Policy Gradient Methods

Milan Straka

November 25, 2019
Instead of predicting expected returns, we could train the method to directly predict the policy 
\[ \pi(a|s; \theta). \]

Obtaining the full distribution over all actions would also allow us to sample the actions according to the distribution \( \pi \) instead of just \( \varepsilon \)-greedy sampling.

However, to train the network, we maximize the expected return \( v_\pi(s) \) and to that account we need to compute its gradient \( \nabla_\theta v_\pi(s) \).
Policy Gradient Methods

In addition to discarding $\varepsilon$-greedy action selection, policy gradient methods allow producing policies which are by nature stochastic, as in card games with imperfect information, while the action-value methods have no natural way of finding stochastic policies (distributional RL might be of some use though).

$$J(\theta) = v_{\pi_\theta}(S)$$

Let $\pi(a|s; \theta)$ be a parametrized policy. We denote the initial state distribution as $h(s)$ and the on-policy distribution under $\pi$ as $\mu(s)$. Let also $J(\theta) \overset{\text{def}}{=} \mathbb{E}_{h,\pi} v_\pi(s)$.

Then

$$\nabla_\theta v_\pi(s) \propto \sum_{s' \in S} P(s \rightarrow \ldots \rightarrow s'|\pi) \sum_{a \in A} q_\pi(s', a) \nabla_\theta \pi(a|s'; \theta)$$

and

$$\nabla_\theta J(\theta) \propto \sum_{s \in S} \mu(s) \sum_{a \in A} q_\pi(s, a) \nabla_\theta \pi(a|s; \theta),$$

where $P(s \rightarrow \ldots \rightarrow s'|\pi)$ is probability of transitioning from state $s$ to $s'$ using $0$, $1$, $\ldots$ steps.
Proof of Policy Gradient Theorem

\[ \nabla v_\pi(s) = \nabla \left[ \sum_a \pi(a|s; \theta) q_\pi(s, a) \right] \]

\[ = \sum_a \left[ \nabla \pi(a|s; \theta) q_\pi(s, a) + \pi(a|s; \theta) \nabla q_\pi(s, a) \right] \]

\[ = \sum_a \left[ \nabla \pi(a|s; \theta) q_\pi(s, a) + \pi(a|s; \theta) \nabla \left( \sum_{s'} p(s'|s, a)(r + v_\pi(s')) \right) \right] \]

\[ = \sum_a \left[ \nabla \pi(a|s; \theta) q_\pi(s, a) + \pi(a|s; \theta) \left( \sum_{s'} p(s'|s, a) \nabla v_\pi(s') \right) \right] \]

We now expand \( v_\pi(s') \).

\[ = \sum_{s' \in S} \sum_{a'} \left[ \nabla \pi(a'|s'|; \theta) q_\pi(s', a') + \pi(a'|s'|; \theta) \nabla \left( \sum_{s''} p(s''|s', a') \nabla v_\pi(s'') \right) \right] \]

Continuing to expand all \( v_\pi(s'') \), we obtain the following:

\[ \nabla v_\pi(s) = \sum_{s' \in S} P(s \rightarrow \ldots \rightarrow s'|\pi) \sum_{a \in A} q_\pi(s', a) \nabla_\theta \pi(a|s'; \theta). \]
Proof of Policy Gradient Theorem

Recall that the initial state distribution is $h(s)$ and the on-policy distribution under $\pi$ is $\mu(s)$. If we let $\eta(s)$ denote the number of time steps spent, on average, in state $s$ in a single episode, we have

$$\eta(s) = h(s) + \sum_{s'} \eta(s') \sum_a \pi(a|s')p(s'|s, a).$$

The on-policy distribution is then the normalization of $\eta(s)$:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}.$$

The last part of the policy gradient theorem follows from the fact that $\mu(s)$ is

$$\mu(s) = \mathbb{E}_{s_0 \sim h(s)} P(s_0 \rightarrow \ldots \rightarrow s|\pi).$$
The REINFORCE algorithm (Williams, 1992) uses directly the policy gradient theorem, maximizing $J(\theta) \overset{\text{def}}{=} \mathbb{E}_{h, \pi} v_{\pi}(s)$. The loss is defined as

$$-\nabla_{\theta} J(\theta) \propto \sum_{s \in S} \mu(s) \sum_{a \in A} q_{\pi}(s, a) \nabla_{\theta} \pi(a|s; \theta)$$

$$= \mathbb{E}_{s \sim \mu} \sum_{a \in A} q_{\pi}(s, a) \nabla_{\theta} \pi(a|s; \theta).$$

