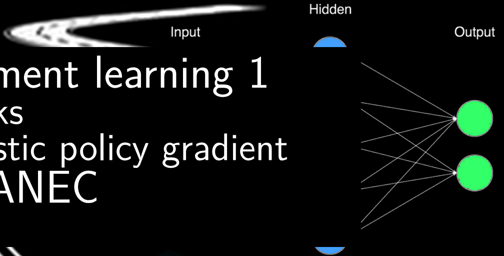
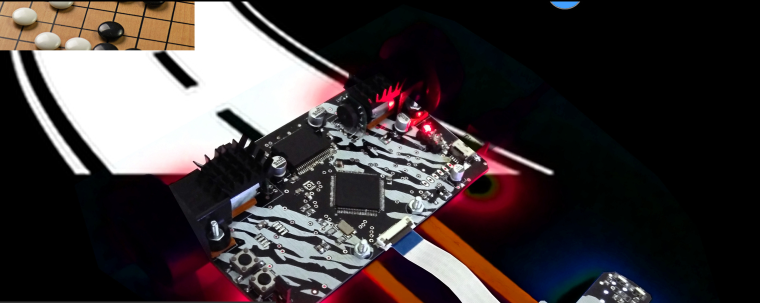
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \lambda \max_{a} Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

(The New Action Value = The Old Value) + The Learning Rate × (The New Information − the Old Information)

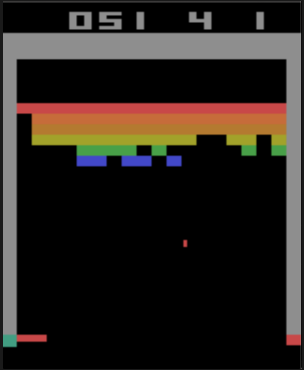


deep reinforcement learning 1
- deep Q networks
- deep deterministic policy gradient

