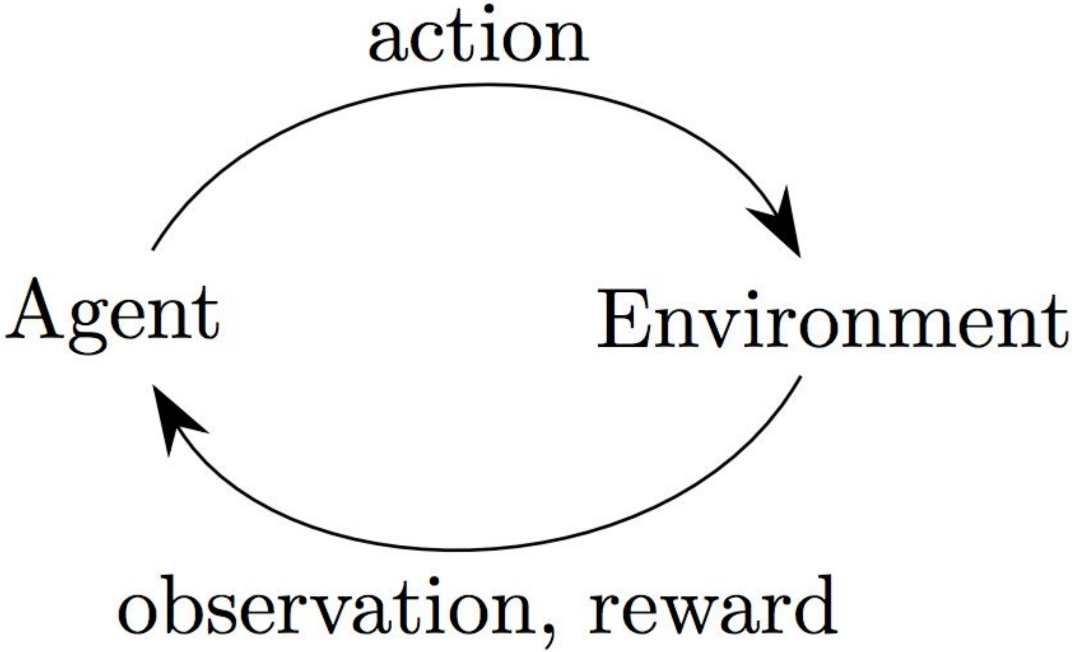
Intro to Reinforcement Learning

Rafal Jozefowicz, 4/28/2017

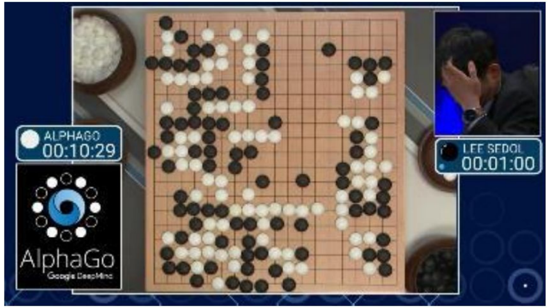
Reinforcement Learning - definition from wikipedia

Reinforcement learning is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward.



Why is this interesting?

- RL methods begin to work really well on many practical applications
- They let us build systems that are difficult or impossible to design by hand



Difference between Supervised Learning (SL) and Reinforcement Learning (RL)

Supervised Learning:

- Environment samples a pair $(x, y) \sim \rho$
- Agent (model) makes a prediction $y' = f(x)$
- Agent pays cost $loss(y, y')$ for its decision

The environment asks agent a question and says what was the true answer.

