Retrace estimator is an recursive defined estimator given by formula table.3. ϱ is still importance weight, Q is current estimate of Q^π.

Retrace algorithm is off-policy, return based, and enabled faster learning. This is the algorithm that we eventually implemented for experience replay. $Q_{\theta v}$ (vector) replaced scalar $V_\theta(Xt)$ in Yuke's algorithm. $Q_{\theta v}$ is used to model critic and is estimated by using Q^{ret} as a target (table.6). The final algorithm looks like in table.7. Notice that the final formula incorporates both importance weight clipping and bias correction (first and second term). Because we approximate the expectation by sample k steps in a trajectory, then for each experience, the gradient term is given by table.8.

The final part of the algorithm includes an efficient trust region policy optimization (TRPO) [20] for policy gradient update. This limits per step change to policy to ensure stability. This is superior to limiting learning rate as we only want to guard against occasional large update and not want to slow down entire training. An efficient way of TRPO involves using FC to generate a statistic for a distribution that ultimately determines the policy [8]. This can be summarized by ϕ_θ:$\pi(.|x) = f(.|\phi_\theta(x))$, where f is the distribution and ϕ_θ is a neural network (in our case FC per each scene+object thread) that generates distribution, and parameterized by θ. We used a weighted soft update for θ.

The full algorithm can be found in **appendix**.

4 Dataset and Features

The image stream data required for training the model is generated by a state of art high quality 3D scenes simulator - AI2 - THOR which includes a physics engine as well. The AI2-THOR framework [11] enables agents

to take actions and interact with objects, thereby making the model training process both cost efficient and easy - when compared with having to do the same tasks with a real robot.

To facilitate training process, a scene dump of all images in a scene can be used. The images are 300x400x3 RGB images and taken at each discrete positions in the training scene. Each position can be thought of as a fixed size tile and the observations are in 0, 90, 180, 270 degree direction. In each direction, the observation perspective could either look up, down or straight ahead. Thus we have 12 images per location. The full training data suite contains 32 different scenes of 4 general types (kitchen, bedroom, living room, and bathroom) on 68 different objects. Since each agent-environment interaction is expensive (taking 100+ hours on GeForce GTX Titan X GPU over 100 million frames), we limited our training to a smaller train and make comparison with Yuke's baseline using the smaller dataset. We selected 4 scenes and 20 objects and trained for around 6 million frames.
To facilitate the training process, we used hdf5 dumps of simulated scenes instead of real time feedback of the simulation environment. We extract the scene image we would have seen by looking it up based on our location and orientation in a scene. Feature extraction is done with pretrained stock Keras model [12] of ResNet-50 [13].
Sample images are shown in figure 4, 5. No preprocessing is needed.

5 Experiments/Results/Discussion

The goal of optimization is to find the shortest trajectory from current location to target destination. We compare performance by measuring the average number of steps the agent takes to reach target. We will train multiple targets in multiple scenes in training. During test time, we will pick several previously unseen objects in the scene to check average generated paths. A final reward of 10 points is associated reaching target and an intermediate reward of -0.01 is used to encourage shorter paths. We will also test on unseen scenes as well, and observe the level of performance degradation. We hope to achieve similar test time performance after training on smaller number of steps.
For comparison, we will evaluate this model against with baseline navigation models based on heuristics, and A3C implementation.

We evaluated our implementation by training on four scenes and five objects per scene. The scenes are of types: bathroom, bedroom , living room and kitchen. The learning rate is XXX. The lambda and gamma for importance sample. The memory size for experience replay. The size of k. We used basic values. We ran the training over 6M steps and used the trained weights on finding a previously untrained object. The average steps is *2254* steps in comparison with Yuke's avg. Trajectory length of *210* steps and random walk length of *2750* steps

6 Conclusion/Future Work (1-3 paragraphs)

For this project, we implemented experience replay algorithm on top of Yuke's target driven A3C model [17] for visual navigation task. Our eventual achieved smaller

trajectory length than random walk but failed to show improvement over Yuke's algorithm performance. We suspect that lack of hyper-parameters tuning particularly the couple of ones impacting trust region policy update was potentially causing a too strong regularization and impacting the learning process. The high turn around time for training the models hindered our progress quite a bit and issues could not be resolved quickly. Further, with a non-functional on policy-model we could not proceed on offline model as was originally scoped. For future work, we need to fix possible bugs so that our model achieves similar result as Yuke's model. We need to further fine tune our parameters to achieve better results.

8 References/Bibliography

[1] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. Nature 518(7540):529–533

[2] Sutton, R. S., and Barto, A. G. 1998. Reinforcement Learning: An Introduction. MIT Press.

[3] Watkins, C. J. C. H., and Dayan, P. 1992. Q-learning. Machine Learning 8(3-4):279–292

[4] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in ICML, 2016.

[5] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J. Lim, Abhinav Gupta, Li Fei-Fei, Ali Farhadi "Target-driven Visual Navigation in Indoor Scenes using Deep Reinforcement Learning"

[6] Matthew Hausknecht , Peter Stone "Deep Recurrent Q-Learning for Partially Observable MDPs "

[7] Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, Honglak Lee "Control of Memory, Active Perception, and Action in Minecraft"

[8] Ziyu Wang, Victor Bapst, NIcolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, Nando de Freitas "Sample Efficient Actor-Critic with Experience Replay"

[9] P. Wawrzyn ´ski. Real-time reinforcement learning by sequential actor–critics and experience replay. *Neural Networks*, 22(10):1484–1497, 2009.

