A Simple Policy Gradient Framework for Deep Reinforcement Learning

Tom Wilson

May 4 2022

The Policy Gradient

Unlike **value-based** methods in reinforcement learning where we are using a function to approximate $V_\theta(s) \approx V^\pi(s) \in \setminus$ or $Q_\theta(s,a) \approx Q^\pi(s,a) \in \mathbb{R}$, we can also use a **policy-based** approach to directly parameterize the policy.

$$\pi_\theta(a|s) \triangleq \mathbb{P}[a|s;\theta]$$

Instead of extracting our policy from a value function, we use an estimation function to model the policy mapping from states to actions, and update our approximation from the sampled transitions of steps taken using the behaviour policy. There are certain advantages to using policy-based algorithm to solve a Markov Decision Process. It has been shown that policy-based rl has better convergence properties, as well as greater effectiveness in high-dimensional continuous action spaces. For these reasons, many of the most popular state of the art reinforcement learnign algorithms are partially based on the policy gradient theorem. However, they typically have high variance, and are not guaranteed to converge to a globally optimal policy. The policy gradient is defined as follows:

$$\nabla_\theta J_\tau \approx \mathbb{E}[\sum_t^\infty \Psi_t \nabla_\theta \pi_\theta(a_t|s_t)]$$

Where J_τ is the utility calculation of the playout τ, and A is the 'advantage' at time t, often represented by the Q function (be it an approximation or the actual reward-to-go).

Score Calculation

We need some notion of what $\nabla_\theta \pi_\theta(a_t|s_t)$ is in the context of our choice for π_θ. Similar to other approximate model-free reinforcement learning algorithms, in this policy gradient framework, gradient optimization is used in order to coverge to the optimal policy using the data obtained from trial-and-error playouts. Since we are taking a probabilistic interpretation of the policy distribution, our gradient optimization at the output of the approximation can be interpreted as a form of Maximum Likelihood Estimation. the most common choices, and the options that are provided in this framework, are the **SoftMax Distribution**: as well as the **Gaussian Distribution**: It is important to remember that with each of these functions, we only need to concern ourselves with the logarithm of the probability

$$SM(x) \triangleq \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Where x_i is the chosen action. this is a version of the logistic function which allows the sum of all mappings from the input to sum to 1. for this reason, it is great for using to approximate probabilities of discrete *action spaces.*

$$N(x, \mu, \sigma) \triangleq \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{x-\mu}{\sigma}^2}$$

Where μ is the mean of the distribution and σ is the standard deviation. This function is a geat choice for approximating continuous action spaces. Note that the term $\frac{x-\mu}{\sigma}$ represents the z-score of the sampled action

desity, as our primary concern in reinforcement learning is the amount of information that is gained with each action taken and each state transition. The score calculation is therefore the gradient of each of these:
Soft-Max:

$$\nabla_\theta \log \pi_\theta(a|s;\theta) = \nabla_\theta[\phi(s) - \mathbb{E}_{a' \ \pi_\theta(a'|s)}[\psi(s, a')]$$

Gaussian:

$$\nabla_{\theta_\mu} \log \pi_\theta(a|s;\theta_\mu) = \frac{(a-\mu)\phi(s)}{\sigma^2}$$

$$\nabla_{\theta_\sigma} \log \pi_\theta(a|s;\theta_\sigma) = (\frac{(a-\mu)^2}{\sigma^2} - 1)\phi(s)$$

Implementation Considerations & Approach:

To correctly compute the score function is perhaps the most crucial feature of this framework, regardless of the algorithm used or the architecture of the approximation. In both cases, it is important to scale the hidden states and to calculate log softmax in such a way that prevents floating point overflow and underflow, as the exponentiation of inputs forces the rounding of the floating point number. This problem can be addresed by using the Log Sum Exponential trick, or by simply clamping the output of the approximation. Additionally, it is worth noting that the product between the state transformation and the loss calculation at the output of the function is an outer product, as the shape of the gradient must of course match the shape of the parameters θ.

The practical implementation of the calculation with Numpy for a **SoftMax approximation** is as follows:

```
def score(self, s_t, a_t):
    ''' In this version of the implementation, computes the gradient wrt to the output of the
     softmax function,
  which follows a single linear combination layer of the state input.
  Params
  ______
    s_t: state at time t
    a_t: action taken from s_t
    '''
    #compute SoftMax
    sm_probs = self.forward(s_t)

    #one-hot encode action
    act_mask = np.zeros(self.action_dim)
    act_mask[a_t] = 1

    #score calculation & gradient update
    score = np.outer(s_t, act_mask) - np.outer(s_t, np.exp(sm_probs))
    self.d_weights += score

    assert not np.isnan(self.d_weights).any()
    return score
```

For the case of the **Gaussian approximation**, it is a different calculation due to the fact that the probability density is calculated differently between these two functions. The variance in practice need not be parameterized by some θ_σ as described above, it may also be a fixed value for the behavior policy. Also note that the mean and action are often squashed between zero and one before being passed to this calculation in order to prevent floating point error. The implementation with Numpy is as follows:

```
def score(self, s_t, a_t):
  ''' In this version of the implementation, computes the gradient wrt to the output of the softmax
      function,
      which follows a single linear combination layer of the state input.
  '''
  mu, sd = self.forward(s_t)

  mu_score = (a_t - mu)/(sd**2)
  sd_score = ((a_t - mu)**2) / sd**2 - 1
  # print(mu, sd, a_t)
  mu_grad = np.outer(s_t, mu_score)
  sd_grad = np.outer(s_t, sd_score)
  # print(sd_grad)

  assert not np.isnan(mu_grad).any()
  assert not np.isnan(sd_grad).any()
  self.d_mu_weight += mu_grad
  self.d_sd_weight += sd_grad
  return [mu_grad, sd_grad]
```

Using the Gaussian distribution to model actions much more expensive and much less sample efficient when using the naive implementation such as REINFORCE. To combat this problem we would want to use batching/vectorization of transitions, saving efficiency by using the 'surrogate loss' score, which means computing the score expectation within the gradient calculation.