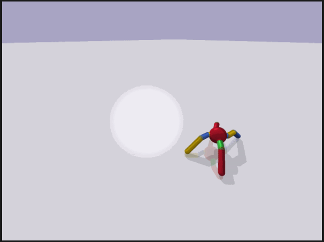
Michal CHOVANEC

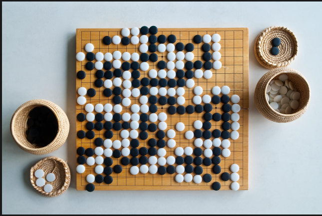


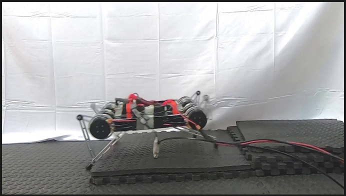
reinforcement learning



reinforcement learning

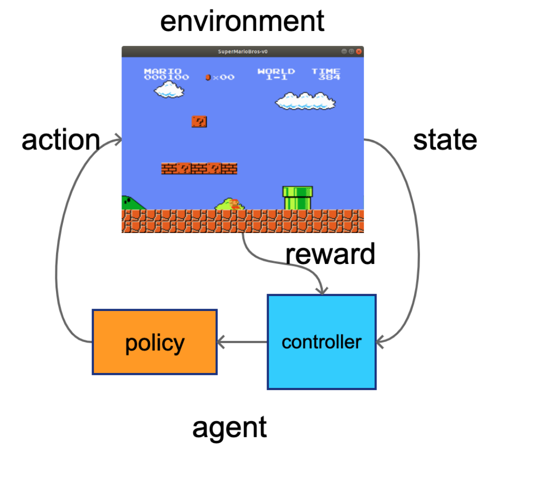




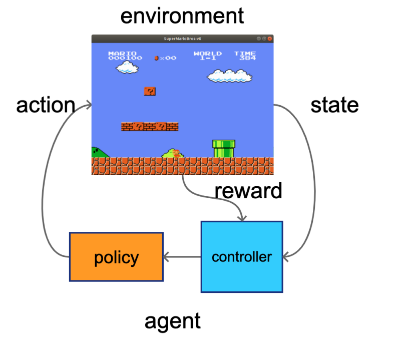


reinforcement learning

learning from punishments and rewards



reinforcement learning



- obtain state
- select action
- exectute action
- learn from experiences

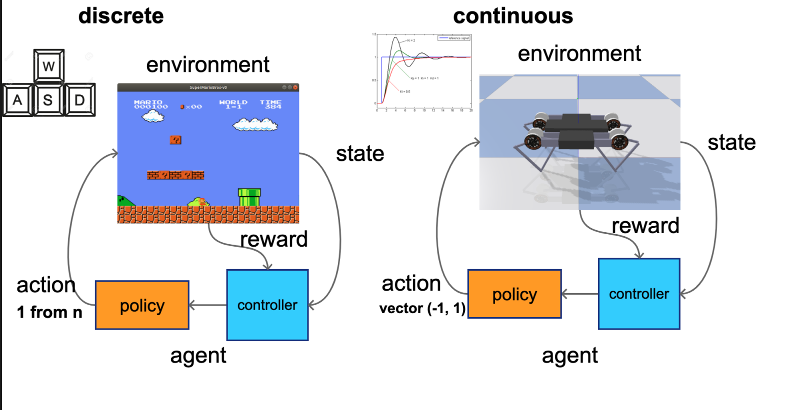
reinforcement learning - algorithms

- discrete actions space
  - Deep Q-network, DQN
  - Dueling DQN
  - Reinbow DQN
- continuous actions space
  - Actor Critic
  - Advantage Actor Critic
  - Proximal policy optimization
  - Soft Actor critic
  - Deep deterministc policy gradient
  - D4PG, SDDPG
- model based
  - curiosity
  - world models
  - imagination agents

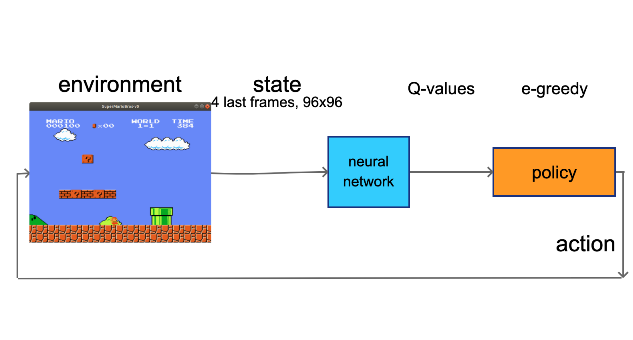
f.e. SDDPG - sampled DDPG, based on Wasserstein loss : Optimal transport, Cédric Villani, 600+ pages

action space

- discrete action space
  - keys, keypad
- continuous action space
  - motors, PWMs, steering, force controll

