Training Mario Experiments Journal

Playing Atari with Deep Reinforcement Learning

Key contributions: Introduced deep Q-networks (DQN) that combine reinforcement learning with deep neural networks, enabling agents to learn directly from high-dimensional sensory input and achieve human-level performance on Atari games using **experience replay** and **target networks**.

In smaller environments one can compute the Q-function ($Q(s, a)$) by tabulating experiences. But for environments with a large number of states to become computationally tractable, like Atari games, this paper successfully leverages neural networks to approximate the Q-function.

One issue that arises when bootstrapping the Q-function with itself, **and** using it for choosing actions is that the agent ends up "going down the rabbit hole" and usually results in divergent or sample inefficient training. For this reason we perform Q-value updates using:

$Q(s, a) \leftarrow r + \gamma * max_{a'}(Q_{target}(s', a'))$

Where $Q_{target} \leftarrow Q$ every K steps.

Additional tricks/notes

Training instability: I've taken two additional steps to stabilize training: * L1+MSE loss function * Gradient clipping

Previous action as input: The agent quickly learns that run and jump buttons together provide a lot of reward, but Mario only jumps if you have not already pressed the jump button. This, in turn, causes Mario to continuously run into the mushroom enemies. To distinguish between states where pressing jump will result in a jump versus continuing because the button was already pressed, I have added the previous action as input. It does improve this behavior slightly.

Softmax action selection: Choosing the highest Q-value action at every step results in Mario getting stuck in parts where the agent is pressing against an object. To perform more randomized actions, I've implemented softmax action selection, which greatly improves agent performance by encouraging more exploration (i.e., it is intermediate between using ϵ-greedy and the max operator for selecting actions).

Comparison between a few agents

Agent	Action Selection Method	Epsilon Schedule	Softmax Temperature	Characteristics
Random Agent	ϵ-greedy	$\epsilon = 1$	N/A	Purely random actions
Max Agent	ϵ-greedy	$1 \rightarrow 0.02$	N/A	Greedy with decaying exploration
Softmax Agent 0.1	ϵ-softmax	$1 \rightarrow 0.02$	0.1	More deterministic, favors highest Q-values
Softmax Agent 0.2	ϵ-softmax	$1 \rightarrow 0.02$	0.2	More exploration, less deterministic than 0.1

Prioritized Experience Replay

Key contribution: This paper introduces a method to prioritize replayed experiences by sampling more important transitions more frequently, based on the magnitude of their temporal-difference (TD) error. This improves learning efficiency and accelerates convergence by focusing updates on the most impactful experiences.

The probability of picking experience i changes from,

$$P(i) = \frac{1}{N}$$

to,

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where p_i is the priority of transition i and α determines how much prioritization is used ($\alpha = 0$ corresponds to uniform sampling). This priority can either be rank-based or simply the TD-error.

Additional tricks/notes

Weighted importance sampling: Since we are changing the i.i.d. nature of uniform sampling of experiences, we will introduce a bias that will need to be corrected. See details in Weighted importance sampling for off-policy learning with linear function approximation. This involves multiplying the gradient updates by:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

where β controls the strength of the importance-sampling correction. Additionally the authors recommend normalizing the IS among the batch, or entire buffer. While it is not clear, I've normalized by the maximum weight over the entire buffer and that worked quite well on a small sample.

Dueling Network Architectures for Deep Reinforcement Learning

Key contribution: This paper proposes to decompose our Q-value function into state value and advantage value function networks, sharing a common torso. This allows for more efficient and generalizable learning across the two function networks. During evaluation we only need to evaluate one sub-network (the advantage), and can expend the value estimation altogether.

So, we rewrite Q-value function as:

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

Where the best action a^* for a given state s is:

$$a^* = \text{argmax}_{a' \in \mathcal{A}}\{Q(s,a')\}$$

$$= \text{argmax}_{a' \in \mathcal{A}}\{V(s) + A(s,a')\}$$

$$= \text{argmax}_{a' \in \mathcal{A}}\{A(s,a')\}$$

The authors note that this particular formulation suffers from unidentifiability (i.e. adding/substracting a constant from V and A respectively) and poor practical performance.

The authors propose to constrain the Q-function to have 0 advantage on the optimal action, leading to the following formulation of our network's output:

$$Q(s,a) = V(s) + A(s,a) - \max_{a' \in \mathcal{A}} A(s,a')$$

Where, for

$$a^* = \text{argmax}_{a' \in \mathcal{A}}\{A(s,a')\}$$

We find

$$Q(s,a^*) = V(s)$$

However the authors note that the max operator suffers from instability during training and suggest the following simplified (and still identifiable) formulation:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in A} A(s,a')$$

Code snippets

```
class DuelingDQN(nn.Module):
  def forward_cnn(self, x):
    # ...

  def forward_dense(self, x)
    value_x, advantage_x = x, x

    for layer in self.advantage_hidden_layers[:-1]:
      advantage_x = self.activation(layer(advantage_x))
    advantage_x = self.advantage_hidden_layers[-1](advantage_x)

    for layer in self.value_hidden_layers[:-1]:
      value_x = self.activation(layer(value_x))
    value_x = self.value_hidden_layers[-1](value_x)

    return value_x + advantage_x - advantage_x.mean(dim=1, keepdim=True)
```

Noisy Networks for Exploration

This paper proposes replacing exploration strategies like ϵ-greddy and entropy reward (ICM) with injecting parametric noise into the network's weights, in particular the linear layers. This noise is learned alongside the weigths of the network.

This turns linear layers from:

$$y = x * w + b$$

Into

$$y = x * (w + \sigma_w * \epsilon) + (b + \sigma_b * \epsilon')$$

In my cartpole experiments noise on the bias term resulted in worse overall learning, so removing it improved convergence speed to an optimal policy, but that might be problem-specific [episode steps screenshot].

Code snippets

```
class NoisyLinear(nn.Module):
  def forward(self, input):
    weight = self.weight_mu + self.weight_sigma * self.weight_epsilon.normal_()
    bias = self.bias + self.bias_sigma * self.bias_epsilon.normal_()
    return nn.functional.linear(input, weight, bias)
```

TODO Reading

- Intrinsic curiosity module – half-implemented by now