What every designer & developer should know about

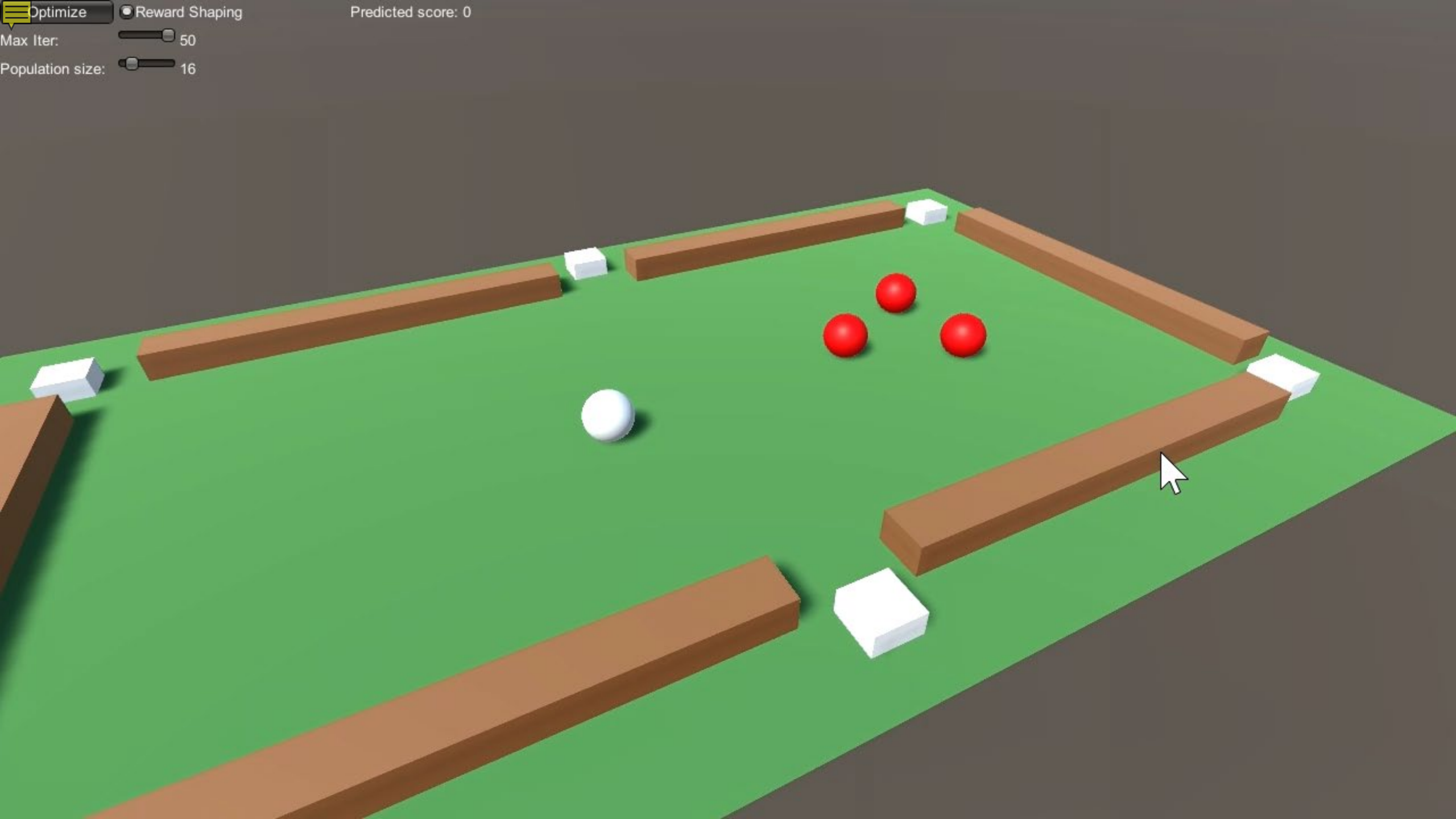
Mathematical Optimization

Intelligent Computational Media, spring 2021

Aalto University

Prof. Perttu Hämäläinen

perttu.hamalainen@aalto.fi


Optimize
Reward Shaping
Predicted score: 0
Max Iter:
50
Population size:
16

Contents

- Optimization: the what and why
- Some intuitions from simple random search
- Gradient-based optimization
- Sampling-based optimization
- From continuous-valued to discrete optimization
- Reinforcement learning
- Forward search methods, e.g., Monte Carlo Tree Search
- Appendix: Optional material like neuroevolution

Optimization: the what and why

What is mathematical optimization?

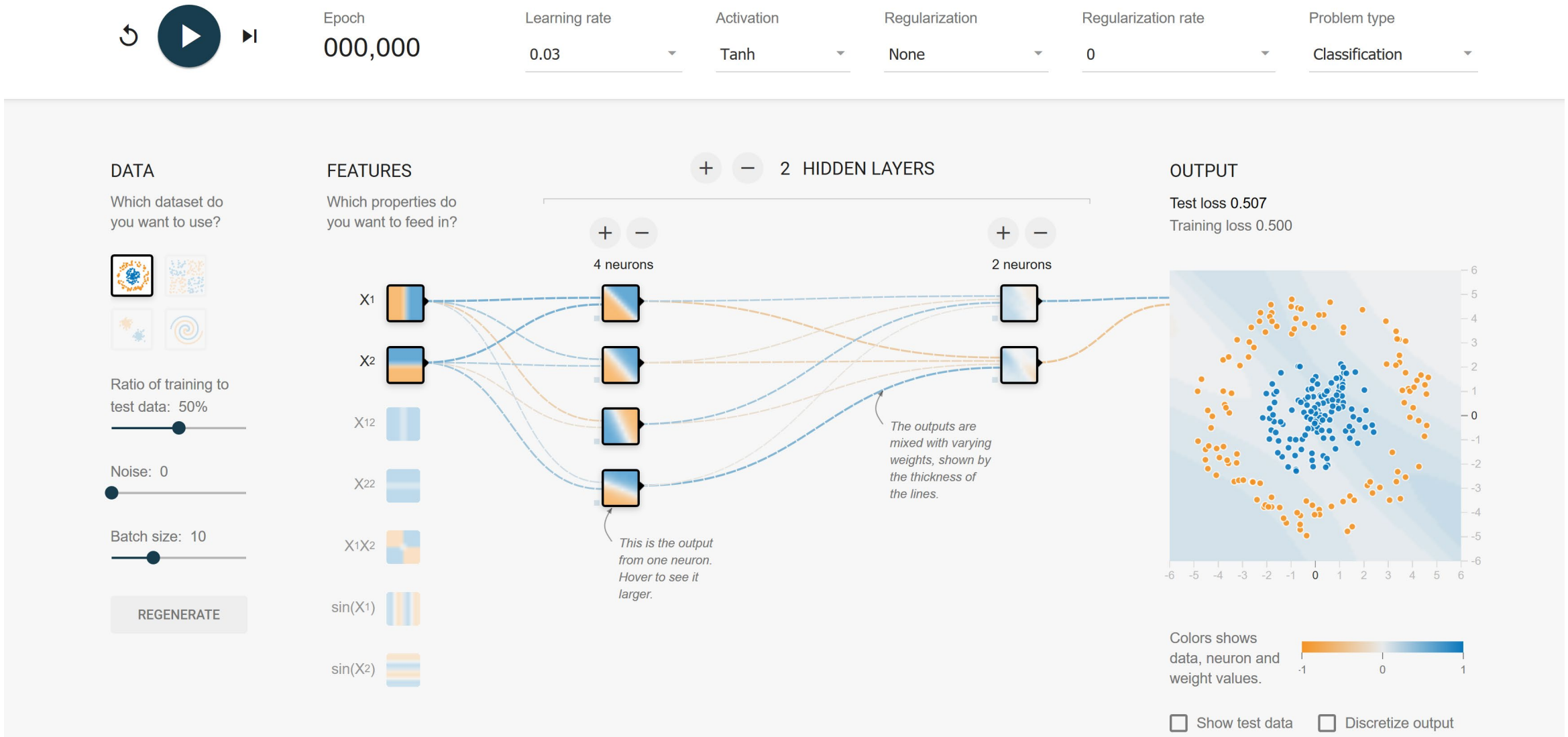
- Optimizing code is just one optimization problem
- In general: find parameters **x** that minimize or maximize some objective function $f(\mathbf{x})$
- We denote vectors of variables with boldface, i.e., $\mathbf{x}=[x_1,x_2,\ldots x_N]$

Why does it matter?

- Game Design is optimization: For example, maximize *enjoyment*(**x**)
- A/B testing is a simple optimization method
- Game playing is (usually) optimization => optimization algorithms can be used for game playing and testing
- More generally, AI = problem solving = optimization (e.g., adjust neural network parameters to minimize some loss function, or find a gameplay strategy that maximizes the probability of winning)