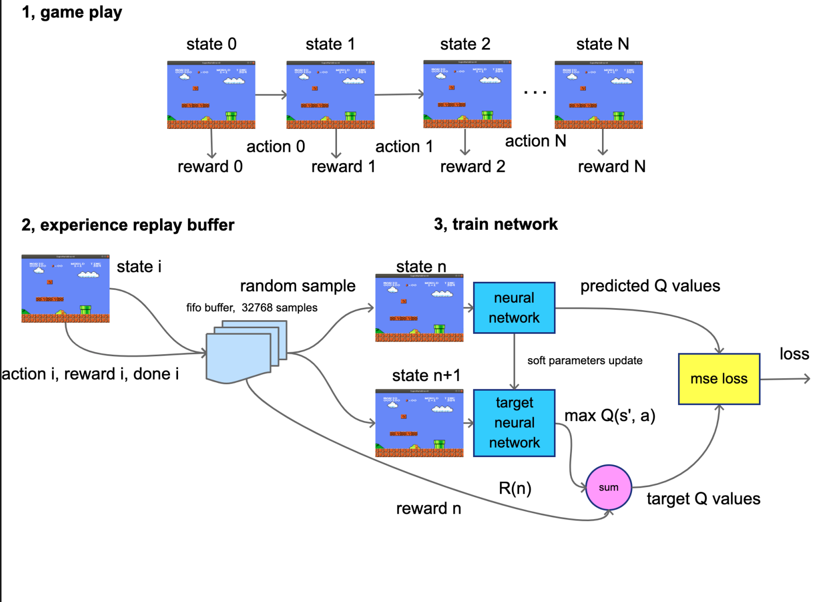


deep Q learning

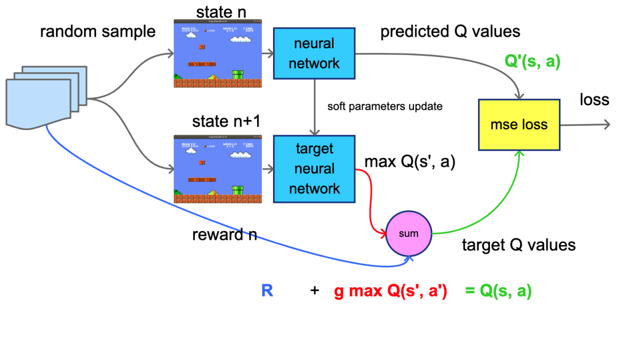


1. play games
2. store transitions into buffer
   - state, action, reward, done
3. learn from buffer

deep Q learning



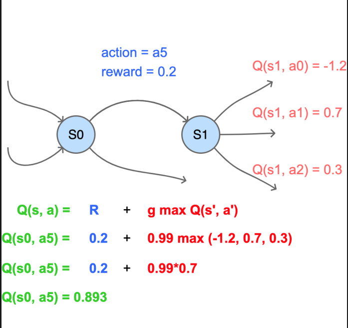
deep Q learning



$$Q(s,a;\theta) = \underbrace{R}_{reward} + \underbrace{\gamma \max_{a'} Q(s',a';\theta^-)}_{discounted\ \ future\ \ reward}$$

$$\mathcal{L}(\theta) = \left(R + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta)\right)^2$$

deep Q learning



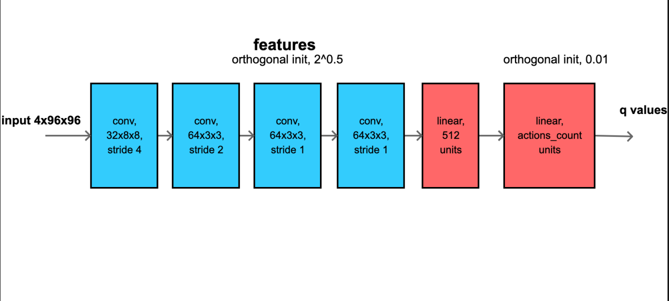
$$Q(s, a; \theta) = \underbrace{R}_{\textit{reward}} + \underbrace{\gamma \max_{a'} Q(s', a'; \theta^-)}_{\textit{discounted future reward}}$$

$$\mathcal{L}(\theta) = \left(R + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2$$

dqn code

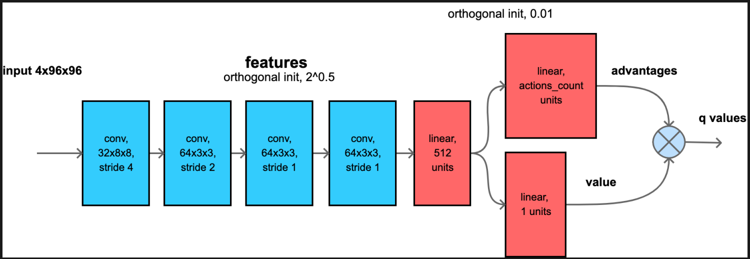
```python
def train_model(self):
    state_t, action_t, reward_t, state_next_t, done_t = self.experience_replay.sample(self.batch_size,

    #q values, state now, state next
    q_predicted      = self.model.forward(state_t)
    q_predicted_next = self.model_target.forward(state_next_t)

    #compute target, n-step Q-learning
    q_target         = q_predicted.clone()
    for j in range(self.batch_size):
        action_idx                = action_t[j]
        q_target[j][action_idx]   = reward_t[j] + torch.max(q_predicted_next[j]*(1 - done_t[j]))


    #train DQN model
    loss = ((q_target.detach() - q_predicted)**2)
    loss  = loss.mean()

    self.optimizer.zero_grad()
    loss.backward()
    for param in self.model.parameters():
        param.grad.data.clamp_(-10.0, 10.0)
    self.optimizer.step()
```