[10] R. Munos, T. Stepleton, A. Harutyunyan, and M. G. Bellemare. Safe and efficient off-policy reinforcement learning. arXiv preprint arXiv:1606.02647, 2016.

[11] http://vuchallenge.org/thor.html
[12] https://keras.io/applications/#resnet50
[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition
[14] Gregory Koch Siamese Neural Networks for One-shot Image Recognition
[15] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *JMLR*, 2016.
[16] Jie and P. Abbeel. On a connection between importance sampling and the likelihood ratio policy gradient. In NIPS, pp. 1000–1008, 2010.
[17] https://github.com/yukezhu/icra2017-visual-navigation
[18] https://github.com/chainer/chainerrl/blob/master/chainerrl/agents/acer.py
[19] https://github.com/devsisters/DQN-tensorflow/blob/master/dqn/replay_memory.py

[20] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel. Trust Region Policy Optimization

Figure 1. Deep Siamese Actor-critic model

Figure 2. Experience replay with random sampling.
https://www.nervanasys.com/demystifying-deep-reinforcement-learning/)

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

Figure 3. Experience replay with prioritized sampling (src: Shaul et al: https://arxiv.org/pdf/1511.05952.pdf)

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Figure 4. Target image

Figure 5. Observation image

Appendix 1.

Algorithm 1 ACER for discrete actions (master algorithm)

// Assume global shared parameter vectors θ and θ_v.
// Assume ratio of replay r.
repeat
  Call ACER on-policy, Algorithm 2.
  $n \leftarrow \text{Possion}(r)$
  for $i \in \{1, \cdots, n\}$ **do**
    Call ACER off-policy, Algorithm 2.
  end for
until Max iteration or time reached.

Algorithm 2 ACER for discrete actions

Reset gradients $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
Initialize parameters $\theta' \leftarrow \theta$ and $\theta'_v \leftarrow \theta_v$.
if not On-Policy **then**
  Sample the trajectory $\{x_0, a_0, r_0, \mu(\cdot|x_0), \cdots, x_k, a_k, r_k, \mu(\cdot|x_k)\}$ from the replay memory.
else
  Get state x_0
end if
for $i \in \{0, \cdots, k\}$ **do**
  Compute $f(\cdot|\phi_{\theta'}(x_i))$, $Q_{\theta'_v}(x_i, \cdot)$ and $f(\cdot|\phi_{\theta_a}(x_i))$.
  if On-Policy **then**
    Perform a_i according to $f(\cdot|\phi_{\theta'}(x_i))$
    Receive reward r_i and new state x_{i+1}
    $\mu(\cdot|x_i) \leftarrow f(\cdot|\phi_{\theta'}(x_i))$
  end if
  $\bar{\rho}_i \leftarrow \min\left\{1, \frac{f(a_i|\phi_{\theta'}(x_i))}{\mu(a_i|x_i)}\right\}$.
end for

$$Q^{ret} \leftarrow \begin{cases} 0 & \text{for terminal } x_k \\ \sum_a Q_{\theta'_v}(x_k, a) f(a|\phi_{\theta'}(x_k)) & \text{otherwise} \end{cases}$$

for $i \in \{k-1, \cdots, 0\}$ **do**
  $Q^{ret} \leftarrow r_i + \gamma Q^{ret}$
  $V_i \leftarrow \sum_a Q_{\theta'_v}(x_i, a) f(a|\phi_{\theta'}(x_i))$
  Computing quantities needed for trust region updating:

$$\begin{aligned} g &\leftarrow \min\{c, \rho_i(a_i)\} \nabla_{\phi_{\theta'}(x_i)} \log f(a_i|\phi_{\theta'}(x_i))(Q^{ret} - V_i) \\ &\quad + \sum_a \left[1 - \frac{c}{\rho_i(a)}\right]_+ f(a|\phi_{\theta'}(x_i)) \nabla_{\phi_{\theta'}(x_i)} \log f(a|\phi_{\theta'}(x_i))(Q_{\theta'_v}(x_i, a_i) - V_i) \\ k &\leftarrow \nabla_{\phi_{\theta'}(x_i)} D_{KL}\left[f(\cdot|\phi_{\theta_a}(x_i)) \| f(\cdot|\phi_{\theta'}(x_i))\right] \end{aligned}$$

  Accumulate gradients wrt θ': $d\theta' \leftarrow d\theta' + \frac{\partial \phi_{\theta'}(x_i)}{\partial \theta'}\left(g - \max\left\{0, \frac{k^T g - \delta}{\|k\|_2^2}\right\} k\right)$
  Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \nabla_{\theta'_v}(Q^{ret} - Q_{\theta'_v}(x_i, a))^2$
  Update Retrace target: $Q^{ret} \leftarrow \bar{\rho}_i\left(Q^{ret} - Q_{\theta'_v}(x_i, a_i)\right) + V_i$
end for
Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
Updating the average policy network: $\theta_a \leftarrow \alpha\theta_a + (1-\alpha)\theta$