Noisy Network

Paper Link: Noisy Networks for Exploration

Key Features

- NoisyNet adds parametric noise to network of deep RL algorithm, so agent can explore efficiently using the induced stochasticity of the agent’s policy.
- Since the parameters of the noise are learned with gradient descent along with the remaining network weights without any hyperparameter tuning
- NoisyNet is straightforward to implement and adds little computational overhead.
- Replacing the conventional exploration heuristics of the existing algorithm with NoisyNet usually results in better performance.

Background

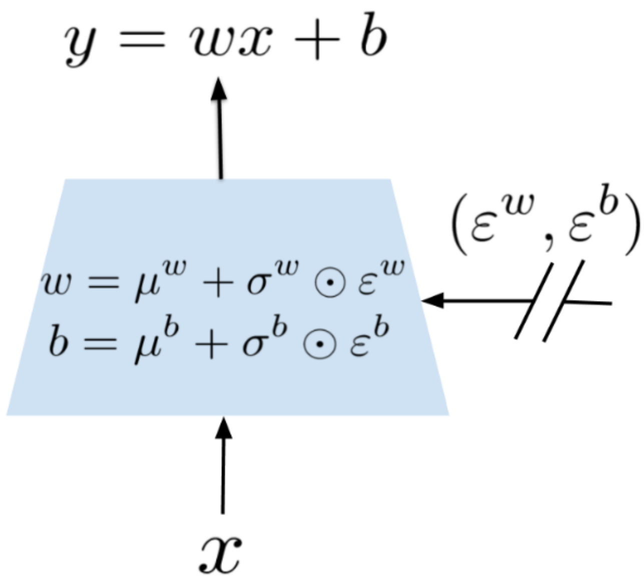
Most exploration heuristics in general reinforcement learning algorithms rely on random perturbations of the agent's policy, such as ε-greedy(Sutton & Barto, 1998) or entropy regularization(Williams, 1992), to induce novel behaviors. However, these methods often limit exploration of the agent to small state-action spaces or explored state-independently.

To solve this problem, the paper propose a simple alternative approach called NoisyNet, where learned perturbations of the network weights are used to drive exploration.

Method

NoisyNets are neural networks that weights and biases are perturbed by a parametric noise. These parameters are trained with gradient descent.

The paper represent the noisy parameters θ as $\theta \stackrel{\text{def}}{=} \mu + \sigma \odot \varepsilon$, where $\zeta \stackrel{\text{def}}{=} (\mu, \sigma)$ is a set of learnable parameters, ε is a vector of zero-mean noise with fixed statistics and $\odot$ represents element-wise multiplication.



Consider a linear layer of a neural network with p inputs and q outputs which can be represented as follows

$$y = wx + b, \tag{1}$$

where $x \in \mathbb{R}^p$ is the layer input, $w \in \mathbb{R}^{q \times p}$ is the weight matrix, and $b \in \mathbb{R}^q$ is the bias.

The corresponding noisy linear layer is defined as follows:

$$y \stackrel{\text{def}}{=} (\mu^w + \sigma^w \odot \varepsilon^w)x + \mu^b + \sigma^b \odot \varepsilon^b,$$

where $\mu^w + \sigma^w \odot \varepsilon^w$ and $\mu^b + \sigma^b \odot \varepsilon^b$ replace w and b in Equation(1), respectively. The parameters $\mu^w \in \mathbb{R}^{q \times p}, \mu^b \in \mathbb{R}^q, \sigma^w \in \mathbb{R}^{q \times p}$ and $\sigma^b \in \mathbb{R}^q$ are learnable whereas $\varepsilon^w \in \mathbb{R}^{q \times p}$ and $\varepsilon^b \in \mathbb{R}^q$ are noise random variables.

The paper describes two options for the parametric noise:

- **Independent Gaussian noise**: the noise applied to each weight and bias is independent, where each entry $\varepsilon_{i,j}^{w}$ of the random matrix ε^{w} is drawn from a unit Gaussian distribution. This means that for each noisy linear layer, there are $pq+q$ noise variables.
- **Factorized Gaussian noise:** Factorize $\varepsilon_{i,j}^{w}$ to use p unit Gaussian variable ε_i for noise of the inputs and q unit Gaussian variable ε_j for noise of the outputs (thus $p+q$ unit Gaussian variables in total). Each $\varepsilon_{i,j}^{w}$ and ε_j^{b} can then be written as:
  $\varepsilon_{i,j}^{w}=f(\varepsilon_i)f(\varepsilon_j),$
  $\varepsilon_j^{b}=f(\varepsilon_j),$ where $f(x)=\operatorname{sgn}(x)|x|$.

