Approximate solutions methods

Nicolas Wicker

Laboratoire Paul Painlevé, Université Lille, 59655, Villeneuve d'Ascq, France.

The basics (1/2)

When the number of states is large

- tabular methods like Q-learning are not efficient anymore
- replace the policy evaluation by an approximation: $\hat{\nu}(s, w) \approx \nu_\pi(s)$ where w is a parameter to be found (like weights in a neural network)
- minimize the value error: $VE(w) = \sum_{s \in \mathcal{S}} \mu(s) \left[\nu_\pi(s) - \hat{\nu}(s, w)\right]^2$ where $\mu(s)$ is a distribution over the states

The basics (2/2)

What next ?

- ν_π is unknown so an approximation is needed
- two main strategies:
  - Monte-Carlo
  - Semi-gradient

Gradient Monte-Carlo for $\hat{\nu} \approx \nu_\pi$ (1/2)

Idea

- replace the $\nu_\pi(s)$ by a stochastic approximation of it
- at time t, $\nu_\pi(S_t)$ is replaced by G_t where $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$

Gradient Monte-Carlo for $\hat{\nu} \approx \nu_\pi$ (2/2)

---

for each episode **do**
  generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  for each step $t \in 0, \ldots, T-1$ of current episode **do**
    $w \leftarrow w + \alpha\Big[G_t - \hat{\nu}(S_t, w)\Big]\nabla\hat{\nu}(S_t, w)$
  end for
end for

Semi-Gradient TD(0) for $\hat{\nu} \approx \nu_\pi$ (1/2)

Idea

- replace the $\nu_\pi(s)$ by its next-step approximation and immediate reward
- at time t, $\nu_\pi(S_t)$ is replaced by $R_{t+1} + \gamma\hat{\nu}(S_{t+1}, w)$

Semi-Gradient TD(0) for $\hat{\nu} \approx \nu_\pi$ (2/2)

initialize weights $w \in \mathbb{R}^d$
for each episode **do**
  S initial state
  for each step of current episode **do**
    choose A from $\pi(.|S)$
    take action A, observe R, S'
    $w \leftarrow w + \alpha\Big[R + \gamma\hat{\nu}(S', w) - \hat{\nu}(S, w)\Big]\nabla\hat{\nu}(S, w)$
    $S \leftarrow S'$
  end for
end for

Episodic Semi-Gradient Sarsa $\hat{q} \approx q_*$ (1/2)

Idea

- extension of semi-gradient TD(0)
- convergence guaranteed as we can make it ϵ-greedy (optimality not guaranteed yet)

Episodic Semi-Gradient Sarsa $\hat{q} \approx q_*$ (2/2)

for each episode **do**
  S state
  for each step $t \in 0, \ldots, T-1$ of current episode **do**
    take action A, observe R, S'
    if S' is terminal **then**
      $w \leftarrow w + \alpha\big[R - \hat{q}(S, A, w)\big]\nabla\hat{q}(S, A, w)$
      go to next episode
    end if
    choose A' as a function of $\hat{q}(S', ., w)$ (e.g. $\epsilon-$greedy)
    $w \leftarrow w + \alpha\big[R + \gamma\hat{q}(S', A', w) - \hat{q}(S, A, w)\big]\nabla\hat{q}(S, A, w)$
    $S \leftarrow S'$
    $A \leftarrow A'$
  end for
end for

ATARI games (1/6)

Motivation

ATARI games (2/6)

Ideas

- use Q learning coupled with a deep neural network
- training along 50 millions frames for each game

Results

- comparison with a professional player
- DQN exceeded the human player on 29 games out of 46 games

ATARI games (3/6)

Neural network

- input: $84 \times 84 \times 4$ corresponding to the 4 most recent frames
- hidden layers: 32 20×20 features maps, then 64 9×9 feature maps and at last 64 7×7 feature maps
- output: 18 units one for each action

ATARI games (4/6)

Training

Rewards are given as follows:

- $+1$ whenever the game score is increased
- -1 whenever the game score is decreased
- 0 otherwise

Semi-gradient form of Q-learning:

$$w_{t+1} = w_t + \alpha \left[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t) \right] \nabla \hat{q}(S_t, A_t, w_t)$$

ATARI games (5/6)

Some twists

- experience replay (Lin, 1992)
- duplicate the neural network
- error term $R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)$ has been clipped to stay in the interval $[-1, 1]$

ATARI games (6/6)

Experience replay (Lin, 1992)

Along the training, the agent accumulate experiences in the form of tuples $(S_t, A_t, R_{t+1}, S_{t+1})$, these tuples are stored in a buffer and sampled from for learning

Network duplication

Every C steps, the network weights are stored into a duplicate network with output $\tilde{q}$. This network is then used as target network:

$$w_{t+1} = w_{t+1} + \alpha\Big[R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)\Big]\nabla\hat{q}(S_t, A_t, w_t)$$

Come back to frozenlake

What is needed ?

