Mastering Atari with an Actor-Critic Model

Sam Greydanus
greydasa@oregonstate.edu

January 19, 2018

Abstract

Reinforcement Learning (RL) with deep neural networks is an exciting area of research, but training a deep RL agent is difficult in practice. In this talk, I will introduce the theories and intuitions of the (Asynchronous) Advantage Actor-Critic (A3C) model and walk through a 180-line implementation in PyTorch. By the end of my talk, you should have the tools you need to train your own Atari agents.

1 A3C in theory

The idea behind reinforcement learning is that, given an *agent* which can interact with its *environment*, one can learn a policy that maximizes the expected reward.

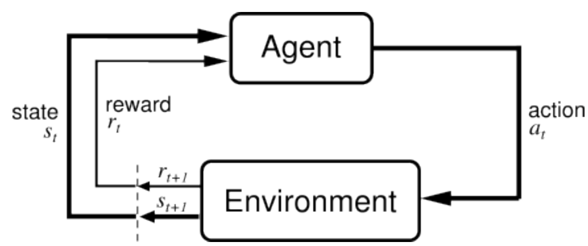

Figure 1: Overview of the RL framework. An agent in state s_t receives reward r_t from the environment. Then, it chooses some internal policy, π to choose action a_t, interacts with the environment, and receives the next state/reward pair.

The idea is simple, but making it work in practice is tricky for a variety of reasons. In this talk, I'll use the Atari video games as example environments because they show off 1) why getting RL to work in practice is tricky 2) why RL is a powerful and interesting tool.

1.1 Policy Gradient Theorem

Suppose we have a reward function $f(x)$ and an agent that acts according to probability distribution $p(x)$. We would like to modify the distribution $p(x)$ to maximize the expectation value $E_x[f(x)]$. Let's assume $p(x)$ is differentiable and we can optimize it using gradient descent with respect to the expected reward.

Now all we need is to take the gradient of the expectation value:

$$
\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation value} && (1)\\
&= \sum_x \nabla_\theta p(x) f(x) && \text{apply gradient to each term} && (2)\\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{multiply by identity} && (3)\\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \text{because} \quad \nabla_\theta \log z = \frac{1}{z}\nabla_\theta z && (4)\\
&= E_x[\nabla_\theta \log p(x) f(x)] && \text{this is the Policy Gradient Theorem} && (5)
\end{aligned}
$$

This gives us the gradient of the expected reward. From now on, let's start using π_θ as our $p(x)$ and call it the agent's *policy*. We'll also refer to the expected reward This gives us the gradient of the expected reward. From now on, let's start using π_θ as our $p(x)$ and call it the agent's *policy*. We'll also refer to the expected reward $E_x[f(x)]$ as J.

1.2 REINFORCE

Many times, our agent will receive a reward not only because of the action a_t which it just took, but also because of the actions which led it into the high-reward state. These actions consist of the set $a_{t-1}, a_{t-2}, \ldots$. We would like to give the agent credit for these actions, especially the ones which helped it obtain the final reward.

There is more than one solution to this problem of *credit assignment*, but here's one idea. As you get closer to the high-reward state, the actions you take are exponentially more likely to have contributed to that reward. If this is the case, then you should spread the reward you received backwards in time. We do this via *gamma-discounted rewards*:

$$R = \sum_{l=0}^{\infty} \gamma^l r_{t+1} \tag{6}$$

Here, γ is a hyperparameter that we, as experimenters, get to set. It generally corresponds to how sparse the rewards are. We will use $\gamma = 0.99$ for all experiments. We'll use the value function $V^\pi(s) = E_s[R]$ to represent the value of the state. If we want to describe the value of taking a particular action once we're in this state, we use the Q function, $Q^\pi(s,a) = R(s,a) + \gamma V^*(\delta(s,a))$.

Now our policy gradient looks like this:

$$\nabla J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^\pi(s,a)] \tag{7}$$

Notice that this is just the gradient of a differentiable function, $\log \pi_\theta(s,a)$, times a scalar which we know how to compute, $Q^\pi(s,a)$. If we want to, we can go ahead and train an agent using gradient descent. This, in fact, is an algorithm called REINFORCE and it works well in some simple cases. Sadly, it doesn't work great for Atari.

Note. Many times, people will clip raw rewards to $[-1, 1]$ before gamma discounting. I tried training on Atari without this step and all my models failed.

1.3 Actor-critic

The big remaining issue is that $Q^\pi(s,a)$ is not a well-behaved function (especially for Atari and other complex environments). It is sparse, high-variance, and totally unnormalized.

