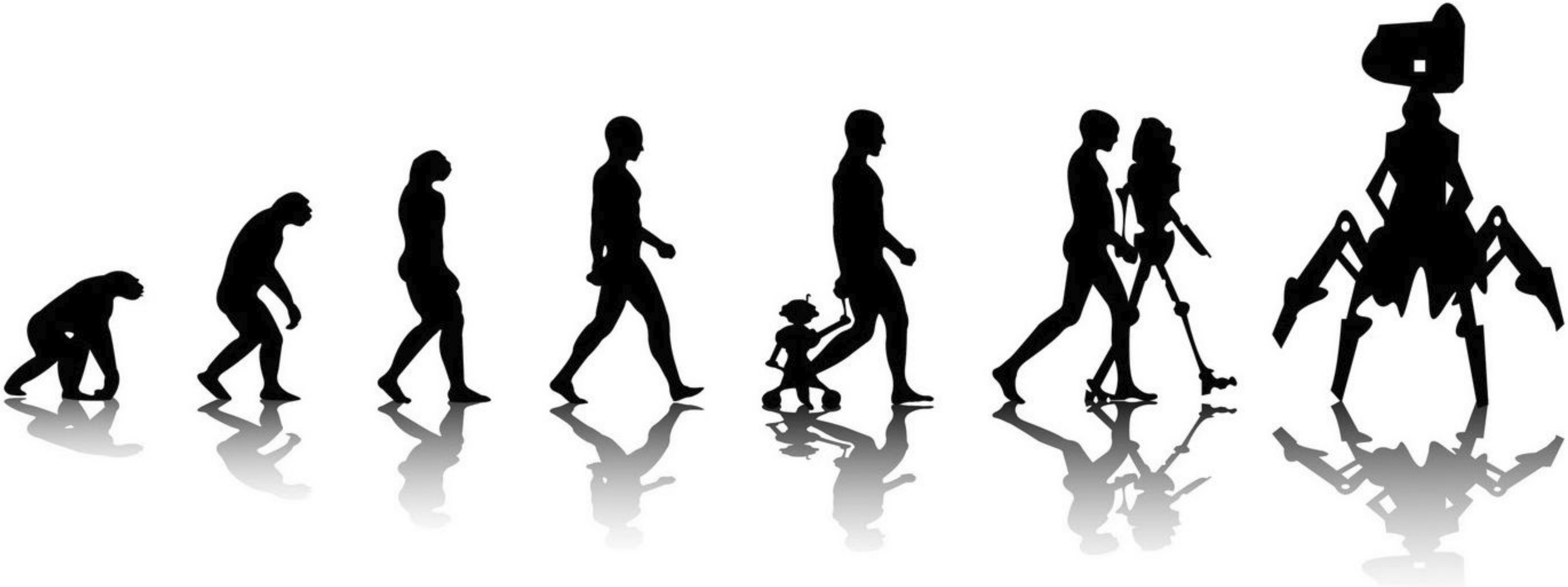


Modern Robotics: Evolutionary Robotics

COSC 4560 / COSC 5560

Professor Cheney

4/20/18

learning value networks

Bellman Equation

- Value function can be unrolled recursively

$$
\begin{aligned}
Q^{\pi}(s, a) &= \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s, a\right] \\
&= \mathbb{E}_{s'}\left[r + \gamma Q^{\pi}(s', a') \mid s, a\right]
\end{aligned}
$$

- Optimal value function $Q^*(s, a)$ can be unrolled recursively

$$
Q^*(s, a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right]
$$

- Value iteration algorithms solve the Bellman equation

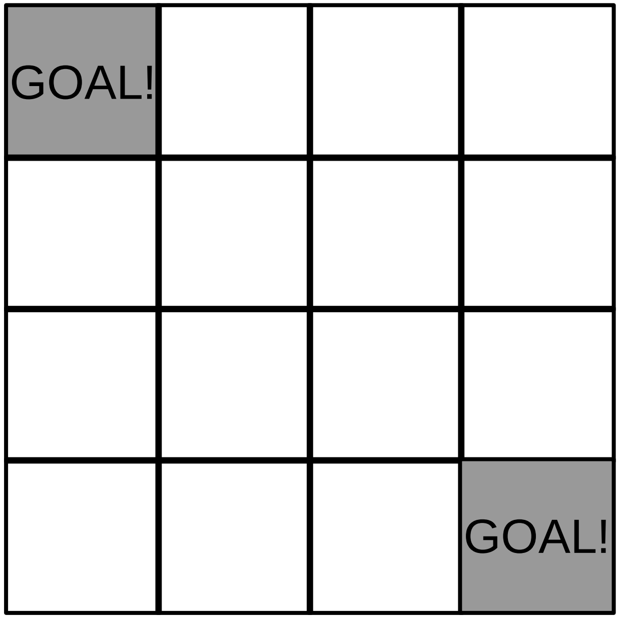
$$
Q_{i+1}(s, a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q_i(s', a') \mid s, a\right]
$$

grid world:

each timestep has -1 reward

the game terminates when you reach a goal state

actions: N, S, E, W



intuitive description: “get to the goal as soon as possible”
(but let's pretend we're a robot, who doesn't know this!)

each value function (V) is defined
with respect to some behavioral policy (π)
V^{π}