Since the loss of a noisy network, $\bar{L}(\zeta)=\mathbb{E}[L(\theta)]$, is an expectation over the noise, the gradients are straightforward to obtain:

$$\nabla\bar{L}(\zeta)=\nabla\mathbb{E}[L(\theta)]=\mathbb{E}[\nabla_{\mu,\sigma}L(\mu+\sigma\odot\varepsilon)].$$

The paper use a Monte Carlo approximation to the above gradients, taking a single sample ξ at each step of optimization:

$$\nabla\bar{L}(\zeta)\approx\nabla_{\mu,\sigma}L(\mu+\sigma\odot\varepsilon).$$

Algorithm

Algorithm 1: NoisyNet-DQN / NoisyNet-Dueling

Input : Env Environment; ε set of random variables of the network
Input : DUELING Boolean; "true" for NoisyNet-Dueling and "false" for NoisyNet-DQN
Input : B empty replay buffer; ζ initial network parameters; ζ^- initial target network parameters
Input : N_B replay buffer size; N_T training batch size; N^- target network replacement frequency
Output : $Q(\cdot, \varepsilon; \zeta)$ action-value function

for *episode* $e \in \{1, \dots, M\}$ **do**
  Initialise state sequence $x_0 \sim Env$
  for $t \in \{1, \dots\}$ **do**
    `/*` $l[-1]$ `is the last element of the list` l `*/`
    Set $x \leftarrow x_0$
    Sample a noisy network $\xi \sim \varepsilon$
    Select an action $a \leftarrow \mathrm{argmax}_{b \in A}\, Q(x, b, \xi; \zeta)$
    Sample next state $y \sim P(\cdot|x, a)$, receive reward $r \leftarrow R(x, a)$ and set $x_0 \leftarrow y$
    Add transition (x, a, r, y) to the replay buffer $B[-1] \leftarrow (x, a, r, y)$
    if $|B| > N_B$ **then**
      Delete oldest transition from B
    end
    `/*` D `is a distribution over the replay, it can be uniform or implementing prioritised replay */`
    Sample a minibatch of N_T transitions $((x_j, a_j, r_j, y_j) \sim D)_{j=1}^{N_T}$
    `/* Construction of the target values. */`
    Sample the noisy variable for the online network $\xi \sim \varepsilon$
    Sample the noisy variables for the target network $\xi' \sim \varepsilon$
    if *DUELING* **then**
      Sample the noisy variables for the action selection network $\xi'' \sim \varepsilon$
    for $j \in \{1, \dots, N_T\}$ **do**
      if y_j *is a terminal state* **then**
        $\widehat{Q} \leftarrow r_j$
      if *DUELING* **then**
        $b^*(y_j) = \arg\max_{b \in \mathcal{A}} Q(y_j, b, \xi''; \zeta)$
        $\widehat{Q} \leftarrow r_j + \gamma Q(y_j, b^*(y_j), \xi'; \zeta^-)$
      else
        $\widehat{Q} \leftarrow r_j + \gamma \max_{b \in A} Q(y_j, b, \xi'; \zeta^-)$
      Do a gradient step with loss $(\widehat{Q} - Q(x_j, a_j, \xi; \zeta))^2$
    end
    if $t \equiv 0 \pmod{N^-}$ **then**
      Update the target network: $\zeta^- \leftarrow \zeta$
    end
  end
end