The actor-critic trick is meant to reduce the variance of the policy gradient. Instead of using $Q^\pi(s,a)$ directly, we learn an estimator,

$$Q_w(s,a) \approx Q^\pi(s,a) \tag{8}$$

This estimate has much lower variance, leading to better results in practice. Now we have:

$$\nabla J(\theta) \approx E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)] \tag{9}$$

Question. Why not simply reduce variance by keeping a running estimate of batch statistics and then using them to reduce variance? Wouldn't this accomplish the same "smoothing" objective?

Answer. I'm not sure. I tried doing this in practice and it did not work as well. I've seen other people do this *within* batches, which seemed to help with REINFORCE.

1.4 Advantage actor-critic

Another way to reduce the variance is to subtract a baseline from the policy gradient. To see how this works, imagine if our expected reward looked like this:

$$E_{\pi_\theta}[\log \pi_\theta(s,a)(f(s,a) - g(s))] \tag{10}$$

I want to convince you that if g has zero mean, then it won't change the expectation value:

$$\begin{aligned}
E_{\pi_\theta}[\log \pi_\theta(s,a) g(s)] =& \sum_{s\in S}\sum_{a\in A} \pi_\theta(s,a) g(s) && \text{from eqn. 2} \qquad (11)\\
=& \sum_{s\in S} g(s) \sum_{a\in A} \pi_\theta(s,a) && (12)\\
=& \sum_{s\in S} g(s) && \text{because } \sum_{a\in A}\pi_\theta(s,a) = 1 \qquad (13)\\
=& 0 && \text{if } g \text{ has zero mean} \qquad (14)
\end{aligned}$$

Now, because $E[A - B] = E[A] - E[B]$, we can conclude that

$$E_{\pi_\theta}[\log \pi_\theta(s,a)(f(s,a) - g(s))] = E_{\pi_\theta}[\log \pi_\theta(s,a) f(s,a)] \tag{15}$$

People call this "subtracting a baseline" and it is another way to reduce variance. A good baseline ends up being the value function, $V^\pi(s)$. Recall that the advantage function is defined as

$$\begin{aligned}
A^\pi(s,a) =& Q^\pi(s,a) - V^\pi(s) && (16)\\
\approx& Q_w(s,a) - V_v(s) && (17)
\end{aligned}$$

In equation 17 I've replaced the value function with a learned estimate, just as we did for the Q-values in section 1.3. So, our new policy gradient becomes

$$\nabla J(\theta) \approx E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) A^\pi(s,a)] \tag{18}$$

1.5 TD actor-critic

One remaining issue is that, if we want to estimate the advantage using a neural network, we need two separate estimators, one for $Q_w(s,a)$ and one for $V_v(s)$. There's a better way: using the TD error, we can estimate the advantage directly from $V_v(s)$. Why? The TD error is an unbiased estimate of the advantage function. Recall that the TD error is given by

$$\delta^{\pi_\theta} = r + \gamma V^\pi(s_{t+1}) - V^\pi(s) \tag{19}$$

The expectation value of the TD error looks like:

$$\begin{aligned} E[\delta^{\pi_\theta}|s,a] =& E[r + \gamma V^\pi(s_{t+1})|s,a] - V^\pi(s) && (20) \\ =& Q^\pi(s,a) - V^\pi(s) && (21) \\ =& A^\pi(s,a) && (22) \end{aligned}$$

Thus, if we use TD error, we only need to train one value estimator to approximate the advantage. The policy gradient for our final advantage actor-critic model looks like:

$$\nabla J(\theta) \approx E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\delta^{\pi_\theta}] \tag{23}$$

This approximation produces gradients with much lower variance, giving a deep neural network enough signal to learn to play Atari games very well.

2 A3C in practice

Making deep RL work in practice an art form unto itself. It takes a lot of clever engineering! As a result, most open-source implementations are large, clunky, and confusing. Otherwise, they are small, overly simplistic, and can't handle Atari. Frustrated by this, I ended up writing my own version called `baby-a3c`[1] which makes solving the Atari environments as simple as possible...but not simpler!

I've appended snapshots of the code below.