let's iteratively find V^{π} for a random policy
in our mini grid world

current value (V_k) for a random policy

k=0

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

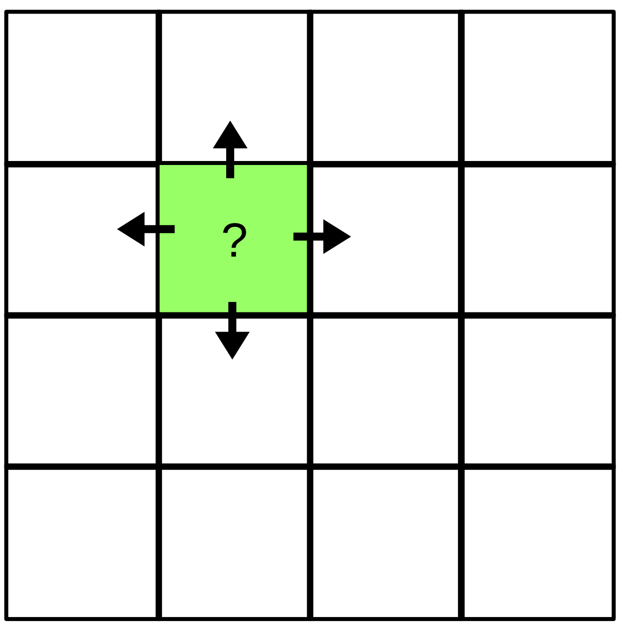
k=1

current value (V_k) for a random policy

k=0

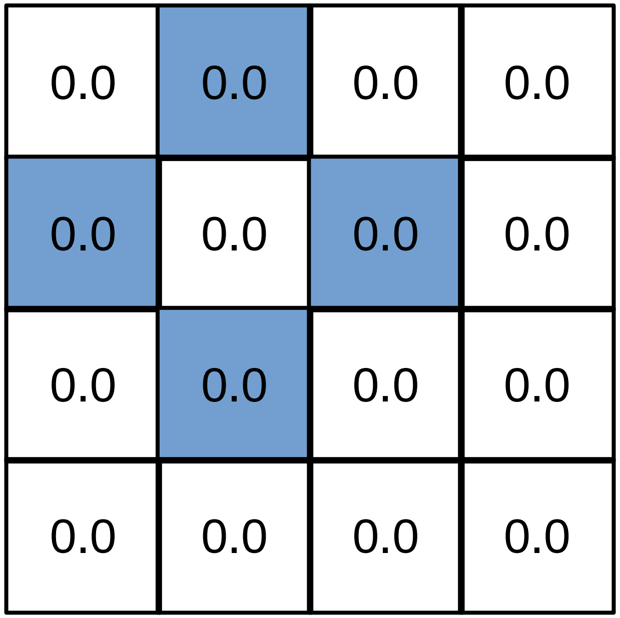
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

k=1

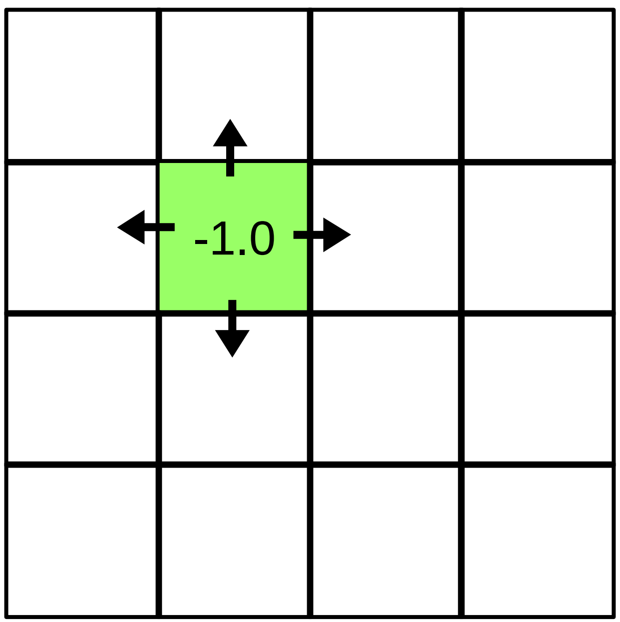


current value (V_k) for a random policy

k=0



k=1



immediate reward

current prediction for cumulative reward in new state

N: $V_{s,N} = -1 + 0 = -1$

S: $V_{s,S} = -1 + 0 = -1$

E: $V_{s,E} = -1 + 0 = -1$

W: $V_{s,W} = -1 + 0 = -1$

with a random policy,
we are equally likely to take any move,
so:

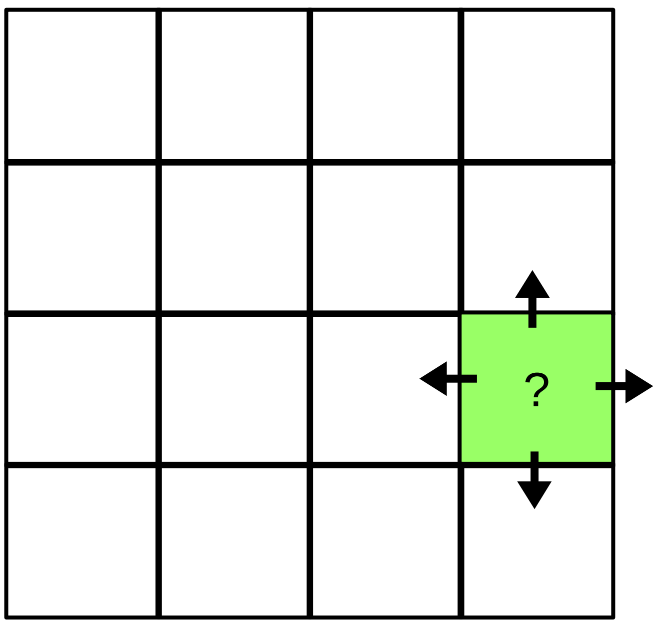
$V_s = (-1 + -1 + -1 + -1)/4 = -1$

current value (V_k) for a random policy

k=0

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

k=1



current value (V_k) for a random policy

k=0

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

k=1

			-1.0

current value (V_k) for a random policy

k=0

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

k=1

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

next iteration…
new value function
becomes
old value function
(“current prediction
for cumulative reward”)

current value (V_k) for a random policy

k=1

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

k=2

current value (V_k) for a random policy

k=1

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

k=2

	-1.75		

N: $V_{s,N} = -1 + -1 = -2$
S: $V_{s,S} = -1 + -1 = -2$
E: $V_{s,E} = -1 + 0 = -1$
W: $V_{s,W} = -1 + -1 = -2$

$V_s = (-2 + -2 + -1 + -2)/4 = -1.75$

current value (V_k) for a random policy

k=1

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

k=2

0.0	-1.75	-2.0	-2.0
-1.75	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.75
-2.0	-2.0	-1.75	0.0

current value (V_k) for a random policy

k=2

0.0	-1.75	-2.0	-2.0
-1.75	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.75
-2.0	-2.0	-1.75	0.0

k=3

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

current value (V_k) for a random policy

k=10

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

k=∞

0.0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0.0

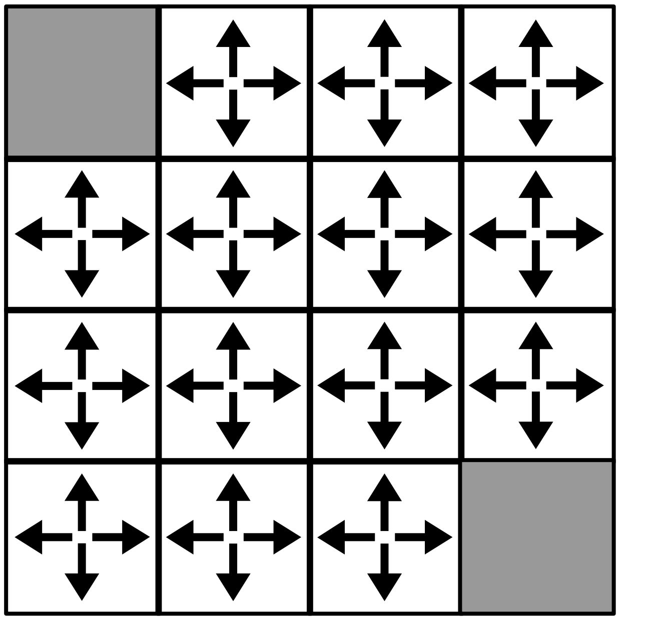
converged to true value function ($V^{\pi\text{-random}}$)