model architecture



- input 96x96 grayscale, 4 stacked frames
- 8x8 and 3x3 convs, with strides
- two fully connected layers
- small learning rate $\eta = 0.0001$, batch size $= 32$
- $\gamma = 0.99$
- exploration ϵ-greedy, 1M samples linear decay from 1 to 0.05
- total training 8..16M samples

dueling DQN, model architecture



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

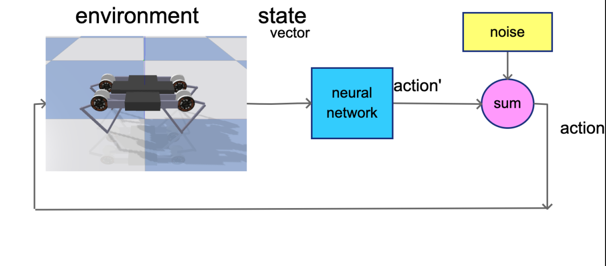
$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a' \in \mathcal{A}} A(s,a')$$

```
WRONG   : q = value + advantage - advantage.mean()
CORRECT : q = value + advantage - advantage.mean(dim=1, keepdim=True)
```

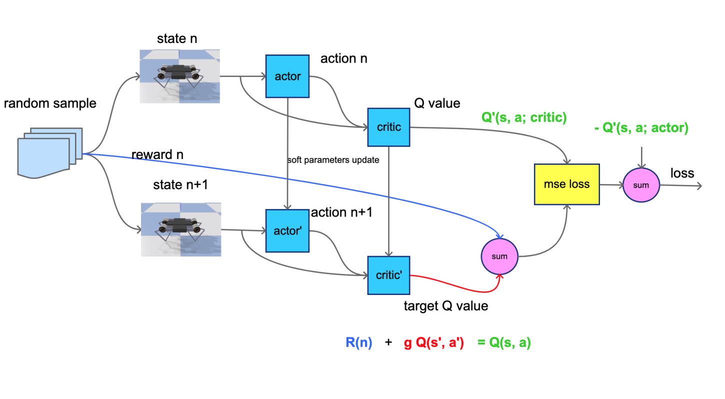
DDPG

deep deterministic policy gradietnt

- continuous action space
- natural extension of DQN
- actor-critic structure

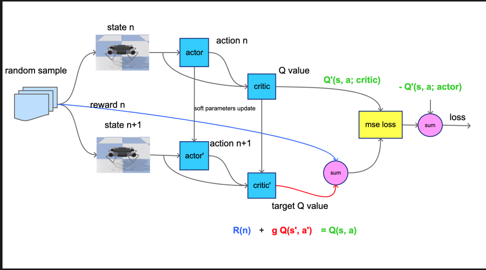


DDPG



state n
actor
action n
random sample
critic
Q value
Q'(s, a; critic)
- Q'(s, a; actor)
reward n
soft parameters update
mse loss
sum
loss
state n+1
actor'
action n+1
sum
critic'
target Q value
R(n) + g Q(s', a') = Q(s, a)

DDPG



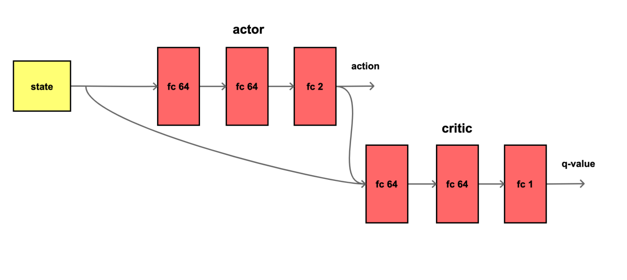
$$\mathcal{L}(\theta) = \left(R + \gamma Q(s', A(s'; \phi^-); \theta^-) - Q(s, A(s; \phi); \theta)\right)^2$$

