sliu1@uvm.edu
Sida Liu

1 Revised Project Proposal

1.1 Response

Neat project proposal! It sounds like the tracer bullet version is one where you:
Code up an RNN to produce high level abstractions, which are functions of the current state.
Learn a policy over these high level states.

Does "tracer bullet" mean proof-of-concept? I googled this phrase but I am still not quite sure. :D Sorry about my English.

And Yes, RNN will be a key component in this architecture.

Will this high level policy interact with the lower-level policy over observations?

In Section 1.5, I will compare multi-RL and multi-RNN. I will come to the conclusion that multi-RNN can be equivalent to multi-RL, and multi-RNN has a major advantage: the gradient is available. So I'll use multi-RNN. (But I am not quite sure so many papers are using hierarchical-RL, maybe I am wrong?)

Do you intend to learn over a family of possible high level policies?

I'll gradually increase the complexity of the model and produce different higher representations. There will be only one RL policy at the top to consume all the representations.

I am particularly interested in whether/how you might forsee using such a model for the embodied agent problem you've talked about.

I might use Multi-Robot Continuous Control task as one of the testbed (control different robots to move forward). But the model developed in this work should be a general RL model that can work beyond robotics.

1.2 Structure

The deliverable will be a piece of open-source software.

1.3 Topic

1.3.1 Motivation

In 2017, Yoshua Bengio submitted an arXiv article named *The Consciousness Prior* [1]. He proposed the *Consciousness Prior Theory*, which is about how, in his opinion, the *Consciousness* works and how to implement the *Global Workspace Theory* using recently developed *Deep Learning* technology. There are two major parts in his theory, the *Consciousness* c and the *Unconsciousness* h.

In Bengio's paper, he thinks the high-level abstraction (a.k.a. the *Unconsciousness*) h is produced by some representation RNN function F, so that

$$h_t = F(x_t, h_{t-1})$$

where x_t is the *observation* at time t, and h_t is the *unconscious representation state* (or *high-level state*) at time t.

I will focus on the h in this course project because it is more practical and is the foundation before we get anything similar to the *Consciousness*.

1.3.2 Big Picture of Deep Reinforcement Learning

Our objective is to get a good h, but how can we get it.

Deep Learning is trying to do hierarchical information abstraction since its invention. CNN, RNN, Attention model, and many other Deep Learning models are very good tools to do this job.

In the setting of Reinforcement Learning, we can simply situate the agent in an environment, and use a high-level reward function to specify the goal.

At each time step, we can pass observation through the neural network model of the agent and get a value of that state (or action); we can also use the Temporal Difference (TD) method to get a value of that state (or action). We assume the TD method gives us the right value, so we adjust the parameters of the neural network model of the agent to make two values consistent. This consistency is what we usually use to train an agent in Deep Reinforcement Learning.

1.3.3 Additional training signal beside TD method

What if the TD method doesn't give us the perfect value function? Do we have other sources of learning signals?

One possible learning signal can come from temporal consistency. For example, we have two observations of two consecutive time steps. We can assume the world doesn't change much in one single step, so the meaning of those two observations should be similar. So we can pass two observations through the neural network model and compare two values produced. They should be consistent. And we can train the neural network model based on this consistency.

This might be an additional source for learning besides the TD method. I mention this because I think this is interesting, but it might not be implemented in this work.

1.3.4 Course Project Plan: Step 0

I'll start the project with an empty framework without any RNNs illustrated in Fig 1.