- approximate $Q(s, a)$ with a neural network
- gradient retropropagation

Gradient retropropagation (1/2)

Notations

- ϕ is the activation function
- w_{ij} stands for the weight between neuron i and j
- e_i and o_i stand respectively for the input and the output of neuron i
- the loss function is given by: $\text{Loss} = \sum_i (y_i - \hat{y}_i)^2$
- $\delta_i = \frac{\partial \text{Loss}}{\partial e_i}$

Gradient retropropagation (2/2)

In a few equations

$$\frac{\partial \text{Loss}}{\partial w_{ij}} = \frac{\partial \text{Loss}}{\partial e_j} \frac{\partial e_j}{\partial w_{ij}} = \delta_j o_i$$

- for the last layer weights:

$$\frac{\partial \text{Loss}}{\partial w_{ij}} = 2(y_j - \hat{y}_j)\phi'(e_j)o_i$$

- for the other weights:

$$\begin{aligned}\frac{\partial \text{Loss}}{\partial w_{ij}} &= \sum_k \frac{\partial \text{Loss}}{\partial e_k} \frac{\partial e_k}{\partial w_{ij}} \\ &= \sum_k \delta_k w_{jk} \phi'(e_j) o_i\end{aligned}$$

FrozenLake Deep Qlearning (1/7)

activation function choice

$$\phi(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

The derivative is then given by:

$$\begin{aligned}
\phi(x)' &= \frac{\big(\exp(x) + exp(-x)\big)\big(\exp(x) + \exp(-x)\big) - \big(\exp(x) - exp(-x)\big)\big(\exp(x) - \exp(-x)\big)}{\big(\exp(x) + \exp(-x)\big)^2} \\
&= \frac{4}{\big(\exp(x) + \exp(-x)\big)^2} \\
&= \big(1 + \phi(x)\big)\big(1 - \phi(x)\big)
\end{aligned}$$

FrozenLake Deep Qlearning (2/7)

```
import gym
import random
import numpy
import math

env = gym.make('FrozenLake-v0')

epsilon=0.1
gamma=0.9
alpha=0.5

def sigmoid(x):
    if x>100:
        returnValue=1
    elif x<-100:
        returnValue=-1
    else:
        returnValue=(math.exp(x)-math.exp(-x))/(math.exp(x)+math.exp(-x))
    return returnValue
```

FrozenLake Deep Qlearning (3/7)

```
class NN:
    def __init__(self,sizeInput,sizeHiddenLayer,sizeOutput):
        self.sizeInput=sizeInput
        self.sizeHiddenLayer=sizeHiddenLayer
        self.sizeOutput=sizeOutput

        #below are the weights
        self.HiddenLayerEntryWeights=numpy.zeros([sizeHiddenLayer,sizeInput])
        self.LastLayerEntryWeights=numpy.zeros([sizeOutput,sizeHiddenLayer])

        #random initialization
        for i in range(0,sizeHiddenLayer):
            for j in range(0,sizeInput):
                self.HiddenLayerEntryWeights[i,j]=random.uniform(-0.1,0.1)

        for i in range(0,sizeOutput):
            for j in range(0,sizeHiddenLayer):
                self.LastLayerEntryWeights[i,j]=random.uniform(-0.1,0.1)

        self.HiddenLayerEntryDeltas=numpy.zeros([sizeHiddenLayer,nbCells])
        self.LastLayerEntryDeltas=numpy.zeros([sizeOutput,sizeHiddenLayer])

        self.HiddenLayerOutput=numpy.zeros(sizeHiddenLayer)
        self.LastLayerOutput=numpy.zeros(sizeOutput)

    def output(self,x):
        for i in range(0, self.sizeHiddenLayer):
            self.HiddenLayerOutput[i]=sigmoid(numpy.dot(self.HiddenLayerEntryWeights[i],x))
        for i in range(0, self.sizeOutput):
            self.LastLayerOutput[i]=  \
            sigmoid(numpy.dot(self.LastLayerEntryWeights[i],self.HiddenLayerOutput))
```

FrozenLake Deep Qlearning (4/7)

```
def retropropagation(self,x,y,actionIndex):
        self.output(x)

        #deltas computation
        for i in range(0,self.sizeHiddenLayer):
            self.LastLayerEntryDeltas[actionIndex,i]=2*(self.LastLayerOutput[actionIndex]-y)* \
            (1+self.LastLayerOutput[actionIndex])*(1-self.LastLayerOutput[actionIndex])

        for i in range(0,self.sizeHiddenLayer):
            for j in range(0,self.sizeInput):
                #here usually you need a sum
                self.HiddenLayerEntryDeltas[i,j]=self.LastLayerEntryDeltas[actionIndex,i]* \
                (1+self.HiddenLayerOutput[i])*(1-self.HiddenLayerOutput[i])* \
                self.LastLayerEntryWeights[actionIndex,i]

        #weights update
        for i in range(0,self.sizeHiddenLayer):
            self.LastLayerEntryWeights[actionIndex,i]-=alpha*self.LastLayerEntryDeltas[actionIndex,i]* \
            self.HiddenLayerOutput[i]

        for i in range(0,self.sizeHiddenLayer):
            for j in range(0,self.sizeInput):
                self.HiddenLayerEntryWeights[i,j]-=alpha*self.HiddenLayerEntryDeltas[i,j]*x[j]
```

FrozenLake Deep Qlearning (5/7)

```
nbCells=16;sizeInput=nbCells;sizeHiddenLayer=10
sizeOutput=4;myNN = NN(sizeInput, sizeHiddenLayer, sizeOutput)

nbEpisodes=50000
for i in range(0, nbEpisodes):
    if i>5000:
        alpha=0.1
    if i>10000:
        alpha=0.05
    if i>20000:
        alpha=0.02
    if i>30000:
        alpha=0.01

    state=env.reset();firstState=state;endOfEpisode = False;

    while not endOfEpisode:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample()
        else:
            x=numpy.zeros(sizeInput)
            x[state-1]=1
            myNN.output(x)
            action = numpy.argmax(myNN.LastLayerOutput)
```