current value (V_k) for a random policy

greedy policy (π_k) for a this value function

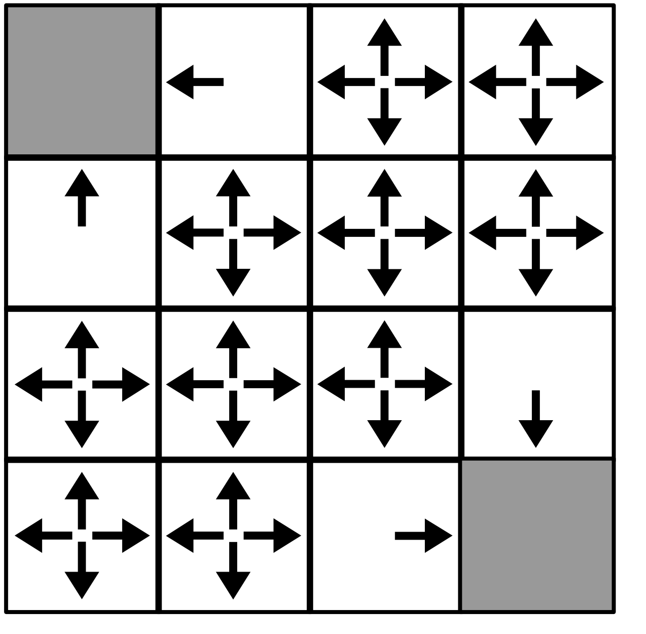
k=0

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



k=1

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

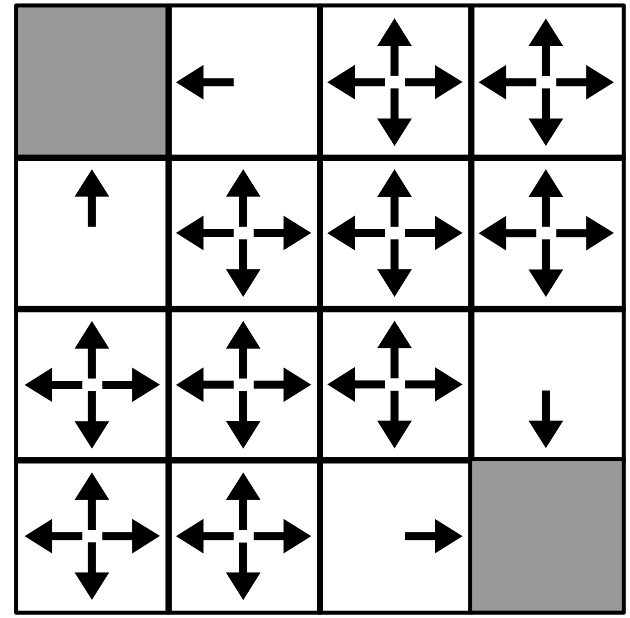


current value (V_k) for a random policy

greedy policy (π_k) for a this value function

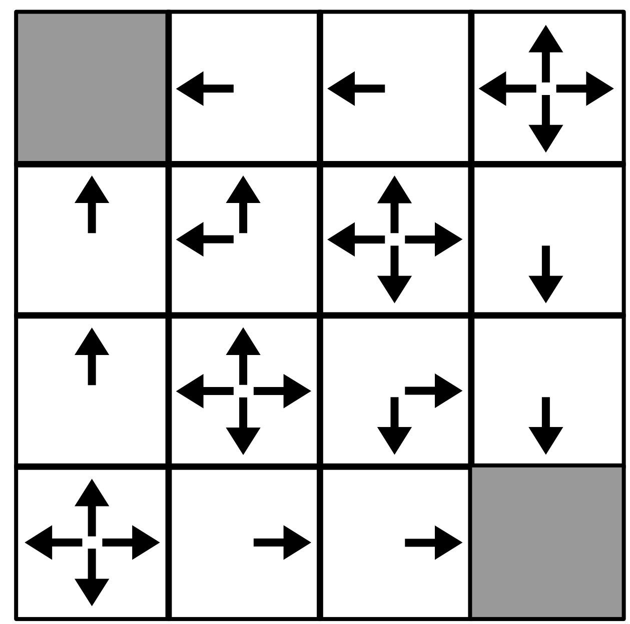
k=1

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0



k=2

0.0	-1.75	-2.0	-2.0
-1.75	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.75
-2.0	-2.0	-1.75	0.0

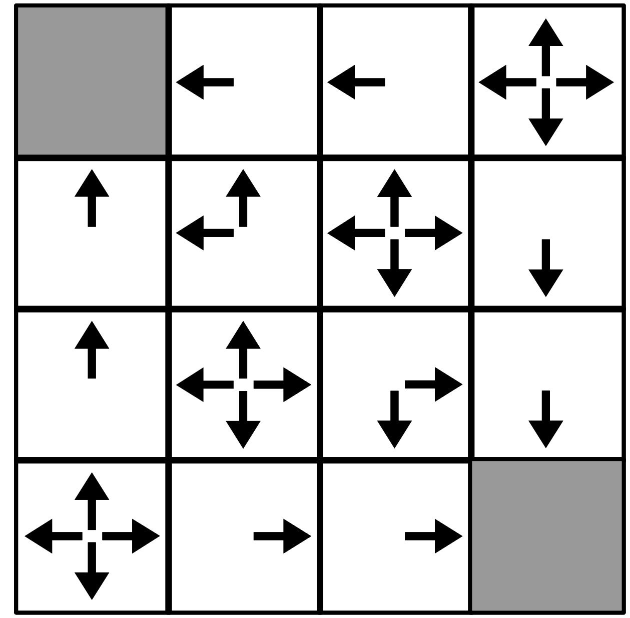


current value (V_k) for a random policy

greedy policy (π_k) for a this value function

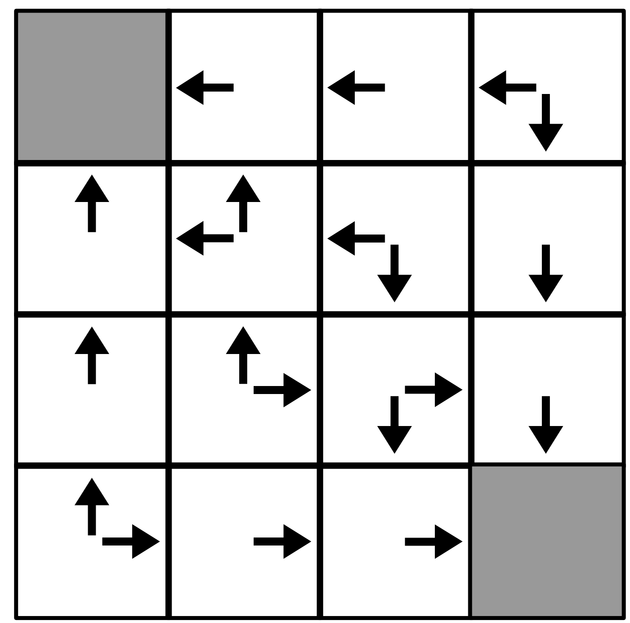
k=2

0.0	-1.75	-2.0	-2.0
-1.75	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.75
-2.0	-2.0	-1.75	0.0