Optimizing actions

Optimizing neural network parameters



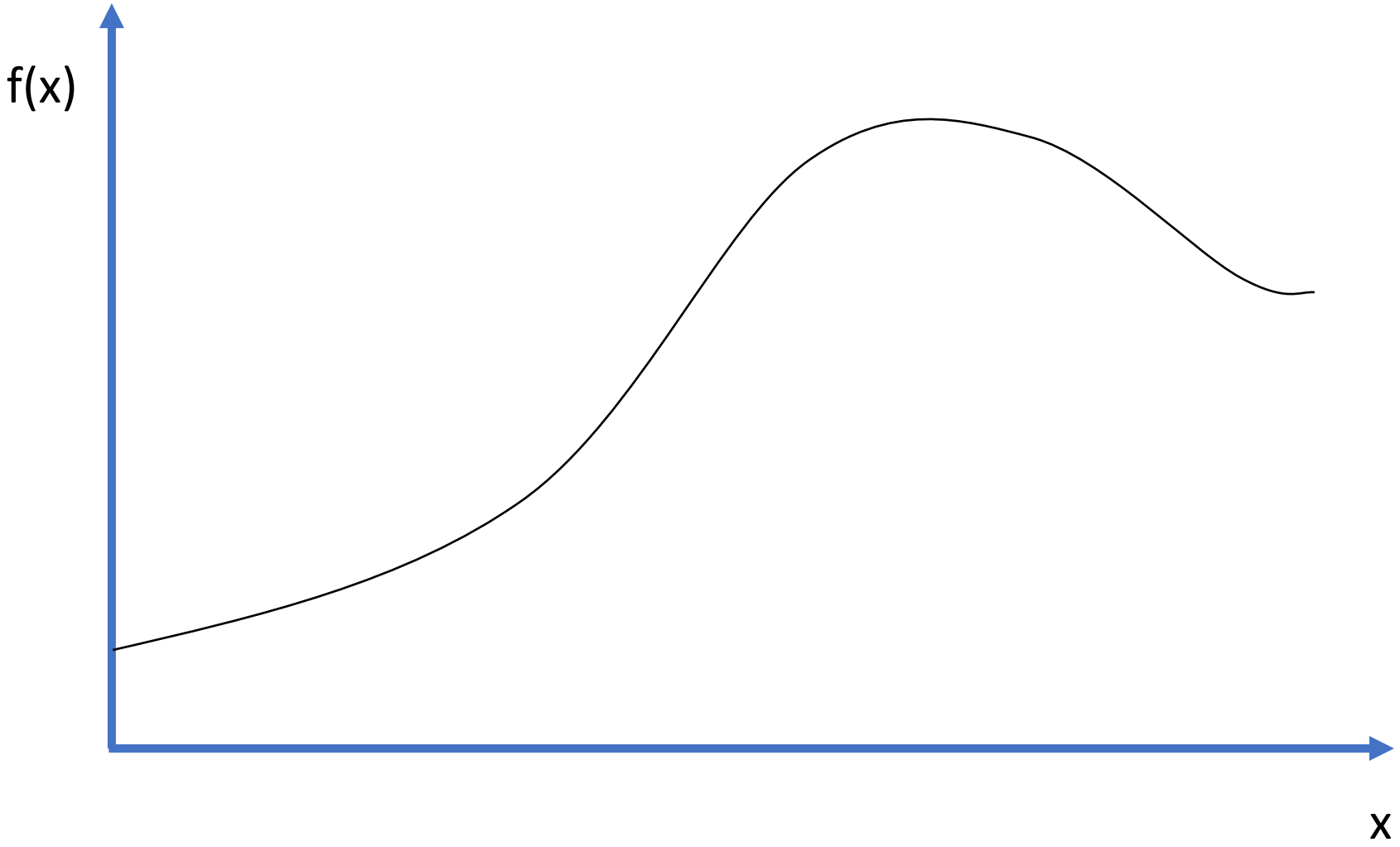
Simple random search

The simplest optimization procedure

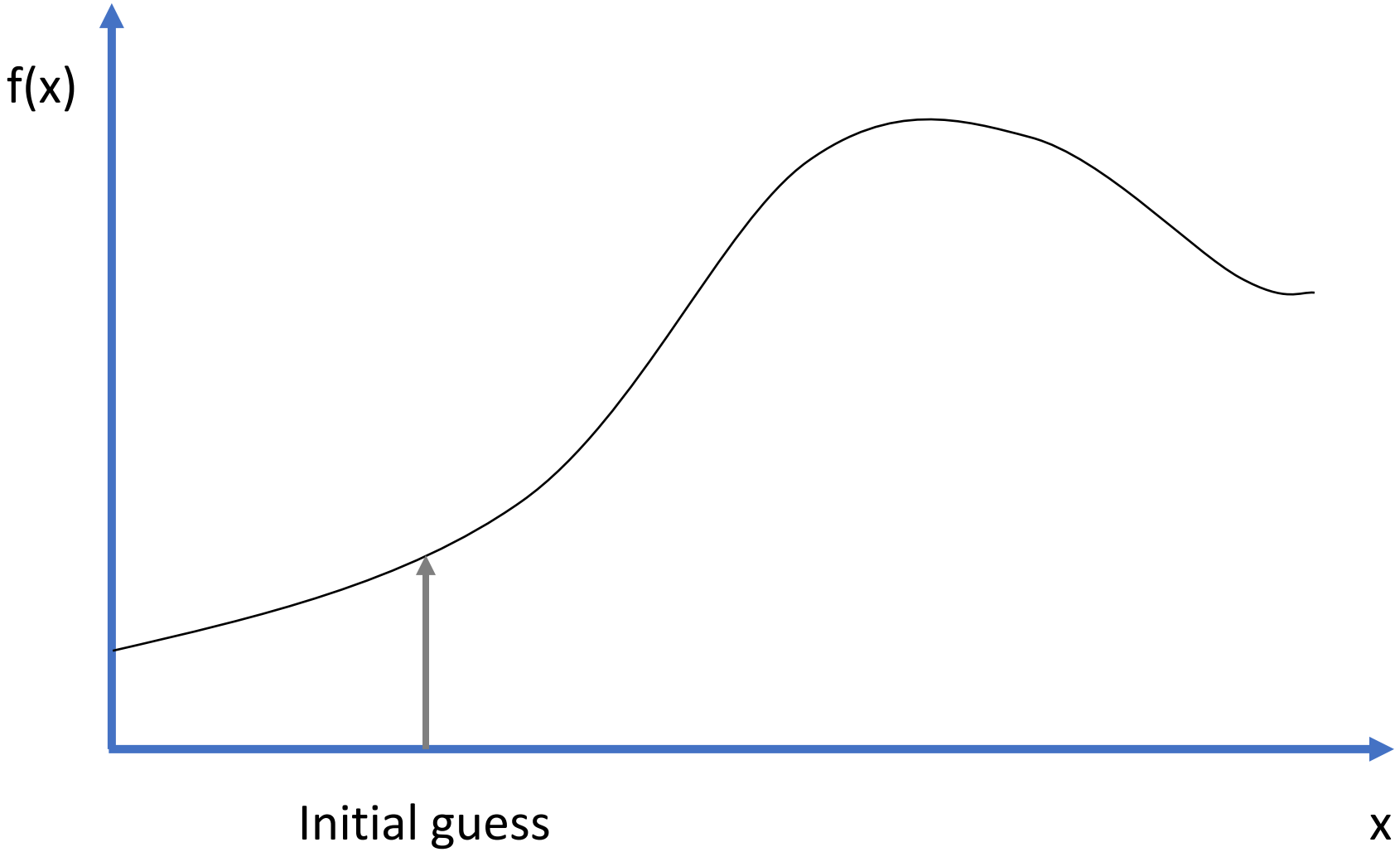
1. Initialize **x** randomly or based on some initial guess
2. Try some new $\mathbf{x}_{new}$ (typically near the current **x**)
3. If $f(\mathbf{x}_{new}) > f(\mathbf{x})$, set $\mathbf{x} = \mathbf{x}_{new}$ //assuming $f()$ is to be maximized
4. Repeat steps 2 & 3