FrozenLake Deep Qlearning (6/7)

```
next_state, reward, endOfEpisode, info = env.step(action)
x=numpy.zeros(sizeInput)
x[next_state-1]=1
myNN.output(x)
next_max = numpy.max(myNN.LastLayerOutput)
target = reward+gamma*next_max

x=numpy.zeros(sizeInput)
x[state-1]=1

if reward==1:
    successesInARow=successesInARow+1
    print("successesInARow: "+str(successesInARow))

myNN.retropropagation(x,target,action)

state = next_state
```

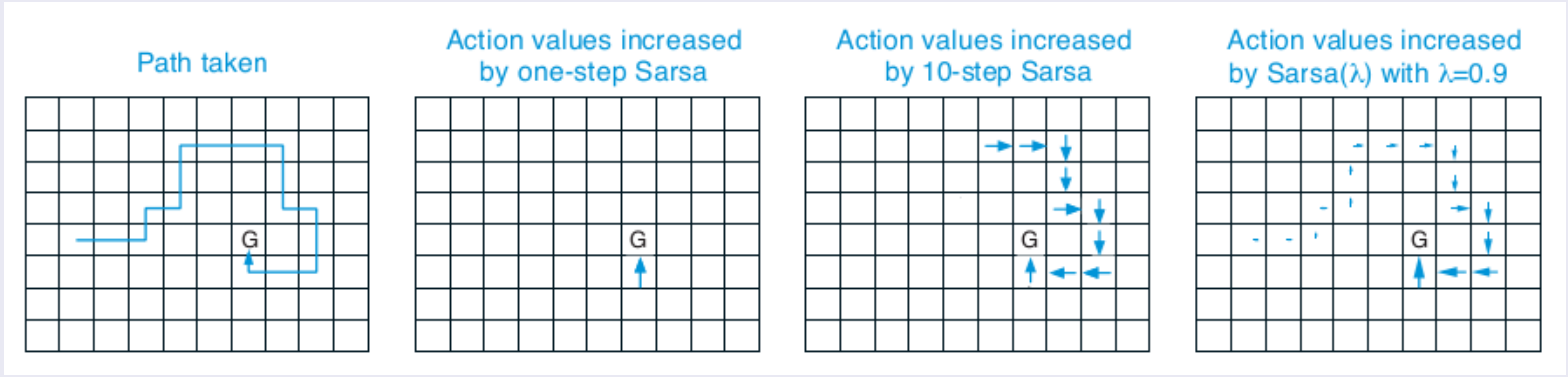
FrozenLake Deep Qlearning) (7/7)

Results

- 16 entry neurons, hidden layer of size 10
- 100000 learning steps
- decreasing α from 0.5 to 0.01
- initial random weights in $[-0.1, 0.1]$
- percentage of successes 73

TD(λ) (1/3)

Motivation



TD(λ) (2/3)

Basic ideas

- extend n-steps method by fading ancient steps!
- vector z capturing the trace decay

$z_1 = 0, z_t = \gamma\lambda z_{t-1} + \nabla\hat{v}(S_t, w_t)$ where γ is still the discount factor and λ the trace-decay parameter.

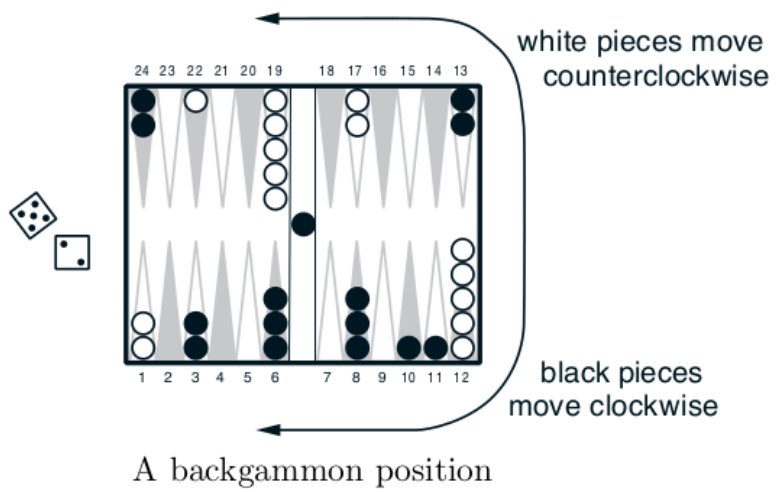
TD(λ) for estimating the policy value $\nu(., w)$ (3/3)

for each episode **do**
  $z \leftarrow 0$
  for each step $t \in 0, \ldots, T-1$ of current episode **do**
    choose $A \sim \pi(.|S)$
    take action A, observe R, S'
    $z \leftarrow \gamma\lambda z + \nabla \hat{v}(S, w)$
    $\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$
    $w \leftarrow w + \alpha\delta z$
    $S \leftarrow S'$
  end for
end for

TD-gammon (1/5)