k=3

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

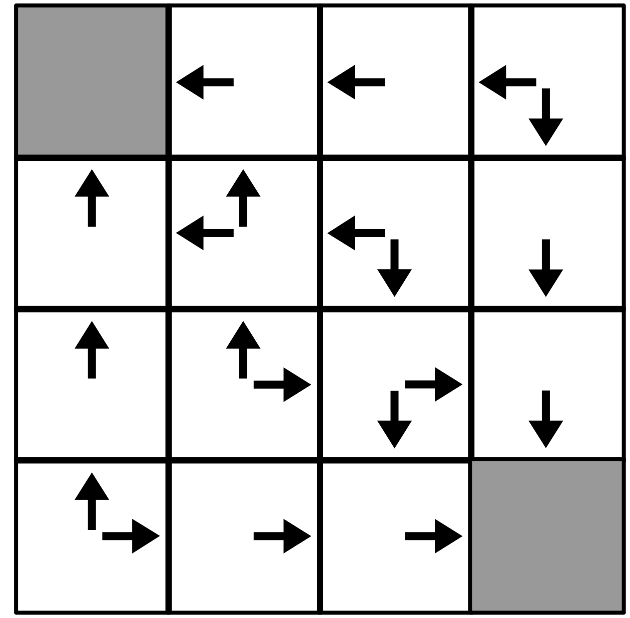


current value (V_k) for a random policy

greedy policy (π_k) for a this value function

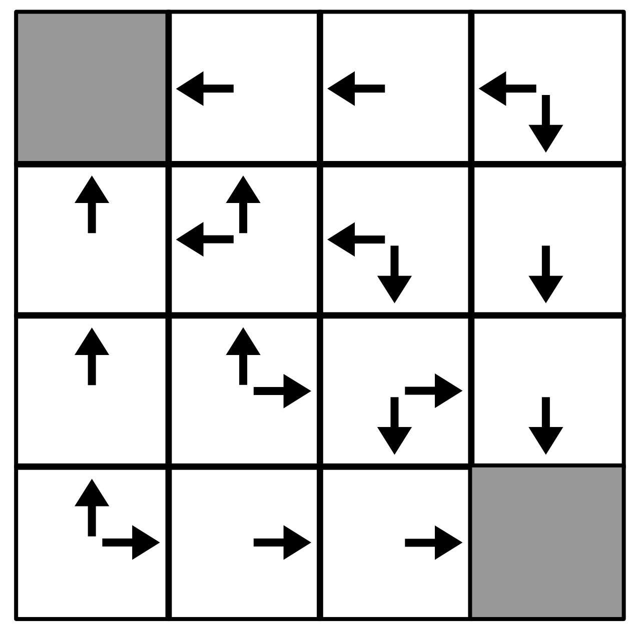
k=10

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0



k=∞

0.0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0.0



Deep Q-Learning

- Represent value function by deep Q-network with weights w

$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

- Define objective function by mean-squared error in Q-values

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

- Leading to the following Q-learning gradient

$$\frac{\partial \mathcal{L}(w)}{\partial w} = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w)\right) \frac{\partial Q(s, a, w)}{\partial w}\right]$$

- Optimise objective end-to-end by SGD, using $\frac{\partial L(w)}{\partial w}$

Stability Issues with Deep RL

Naive Q-learning oscillates or diverges with neural nets

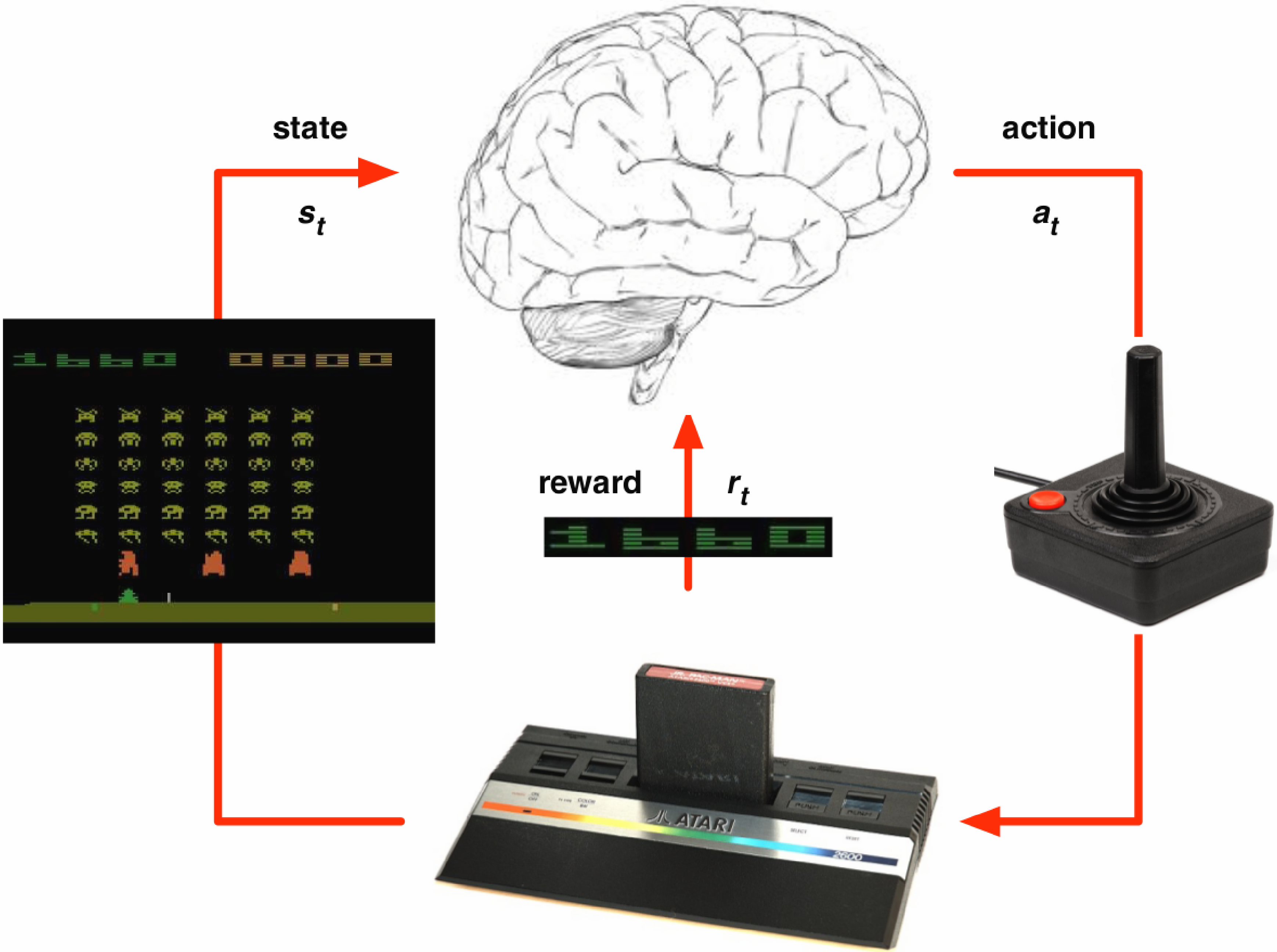
1. Data is sequential
   - Successive samples are correlated, non-iid
2. Policy changes rapidly with slight changes to Q-values
   - Policy may oscillate
   - Distribution of data can swing from one extreme to another
3. Scale of rewards and Q-values is unknown
   - Naive Q-learning gradients can be large
     unstable when backpropagated