However, the sum over all actions is problematic. Instead, we rewrite it to an expectation which we can estimate by sampling:

$$-\nabla_{\theta} J(\theta) \propto \mathbb{E}_{s \sim \mu} \mathbb{E}_{a \sim \pi} q_{\pi}(s, a) \nabla_{\theta} \ln \pi(a|s; \theta),$$

where we used the fact that

$$\nabla_{\theta} \ln \pi(a|s; \theta) = \frac{1}{\pi(a|s; \theta)} \nabla_{\theta} \pi(a|s; \theta).$$
REINFORCE Algorithm

REINFORCE therefore minimizes the loss

$$-\mathbb{E}_{s \sim \mu} \mathbb{E}_{a \sim \pi} q_\pi(s, a) \nabla_\theta \ln \pi(a | s; \theta),$$

estimating the $q_\pi(s, a)$ by a single sample.

Note that the loss is just a weighted variant of negative log likelihood (NLL), where the sampled actions play a role of gold labels and are weighted according to their return.

**REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for $\pi_*$**

- **Input:** a differentiable policy parameterization $\pi(a | s, \theta)$
- **Algorithm parameter:** step size $\alpha > 0$
- **Initialize policy parameter** $\theta \in \mathbb{R}^{d'}$ (e.g., to 0)

Loop forever (for each episode):

1. Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot | \cdot, \theta)$
2. Loop for each step of the episode $t = 0, 1, \ldots, T - 1$:
   
   $$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$
   
   $$\theta \leftarrow \theta + \alpha G \nabla \ln \pi(A_t | S_t, \theta)$$

Modification of Algorithm 13.3 of "Reinforcement Learning: An Introduction, Second Edition".
REINFORCE with Baseline

The returns can be arbitrary – better-than-average and worse-than-average returns cannot be recognized from the absolute value of the return.

Hopefully, we can generalize the policy gradient theorem using a baseline $b(s)$ to

$$\nabla_\theta J(\theta) \propto \sum_{s \in S} \mu(s) \sum_{a \in A} (q_\pi(s, a) - b(s)) \nabla_\theta \pi(a|s; \theta).$$

The baseline $b(s)$ can be a function or even a random variable, as long as it does not depend on $a$, because

$$\sum_a b(s) \nabla_\theta \pi(a|s; \theta) = b(s) \sum_a \nabla_\theta \pi(a|s; \theta) = b(s) \nabla 1 = 0.$$
REINFORCE with Baseline

A good choice for $b(s)$ is $v_\pi(s)$, which can be shown to minimize variance of the estimator. Such baseline reminds centering of returns, given that

$$v_\pi(s) = \mathbb{E}_{a \sim \pi} q_\pi(s, a).$$

Then, better-than-average returns are positive and worse-than-average returns are negative. The resulting $q_\pi(s, a) - v_\pi(s)$ function is also called an advantage function

$$a_\pi(s, a) \overset{\text{def}}{=} q_\pi(s, a) - v_\pi(s).$$

Of course, the $v_\pi(s)$ baseline can be only approximated. If neural networks are used to estimate $\pi(a|s; \theta)$, then some part of the network is usually shared between the policy and value function estimation, which is trained using mean square error of the predicted and observed return.
REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \theta)$
Input: a differentiable state-value function parameterization $\hat{v}(s, w)$
Algorithm parameters: step sizes $\alpha^\theta > 0$, $\alpha^w > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$ (e.g., to 0)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\delta \leftarrow G - \hat{v}(S_t, w)$
    $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S_t, w)$
    $\theta \leftarrow \theta + \alpha^\theta \delta \nabla \ln \pi(A_t|S_t, \theta)$
REINFORCE with Baseline

**Figure 13.2 of “Reinforcement Learning: An Introduction, Second Edition”**

![Graph showing REINFORCE with baseline](image)

- **$G_0$**
  - Total reward on episode
  - Averaged over 100 runs

**Policy Gradient Methods**
- REINFORCE
- Baseline
- Actor-Critic
- A3C
- PAAC
- Continuous Action Space
Actor-Critic

It is possible to combine the policy gradient methods and temporal difference methods, creating a family of algorithms usually called *actor-critic* methods.