The game

- 15 white and 15 black pieces in 24 locations



A backgammon position

pieces are moved by throwing 2 dice indicating the number of moves of two pieces

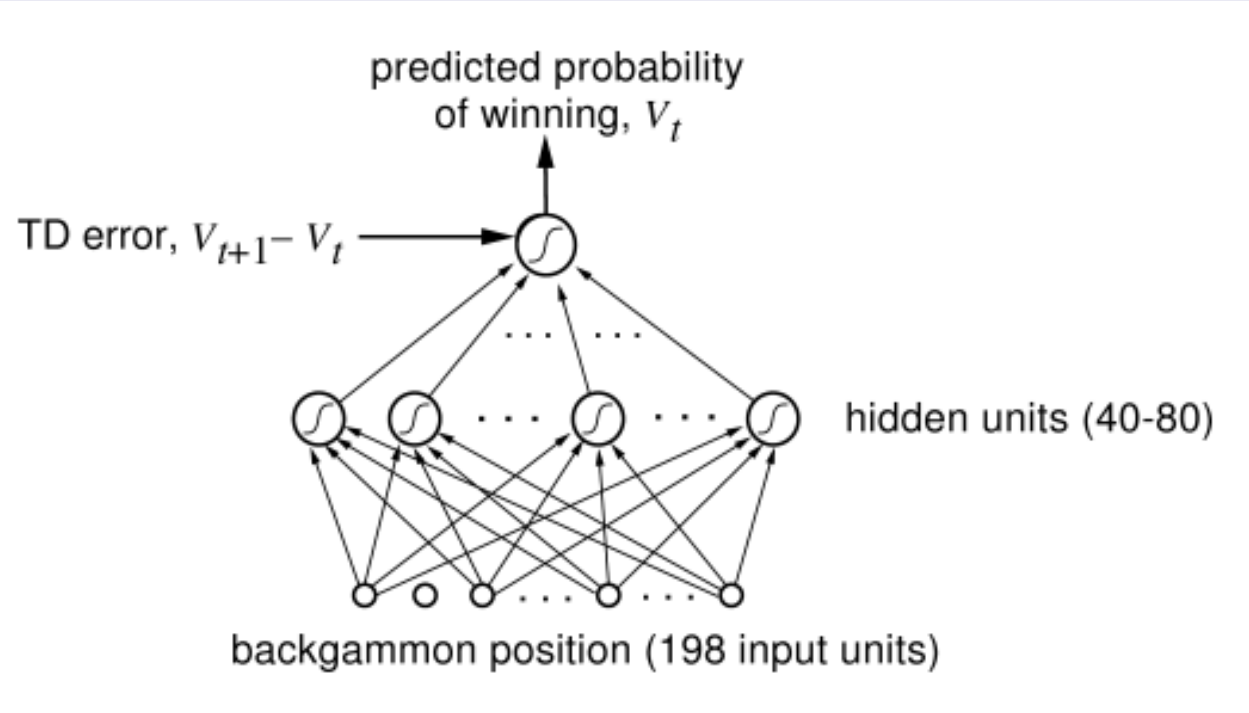
TD-gammon (2/5)

The method

- TD(λ)
- estimating position value $\hat{\nu}(s, w)$ for state s and parameter w
- use of a neural network
- $w_{t+1} = w_t + \alpha\Big[R_{t+1} + \gamma\hat{\nu}(S_{t+1}, w_t) - \hat{\nu}(S_t, w_t)\Big]z_t$
- eligibility traces update by $z_t = \gamma\lambda z_{t-1} + \nabla\hat{\nu}(S_t, w_t)$ with $z_0 = 0$
- the gradient is computed by backpropagation

TD-gammon (3/5)

Neural network



- 198 input units, coding is important, described in next slide
- activation function for unit j is $h(j) = \frac{1}{1+\exp\left\{-\sum_i w_{ij} x_i\right\}}$
- estimating position value $\hat{\nu}(s, w)$ for state s and parameter w

TD-gammon (4/5)

Neural network 198 input units

There are 24 points, each is encoded by 4 units for white and 4 units for black pieces in the following way:

- 1 piece, then first unit equal to 1: the piece can be hit by the opponent
- 2 pieces, then the second unit is also equal to 1: concept of protection, the opponent cannot hit these pieces (made point)
- 3 pieces then the third unit is equal to 1: concept of one spare piece, it can move without leaving the point defenseless
- more than 3 pieces, then the fourth unit encodes this excess of pieces using formula $(n-3)/2$

This yields $192 = 8 * 24$,three additional units for the white and the black encode:

- turn to play
- number of pieces on the bar

TD-gammon (5/5)

Results

Program	Hidden Units	Training Games	Opponents	Results
TD-Gammon 0.0	40	300,000	other programs	tied with the bests
TD-Gammon 1.0	80	300,000	Robertie, Magriel	-13pts/51 games
TD-Gammon 2.0	40	800,000	various grandmasters	-7pts/38 games
TD-Gammon 2.1	80	1,500,000	Robertie	-1pt/40 games
TD-Gammon 3.0	80	1,500,000	Kazaros	+6pts/20 games

Policy gradient methods (1/2)

$q(s, a)$ or $\nu(s)$ approximation means

- $\hat{\nu}(s, w) \approx \nu(s)$
- $\hat{q}(s, a, w) \approx q(s, a)$

so policy gradient is not this, even if a common tool is neural network.

Policy gradient methods (2/2)

Policy gradient answers positively to this question: can we have a policy which does not estimate q or ν ?