Deep Q-Networks

DQN provides a stable solution to deep value-based RL

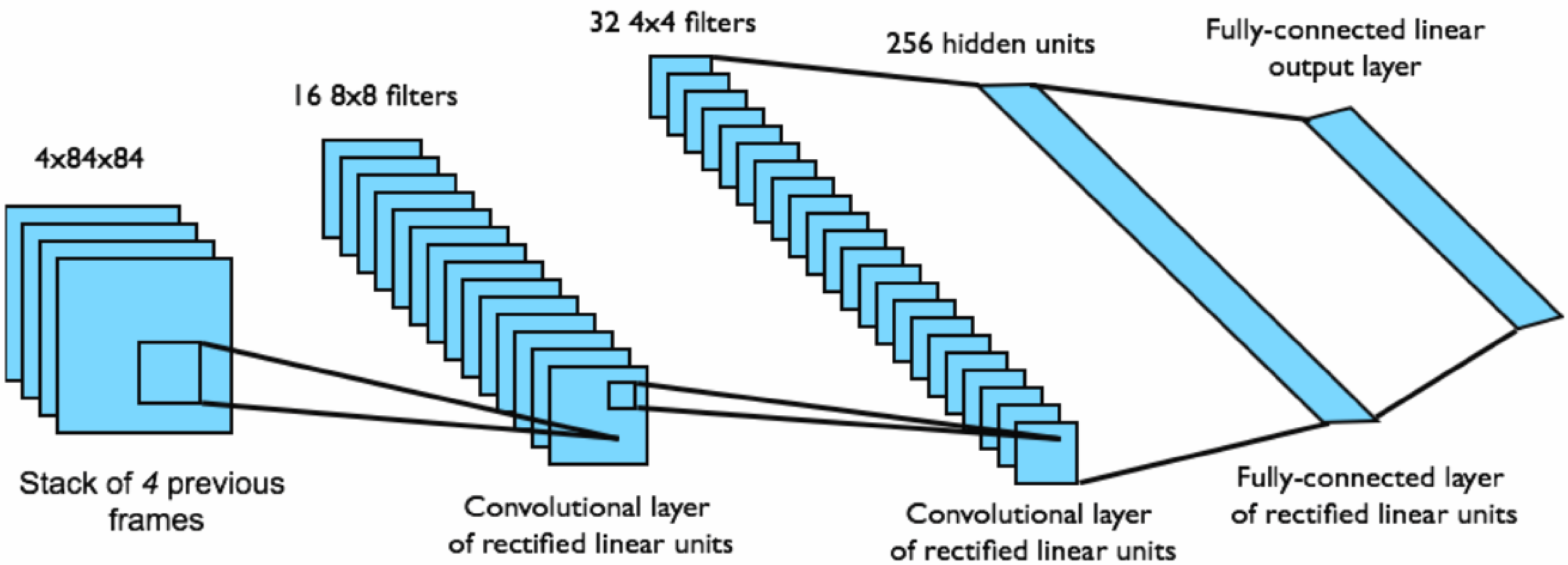
1. Use experience replay
   - Break correlations in data, bring us back to iid setting
   - Learn from all past policies
2. Freeze target Q-network
   - Avoid oscillations
   - Break correlations between Q-network and target
3. Clip rewards or normalize network adaptively to sensible range
   - Robust gradients

Reinforcement Learning in Atari



DQN in Atari

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



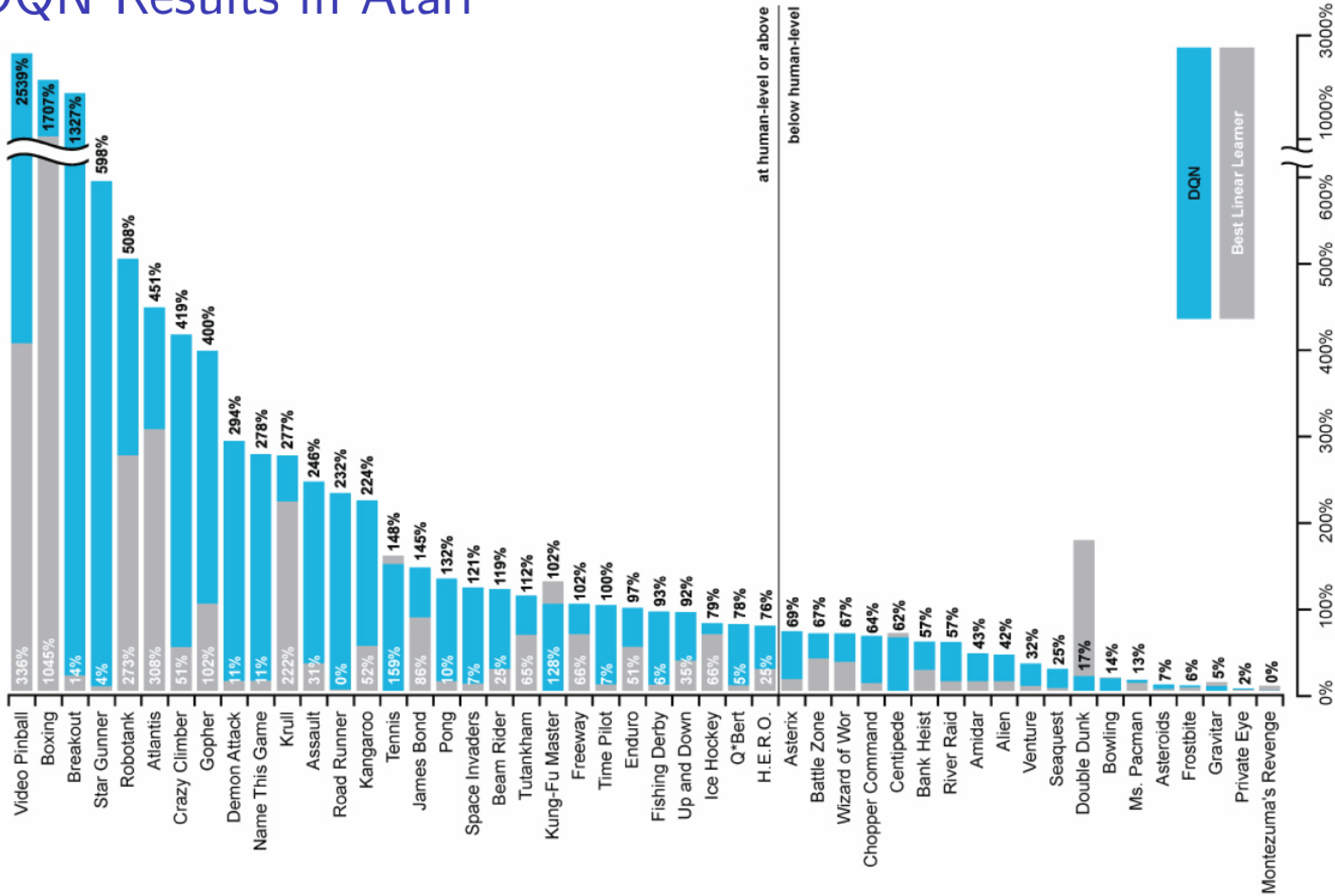
Network architecture and hyperparameters fixed across all games