This is similar to *simulated annealing* with specific parameters

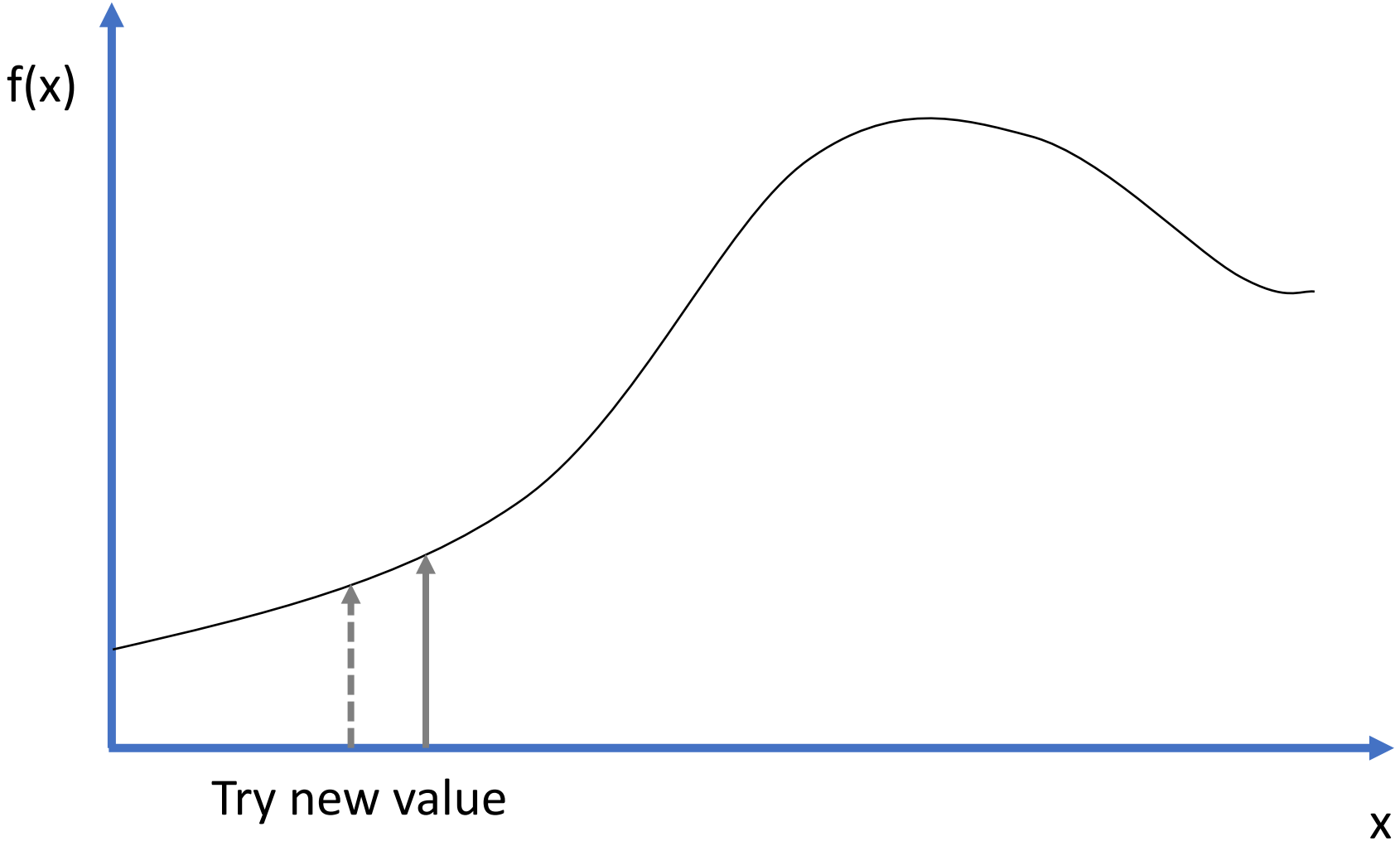
The simplest optimization procedure



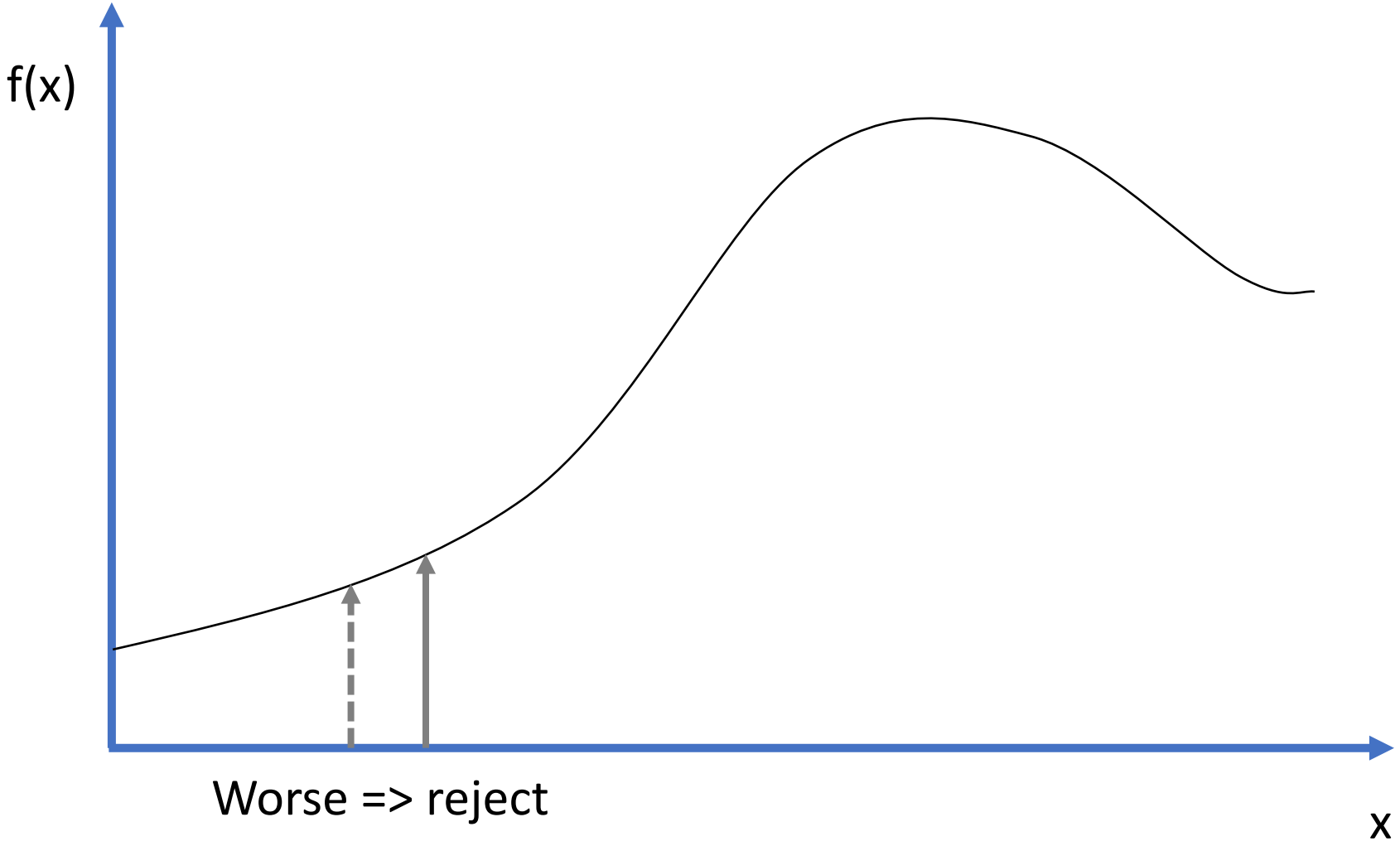
The simplest optimization procedure



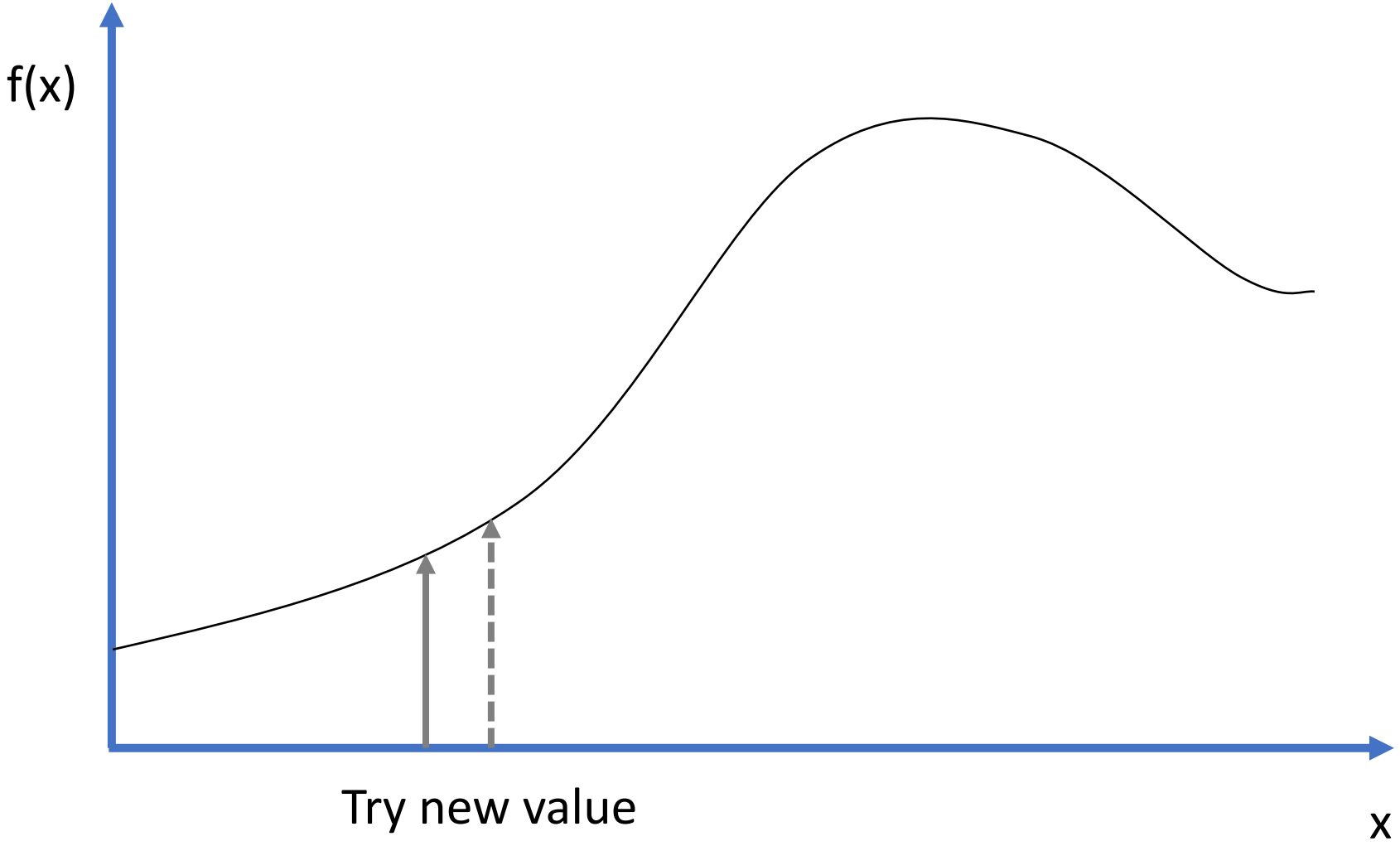
The simplest optimization procedure



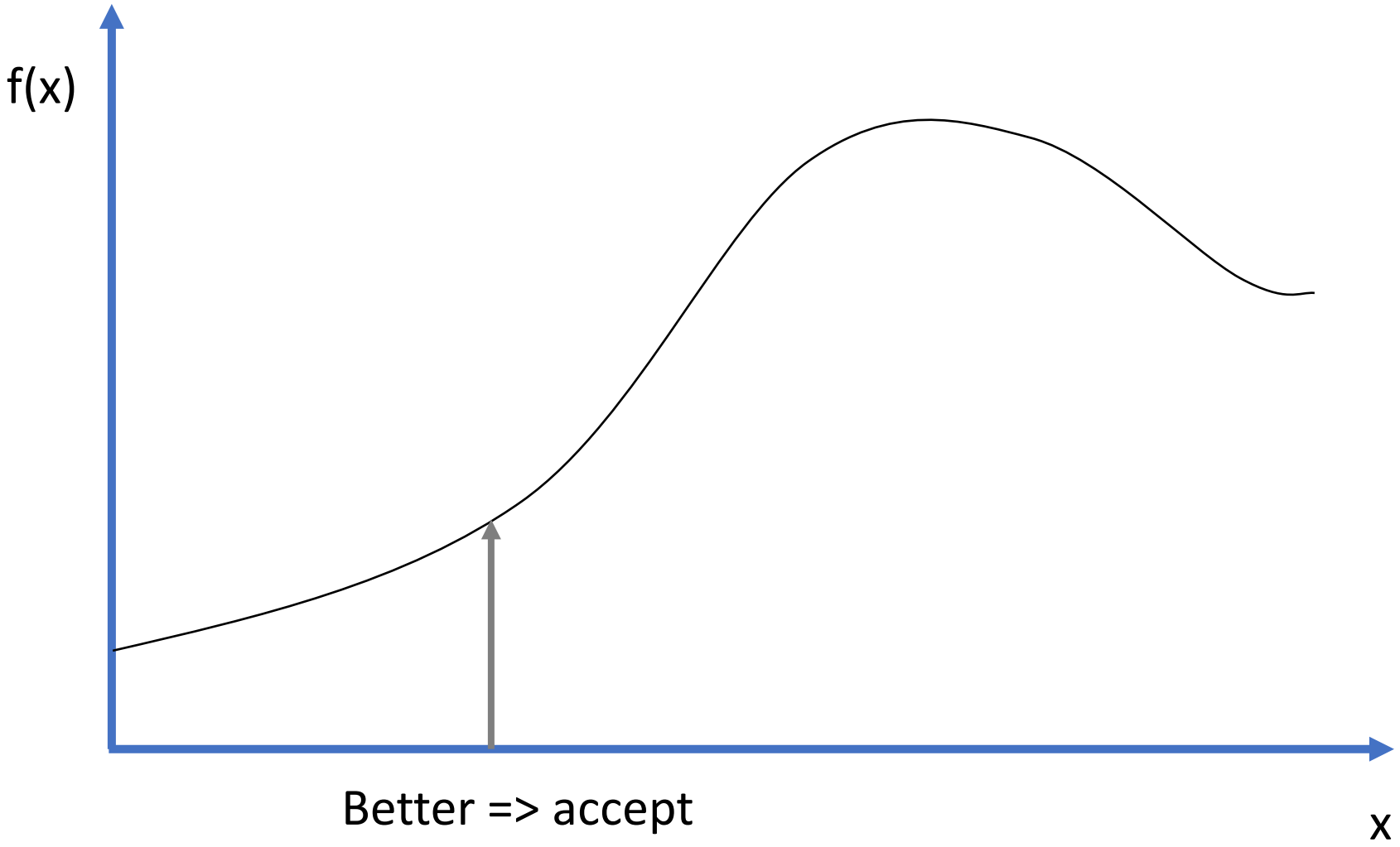
The simplest optimization procedure



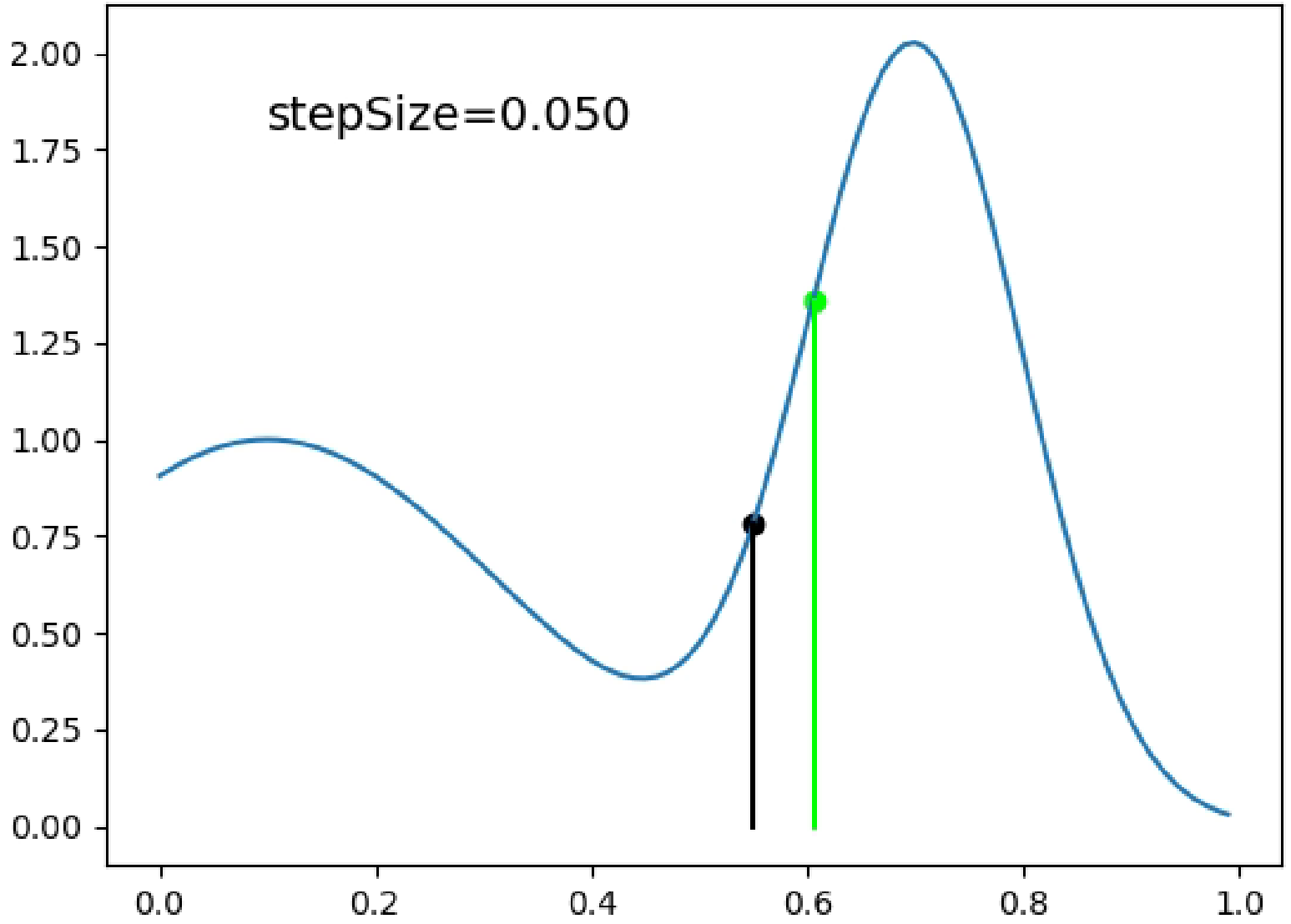
The simplest optimization procedure



The simplest optimization procedure



The simplest optimization procedure



A/B testing

- The simple optimization procedure where
  - **x** denotes some game design parameters (e.g., monetization strategy)
  - $f(\mathbf{x})$ is evaluated through deploying the game to some players, and measuring impact, e.g., monetization