The idea is straightforward – instead of estimating the episode return using the whole episode rewards, we can use $n$-step temporal difference estimation.
One-step Actor–Critic (episodic), for estimating $\pi_\theta \approx \pi^*$

Input: a differentiable policy parameterization $\pi(a|s, \theta)$
Input: a differentiable state-value function parameterization $\hat{v}(s,w)$
Parameters: step sizes $\alpha^\theta > 0$, $\alpha^w > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$ (e.g., to 0)
Loop forever (for each episode):
  Initialize $S$ (first state of episode)
  
  Loop while $S$ is not terminal (for each time step):
  
  $A \sim \pi(\cdot|S, \theta)$
  Take action $A$, observe $S', R$
  $\delta \leftarrow R + \gamma \hat{v}(S',w) - \hat{v}(S,w)$ (if $S'$ is terminal, then $\hat{v}(S',w) \equiv 0$)
  $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S,w)$
  $\theta \leftarrow \theta + \alpha^\theta \delta \nabla \ln \pi(A|S, \theta)$
  $S \leftarrow S'$


Policy Gradient Methods  REINFORCE  Baseline  Actor-Critic  A3C  PAAC  Continuous Action Space
Asynchronous Methods for Deep RL

A 2015 paper from Volodymyr Mnih et al., the same group as DQN. The authors propose an asynchronous framework, where multiple workers share one neural network, each training using either an off-line or on-line RL algorithm.

They compare 1-step Q-learning, 1-step Sarsa, $n$-step Q-learning and A3C (an \textit{asynchronous advantage actor-critic} method). For A3C, they compare a version with and without LSTM. The authors also introduce \textit{entropy regularization term} $\beta H(\pi(s; \theta))$ to the loss to support exploration and discourage premature convergence.
Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

//Assume global shared $\theta$, $\theta^-$, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state $s$

repeat
    Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
    Receive new state $s'$ and reward $r$
    $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
    Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial(y - Q(s,a;\theta))^2}{\partial \theta}$
    $s = s'$
    $T \leftarrow T + 1$ and $t \leftarrow t + 1$
    if $T \mod I_{target} == 0$ then
        Update the target network $\theta^- \leftarrow \theta$
    end if
    if $t \mod I_{AsyncUpdate} == 0$ or $s$ is terminal then
        Perform asynchronous update of $\theta$ using $d\theta$.
        Clear gradients $d\theta \leftarrow 0$.
    end if
until $T > T_{max}$
Asynchronous Methods for Deep RL

Algorithm S2: Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vector \( \theta \).
// Assume global shared target parameter vector \( \theta^- \).
// Assume global shared counter \( T = 0 \).
Initialize thread step counter \( t \leftarrow 1 \)
Initialize target network parameters \( \theta^- \leftarrow \theta \)
Initialize thread-specific parameters \( \theta' = \theta \)
Initialize network gradients \( d\theta \leftarrow 0 \)