[Mnih et al.]

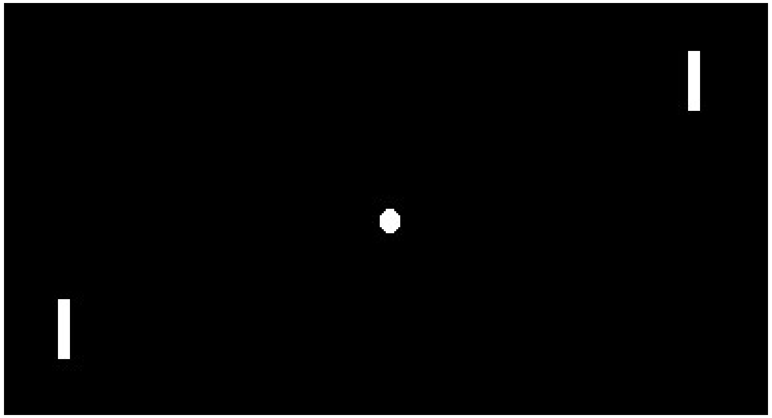
Space Invaders

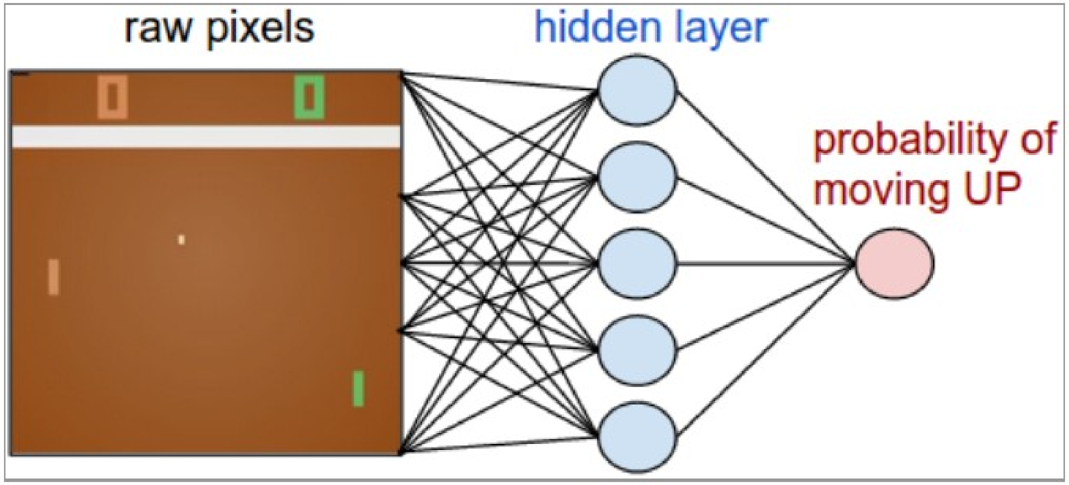
DQN controls the green laser cannon to clear columns of space invaders descending from the sky and also destroys two pink motherships at the top of the screen