  - $\mathbf{x}_{new}$ is selected by a human designer based on some hypotheses of player preferences and behavior (typically more efficient than algorithms that don't understand player psychology)

The simplest optimization procedure

1. Initialize **x** randomly or based on some initial guess
2. Try some new $\mathbf{x}_{new}$ (typically near the current **x**)
3. If $f(\mathbf{x}_{new}) > f(\mathbf{x})$, set $\mathbf{x} = \mathbf{x}_{new}$ //assuming $f()$ is to be maximized
4. Repeat steps 2 & 3

The simplest optimization procedure

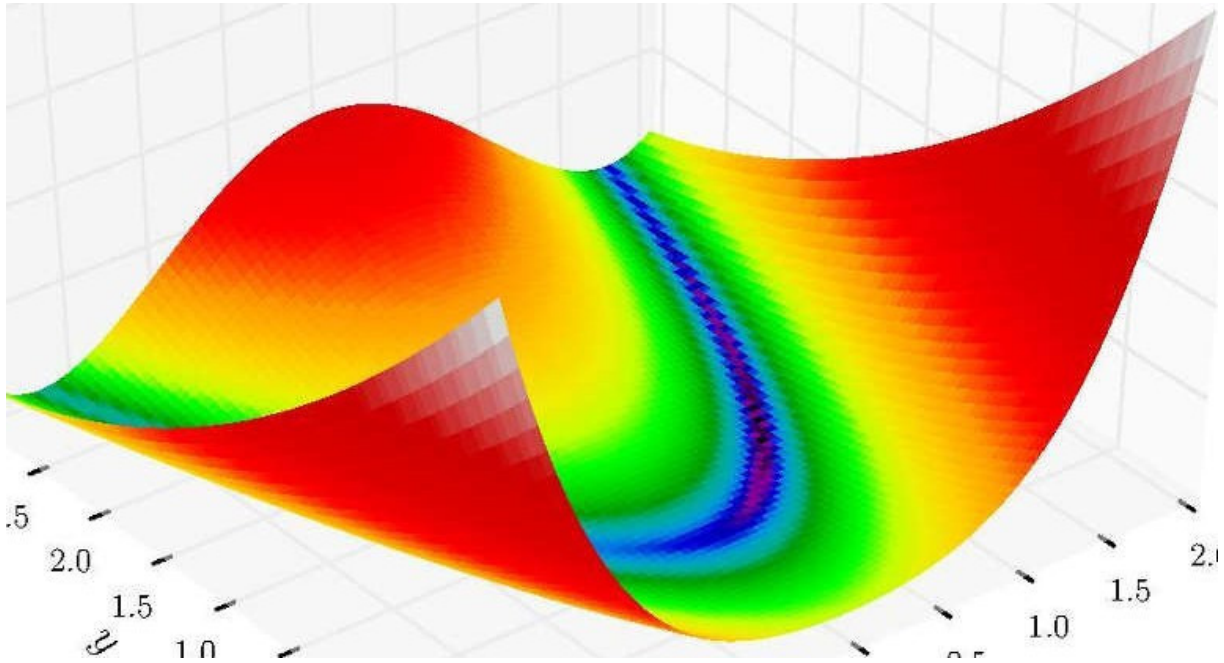
1. Initialize **x** randomly or based on some initial guess
2. Try some new $\mathbf{x}_{new}$ (typically near the current **x**)
3. If $f(\mathbf{x}_{new}) > f(\mathbf{x})$, set $\mathbf{x} = \mathbf{x}_{new}$ //assuming $f()$ is to be maximized
4. Repeat steps 2 & 3