[1] github.com/greydanus/baby-a3c

```python
# Baby Advantage Actor-Critic | Sam Greydanus | October 2017 | MIT License

from __future__ import print_function
import torch, os, gym, time, glob, argparse
import numpy as np
from scipy.signal import lfilter
from scipy.misc import imresize # preserves single-pixel info _unlike_ img = img[::2,::2]

import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F
import torch.multiprocessing as mp
os.environ['OMP_NUM_THREADS'] = '1'

parser = argparse.ArgumentParser(description=None)
parser.add_argument('--env', default='Breakout-v0', type=str, help='gym environment')
parser.add_argument('--processes', default=20, type=int, help='number of processes to train with')
parser.add_argument('--render', default=False, type=bool, help='renders the atari environment')
parser.add_argument('--test', default=False, type=bool, help='test mode sets lr=0, chooses most likely actions')
parser.add_argument('--lstm_steps', default=20, type=int, help='steps to train LSTM over')
parser.add_argument('--lr', default=1e-4, type=float, help='learning rate')
parser.add_argument('--seed', default=1, type=int, help='seed random # generators (for reproducibility)')
parser.add_argument('--gamma', default=0.99, type=float, help='discount for gamma-discounted rewards')
parser.add_argument('--tau', default=1.0, type=float, help='discount for generalized advantage estimation')
parser.add_argument('--horizon', default=0.99, type=float, help='horizon for running averages')
args = parser.parse_args()

args.save_dir = '{}/'.format(args.env.lower()) # keep the directory structure simple
if args.render:  args.processes = 1 ; args.test = True # render mode -> test mode w one process
if args.test:  args.lr = 0 # don't train in render mode
args.num_actions = gym.make(args.env).action_space.n # get the action space of this game
os.makedirs(args.save_dir) if not os.path.exists(args.save_dir) else None # make dir to save models etc.

discount = lambda x, gamma: lfilter([1],[1,-gamma],x[::-1])[::-1] # discounted rewards one liner
prepro = lambda img: imresize(img[35:195].mean(2), (80,80)).astype(np.float32).reshape(1,80,80)/255.

def printlog(args, s, end='\n', mode='a'):
    print(s, end=end) ; f=open(args.save_dir+'log.txt',mode) ; f.write(s+'\n') ; f.close()

class NNPolicy(torch.nn.Module): # an actor-critic neural network
    def __init__(self, channels, num_actions):
        super(NNPolicy, self).__init__()
        self.conv1 = nn.Conv2d(channels, 32, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(32, 32, 3, stride=2, padding=1)
        self.conv4 = nn.Conv2d(32, 32, 3, stride=2, padding=1)
        self.lstm = nn.LSTMCell(32 * 5 * 5, 256)
        self.critic_linear, self.actor_linear = nn.Linear(256, 1), nn.Linear(256, num_actions)

    def forward(self, inputs):
        inputs, (hx, cx) = inputs
        x = F.elu(self.conv1(inputs))
        x = F.elu(self.conv2(x))
        x = F.elu(self.conv3(x))
        x = F.elu(self.conv4(x))
        hx, cx = self.lstm(x.view(-1, 32 * 5 * 5), (hx, cx))
        return self.critic_linear(hx), self.actor_linear(hx), (hx, cx)

    def try_load(self, save_dir):
        paths = glob.glob(save_dir + '*.tar') ; step = 0
        if len(paths) > 0:
            ckpts = [int(s.split('.')[-2]) for s in paths]
            ix = np.argmax(ckpts) ; step = ckpts[ix]
            self.load_state_dict(torch.load(paths[ix]))
        print("\tno saved models") if step is 0 else print("\tloaded model: {}".format(paths[ix]))
        return step
```

```python
class SharedAdam(torch.optim.Adam): # extend a pytorch optimizer so it shares grads across processes
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight_decay=0):
        super(SharedAdam, self).__init__(params, lr, betas, eps, weight_decay)
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['shared_steps'], state['step'] = torch.zeros(1).share_memory_(), 0
                state['exp_avg'] = p.data.new().resize_as_(p.data).zero_().share_memory_()
                state['exp_avg_sq'] = p.data.new().resize_as_(p.data).zero_().share_memory_()

        def step(self, closure=None):
            for group in self.param_groups:
                for p in group['params']:
                    if p.grad is None: continue
                    self.state[p]['shared_steps'] += 1
                    self.state[p]['step'] = self.state[p]['shared_steps'][0] - 1 # there's a "step += 1" later
            super.step(closure)

torch.manual_seed(args.seed)
shared_model = NNPolicy(channels=1, num_actions=args.num_actions).share_memory()
shared_optimizer = SharedAdam(shared_model.parameters(), lr=args.lr)

info = {k : torch.DoubleTensor([0]).share_memory_() for k in ['run_epr', 'run_loss', 'episodes', 'frames']}