Policy Approximations

In order to approximate an action in a complex environment, we need to break down each element in both our state space $s \in \mathcal{S}$ and action space $a \in \mathcal{A}$ into a set of features. A flattened* state s_t can be represented as a vector and passed into a neural network. This is the approximation setup that we will be using. Each factor of the action space can be represented as a full linear combination of the state: Computing the estimation with a setup

$$\log \pi(\begin{bmatrix} \theta_{11} & \theta_{12} & ... & \theta_{1m} \\ \theta_{21} & \theta_{22} & ... & \theta_{2m} \\ \vdots & & \ddots & \vdots \\ \theta_{n1} & \theta_{n2} & ... & \theta_{nm} \end{bmatrix} \begin{bmatrix} \phi(s_1) \\ \phi(s_2) \\ \vdots \\ \phi(s_m) \end{bmatrix}) = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

Where π is the our chosen probability density as described above (SoftMax, Gaussian).

like the one described above is sufficient and effective for many problems, and it can be implemented quite easily with Numpy by taking a matrix dot product with state. This matrix can be updating using the gradient computation $\theta = \theta + \nabla_\theta \log \pi(s|a)$ described in the previous section. However, This is not the most rich representation possible, and it must be assumed that each factor of the action can be chosen correctly by a policy that is a linear function of the state. We provide the option of using a full feed forward neural network to approximate the state transformation, with some number (2) of intermediate hidden layers (default 32 neurons each), each followed by ReLU nonlinearity. For higher dimensional problems with more complex solutions, this is may allow for a higher peak of performance or different convergence properties, but it is highly problem dependant. Empirically, networks that are exceeding large cause overfitting, which appears as saddles in the optimization curve. Increasing the size of the parameter representation also allows us to have some control over the complexity of the approximation.

**For image-like observations, a convolutional neural network feature extractor is provided, see code for exact architecture*

The REINFORCE Algorithm

With this definition of a policy gradient optimization method that updates an approximation based on an action taken from a state, we may define a simple optimization loop which, every iteration, collects a full playout based on the current behaviour policy, calculates the score of the approximation, and calculates the reward-to-go from each state in the playout. The parameters of the approximation are updated by the score, scaled by the reward-to-go and the step size. This method was first proposed by RJ Williams in 1992, and outlines the basic optimization loop that follows in a general form in many other more sophisticated policy gradient methods.

Algorithm 1 REINFORCE

Initialize θ_π
while $\pi_\theta \neq \pi^*$ **do**
  $\tau = \{s_0, a_0, r_0, ...s_T, a_T, r_T\}$
  for $\{s_t, a_t, r_t\} \in \tau$ **do**
    $\theta_\pi = \theta_\pi + \alpha \nabla_\theta \log \pi_\theta(a|s) \sum_{t=i}^{T} \gamma^{t-i} r_t$
  end for
end while

Practical Implementation

For the implementation of this algorithm, we define a general function $G_\tau(s)$ which simply represents the reward-to-go from s_t given τ. We also define π as the policy approximation with the methods *score*, which computes $\nabla_\theta \log \pi_\theta(a|s)$, and *optimize*, which performs the optimizing step and updates the parameters θ of the approximation.

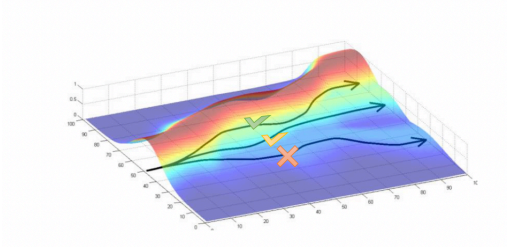
```
class REINFORCEAgent(PolicyGradientAgent):

    def __init__(self,
                 policy_fn:Callable,
                 gamma: float = 0.99,
                 env: gym.Env = None) -> None:
        super().__init__(gamma, env)

        self.pi = policy_fn
        self.G = MonteCarloReturns(self.gamma)


    def train(self, n_iter):
        pbar = tqdm(range(1, n_iter+1), file=sys.stdout)
        for episode in pbar:
            tau = self.playout()
            self.G.update_baseline(tau)
            for s_t, a_t, r in tau:
                score = self.pi.score(s_t, a_t)
                self.pi.optimize(
                        score,
                        self.G(s_t))
```

Using just this setup and a single dense feed-forward approximation, we are able to solve most problems with discrete action spaces **Performance:** Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Value Functions

Monte-Carlo Playouts

With any policy gradient based algorithm, it is important to specify the value of any state transition, so that we may scale the rewarding actions accordingly, adjusting our policy estimation more towards choosing these actions, and less towards less rewarding actions.



Paths across optimization landscape

Using the REINFORCE Algorithm, we collect playouts at the beginning of each loop, so the value function of any state in the playout can be the actual reward-to go calulation, simply the sum of all future rewards following that observation. We may optionally specify a discount factor γ for this calculation, however, in the case of sparse rewards, or uniform-dense rewards, it is almost always better to have a discount factor close to 1.

References

Karpathy, A. (2016, May 31). *Deep reinforcement learning: Pong from pixels.* Andrej Karpathy blog. Retrieved from `https://karpathy.github.io/2016/05/31/rl/`

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

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *International conference on machine learning* (pp. 387–395).

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*(3), 229–256.