DQN Results in Atari



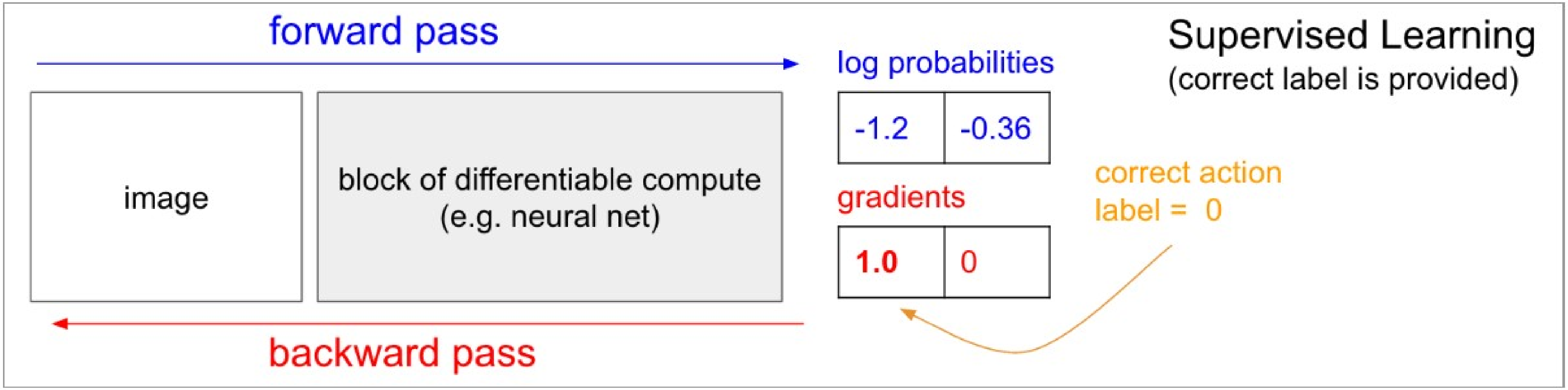
Policy Gradients

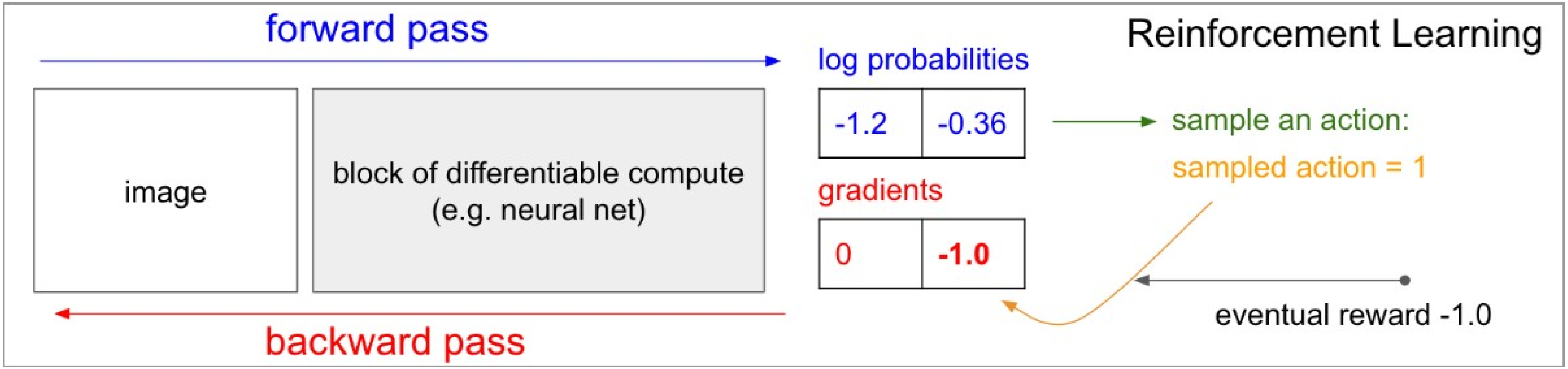




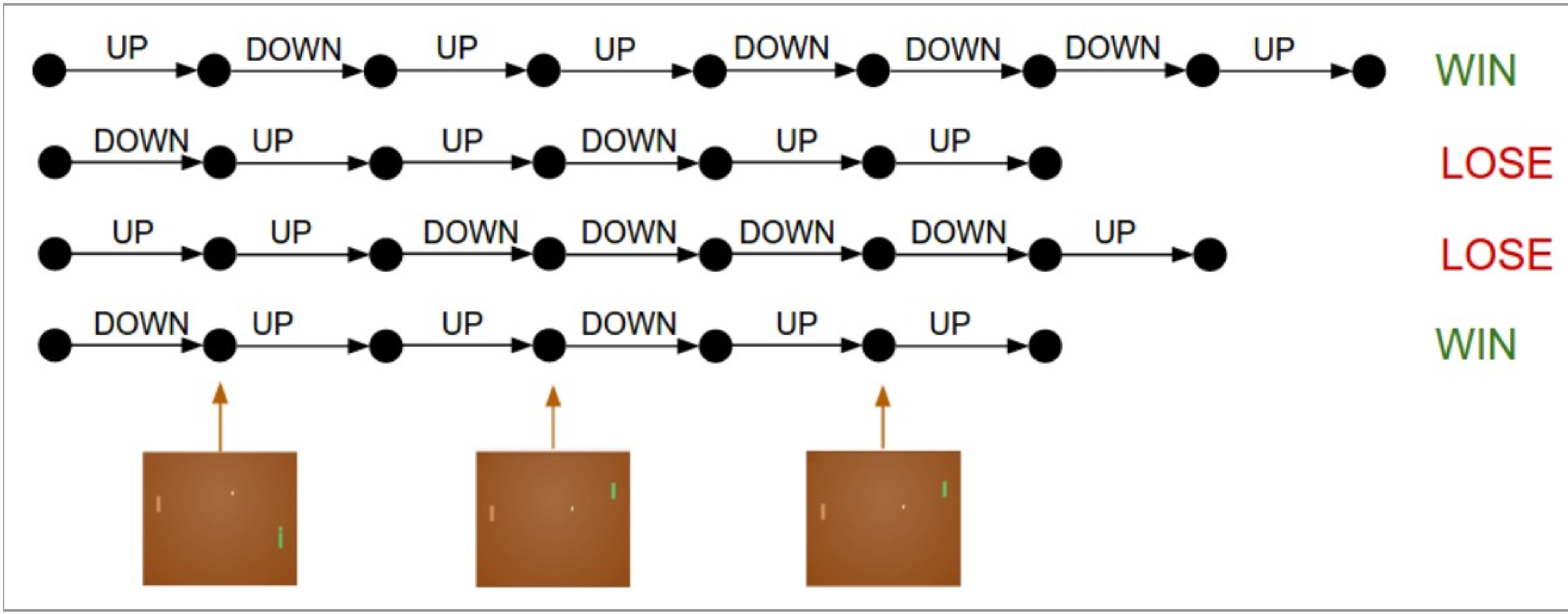
Our policy network is a 2-layer fully-connected net.

```
h = np.dot(W1, x) # compute hidden layer neuron activations
h[h<0] = 0 # ReLU nonlinearity: threshold at zero
logp = np.dot(W2, h) # compute log probability of going up
p = 1.0 / (1.0 + np.exp(-logp)) # sigmoid function (gives probability of going up)
```


forward pass
log probabilities
Supervised Learning
(correct label is provided)
image
block of differentiable compute
(e.g. neural net)
-1.2
-0.36
correct action
label = 0
gradients
1.0
0
backward pass


forward pass
Reinforcement Learning
log probabilities
image
block of differentiable compute
(e.g. neural net)
-1.2
-0.36
sample an action:
sampled action = 1
gradients
0
-1.0
eventual reward -1.0
backward pass

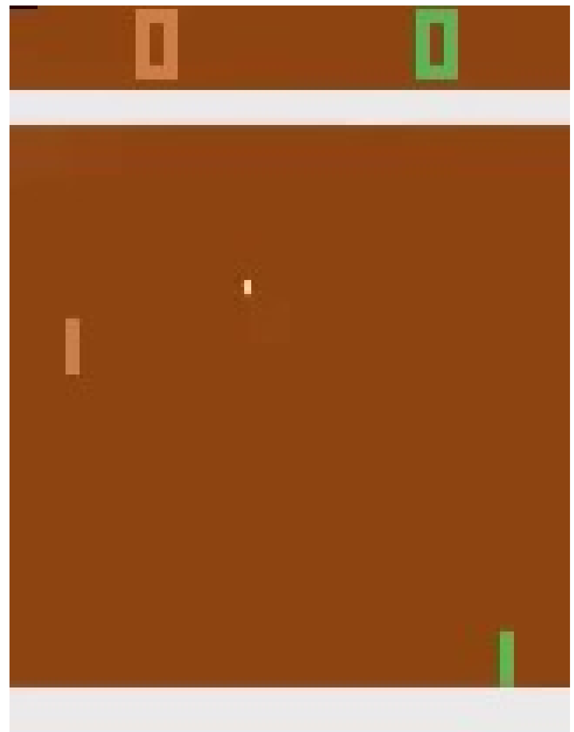
> *Policy Gradients: Run a policy for a while. See what actions led to high rewards. Increase their probability.*