repeat
    Clear gradients \( d\theta \leftarrow 0 \)
    Synchronize thread-specific parameters \( \theta' = \theta \)
    \( t_{start} = t \)
    Get state \( s_t \)
    \( T \leftarrow T + 1 \)
    \( R = \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q(s_t, a; \theta^-) & \text{for non-terminal } s_t \end{cases} \)
    for \( i \in \{t - 1, \ldots, t_{start}\} \) do
        \( r_i \leftarrow r_i + \gamma R \)
        Accumulate gradients wrt \( \theta' \): \( d\theta \leftarrow d\theta + \frac{\partial}{\partial \theta}(R - Q(s_i, a_i; \theta'))^2 \)
    end for
    Perform asynchronous update of \( \theta \) using \( d\theta \).
    if \( T \mod T_{\text{target}} == 0 \) then
        \( \theta^- \leftarrow \theta \)
    end if
until \( T > T_{\text{max}} \)
```

Algorithm S2 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.
Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$

repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    Repeat
        Perform $a_t$ according to policy $\pi(a_t | s_t; \theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} = = t_{max}$
    $R = \begin{cases} 
        0 & \text{for terminal } s_t \\
        V(s_t, \theta'_v) & \text{for non-terminal } s_t\text{/ Bootstrap from last state}
    \end{cases}$

for $i \in \{t - 1, \ldots, t_{start}\}$ do
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i | s_i; \theta') (R - V(s_i, \theta'_v))$
    Accumulate gradients wrt $\theta_v$: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i, \theta'_v))^2 / \partial \theta'_v$
end for

Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.

until $T > T_{max}$

*Algorithm S3 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.*
Asynchronous Methods for Deep RL

All methods performed updates every 5 actions ($t_{\text{max}} = I_{\text{AsyncUpdate}} = 5$), updating the target network each 40,000 frames.

The Atari inputs were processed as in DQN, using also action repeat 4.

The network architecture is: 16 filters $8 \times 8$ stride 4, 32 filters $4 \times 4$ stride 2, followed by a fully connected layer with 256 units. All hidden layers apply a ReLU non-linearity. Values and/or action values were then generated from the (same) last hidden layer.

The LSTM methods utilized a 256-unit LSTM cell after the dense hidden layer.

All experiments used a discount factor of $\gamma = 0.99$ and used RMSProp with momentum decay factor of 0.99.
Asynchronous Methods for Deep RL

Table 1 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

<table>
<thead>
<tr>
<th>Method</th>
<th>Training Time</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td>8 days on GPU</td>
<td>121.9%</td>
<td>47.5%</td>
</tr>
<tr>
<td>Gorila</td>
<td>4 days, 100 machines</td>
<td>215.2%</td>
<td>71.3%</td>
</tr>
<tr>
<td>D-DQN</td>
<td>8 days on GPU</td>
<td>332.9%</td>
<td>110.9%</td>
</tr>
<tr>
<td>Dueling D-DQN</td>
<td>8 days on GPU</td>
<td>343.8%</td>
<td>117.1%</td>
</tr>
<tr>
<td>Prioritized DQN</td>
<td>8 days on GPU</td>
<td>463.6%</td>
<td>127.6%</td>
</tr>
<tr>
<td>A3C, FF</td>
<td>1 day on CPU</td>
<td>344.1%</td>
<td>68.2%</td>
</tr>
<tr>
<td>A3C, LSTM</td>
<td>4 days on CPU</td>
<td>496.8%</td>
<td>116.6%</td>
</tr>
<tr>
<td>A3C, FF</td>
<td>4 days on CPU</td>
<td>623.0%</td>
<td>112.6%</td>
</tr>
</tbody>
</table>

Table 2 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-step Q</td>
<td>1.0</td>
</tr>
<tr>
<td>1-step SARSA</td>
<td>1.0</td>
</tr>
<tr>
<td>n-step Q</td>
<td>1.0</td>
</tr>
<tr>
<td>A3C</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Figure 1 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.
Asynchronous Methods for Deep RL

Figure 3 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.
Asynchronous Methods for Deep RL

Policy Gradient Methods

- REINFORCE
- Baseline
- Actor-Critic
- A3C
- PAAC

Continuous Action Space
Asynchronous Methods for Deep RL

Figure 2 of the paper "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.
Parallel Advantage Actor Critic

An alternative to independent workers is to train in a synchronous and centralized way by having the workers to only generate episodes. Such approach was described in May 2017 by Clemente et al., who named their agent *parallel advantage actor-critic* (PAAC).

---

*Figure 1 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.*
Algorithm 1 Parallel advantage actor-critic

1: Initialize timestep counter $N = 0$ and network weights $\theta, \theta_v$
2: Instantiate set $e$ of $n_e$ environments
3: repeat
4:   for $t = 1$ to $t_{\text{max}}$ do
5:     Sample $\alpha_t$ from $\pi(\alpha_t \mid s_t; \theta)$
6:     Calculate $v_t$ from $V(s_t; \theta_v)$
7:     parallel for $i = 1$ to $n_e$ do
8:       Perform action $a_{t,i}$ in environment $e_i$
9:       Observe new state $s_{t+1,i}$ and reward $r_{t+1,i}$
10:    end parallel for
11:   end for
12: $R_{t_{\text{max}}+1} = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_{t_{\text{max}}+1}; \theta) & \text{for non-terminal } s_t \end{cases}$
13: for $t = t_{\text{max}}$ down to 1 do
14:   $R_t = r_t + \gamma R_{t+1}$
15: end for
16: $d\theta = \frac{1}{n_e \cdot t_{\text{max}}} \sum_{i=1}^{n_e} \sum_{t=1}^{t_{\text{max}}} (R_{t,i} - v_{t,i}) \nabla_\theta \log \pi(\alpha_{t,i} \mid s_{t,i}; \theta) + \beta \nabla_\theta H(\pi(s_{e,t}; \theta))$
17: $d\theta_v = \frac{1}{n_e \cdot t_{\text{max}}} \sum_{i=1}^{n_e} \sum_{t=1}^{t_{\text{max}}} \nabla_{\theta_v} (R_{t,i} - V(s_{t,i}; \theta_v))^2$
18: Update $\theta$ using $d\theta$ and $\theta_v$ using $d\theta_v$
19: $N \leftarrow N + n_e \cdot t_{\text{max}}$
20: until $N \geq N_{\text{max}}$

Algorithm 1 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.
The authors use 8 workers, $n_e = 32$ parallel environments, 5-step returns, $\gamma = 0.99$, $\varepsilon = 0.1$, $\beta = 0.01$ and a learning rate of $\alpha = 0.0007 \cdot n_e = 0.0224$.

The $\text{arch}_\text{nips}$ is from A3C: 16 filters $8 \times 8$ stride 4, 32 filters $4 \times 4$ stride 2, a dense layer with 256 units. The $\text{arch}_\text{nature}$ is from DQN: 32 filters $8 \times 8$ stride 4, 64 filters $4 \times 4$ stride 2, 64 filters $3 \times 3$ stride 1 and 512-unit fully connected layer. All nonlinearities are ReLU.
Parallel Advantage Actor Critic

Figure 3 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.

- Beam Rider
- Pong
- QBert
- Boxing
- Breakout

NPFL122, Lecture 7
Parallel Advantage Actor Critic

Figure 4 of the paper “Efficient Parallel Methods for Deep Reinforcement Learning” by Alfredo V. Clemente et al.

NPFL122, Lecture 7 Policy Gradient Methods REINFORCE Baseline Actor-Critic A3C PAAC Continuous Action Space
Parallel Advantage Actor Critic

Figure 2 of the paper "Efficient Parallel Methods for Deep Reinforcement Learning" by Alfredo V. Clemente et al.
Continuous Action Space

Until now, the actions were discrete. However, many environments naturally accept actions from continuous space. We now consider actions which come from range \([a, b]\) for \(a, b \in \mathbb{R}\), or more generally from a Cartesian product of several such ranges:

\[
\prod_{i} [a_i, b_i].
\]

A simple way how to parametrize the action distribution is to choose them from the normal distribution. Given mean \(\mu\) and variance \(\sigma^2\), probability density function of \(\mathcal{N}(\mu, \sigma^2)\) is

\[
p(x) \equiv \frac{1}{\sqrt{2\pi \sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.
\]
Utilizing continuous action spaces in gradient-based methods is straightforward. Instead of the softmax distribution we suitably parametrize the action value, usually using the normal distribution. Considering only one real-valued action, we therefore have

\[ \pi(a|s; \theta) \overset{\text{def}}{=} P\left(a \sim \mathcal{N}(\mu(s; \theta), \sigma(s; \theta)^2)\right), \]

where \( \mu(s; \theta) \) and \( \sigma(s; \theta) \) are function approximation of mean and standard deviation of the action distribution.

The mean and standard deviation are usually computed from the shared representation, with

- the mean being computed as a regular regression (i.e., one output neuron without activation);
- the standard variance (which must be positive) being computed again as a regression, followed most commonly by either \( \exp \) or \( \text{softplus} \), where \( \text{softplus}(x) \overset{\text{def}}{=} \log(1 + e^x) \).
Continuous Action Space in Gradient Methods

During training, we compute $\mu(s; \theta)$ and $\sigma(s; \theta)$ and then sample the action value (clipping it to $[a, b]$ if required). To compute the loss, we utilize the probability density function of the normal distribution (and usually also add the entropy penalty).