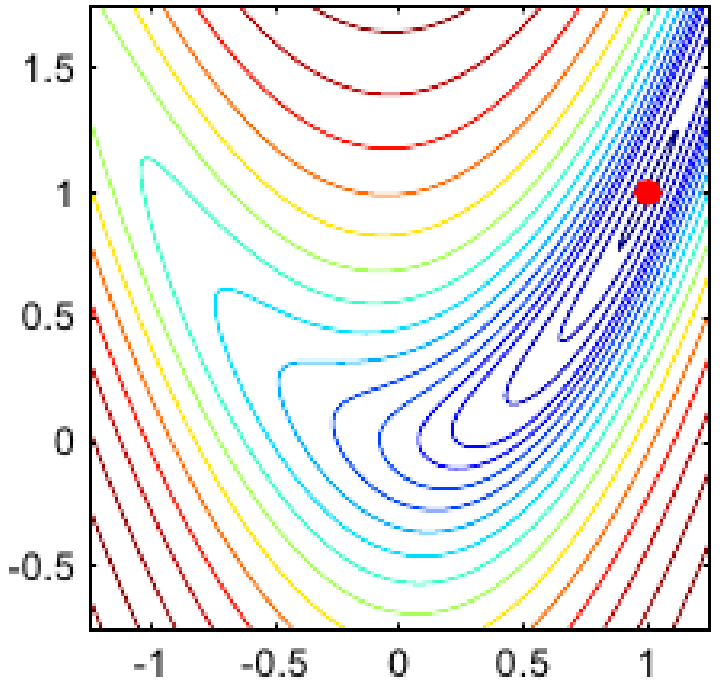
Modifications:

- Generic: How to select $\mathbf{x}_{new}$?
- Generic: How to adjust algorithm parameters?
- Problem-specific: How to initialize?
- Problem-specific: How to modify $f(\mathbf{x})$ or parameterize **x** such that optimization is easier?

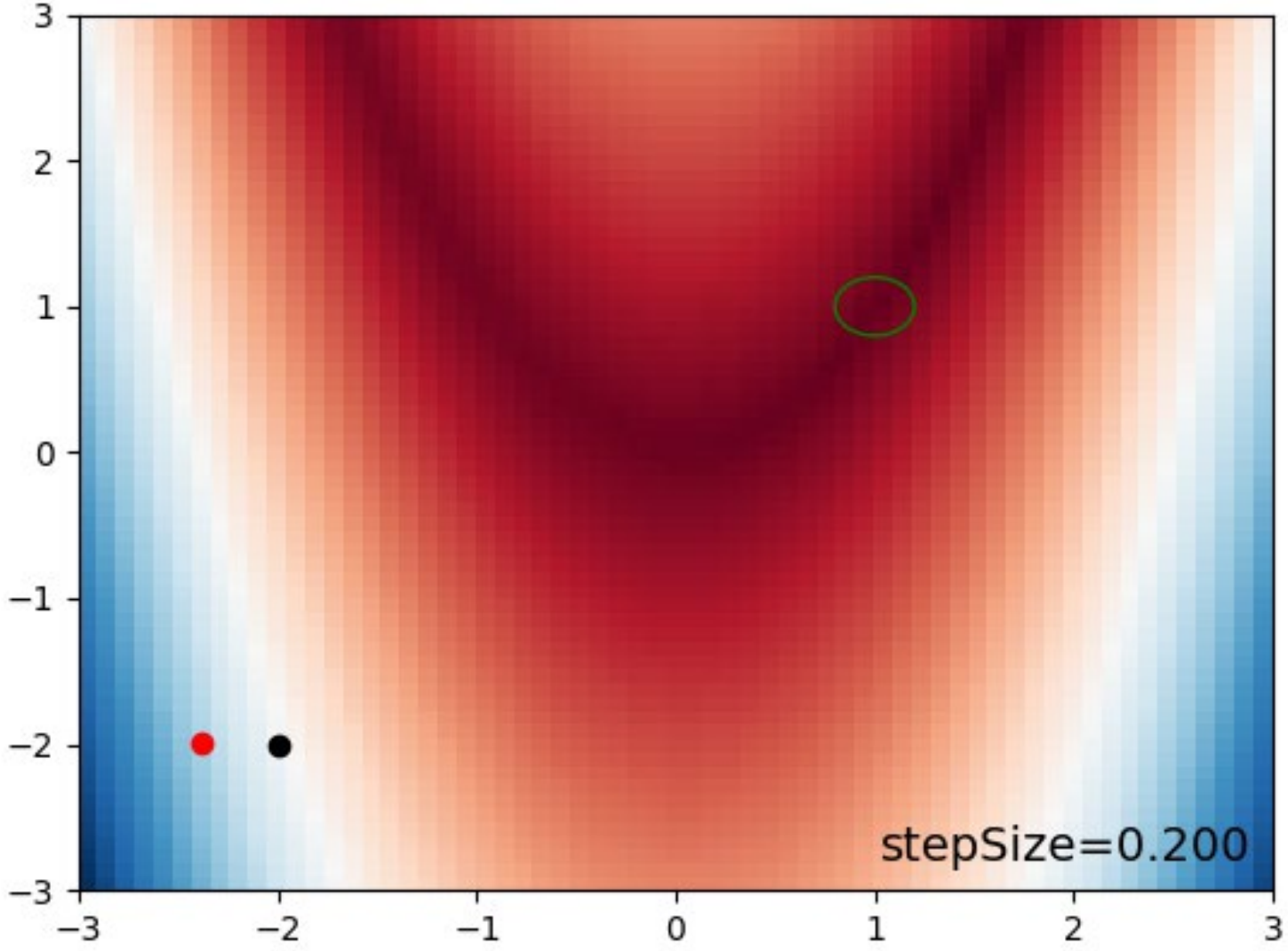
Curse of dimensionality

- Problem: in high-dimensional problems, random search is bad at finding the direction of improvements
- A good test function: Rosenbrock





Progress slows down in the Rosenbrock valley



Curse of dimensionality

- Problem: in high-dimensional problems, random search is bad at finding the direction of improvements
- 2D Rosenbrock: a narrow valley
- 3D Rosenbrock: a narrow ”tunnel”
- The search space grows exponentially with dimensionality! (e.g., if one tries even just 2 values for each x_1, x_2, ..., x_N, there’s 2^N possible combinations.

Contents

- Optimization: the what and why
- Some intuitions from simple random search
- **Gradient-based optimization**
- Population-based optimization
- Approach: A visual introduction. Some math included for self-study, but will be skipped to save time.

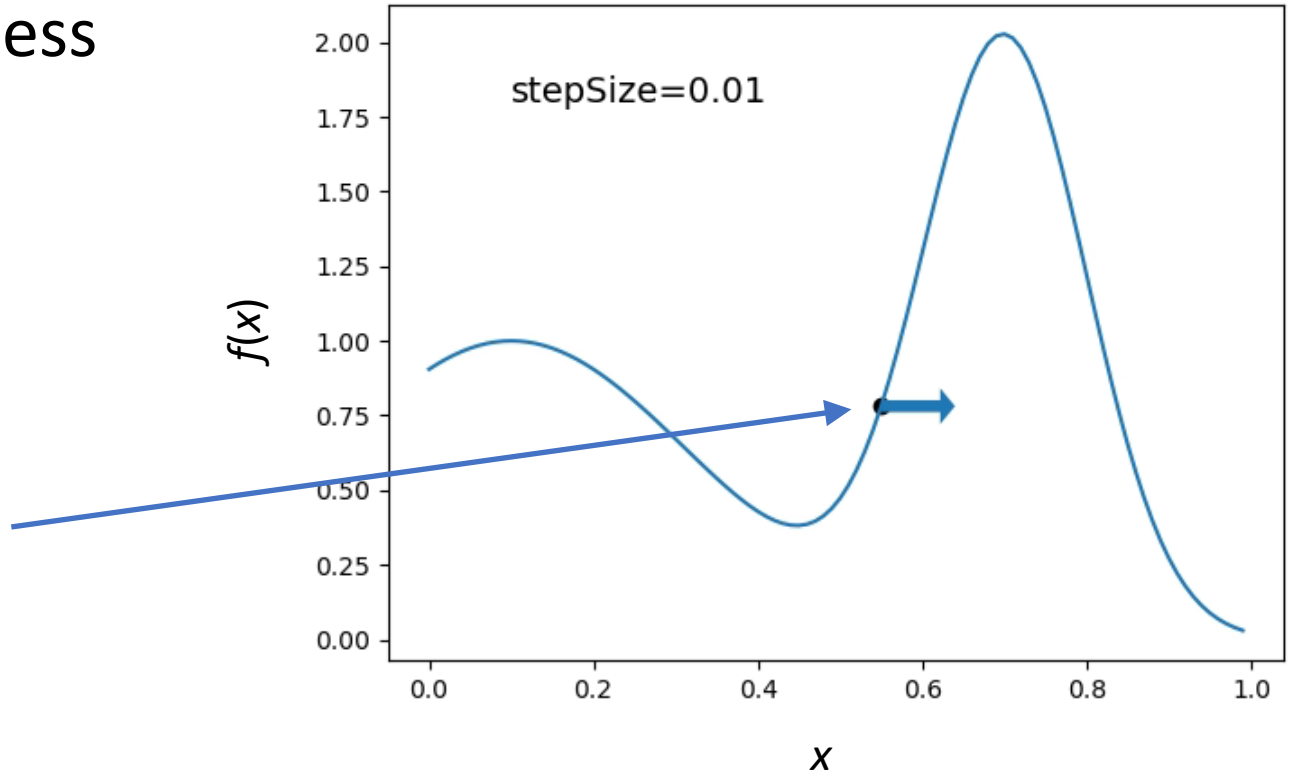
Gradient-based optimization

Utilizing gradient information

- Finding the direction of improvement => use *gradient* information
- Gradient is *the multivariate generalization of the derivative*
- A vector that points to direction of steepest ascent of f(**x**) in the space of **x**. Denoted ∇f(**x**).
- Length proportional to steepness

Gradient vector:

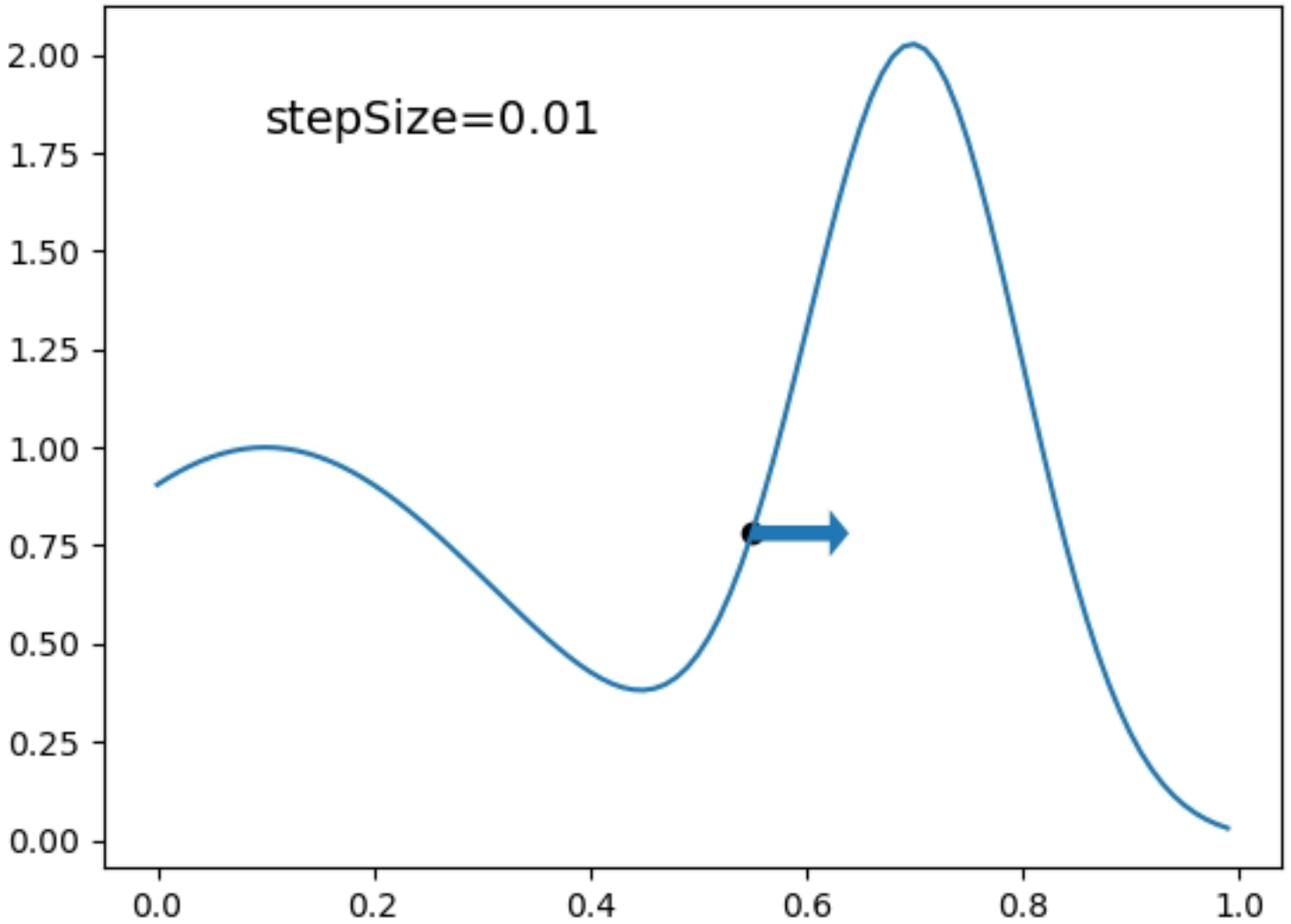
To make *f*(*x*) larger,
move *x* to the right



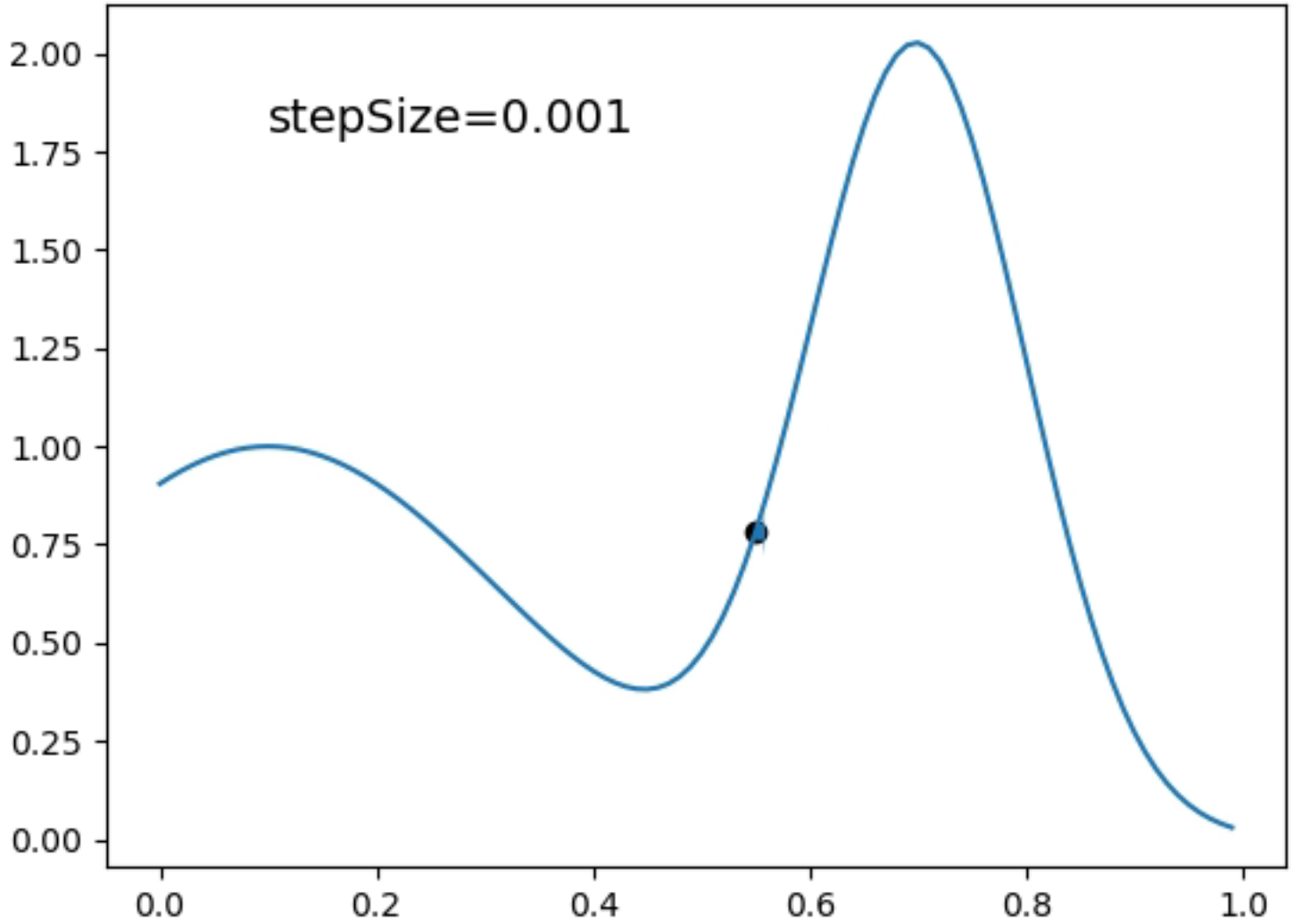
Computing the gradient

- Analytically (TensorFlow, PyTorch etc. do this automatically if your objective function math is implemented using the provided operations)
- Finite differences: measure the change of $f(\mathbf{x})$ when $\mathbf{x}$ perturbed along each coordinate axis
- $g_i(\mathbf{x}) = [f(\mathbf{x}) - f(\mathbf{x}+k\mathbf{u}_i)] \,/\, k$, where $\mathbf{u}_i$ is a unit vector pointing along the i:th coordinate axis, k is a small number
- For N dimensions, this requires N+1 evaluations of $f(\mathbf{x})$

Gradient ascend: $\mathbf{x}_{new} = \mathbf{x} + stepSize * \nabla f(\mathbf{x})$