info['frames'] += shared_model.try_load(args.save_dir)*1e6
if int(info['frames'][0]) == 0: printlog(args,'', end='', mode='w') # clear log file

def cost_func(values, logps, actions, rewards):
    np_values = values.view(-1).data.numpy()

    # generalized advantage estimation (a policy gradient method)
    delta_t = np.asarray(rewards) + args.gamma * np_values[1:] - np_values[:-1]
    gae = discount(delta_t, args.gamma * args.tau)
    logpys = logps.gather(1, Variable(actions).view(-1,1))
    policy_loss = -(logpys.view(-1) * Variable(torch.Tensor(gae))).sum()

    # l2 loss over value estimator
    rewards[-1] += args.gamma * np_values[-1]
    discounted_r = discount(np.asarray(rewards), args.gamma)
    discounted_r = Variable(torch.Tensor(discounted_r))
    value_loss = .5 * (discounted_r - values[:-1,0]).pow(2).sum()

    entropy_loss = -(-logps * torch.exp(logps)).sum() # encourage lower entropy
    return policy_loss + 0.5 * value_loss + 0.01 * entropy_loss
```

```python
def train(rank, args, info):
    env = gym.make(args.env) # make a local (unshared) environment
    env.seed(args.seed + rank) ; torch.manual_seed(args.seed + rank) # seed everything
    model = NNPolicy(channels=1, num_actions=args.num_actions) # init a local (unshared) model
    state = torch.Tensor(prepro(env.reset())) # get first state

    start_time = last_disp_time = time.time()
    episode_length, epr, eploss, done  = 0, 0, 0, True # bookkeeping

    while info['frames'][0] <= 8e7 or args.test: # openai baselines uses 40M frames...we'll use 80M
        model.load_state_dict(shared_model.state_dict()) # sync with shared model

        cx = Variable(torch.zeros(1, 256)) if done else Variable(cx.data) # lstm memory vector
        hx = Variable(torch.zeros(1, 256)) if done else Variable(hx.data) # lstm activation vector
        values, logps, actions, rewards = [], [], [], [] # save values for computing gradientss

        for step in range(args.lstm_steps):
            episode_length += 1
            value, logit, (hx, cx) = model((Variable(state.view(1,1,80,80)), (hx, cx)))
            logp = F.log_softmax(logit)

            action = logp.max(1)[1].data if args.test else torch.exp(logp).multinomial().data[0]
            state, reward, done, _ = env.step(action.numpy()[0])
            if args.render: env.render()

            state = torch.Tensor(prepro(state)) ; epr += reward
            reward = np.clip(reward, -1, 1) # reward
            done = done or episode_length >= 1e4 # keep agent from playing one episode too long

            info['frames'] += 1 ; num_frames = int(info['frames'][0])
            if num_frames % 2e6 == 0: # save every 2M frames
                printlog(args, '\n\t{:.0f}M frames: saved model\n'.format(num_frames/1e6))
                torch.save(shared_model.state_dict(), args.save_dir+'model.{:.0f}.tar'.format(num_frames/1e6))

            if done: # update shared data. maybe print info.
                info['episodes'] += 1
                interp = 1 if info['episodes'][0] == 1 else 1 - args.horizon
                info['run_epr'].mul_(1-interp).add_(interp * epr)
                info['run_loss'].mul_(1-interp).add_(interp * eploss)

                if rank ==0 and time.time() - last_disp_time > 60: # print info ~ every minute
                    elapsed = time.strftime("%Hh %Mm %Ss", time.gmtime(time.time() - start_time))
                    printlog(args, 'time {}, episodes {:.0f}, frames {:.1f}M, run epr {:.2f}, run loss {:.2f}'
                        .format(elapsed, info['episodes'][0], num_frames/1e6, info['run_epr'][0], info['run_loss'][0]))
                    last_disp_time = time.time()

                episode_length, epr, eploss = 0, 0, 0
                state = torch.Tensor(prepro(env.reset()))

            values.append(value) ; logps.append(logp) ; actions.append(action) ; rewards.append(reward)

        next_value = Variable(torch.zeros(1,1)) if done else model((Variable(state.unsqueeze(0)), (hx, cx)))[0]
        values.append(Variable(next_value.data))

        loss = cost_func(torch.cat(values), torch.cat(logps), torch.cat(actions), np.asarray(rewards))
        eploss += loss.data[0]
        shared_optimizer.zero_grad() ; loss.backward()
        torch.nn.utils.clip_grad_norm(model.parameters(), 40)

        for param, shared_param in zip(model.parameters(), shared_model.parameters()):
            if shared_param.grad is None: shared_param._grad = param.grad # sync gradients with shared model
        shared_optimizer.step()

processes = []
for rank in range(args.processes):
    p = mp.Process(target=train, args=(rank, args, info))
    p.start() ; processes.append(p)
for p in processes:
    p.join()
```