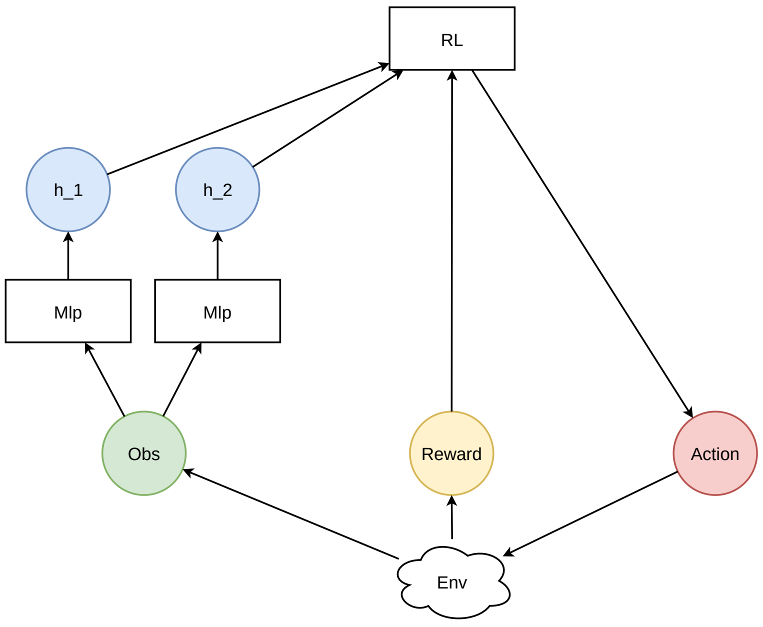

Figure 1: Starting point: without any RNN

This is a standard RL model with two different MultiLayer Perceptron (MLP) modules. The representation (in blue) will be concatenated and pass into the RL core algorithm.

Later in this project, my job is to produce more and better representations h's for the RL core algorithm on the left side. I'll introduce RNNs and other tools available in Deep Learning.

Once we have more than enough h's, and if there is time left, we can introduce attention mechanism to filter them before passing into the RL.

I am not very sure about the right side of the diagram yet. For example, in conventional hierarchical RL, people use options to execute a sequence of actions, but I am not sure if it is necessary in this framework. Maybe in this project, I'll just leave it as simple as it is.

1.3.5 Testing Environments

My algorithm should be general, so it can work in different environments.

One possible testbed is the Continuous Control problem. For example, locomotion is my most familiar task.

I also wish to test it in some multi-player games, so maybe I can ask my agent to play with someone else's agent.

1.4 World Model: An Example

David Ha et al. 2018 proposed the World Model [2]. As illustrated in Fig 3, it utilizes a VAE and an RNN to make a model so that the agent can not just see the compressed current observation but can also predict the future. Both current observation and the prediction of the future are passed into an RL agent.

Interestingly, the RNN takes in the action and h_1 from previous time step. I understand the action of current step is not available, but why the previous h_1?

Also, the inputting of previous action indicates that the reward could also be an input. hmm... My interpretation is that, since we are facing a POMDP, the observation doesn't capture all the information, for example, the previous action. So I'll treat previous action and previous reward as additional observations.

Overall, this model is a good demonstration of the framework we constructed here.

However, the paper only has one VAE and one RNN, so it's not possible to capture features at different scales. As illustrated in Fig 1, I prefer multiple modules that can extract representations at multiple scales.

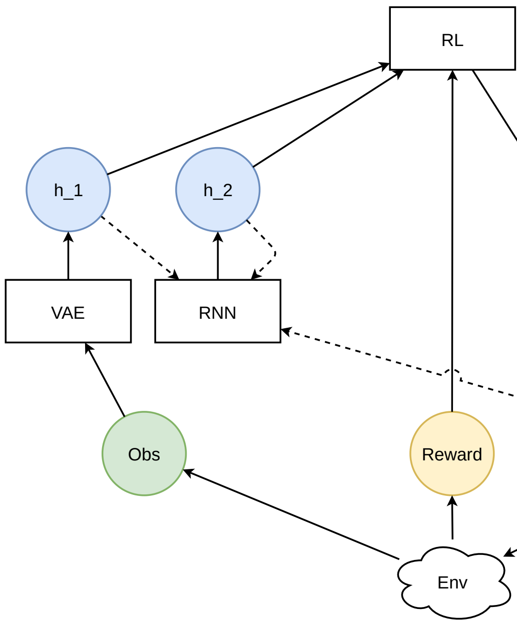

Figure 2: The World Model, re-illustrated to fit in our framework. Dashed line stands for the information is from the previous time step.

1.5 Multiple RLs or Multiple RNNs?

Before I read these two papers [3][4], I thought RL and RNN are simply different creatures. And then, I looked at these two approaches of making learned optimizers to replace hand-designed optimizer such as Adam, and I realized there are so many similarities.