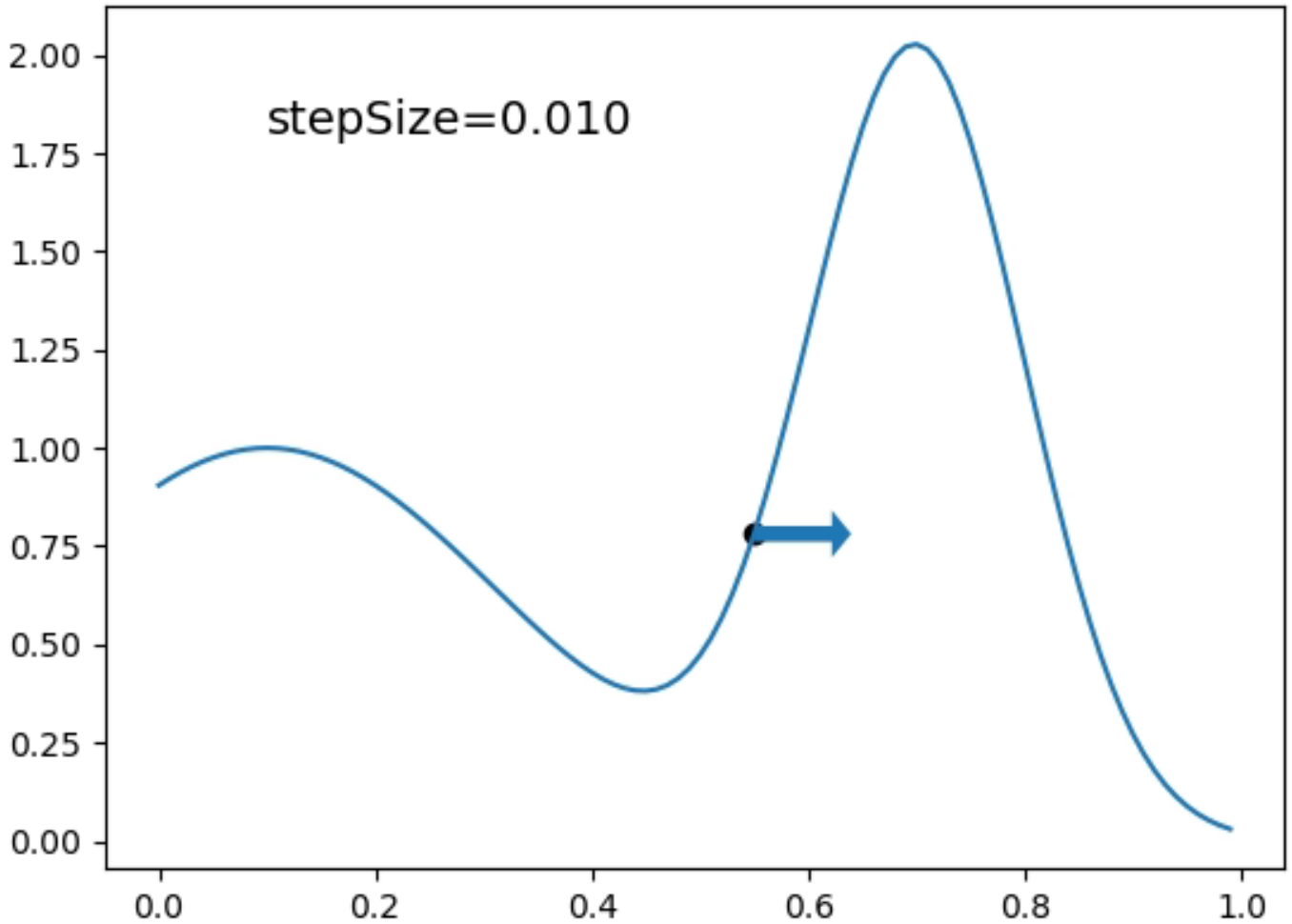
Smaller stepsize: no oscillations, but slow convergence