Difference between Supervised Learning (SL) and Reinforcement Learning (RL)

Reinforcement Learning:

- Environment samples $x_t \sim P(x_t \mid x_{t-1}, y_{t-1})$
  - x_t depends on previous actions!
- Agent makes decision $y_t = f(x_t)$
- Agent pays cost $c_t \sim P(c_t, x_t, y_t)$, but it doesn't know distribution P

Difference between Supervised Learning (SL) and Reinforcement Learning (RL)

In short:

- With RL we don’t have full access to the function we’re trying to optimize. We learn it through interactions.
- Input states and interaction costs depend on previous decisions

First steps - OpenAI gym

In [2]:

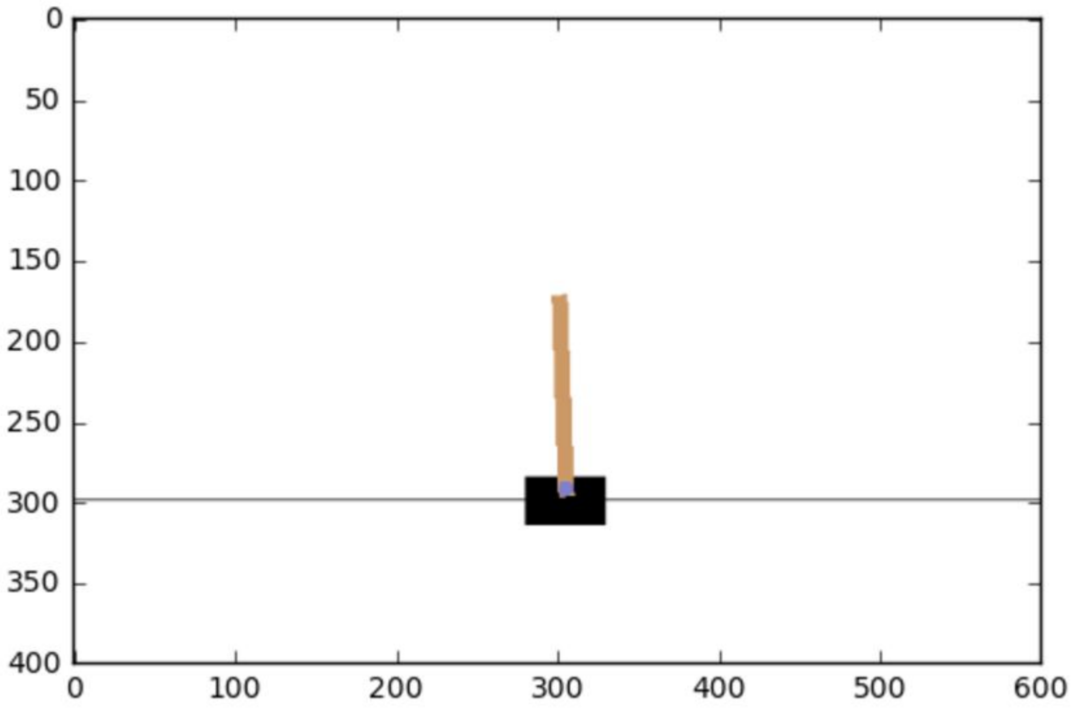
```
import gym
env = gym.make('CartPole-v1')
env.reset()
```

```
[2017-04-19 21:03:08,649] Making new env: CartPole-v1
```

First steps - OpenAI gym

```
In [3]: plt.imshow(env.render(mode='rgb_array'))

Out[3]: <matplotlib.image.AxesImage at 0x1131ed2e8>
```



First steps - random agent

In [6]:
```
def agent(observation):
    return env.action_space.sample()

done = False
observation = env.reset()
while not done:
    action = agent(observation)
    observation, reward, done, _ = env.step(action)
    env.render()
```

Available environments - https://gym.openai.com

MuJoCo

Continuous control tasks, running in a fast physics simulator.

InvertedPendulum-v1

Balance a pole on a cart.

InvertedDoublePendulum-v1

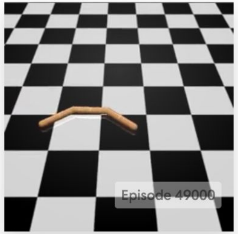
Balance a pole on a pole on a cart.

Reacher-v1

Make a 2D robot reach to a randomly located target.

HalfCheetah-v1

Make a 2D cheetah robot run.