One views the optimizer as an RL agent[3], and the other views the optimizer as an RNN network[4]. Here is a brief comparison:

1.5.1 RL: Learning to Optimize

Li et al. proposed this idea in 2016 [5], which is very similar to my idea. In Li et al., 2016, the observation contains: (1) Current parameters, θ, (2) The $\Delta\mathcal{L}$ in recent H time steps, (3) The $\nabla_\theta\mathcal{L}$ in recent H time steps. (I changed the notation to be aligned with this document.) The action contains: (1) the update steps, $\Delta\theta = \alpha\nabla_\theta\mathcal{L}$. The goal is to minimize the total area under the learning curve at meta-test time. The reward function (cost function) is the negative value of outer loss function, $-\mathcal{L}$. They chose Guided Policy Search (GPS) as the RL algorithm for the optimizer.

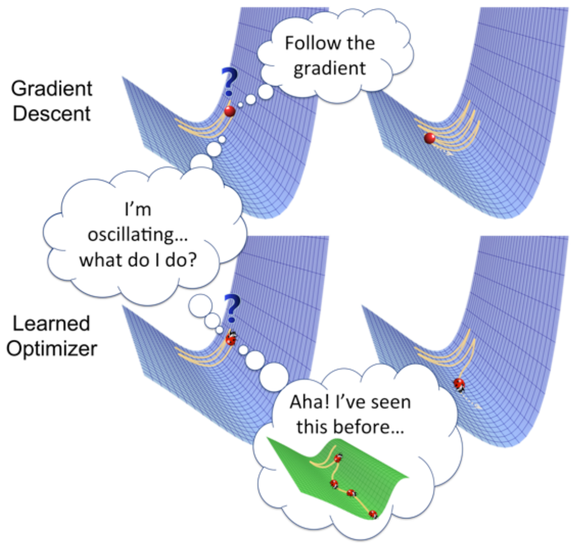
Gradient Descent
Follow the gradient
I'm oscillating... what do I do?
Learned Optimizer
Aha! I've seen this before...

Figure 3: Illustration of the Paper *Learning to Optimize*. Source: Blog

Li et al. 2017 provide more details on implementation [3], though source code is not available. The learned optimizer was then tested on several image datasets, which outperforms Adam, AdaGrad, RMSprop in the experiments.

In general, I think Li's implementation can be improved somehow. At least I should be able to produce a reusable package that can be easily used in PyTorch.

1.5.2 RNN: Learning to learn by gradient descent by gradient descent

In contrast, Andrychowicz et al. 2016[4] formulate the optimizer as an RNN neural network. At every time step, the current parameters θ, and the gradients $\nabla_\theta \mathcal{L}$ are the input to the RNN, along with the hidden state $hidden_{t-1}$ (to avoid using the notation h here) of the last step. Just like the RL model, the RNN also produces the update steps, $\Delta\theta$, and the outer network applies the update steps. Critically, because the whole model is differentiable, so they can directly compute gradients for the RNN optimizer from that step.

Also, to reduce the requirement of memory and make the model more flexible, the RNN model doesn't take concatenated parameters and gradients but take one parameter and one gradient at a time, so all parameters from outer networks pass through the RNN model, producing multiple hidden states and the update steps for their corresponding parameters, respectively.

1.5.3 My Lesson

I am not planning to use learned optimizers in my course project. I'll still use Adam. However, from this comparison, the takeaway for me is that, in my hierarchical networks, there will be no need to construct multiple RL agents. We should not formulate problems as an RL problem if the gradients are directly available.

References

[1] Yoshua Bengio. The Consciousness Prior. *arXiv:1709.08568 [cs, stat]*, December 2019. arXiv: 1709.08568.

[2] David Ha and Jürgen Schmidhuber. World Models. *arXiv:1803.10122 [cs, stat]*, March 2018. arXiv: 1803.10122.

[3] Ke Li and Jitendra Malik. Learning to optimize neural nets, 2017.

[4] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to learn by gradient descent by gradient descent, 2016.

[5] Ke Li and Jitendra Malik. Learning to optimize, 2016.