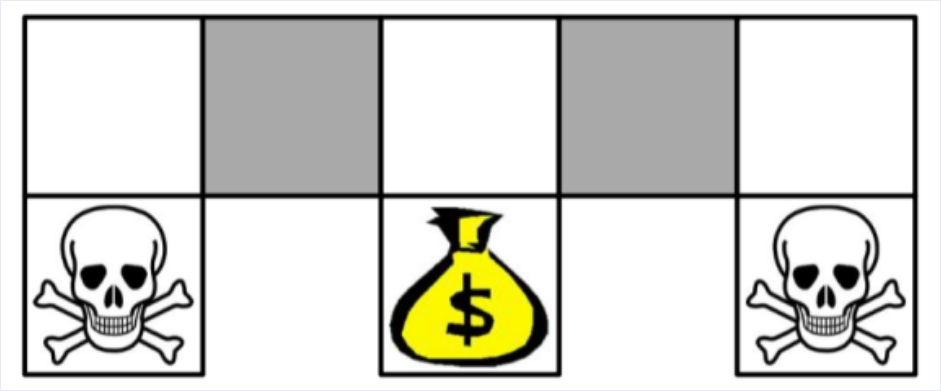
- The policy is parametrized by a vector $\theta \in \mathbb{R}^{d'}$ so that the probability of choosing action a is given by: $\pi(a|s,\theta) = P(A_t = a|S_t = s, \theta_t = \theta)$
- $$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}$$

  where $J(\theta)$ is some criterion to be optimized involving rewards.

Motivation (1/2)

Example 1

sometimes random is better ! Think of rock/paper/scissors or the following example:



Motivation (2/2)

Example 2

Policy gradient enables to work with a large number (possibly infinite) of actions, just consider learning the probability distribution of actions as for instance:

$$\pi(a|s,\theta) = \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2)}\right)$$

with $\mu(s,\theta) = \theta_\mu^\top x_\mu(s)$ and $\sigma(s,\theta) = \exp\left(\theta_\sigma^\top x_\sigma(s)\right)$ where $x_\mu(s)$ and $\sigma(s,\theta)$ feature vectors modeled possibly by neural networks

REINFORCE algorithm (episodic case) (2/4)

---

initialize $\theta \in \mathbb{R}^{d'}$

for ever **do**

  generate episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi(.|., \theta)$

  for each step $t \in 0, \ldots, T-1$ of current episode **do**

    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$

    $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \log \pi(A_t|S_t, \theta)$

  end for

end for

REINFORCE algorithm with baseline (episodic case) (3/4)

Application to Chinese-English translation

Adaptive Multi-pass decoder for neural machine translation X. Geng et al.

Source	*héshìzhù xiānshēng dē wěirènqī wéi yī nián , yǐ pèihé qí wéi fángwěihuì wěiyuán dē rèngqī jièmǎn rìqī , qítā xīn wěirèngwěiyuān dē rèngqī zé wéi liǎngnían .*
Reference	*all appointments are for two years , except that of mr ho sai - chu 's which is for one year in order to tie in with the expiry date of his appointment as an ha member .*
1st-pass	*mr ho sai - chu 's UNK is a year - long term of two years with a term of two years as the term of his term of office of the ha .*
2nd-pass	*mr ho sai - chu 's UNK is a year - long term of two years with a term of two years to serve as the term of office of the ha .*
3rd-pass	*mr ho sai - chu 's UNK is a year - long term of two years with a term of two years to tie in with the expiry date of his term of office .*
4th-pass	*mr ho sai - chu has been serving as a member of authority for a term of two years with a term of two years .*

REINFORCE algorithm (episodic case) (3/4)

In a few words:

- REINFORCE is just used to decide the number of passes
- States: $\{1, 2, 3, \dots\}$ for the pass number
- Actions: Stop or Continue
- policy parameter: $\theta = (W_p, b_p)$
- decision taken according to: $\text{softmax}(W_p s + b_p)$ where s is the current state
- rewards are given according to the metric BLEU (Papineni et al., 2002)

Mathematical background for episodic REINFORCE (1/2)

$$
\begin{aligned}
\nabla \nu_\pi(s) &= \nabla \Big[\sum_a \pi(a|s) q_\pi(s,a) \Big], \forall s \in S \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r|s,a)(r + \nu_\pi(s')) \Big] \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s',r} p(s'|s,a) \nabla \nu_\pi(s') \Big] \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s',r} p(s'|s,a) \\
&\quad \Big[\sum_{a'} \Big[\nabla \pi(a'|s') q_\pi(s',a') + \pi(a'|s') \sum_{s'',r} p(s''|s',a') \nabla \nu_\pi(s'') \Big] \Big] \\
&\quad \text{by iterating} \\
&= \sum_{s \in S} \sum_{k=0}^{+\infty} P(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x,a)
\end{aligned}
$$

Mathematical background for episodic REINFORCE (2/2)

$$
\begin{aligned}
\nabla J(\theta) &= \nabla \nu_\pi(s_0) \\
&= \sum_s \left(\sum_{k=0}^{+\infty} P(s_0 \to s, k\pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) \text{ where } \eta(s) \text{ stands for the number} \\
&\quad \text{of times state } s \text{ has been encountered} \\
&\approx \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a)
\end{aligned}
$$

REINFORCE algorithm with baseline (episodic case) (1/5)

Reducing variance

Simple idea: doing something similar to control variables in estimation. In details:

$$\nabla J(\theta) \approx \sum_s \mu(s) \sum_a \big(q_\pi(s,a) - b(s)\big) \nabla \pi(a|s,\theta)$$

Indeed,

$$\sum_a b(s) \nabla \pi(a|s,\theta) = b(s) \nabla \sum_a \pi(a|s,\theta) = b(s) \nabla 1 = 0$$

Then, the update is:

$$\theta_{t+1} = \theta_t + \alpha\big(G_t - b(S_t)\big) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

REINFORCE algorithm with baseline (episodic case) (2/5)

