ヤロミル

Let’s make an A3C: Theory

Posted on **February 16, 2017March 26, 2017**
This article is part of series *Let’s make an A3C*.

1. **Theory (/2017/02/16/lets-make-an-a3c-theory/)**
2. **Implementation (/2017/03/26/lets-make-an-a3c-implementation/)**

Introduction

Policy Gradient Methods is an interesting family of Reinforcement Learning algorithms. They have a long history[1], but only recently were backed by neural networks and had success in high-dimensional cases. A3C algorithm was published in 2016 and can do better than DQN with a fraction of time and resources[2].

In this series of articles we will explain the theory behind Policy Gradient Methods, A3C algorithm and develop a simple agent in Python.

It is very recommended to have read at least the first **Theory (/2016/09/27/lets-make-a-dqn-theory/)** article from *Let’s make a DQN* series which explains theory behind Reinforcement Learning (RL). We will also make comparison to DQN and make references to these older series.

Background

Let’s review the RL basics. An agent exists in an environment, which evolves in discrete time steps. Agent can influence the environment by taking an action *a* each time step, after which it receives a reward *r* and an observed state *s*. For simplification, we only consider deterministic environments. That means that taking action *a* in state *s* always results in the same state *s’*.

Although these high level concepts stay the same as in DQN case, they are some important changes in Policy Gradient (PG) Methods. To understand the following, we have to make some definitions.

First, agent’s actions are determined by a stochastic policy *π(s)*. Stochastic policy means that it does not output a single action, but a *distribution of probabilities over actions*, which sum to 1.0. We’ll also use a notation *π(a|s)* which means the probability of taking action *a* in state *s*.

For clarity, note that there is no concept of greedy policy in this case. The policy *π* does not maximize any value. It is simply a function of a state *s*, returning probabilities for all possible actions.

We will also use a concept of expectation of some value. Expectation of value *X* in a probability distribution *P* is:

$$E_P[X] = \sum_i P_i X_i$$

where X_i are all possible values of *X* and P_i their probabilities of occurrence. It can also be viewed as a weighted average of values X_i with weights P_i.

The important thing here is that if we had a pool of values *X*, ratio of which was given by *P*, and we randomly picked a number of these, we would *expect* the mean of them to be $E_P[X]$. And the mean would get closer to $E_P[X]$ as the number of samples rise.

We’ll use the concept of expectation right away. We define a value function *V(s)* of policy *π* as an expected discounted return, which can be viewed as a following recurrent definition:

$$V(s) = E_{\pi(s)}[r + \gamma V(s')]$$

Basically, we weight-average the $r + \gamma V(s')$ for every possible action we can take in state *s*. Note again that there is no max, we are simply averaging.

Action-value function *Q(s, a)* is on the other hand defined plainly as:

$$Q(s, a) = r + \gamma V(s')$$

simply because the action is given and there is only one following *s’*.

Now, let’s define a new function *A(s, a)* as:

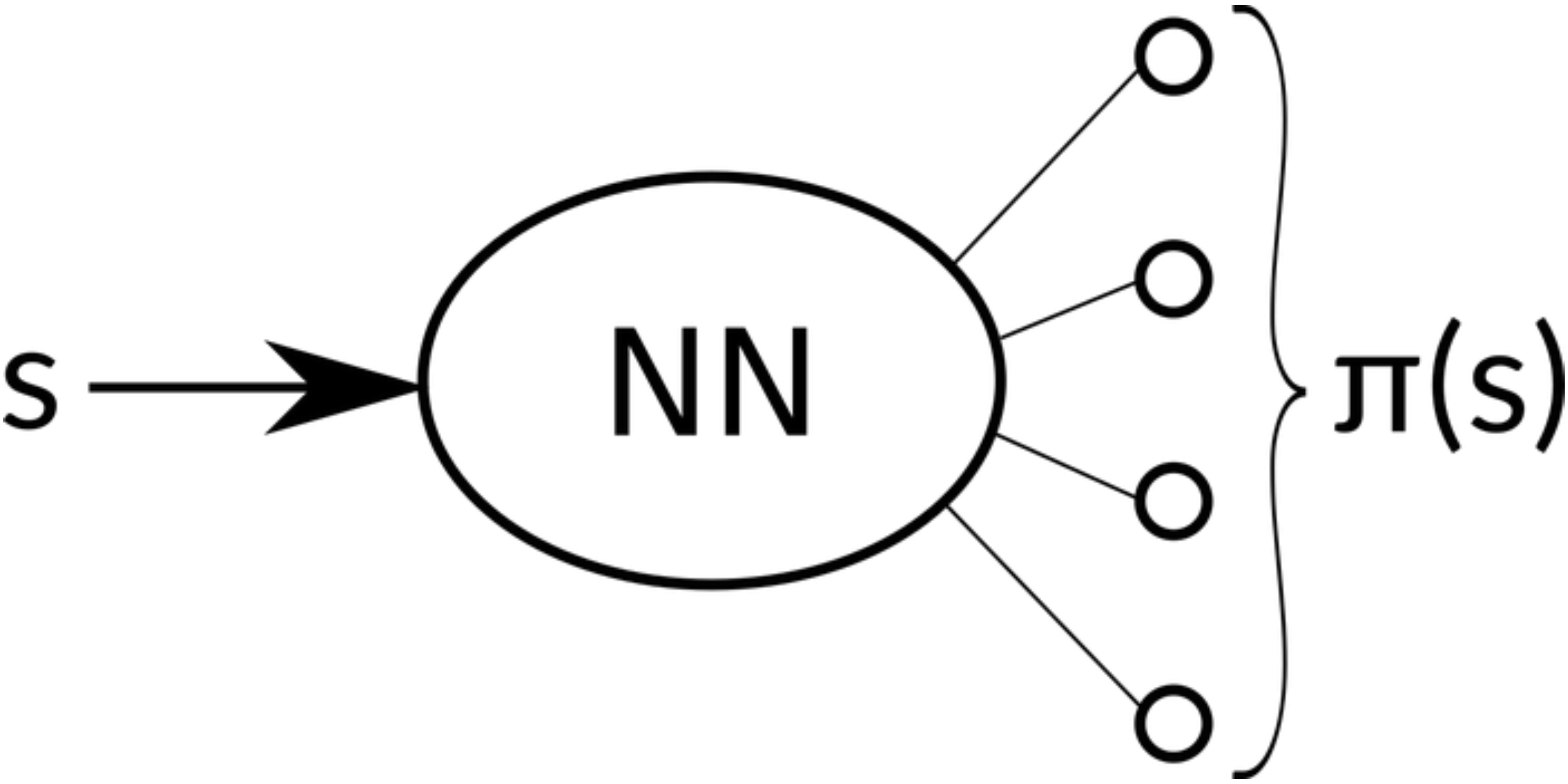
$$A(s, a) = Q(s, a) - V(s)$$

We call *A(s, a)* an advantage function and it expresses how good it is to take an action *a* in a state *s* compared to average. If the action *a* is better than average, the advantage function is positive, if worse, it is negative.

And last, let's define ρ as some distribution of states, saying what the probability of being in some state is. We'll use two notations – ρ^{s_0}, which gives us a distribution of starting states in the environment and ρ^{π}, which gives us a distribution of states under policy π. In other words, it gives us probabilities of being in a state when following policy π.

Policy Gradient

When we built the DQN agent, we used a neural network to approximate the *Q(s, a)* function. But now we will take a different approach. The policy π is just a function of state *s*, so we can approximate directly that. Our neural network with weights θ will now take an state *s* as an input and output an action probability distribution, π_θ. From now on, by writing π it is meant π_θ, a policy parametrized by the network weights θ.



In practice, we can take an action according to this distribution or simply take the action with the highest probability, both approaches have their pros and cons.

But we want the policy to get better, so how do we optimize it? First, we need some metric that will tell us how good a policy is. Let's define a function $J(\pi)$ as a discounted reward that a policy π can gain, averaged over all possible starting states s_0.

$$J(\pi) = E_{\rho^{s_0}}[V(s_0)]$$

We can agree that this metric truly expresses, how good a policy is. The problem is that it's hard to estimate. Good news are, that we don't have to.

What we truly care about is how to improve this quantity. If we knew the gradient of this function, it would be trivial. Surprisingly, it turns out that there's easily computable gradient of $J(\pi)$ function in the following form:

$$\nabla_\theta\, J(\pi) = E_{s\sim\rho^\pi,\, a\sim\pi(s)}[A(s,a) \cdot \nabla_\theta\, log\ \pi(a|s)]$$

I understand that the step from $J(\pi)$ to $\nabla_\theta\, J(\pi)$ looks a bit mysterious, but a proof is out of scope of this article. The formula above is derived in the Policy Gradient Theorem[3] and you can look it up if you want to delve into quite a piece of mathematics. I also direct you to a more digestible online lecture[4], where David Silver explains the theorem and also a concept of baseline, which I already incorporated.