```python
mu = tf.layers.dense(hidden_layer, 1)[:, 0]
sd = tf.layers.dense(hidden_layer, 1)[:, 0]
sd = tf.exp(log_sd)  # or sd = tf.nn.softplus(sd)
normal_dist = tf.distributions.Normal(mu, sd)

# Loss computed as - log $\pi(a|s)$ - entropy_regularization
loss = - normal_dist.log_prob(self.actions) * self.returns \
- args.entropy_regularization * normal_dist.entropy()
```
Continuous Action Space

When the action consists of several real values, i.e., action is a suitable subregion of $\mathbb{R}^n$ for $n > 1$, we can:

- either use multivariate Gaussian distribution;
- or factorize the probability into a product of univariate normal distributions.

Modeling the action distribution using a single normal distribution might be insufficient, in which case a mixture of normal distributions is usually used.

Sometimes, the continuous action space is used even for discrete output -- when modeling pixels intensities (256 values) or sound amplitude ($2^{16}$ values), instead of a softmax we use discretized mixture of distributions, usually logistic (a distribution with a sigmoid cdf). Then,

$$\pi(a) = \sum_i p_i \left( \sigma \left( (a + 0.5 - \mu_i) / \sigma_i \right) - \sigma \left( (a - 0.5 - \mu_i) / \sigma_i \right) \right).$$

However, such mixtures are usually used in generative modeling, not in reinforcement learning.