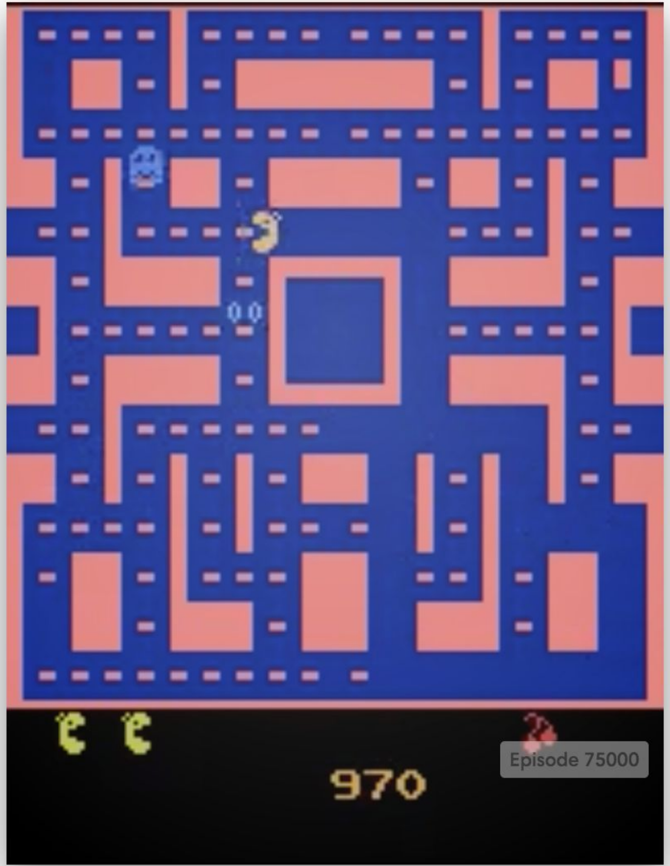


Swimmer-v1

Make a 2D robot swim.

Hopper-v1

Make a 2D robot hop.



Available environments - https://universe.openai.com/

Let’s solve CartPole - Cross-Entropy Method

Our agent will be a small neural network that predicts left/right actions.

General approach:

1. Let’s generate 100 random weights of NN and run each of them in env
2. Collect top 20% results and average their weights
3. Iterate forever

Weights are initially sampled from $\theta \sim N(\theta_{init}, std=1.0)$ and at each iteration we’ll update:
Θ = mean(\`top 20% best weights\`)
std = std(\`top 20% best weights\`)

Everything can be implement in 10 lines of Python code.

CEM on CartPole

Training typically takes <3 minutes on my laptop to get a perfect score.

CEM needs about 15 iterations to get there (x 100 episodes)

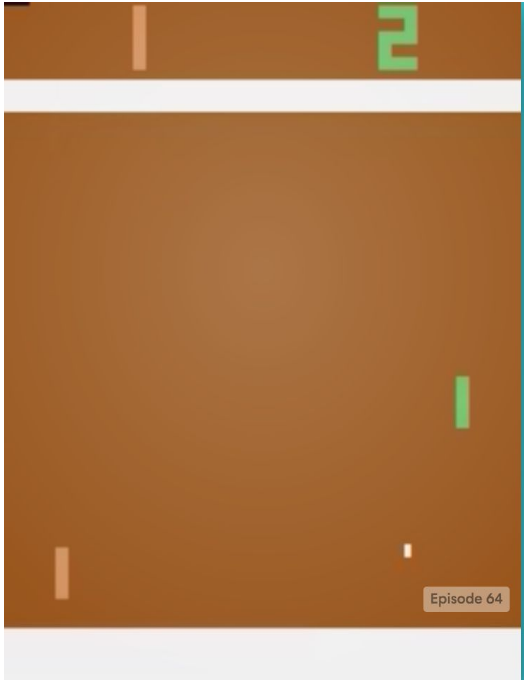
Mean scores after 1 iterations: 9.46
Mean scores after 6 iterations: 143.62
Mean scores after 11 iterations: 493.41
Mean scores after 16 iterations: 500.00

CEM

- Worth pointing out that we trained a neural network without using gradient information!
- CEM works surprisingly well on many low-dimensional problems