$$\mathcal{L}(\phi) = -Q(s, A(s; \phi); \theta)$$

where

- Q is critic network with parameters θ
- A is actor network with parameters ϕ

model architecture



ddpg actor code

```python
class Model(torch.nn.Module):
    def __init__(self, input_shape, outputs_count, hidden_count = 256):
        super(Model, self).__init__()
        self.device = "cpu"

        self.layers = [
                            nn.Linear(input_shape[0], hidden_count),
                            nn.ReLU(),
                            nn.Linear(hidden_count, hidden_count//2),
                            nn.ReLU(),
                            nn.Linear(hidden_count//2, outputs_count),
                            nn.Tanh()
        ]

        torch.nn.init.xavier_uniform_(self.layers[0].weight)
        torch.nn.init.xavier_uniform_(self.layers[2].weight)
        torch.nn.init.uniform_(self.layers[4].weight, -0.3, 0.3)

        self.model = nn.Sequential(*self.layers)
        self.model.to(self.device)

        print(self.model)

    def forward(self, state):
        return self.model(state)
```

ddpg critic code

```python
class Model(torch.nn.Module):
    def __init__(self, input_shape, outputs_count, hidden_count = 256):
        super(Model, self).__init__()

        self.device = "cpu"

        self.layers = [
                        nn.Linear(input_shape[0] + outputs_count, hidden_count),
                        nn.ReLU(),
                        nn.Linear(hidden_count, hidden_count//2),
                        nn.ReLU(),
                        nn.Linear(hidden_count//2, 1)
        ]

        torch.nn.init.xavier_uniform_(self.layers[0].weight)
        torch.nn.init.xavier_uniform_(self.layers[2].weight)
        torch.nn.init.uniform_(self.layers[4].weight, -0.003, 0.003)

        self.model = nn.Sequential(*self.layers)
        self.model.to(self.device)

        print(self.model)


    def forward(self, state, action):
        x = torch.cat([state, action], dim = 1)
        return self.model(x)
```

ddpg code

```python
def train_model(self):
    state_t, action_t, reward_t, state_next_t, done_t = self.experience_replay.sample(self.batch_size, self.model_critic.device)

    reward_t = reward_t.unsqueeze(-1)
    done_t   = (1.0 - done_t).unsqueeze(-1)

    action_next_t   = self.model_actor_target.forward(state_next_t).detach()
    value_next_t    = self.model_critic_target.forward(state_next_t, action_next_t).detach()

    #critic loss
    value_target    = reward_t + self.gamma*done_t*value_next_t
    value_predicted = self.model_critic.forward(state_t, action_t)

    critic_loss     = ((value_target - value_predicted)**2)
    critic_loss     = critic_loss.mean()

    #update critic
    self.optimizer_critic.zero_grad()
    critic_loss.backward()
    self.optimizer_critic.step()

    #actor loss
    actor_loss      = -self.model_critic.forward(state_t, self.model_actor.forward(state_t))
    actor_loss      = actor_loss.mean()

    #update actor
    self.optimizer_actor.zero_grad()
    actor_loss.backward()
    self.optimizer_actor.step()
```

wise Wizard's DDPG spell chart

- **neurons count** on 1st layer = 10x state vector size
- **neurons count** on 2nd layer = 0.5x neurons on 1st layer
- **weight init** for hidden layers : use Xavier
- **weight init** actor output : use uniform $\langle -0.3, 0.3 \rangle$
- **weight init** critic output : use uniform $\langle -0.003, 0.003 \rangle$
- **gaussian noise** : linear decay variance, from 0.5 to 0.1, for 1M steps, or noisy layers
- use **soft** target network update, $\tau = 0.001$
- actor learning rate $\eta_a = 0.0001$
- critic learning rate $\eta_c = 0.0002$

wise Wizard's magic staff

- fully connected nets (robotic envs) **train on CPU** - AMD Ryzen
- convolutional nets (visual inputs envs) **train on GPU**
- use fast CPU - envs are slow
- 32GB of RAM is enough
- for small visual envs (Atari, DOOM, Nec) - GTX1060, GTX1080ti, RTX2080 ...

books to read

- Maxim Lapan, 2020, Deep Reinforcement Learning Hands-On second edition
- Maxim Lapan, 2018, Deep Reinforcement Learning Hands-On
- Praveen Palanisamy, 2018, Hands-On Intelligent Agents with OpenAI Gym
- Andrea Lonza, 2019, Reinforcement Learning Algorithms with Python
- Rajalingappaa Shanmugamani, 2019, Python Reinforcement Learning
- Micheal Lanham, 2019, Hands-On Deep Learning for Games

Q&A