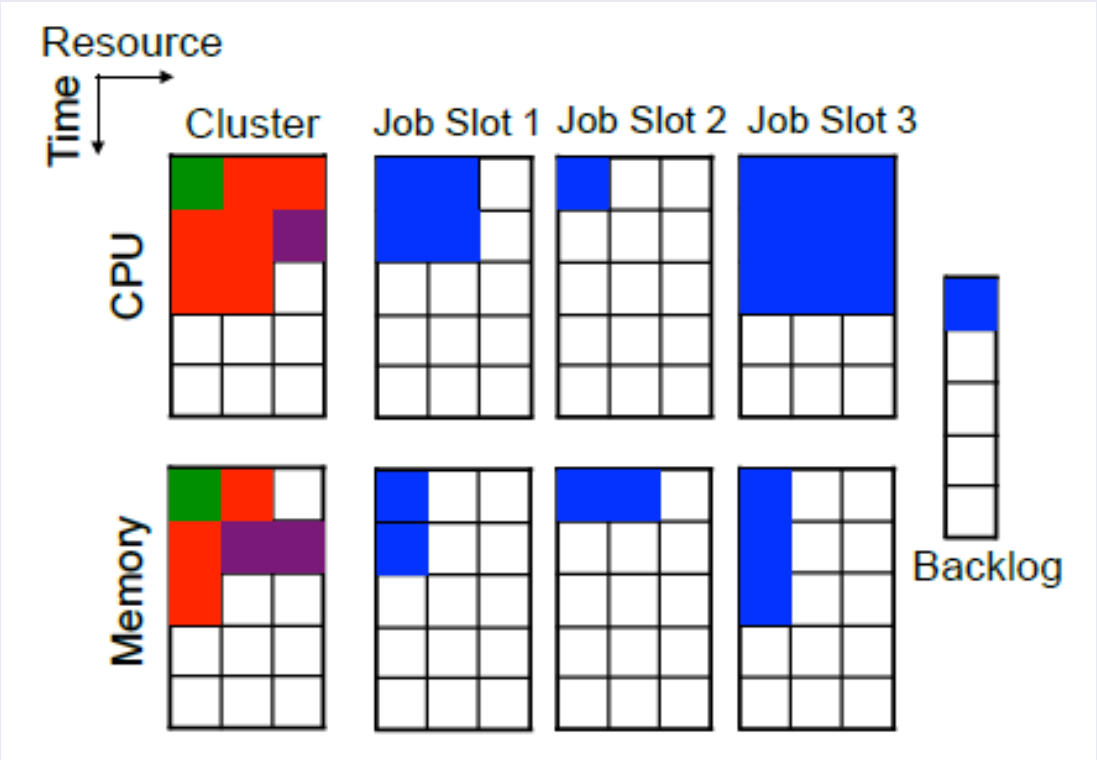
initialize $\theta \in \mathbb{R}^{d'}$ and $w \in \mathbb{R}^{d}$
for ever **do**
  generate episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi(.|., \theta)$
  for each step $t \in 0, \ldots, T-1$ of current episode **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\delta \leftarrow G - \hat{\nu}(S_t, w)$
    $w \leftarrow w + \alpha^w \delta \nabla \hat{\nu}(S_t, w)$
    $\theta \leftarrow \theta + \alpha^\theta \gamma^t \delta \nabla \log \pi(A_t|S_t, \theta)$
  end for
end for

REINFORCE algorithm with baseline (episodic case) (3/5)

Application to ressource management

Resource management with deep reinforcement learning (Mao et al., 2016)

- M backlog jobs
- d ressources



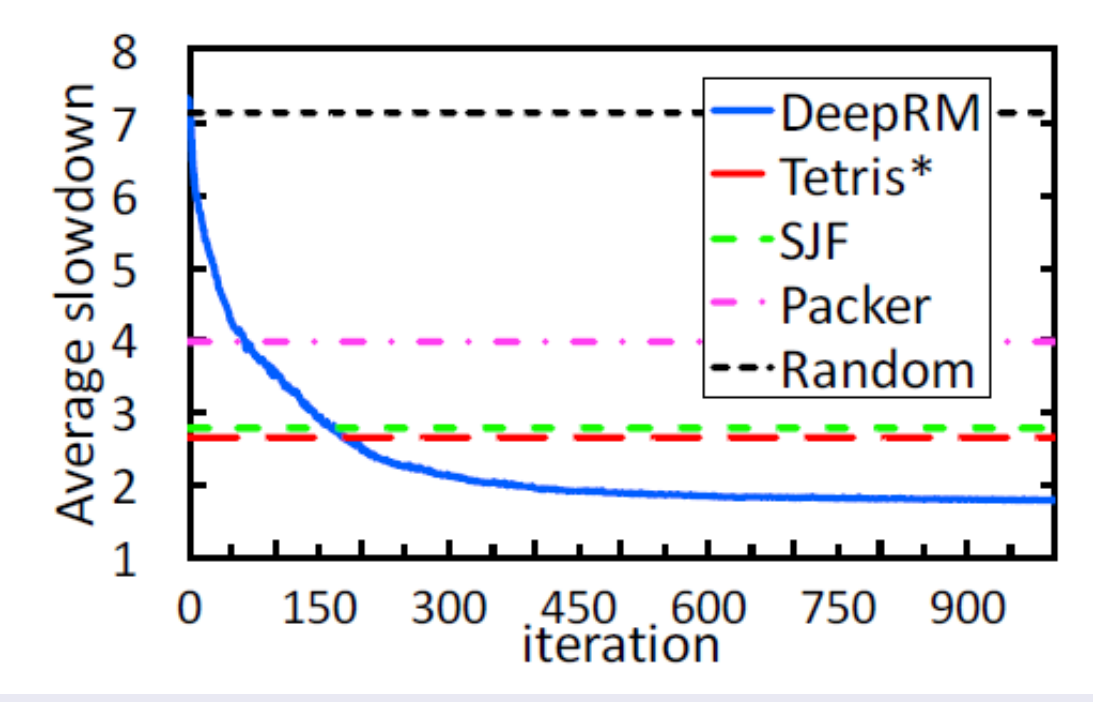
REINFORCE algorithm with baseline (episodic case) (4/5)