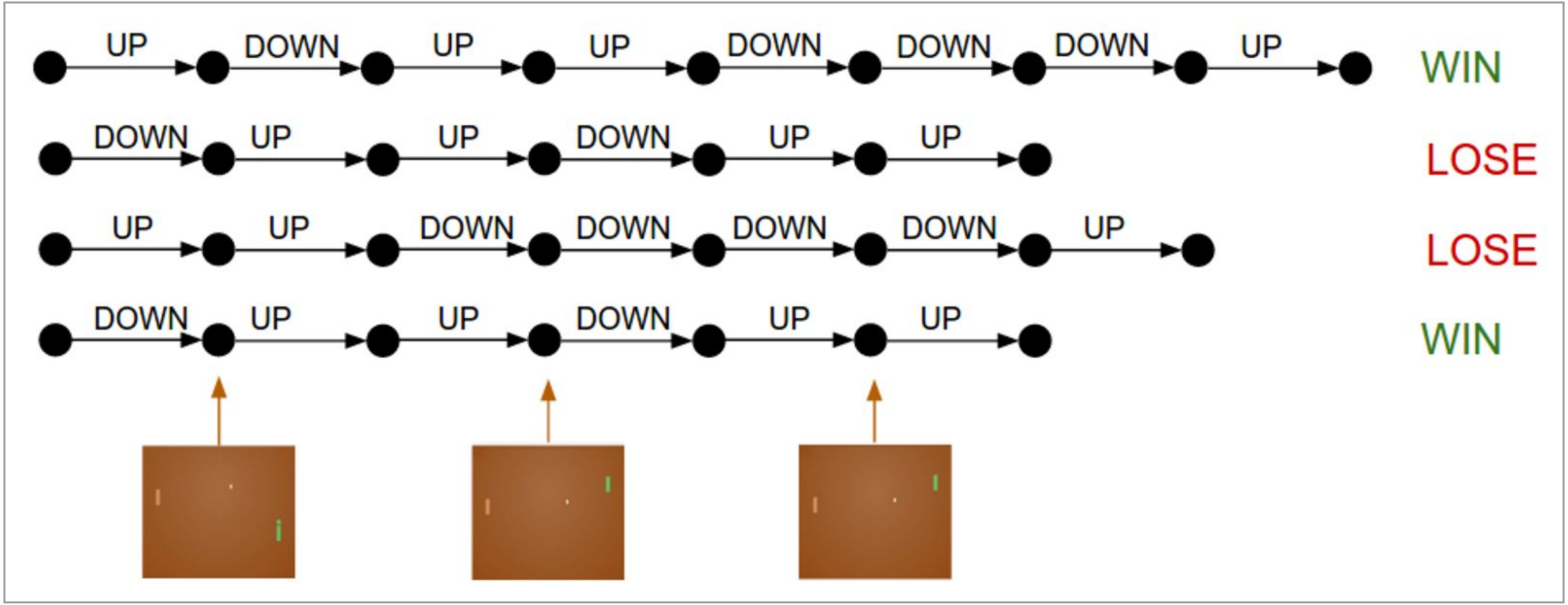
Pong

https://gym.openai.com/envs/Pong-v0

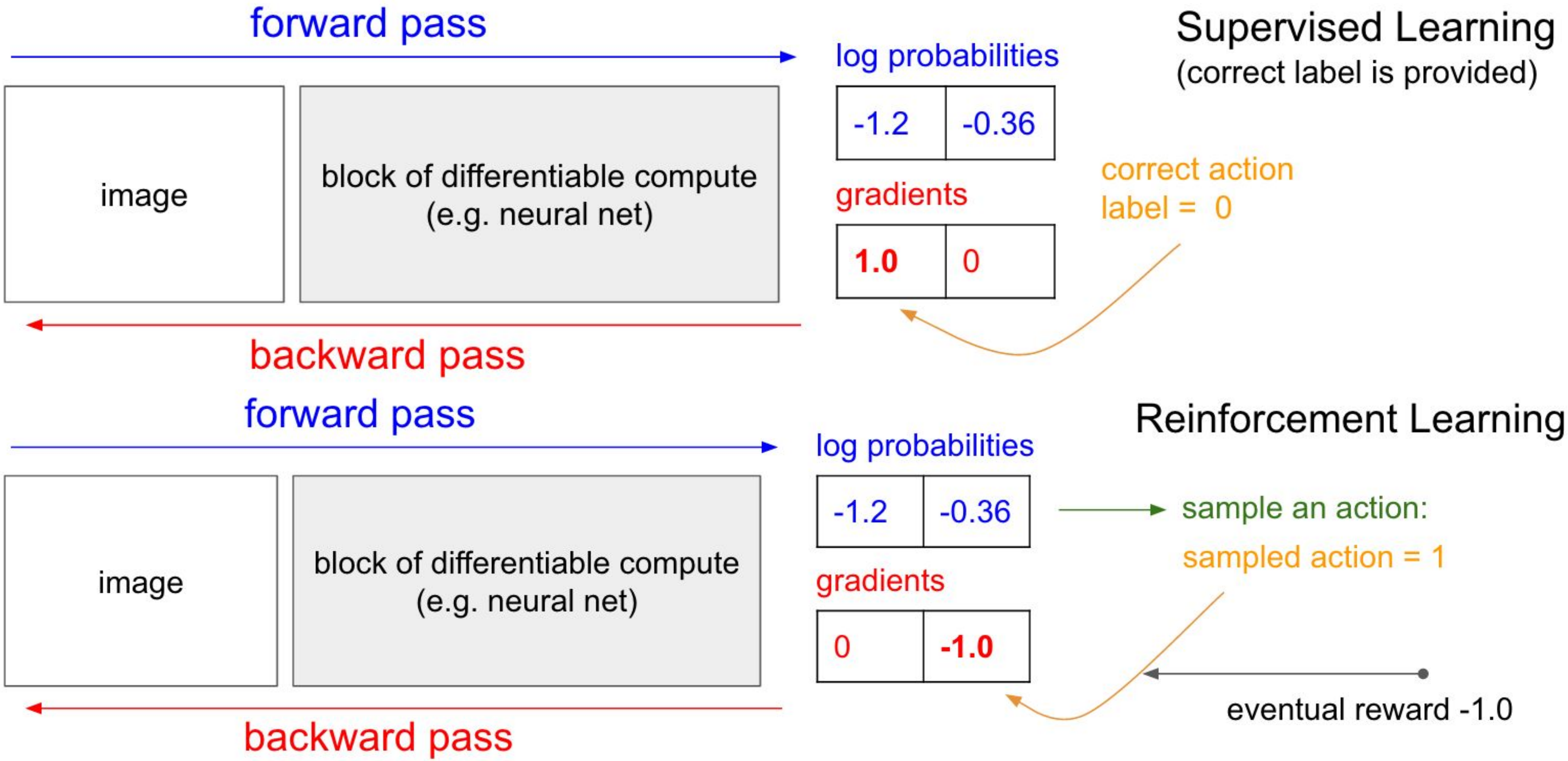


Policy Gradients

> *Policy Gradients: Run a policy for a while. See what actions led to high rewards. Increase their probability.*



Policy Gradients



Policy Gradients - more formally

$$
\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation} \\
&= \sum_x \nabla_\theta p(x) f(x) && \text{swap sum and gradient} \\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{both multiply and divide by } p(x) \\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z} \nabla_\theta z \\
&= E_x[f(x) \nabla_\theta \log p(x)] && \text{definition of expectation}
\end{aligned}
$$

Policy Gradients

It means that we can compute an unbiased estimator of gradients of f.

- This derivation works even if f(x) itself is discontinuous/non-differentiable
- Setting $f = \Sigma\, r_i$ gives us a method for optimizing the sum of future rewards

Algorithm

1. Run NN 10 times on a given environment, sample actions in each step according to model's distribution and collect data (states, actions, rewards)
2. Treat these sampled actions as labels and compute $g = [\log p(\text{action}|\text{state})]' * \Sigma\, r_i$, averaged over the data
3. Update model's weights using g and go back to 1)

This version of the algorithm is called REINFORCE (1991)

AlphaGo - http://airesearch.com/wp-content/uploads/2016/01/deepmind-mastering-go.pdf

AlphaGo

Why did it take so long to beat humans?