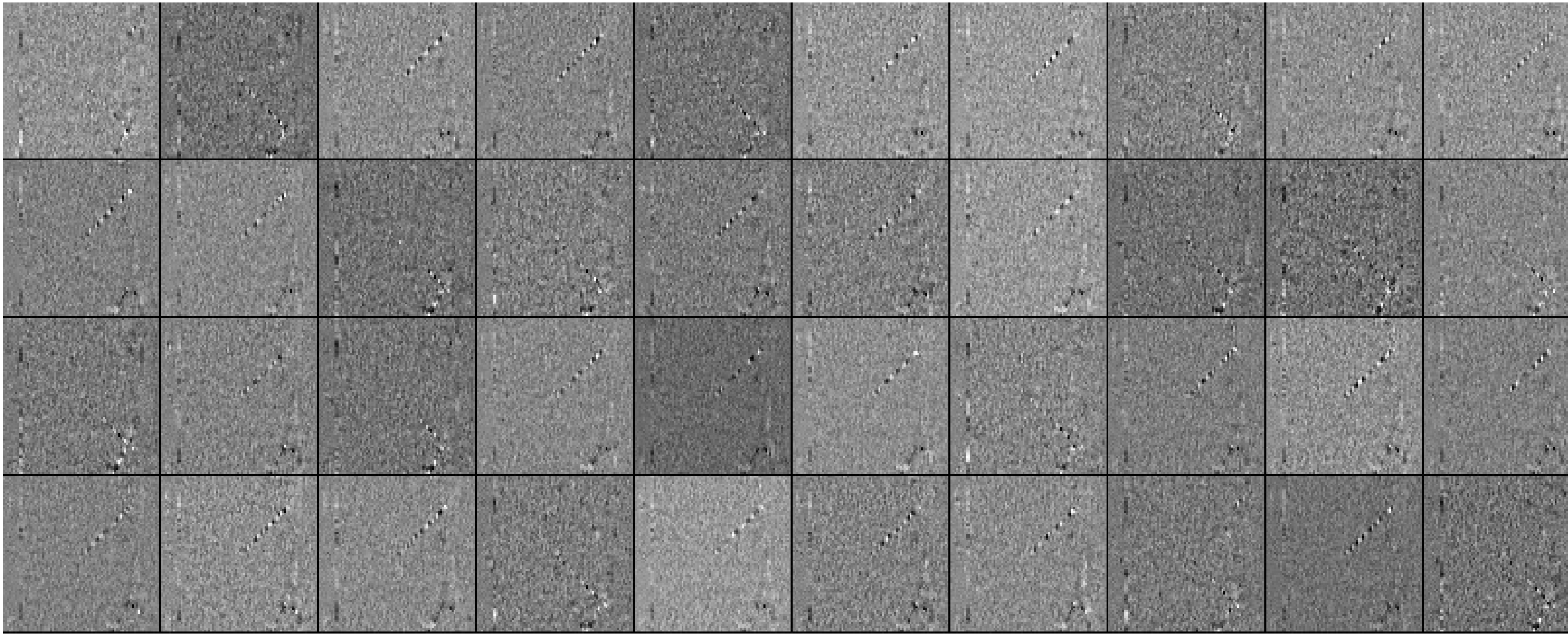


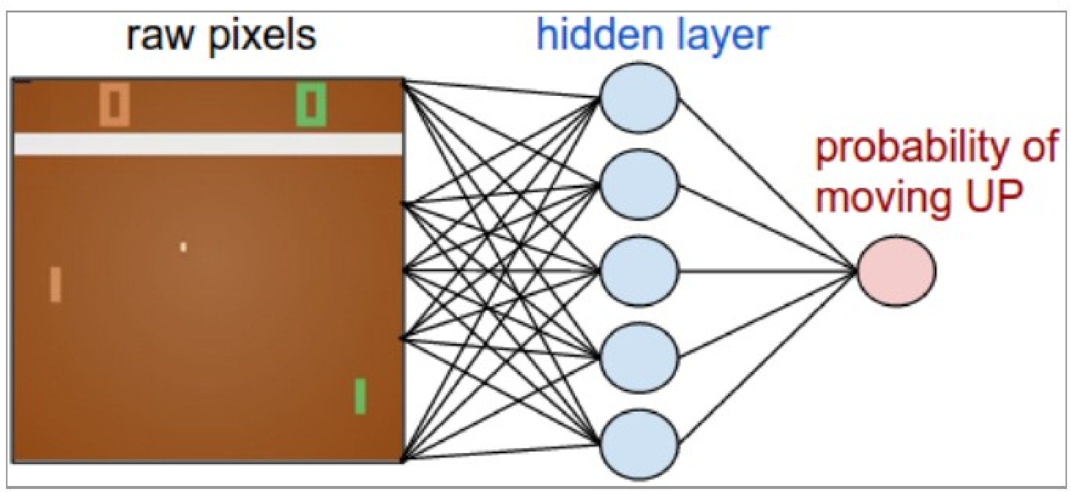
Cartoon diagram of 4 games. Each black circle is some game state (three example states are visualized on the bottom), and each arrow is a transition, annotated with the action that was sampled. In this case we won 2 games and lost 2 games. With Policy Gradients we would take the two games we won and slightly encourage every single action we made in that episode. Conversely, we would also take the two games we lost and slightly discourage every single action we made in that episode.

Deriving Policy Gradients. I'd like to also give a sketch of where Policy Gradients come from mathematically. Policy Gradients are a special case of a more general *score function gradient estimator*. The general case is that when we have an expression of the form $E_{x \sim p(x|\theta)}[f(x)]$ - i.e. the expectation of some scalar valued score function $f(x)$ under some probability distribution $p(x;\theta)$ parameterized by some θ. Hint hint, $f(x)$ will become our reward function (or advantage function more generally) and $p(x)$ will be our policy network, which is really a model for $p(a \mid I)$, giving a distribution over actions for any image I. Then we are interested in finding how we should shift the distribution (through its parameters θ) to increase the scores of its samples, as judged by f (i.e. how do we change the network's parameters so that action samples get higher rewards). We have that:

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


0
0





Our policy network is a 2-layer fully-connected net.

On using PG in practice. As a last note, I'd like to do something I wish I had done in my RNN blog post. I think I may have given the impression that RNNs are magic and automatically do arbitrary sequential problems. The truth is that getting these models to work can be tricky, requires care and expertise, and in many cases could also be an overkill, where simpler methods could get you 90%+ of the way there. The same goes for Policy Gradients. They are not automatic: You need a lot of samples, it trains forever, it is difficult to debug when it doesn't work. One should always try a BB gun before reaching for the Bazooka. In the case of Reinforcement Learning for example, one strong baseline that should always be tried first is the cross-entropy method (CEM), a simple stochastic hill-climbing "guess and check" approach inspired loosely by evolution. And if you insist on trying out Policy Gradients for your problem make sure you pay close attention to the *tricks* section in papers, start simple first, and use a variation of PG called TRPO, which almost always works better and more consistently than vanilla PG in practice. The core idea is to avoid parameter updates that change your policy too much, as enforced by a constraint on the KL divergence between the distributions predicted by the old and the new policy on a batch of data (instead of conjugate gradients the simplest instantiation of this idea could be implemented by doing a line search and checking the KL along the way).