Strategy in a few words

- training is episodic using various jobsets with several episodes
- policy $\pi_\theta(s, a)$ provided by a neural network
- actions $\{\varnothing, 1, \ldots, M\}$ where action $i \in \mathbb{N}$ means schedule job i, $\varnothing$ means schedule no job
- state: resource profile of each job i, $r_{i1}, \ldots, r_{id}$ and current occupancy of resources
- rewards: $\sum_{j \in \mathcal{J}} \frac{-1}{T_j}$ where $\mathcal{J}$ represents all the jobs in the system
- baseline: mean rewards at time t along all episodes of a jobset

REINFORCE algorithm with baseline (episodic case) (5/5)

Results



Average slowdown: $\sum_j \frac{C_j}{T_j}$ where C_j is the observed completion time of job j and T_j the theoretical duration of the job

ACTOR-Critic

Basic idea

replace the baseline by an estimation

One-step ACTOR-Critic (episodic case) (1/2)

initialize $\theta \in \mathbb{R}^{d'}$, $w \in \mathbb{R}^{d}$ and S
for ever **do**
  initialize S and $I \leftarrow 1$
  for each step $t \in 0, \ldots, T-1$ of current episode **do**
    $A \sim \pi(.|S, \theta)$
    take action A, observe S' and R
    $\delta \leftarrow R + \gamma \hat{\nu}(S', w) - \hat{\nu}(S, w)$
    $w \leftarrow w + \alpha^{w} \delta \nabla \hat{\nu}(S, w)$
    $\theta \leftarrow \theta + \alpha^{\theta} I \delta \nabla \log \pi(A|S, \theta)$
    $I \leftarrow \gamma I$ and $S \leftarrow S'$
  end for
end for

One-step ACTOR-Critic (episodic case) (2/2)

Application to Atari 2600 games

Asynchronous methods for deep reinforcement learning (V. Mnih et al., 2016)

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

- 4 hidden layers
- outputs for decision and outputs for state value

ACTOR-Critic with eligibility traces (episodic case)

initialize $\theta \in \mathbb{R}^{d'}$, $w \in \mathbb{R}^{d}$ and S
for ever **do**
  initialize S, $I \leftarrow 1$, $z^{\theta} \leftarrow 0$ and $z^{w} \leftarrow 0$
  for each step $t \in 0, \ldots, T-1$ of current episode **do**
    $A \sim \pi(.|S, \theta)$
    take action A, observe S' and R
    $\delta \leftarrow R + \gamma \hat{\nu}(S', w) - \hat{\nu}(S, w)$
    $z^{w} \leftarrow \gamma \lambda^{w} z^{w} + \nabla \hat{v}(S, w)$
    $z^{\theta} \leftarrow \gamma \lambda^{\theta} z^{\theta} + I \nabla \log \pi(A|S, \theta)$
    $w \leftarrow w + \alpha^{w} \delta z^{w}$
    $\theta \leftarrow \theta + \alpha^{\theta} \delta z^{\theta}$
    $I \leftarrow \gamma I$ and $S \leftarrow S'$
  end for
end for

ACTOR-Critic with eligibility traces (continuing case)

initialize $\theta \in \mathbb{R}^{d'}$, $w \in \mathbb{R}^{d}$ and S
for ever **do**
  initialize S, $z^{\theta} \leftarrow 0$ and $z^{w} \leftarrow 0$
  for each step $t \in 0, \ldots, T-1$ of current episode **do**
    $A \sim \pi(.|S, \theta)$
    take action A, observe S' and R
    $\delta \leftarrow R - \bar{R} + \hat{\nu}(S', w) - \hat{\nu}(S, w)$
    $\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}} \delta$
    $z^{w} \leftarrow \lambda^{w} z^{w} + \nabla \hat{v}(S, w)$
    $z^{\theta} \leftarrow \lambda^{\theta} z^{\theta} + \nabla \log \pi(A|S, \theta)$
    $w \leftarrow w + \alpha^{w} \delta z^{w}$
    $\theta \leftarrow \theta + \alpha^{\theta} \delta z^{\theta}$
    $S \leftarrow S'$
  end for
end for

Maths for continuing policy gradient (1/3)

$$\begin{aligned}
\nabla \nu_\phi(s) &= \nabla\Big[\sum_a \pi(a|s) q_\pi(s,a)\Big], \forall s \in S \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a)\Big] \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r|s,a)(r - r(\theta) + \nu_\pi(s')\Big] \\
&= \sum_a \Bigg[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s)\Big[-\nabla r(\theta) + \sum_{s',r} p(s'|s,a) \nabla \nu_\pi(s')\Big]\Bigg]
\end{aligned}$$

This implies:

$$\nabla r(\theta) = \sum_a \Bigg[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla \nu_\pi(s')\Bigg] - \nabla \nu_\pi(s)$$

Maths for continuing policy gradient (2/3)