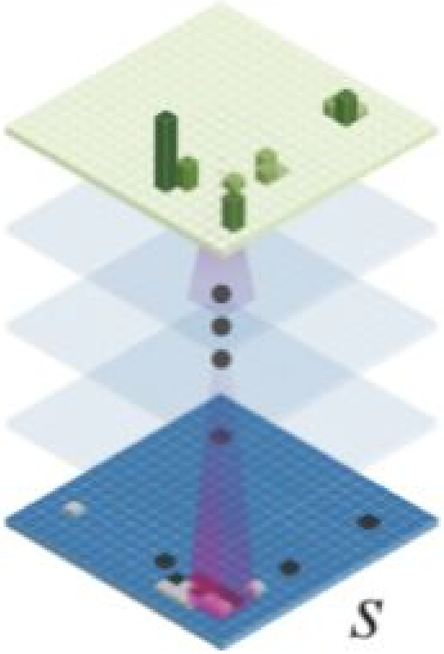
- State space is much larger than in chess (361 on empty board) and searching through game tree becomes computationally infeasible
- Decisions made in early game can affect results over a 100 steps ahead
- It’s not easy to tell who’s winning during the game, even for very good players

AlphaGo - architecture

- Neural net with 13 layers, most of them convolutional
- It outputs a probability distribution over all valid responses

Policy network

$$p_{\sigma/\rho}(a \mid s)$$



Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Extended Data Table 2: **Input features for neural networks.** Feature planes used by the policy network (all but last feature) and value network (all features).

AlphaGo - supervised pre-training

- At first, the model was trained on 30M positions from a database with good player games
- Best model reached 57% accuracy on this problem (copying human decisions); 55.7% for the version that used raw inputs. It needs about 3ms to evaluate position
- They also had a much smaller version with 24.2% accuracy but 1500x faster

AlphaGo - reinforcement learning

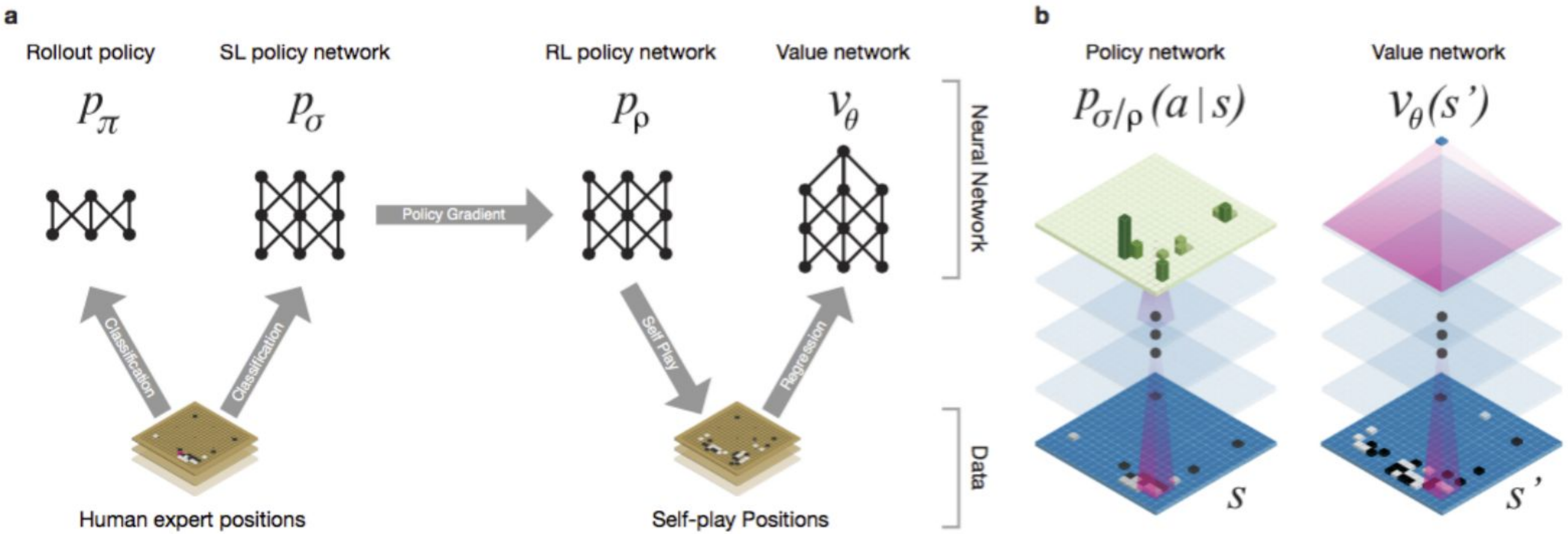
- REINFORCE was used, starting from neural network weights trained on historical games.
- The rewards were equal to 0 during the game and +1 or -1 at the last step
- NN was playing against randomly sampled previous version of the model
- After training this way, best model won 80% games against supervised learning baseline