The formula might seem intimidating, but it's actually quite intuitive when it's broken down. First, what does it say? It informs us in what direction we have to change the weights of the neural network if we want the function $J(\pi)$ to improve.

Let's look at the right side of the expression. The second term inside the expectation, $\nabla_\theta\, log\ \pi(a|s)$, tells us a direction in which logged probability of taking action *a* in state *s* rises. Simply said, how to make this action in this context more probable.

The first term, $A(s,a)$, is a scalar value and tells us what's the advantage of taking this action. Combined we see that likelihood of actions that are better than average is increased, and likelihood of actions worse than average is decreased. That sounds like a right thing to do.

Both terms are inside an expectation over state and action distribution of π. However, we can't exactly compute it over every state and every action. Instead, we can use that nice property of expectation that the mean of samples with these distributions lays near the expected value.

Fortunately, running an episode with a policy π yields samples distributed exactly as we need. States encountered and actions taken are indeed an unbiased sample from the ρ^π and π *(s)* distributions.

That's great news. We can simply let our agent run in the environment and record the *(s, a, r, s')* samples. When we gather enough of them, we use the formula above to find a good approximation of the gradient $\nabla_\theta \, J(\pi)$. We can then use any of the existing techniques based on gradient descend to improve our policy.

Actor-critic

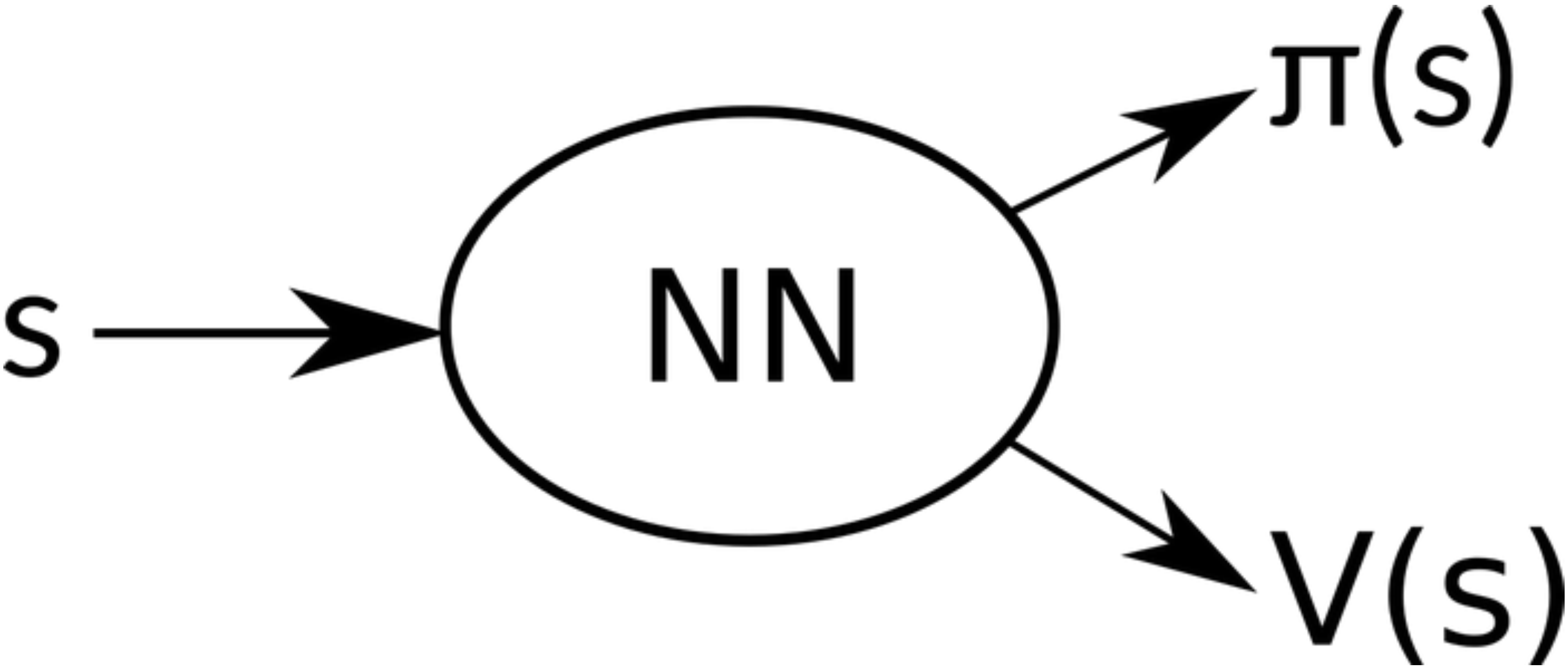
One thing that remains to be explained is how we compute the *A(s, a)* term. Let's expand the definition:

$$A(s,a) = Q(s,a) - V(s) = r + \gamma V(s') - V(s)$$

A sample from a run can give us an unbiased estimate of the *Q(s, a)* function. We can also see that it is sufficient to know the value function *V(s)* to compute *A(s, a)*.

The value function can also be approximated by a neural network, just as we did with action-value function in DQN. Compared to that, it's easier to learn, because there is only one value for each state.

What's more, we can use the same neural network for estimating π*(s)* to estimate *V(s)*. This has multiple benefits. Because we optimize both of these goals together, we learn much faster and effectively. Separate networks would very probably learn very similar low level features, which is obviously superfluous. Optimizing both goals together also acts as a regularizing element and leads to a greater stability. Exact details on how to train our network will be explained in the next article. The final architecture then looks like this:



Our neural network share all hidden layers and outputs two sets – π*(s)* and *V(s)*.

So we have two different concepts working together. The goal of the first one is to optimize the policy, so it performs better. This part is called actor. The second is trying to estimate the value function, to make it more precise. That is called critic. I believe these terms arose from the Policy Gradient Theorem:

$$\nabla_\theta \, J(\pi) = E_{s\sim\rho^\pi,\, a\sim\pi(s)}[A(s,a) \cdot \nabla_\theta \, log \, \pi(a|s)]$$

The actor acts, and the critic gives insight into what is a good action and what is bad.

Parallel agents

The samples we gather during a run of an agent are highly correlated. If we use them as they arrive, we quickly run into issues of online learning. In DQN, we used a technique named *Experience Replay* to overcome this issue. We stored the samples in a memory and retrieved them in random order to form a batch.

But there's another way to break this correlation while still using online learning. We can run several agents in parallel, each with its own copy of the environment, and use their samples as they arrive. Different agents will likely experience different states and transitions, thus avoiding the correlation[2]. Another benefit is that this approach needs much less memory, because we don't need to store the samples.

This is the approach the A3C algorithm takes. The full name is *Asynchronous advantage actor-critic* (A3C) and now you should be able to understand why.

Conclusion

We learned the fundamental theory behind PG methods and will use this knowledge to implement an agent in the next article. We will explain how to use the gradients to train the neural network with our familiar tools, Python, Keras, OpenAI Gym and newly TensorFlow.

References

1. Williams, R., *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, Machine Learning, 1992 **↩**
2. Mnih, V. et al., *Asynchronous methods for deep reinforcement learning*, ICML, 2016 **↩** **↩ (544-2)**
3. Sutton, R. et al., *Policy Gradient Methods for Reinforcement Learning with Function Approximation*, NIPS, 1999 **↩**
4. Silver, D., *Policy Gradient Methods*, **https://www.youtube.com/watch?v=KHZVXao4qXs (https://www.youtube.com/watch?v=KHZVXao4qXs)**, 2015 **↩**

Posted in **A3C** **8 Comments**

8 thoughts on “Let’s make an A3C: Theory”

1. **DarkZero** says:
   February 19, 2017 at 10:49 am
   Hi ヤロミル,

   This is indeed the most clear post I have seen on Policy Gradient and A3C.
   Many other posts just throw a bunch of formula to me and you are the only one speaking human English.
   I’m strongly looking forward to reading your next article on implementation.

   Thank you!

   REPLY

2. **Fabian** says:
   February 17, 2017 at 4:46 pm
   Thanks for the quick response As I can not reply to your Reply, here is a new comment!

   1) That is exactly what I expected, thanks for clarification. So, stable on-policy training only comes through asyncronous workers, right?