Preceding result is true for any s

$$\begin{aligned}
\nabla J(\theta) &= \sum_s \mu(s) \Bigg(\sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \\
&\qquad \pi(a|s) \sum_{s'} p(s'|s,a) \nabla \nu_\pi(s') \Big] - \nabla \nu_\pi(s) \Bigg) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) + \\
&\quad \sum_s \nu(s) \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \nabla \nu_\pi(s') - \sum_s \mu(s) \nabla \nu_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) + \\
&\quad + \sum_{s'} \sum_s \mu(s) \sum_a \pi(a|s) p(s'|s,a) \nabla \nu_\pi(s') - \sum_s \mu(s) \nabla \nu_\pi(s)
\end{aligned}$$

Maths for continuing policy gradient (3/3)

Then, remembering that $\mu(s)$ is the stationary distribution, we get:

$$\begin{aligned}\nabla J(\theta) &= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) + \sum_{s'} \mu(s') \nabla \nu_\pi(s') - \sum_s \mu(s) \nabla \nu_\pi(s) \\ &= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a)\end{aligned}$$

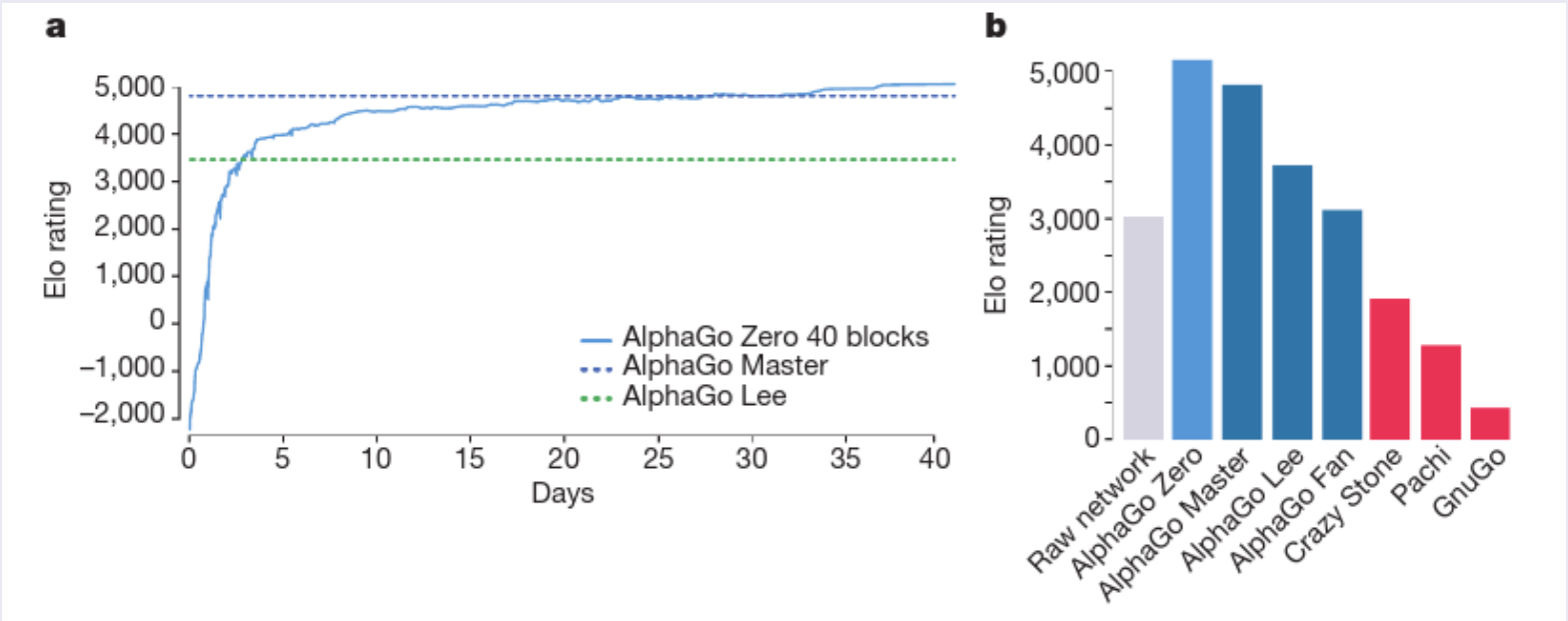
AlphaGo Zero (1/4)

In a few lines

- breakthrough in 2017 in the go game (Silver et al., 2017)
- only reinforcement learning without prior knowledge
- defeated all top go players

AlphaGo Zero (2/4)

Results



AlphaGo Zero (3/4)

The maths behind

- at its core a neural network whose output is $(p, v) = f_\theta(s)$ where the loss function is given by $l = (z - v)^2 - \pi^\top p + c\|\theta\|^2$
- for each move a, $\pi_a(s) \propto N(s,a)^{1/\tau}$ where τ is a temperature parameter
- $N(s,a)$ number of visits of (s,a)
- nodes visited according to Monte-Carlo Tree Search and score $Q(s,a) + U(s,a)$ with $Q(s,a) = \frac{1}{N(s,a)} \sum_{s'|s,a \rightsquigarrow s'} V(s')$ and $U(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$

AlphaGo Zero (4/4)

The training

- 29 millions self-play games during 40 days, though the program was already strong after 3 days and 5 millions games
- input $19 \times 19 \times 17$ images: 8 for the current player for describing the current position and the 7 previous positions, the same for the opponent, and one for the color to play
- 1600 games to select each move