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t$$

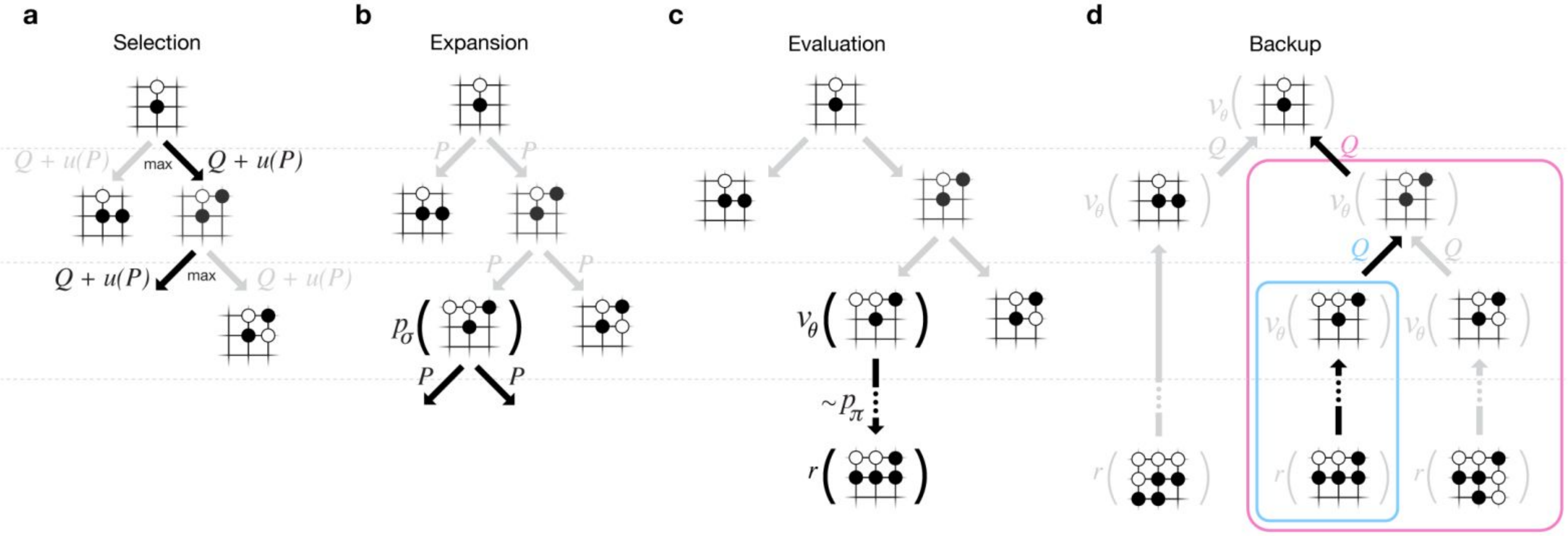
AlphaGo - value function

- In the last stage, we train a new network that will be used to score current position on the board
- Ideally we'd know win probability against optimal player but we just approximate it with our strongest model
- Using existing game database led to strong overfitting and instead they created a new dataset with 30M positions sampled from games played against itself. Each game contributed only one state

Training pipeline



AlphaGo - MCTS



AlphaGo - MCTS

- Actions in MCTS are chosen based on the value function plus additional exploration bonus, proportional to p(action|state) / (1 + N(state))
- Positions are evaluated using value function and a very fast rollout network that simulates the game from given state until the end

AlphaGo - Statystyki