   2) Indeed that is one of the cleanest A3C codes I found, didn’t know it was yours

   Anyway, doesn’t Keras allow to define custom loss functions? In the form of **https://github.com/fchollet/keras/blob/master/keras/objectives.py**

   Can’t we put the nasty loss function in one of these, and then proceed to use Keras’ high level train_on_batch function?

   It would make the whole thing more or less backend-agnostic and better understandable

   Regards,
   Fabian

   REPLY

   1. ヤロミル says:
      February 18, 2017 at 12:17 am
      Hi again.

      The problem with Keras loss function is that it is that it always expect (y_true, y_pred) arguments. In our case, however, we don’t have these – we only derived a formula for the gradient. In other words we don’t have targets for our policy. Think about it, y_pred is the distribution given by the network, but what should you pass as y_true?

      That’s why some TensorFlow is necessary, to define the gradient and let it compute it.

   2. **Fabian** says:
      February 18, 2017 at 10:16 am
      Hello jaromiru,

      I don’t know if I just don’t see it, or if its not possible to directly answer to replys, so here is a new comment – once again.

      First of all, thank you very much for taking the time for my questions, I appreciate that a lot

      You say that Keras’ loss functions expect (y_true, y_pred), but can’t we just ignore the y_true part and proceed to define the same loss calculations as in Tensorflow?

      Or is there a difference of how Keras’ handles the gradient calculation/descent afterwards?

      Regards,
      Fabian

   3. ヤロミル says:
      February 22, 2017 at 9:57 am

Yes, comment system seems to be inconvenient.

You have to define the loss function by means of TensorFlow tensors, even in Keras. You have to pass (s, a, r, s_) to the loss function somehow, and you can’t pass it in y_true, because that’s already a TensorFlow tensor. I did some quick experiments but still don’t see any way to do that.

4\.

Fabian says:
February 22, 2017 at 9:58 am
I just wanted to say thanks for all your replys to my questions 🙂 Take care, looking forward to your next article!

3\.

Fabian says:
February 17, 2017 at 10:19 am
As someone that is currently right in the process of learning about RL in general and A3C in particular, I can telly you: This is a really great article, thank you very much! Right now I am learning from reading the original paper and source code, and this here made a lot of stuff clear 🙂

But I also have one question and one request for you, if you dont mind.

The question is:
Your “Parallel agents”-paragraph sounds like replay-memory might word with A3C, but the original paper states the following:
“Since we no longer rely on experience replay for stabilizing learning we are able to use on-policy reinforcement learning methods such as … actor-critic to train … in a stable way. “.

This sounds like the on-policy nature of the algorithm does not work well with memory replay. Do you have any experience on that?
Because if one could use replay-memory, that would not only make some engineering tasks easier (each actor could only take care about itself), but would also allow to pre-train a new NN architecture with the memory of a past experiment. Do you have any experience on this topic? 🙂

The request is:
As I am currently learning about A3C a lot from reading code, I have to say that most code about this topic is not very understandable.

Is it possible that you wrap the whole Tensorflow-Part away and stick to using Keras’ High Level API? It is so much simpler to read and understand 🙂

Correct me if I am wrong, but by introducing your own objective/loss function in Keras – which would then do the exotic gradient calculation – that should work, right?

Anyway, thanks again, this is one of the best RL articles that I have red so far!

Best Regards,
Fabian

REPLY

1\.

ヤロミル says:
February 17, 2017 at 4:34 pm
Hi Fabian, thank you for the comment.

1) The policy gradient formula demands that the state and action distribution are according to the current policy. The main problem with experience replay is that the older samples are made with a different (older) policy, so that equation does not hold anymore. Basically the estimation will give you a different gradient which might not be in the right direction. It is a question how much different this gradient would be and I can imagine that it depends on how big your memory is. It might work to some extent or not. I haven’t experimented with it.

2) Don’t worry, if you have read my first DQN series you know that I try to write clean, readable and understandable code. I too like Keras because it encapsulates all those nasty low level details, however TensorFlow part is necessary in this case. Precisely to define the loss function, which can’t be done with Keras alone. If you browse my github account’s ai_examples repo, you will find my current implementation of A3C algorithm. I don’t give direct link on purpose, that code will be cleaned and edited to fit the next article. But if you really want, feel free to use that to learn now.

Regards,
jaromiru