Implementation on JORLDY

- [Noisy Network JORLDY Implementation]()

```python
# It can be created in two types: independent and factorized.

class Noisy(BaseNetwork):
    def __init__(self, D_in, D_out, noise_type="factorized", D_hidden=512, head="mlp"):
        assert noise_type in ["independent", "factorized"]

        D_head_out = super(Noisy, self).__init__(D_in, D_hidden, head)
        self.noise_type = noise_type

        self.mu_w1, self.sig_w1, self.mu_b1, self.sig_b1 = init_weights(
            (D_head_out, D_hidden), noise_type
        )
        self.mu_w2, self.sig_w2, self.mu_b2, self.sig_b2 = init_weights(
            (D_hidden, D_out), noise_type
        )

    def forward(self, x, is_train):
        x = super(Noisy, self).forward(x)
        x = F.relu(
            noisy_l(
                x,
                self.mu_w1,
                self.sig_w1,
                self.mu_b1,
                self.sig_b1,
                self.noise_type,
                is_train,
            )
        )
        x = noisy_l(
            x,
            self.mu_w2,
            self.sig_w2,
            self.mu_b2,
            self.sig_b2,
            self.noise_type,
            is_train,
        )
        return x

    def get_sig_w_mean(self):
        sig_w_abs_mean1 = torch.abs(self.sig_w1).mean()
        sig_w_abs_mean2 = torch.abs(self.sig_w2).mean()

        return sig_w_abs_mean1, sig_w_abs_mean2
```

- [NoisyNet utils JORLDY Implementation]

```
# If is_train=False, only weight μ are used.

def noisy_l(x, mu_w, sig_w, mu_b, sig_b, noise_type, is_train):
    if noise_type == "factorized":
        # Factorized Gaussian Noise
        if is_train:
            eps_i = torch.randn(mu_w.size()[0]).to(x.device)
            eps_j = torch.randn(mu_b.size()[0]).to(x.device)

            f_eps_i = torch.sign(eps_i) * torch.sqrt(torch.abs(eps_i))
            f_eps_j = torch.sign(eps_j) * torch.sqrt(torch.abs(eps_j))

            eps_w = torch.matmul(
                torch.unsqueeze(f_eps_i, 1), torch.unsqueeze(f_eps_j, 0)
            )
            eps_b = f_eps_j
        else:
            eps_w = torch.zeros(mu_w.size()[0], mu_b.size()[0]).to(x.device)
            eps_b = torch.zeros(1, mu_b.size()[0]).to(x.device)
    else:
        # Independent Gaussian Noise
        if is_train:
            eps_w = torch.randn(mu_w.size()).to(x.device)
            eps_b = torch.randn(mu_b.size()).to(x.device)
        else:
            eps_w = torch.zeros(mu_w.size()).to(x.device)
            eps_b = torch.zeros(mu_b.size()).to(x.device)

    weight = mu_w + sig_w * eps_w
    bias = mu_b + sig_b * eps_b

    y = torch.matmul(x, weight) + bias

    return y


def init_weights(shape, noise_type):
    if noise_type == "factorized":
        mu_init = 1.0 / (shape[0] ** 0.5)
        sig_init = 0.5 / (shape[0] ** 0.5)
    else:
        mu_init = (3.0 / shape[0]) ** 0.5
        sig_init = 0.017

    mu_w = torch.nn.Parameter(torch.empty(shape))
    sig_w = torch.nn.Parameter(torch.empty(shape))
    mu_b = torch.nn.Parameter(torch.empty(shape[1]))
    sig_b = torch.nn.Parameter(torch.empty(shape[1]))

    mu_w.data.uniform_(-mu_init, mu_init)
```

```
    mu_b.data.uniform_(-mu_init, mu_init)
    sig_w.data.uniform_(sig_init, sig_init)
    sig_b.data.uniform_(sig_init, sig_init)

    return mu_w, sig_w, mu_b, sig_b
```

- In JORLDY, it is implemented by applying NoisyNet to the DQN algorithm.
- NoisyNet-DQN JORLDY Implementation

```
## Noisy-DQN Agent act function ##

def act(self, state, training=True):
    self.network.train(training)

    if training and self.memory.size < max(self.batch_size, self.start_train_step):
        action = np.random.randint(0, self.action_size, size=(state.shape[0], 1))
    else:
        action = (
            torch.argmax(
                self.network(self.as_tensor(state), training), -1, keepdim=True
            )
            .cpu()
            .numpy()
        )
    return {"action": action}
```

References

Relevant papers

- Reinforcement Learning: An Introduction
- Simple statistical gradient-following algorithms for connectionist reinforcement learning