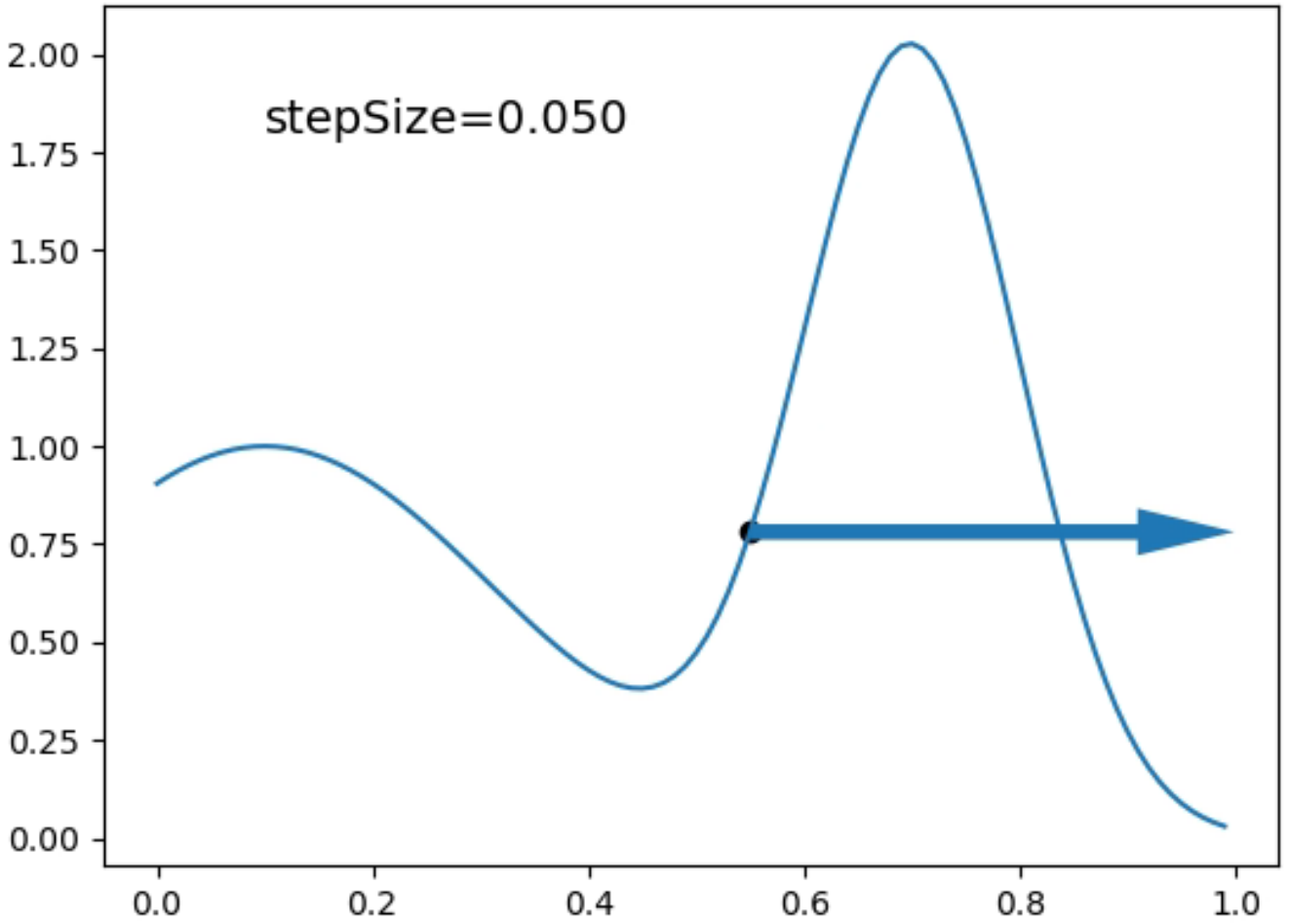
Speeding up convergence

- Step size decay
- Gradient clipping
- Momentum

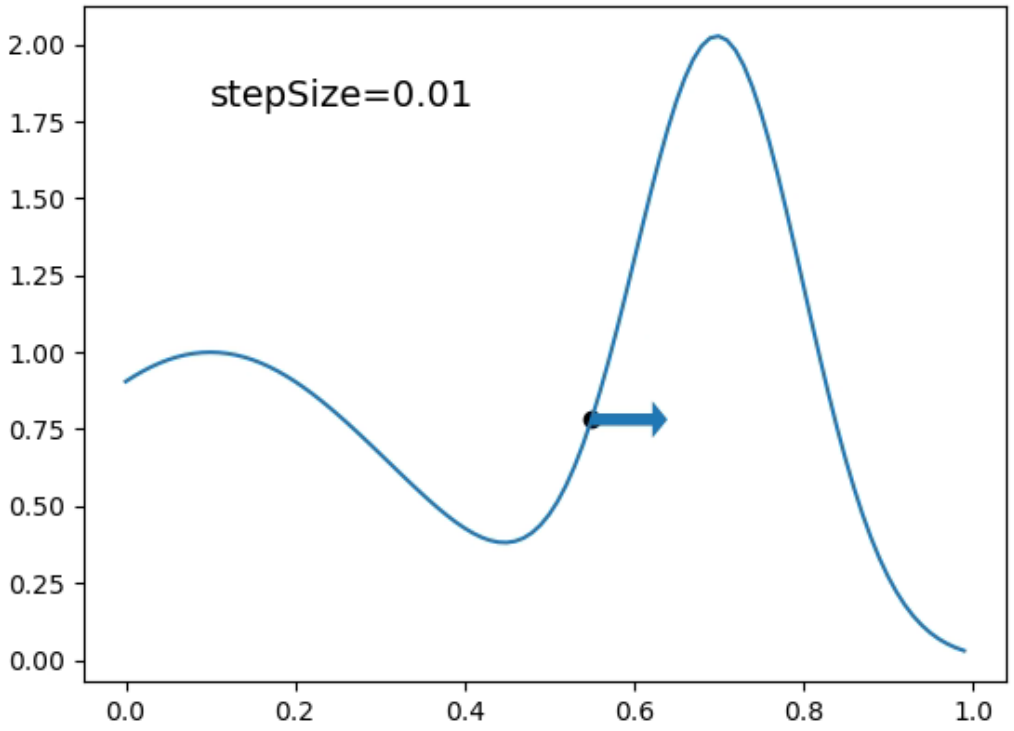
Step size decay

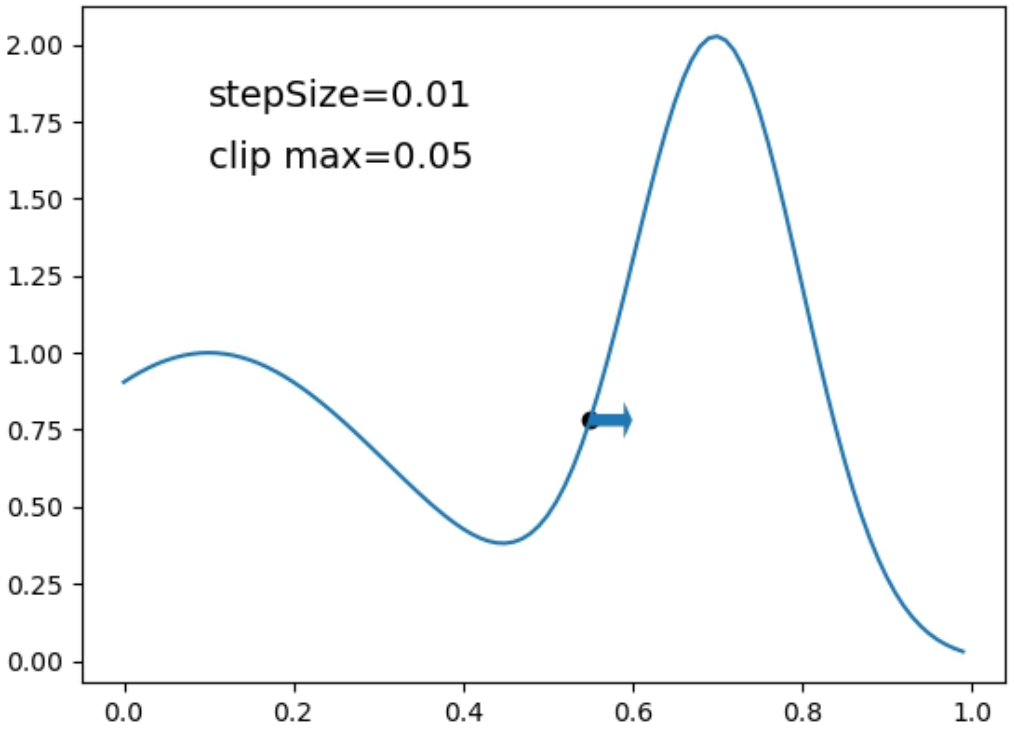


Step size decav



Gradient clipping

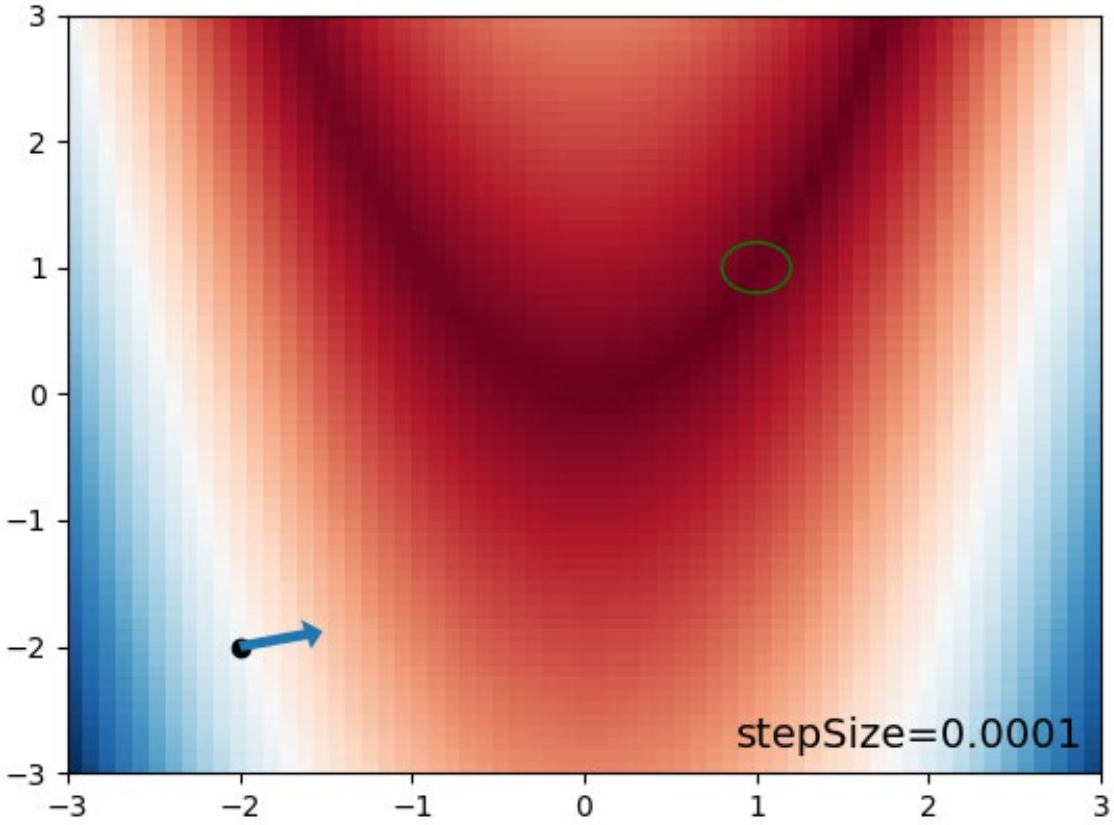




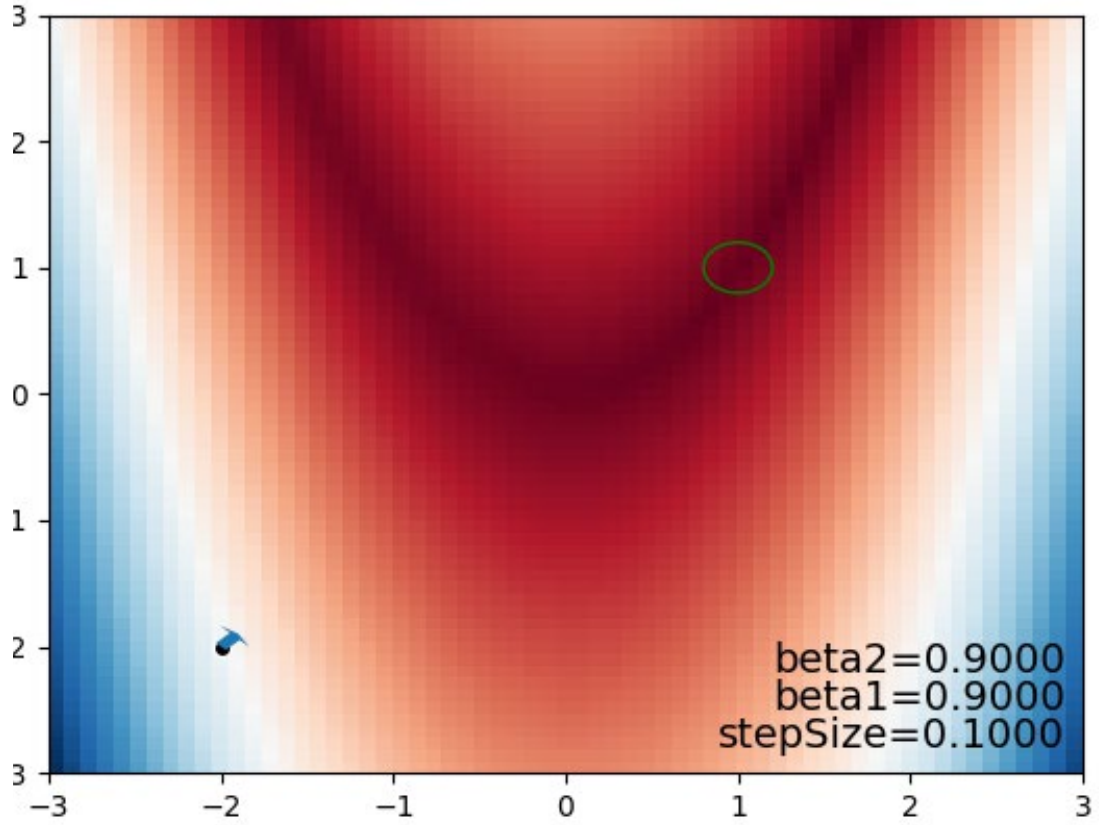
Momentum

- Gradient interpreted as acceleration instead of velocity
- $\mathbf{v}$ = momentum*$\mathbf{v}$ + $\nabla f(\mathbf{x})$
- $\mathbf{x}_{new}$ = -stepSize * $\mathbf{v}$
- 1-momentum = friction, damping of velocity between updates
- Can be used with gradient clipping
- Adam is a modern momentum-based method with some clever additional tricks. A good default choice for optimizing neural networks

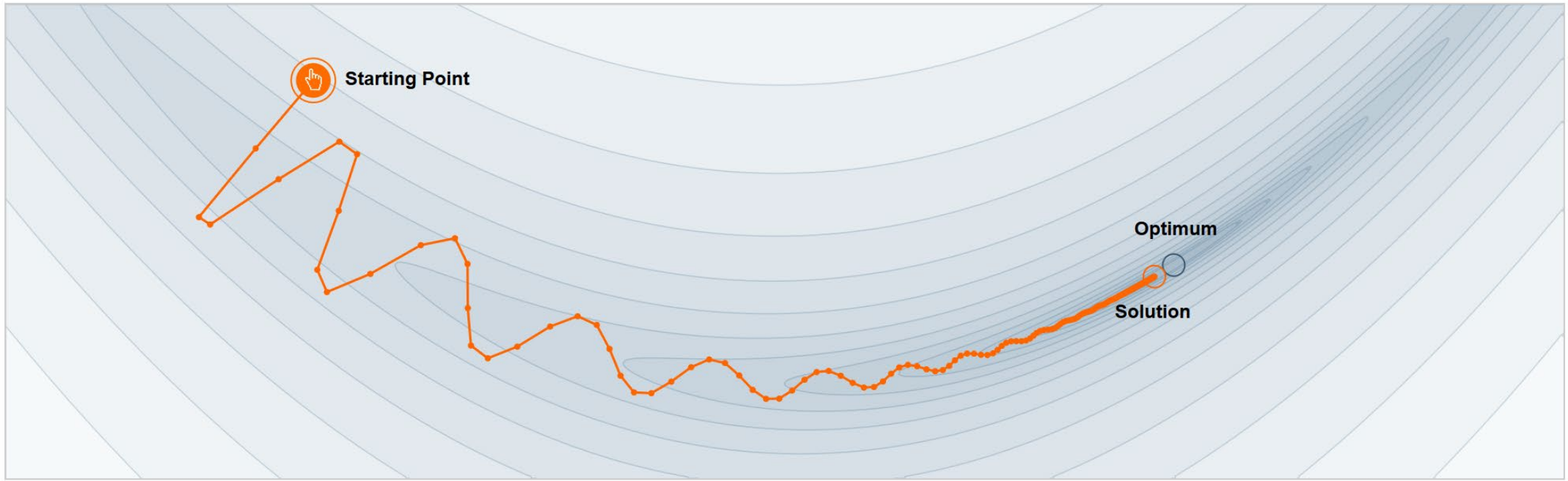
Without momentum



Adam



Why Momentum Really Works



Step-size α = 0.0015

0 0.003 0.006

Momentum β = 0.88

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

GABRIEL GOH
UC Davis

April. 4
2017

Citation:
Goh, 2017

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*
University of Amsterdam
dpkingma@uva.nl

Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca

ABSTRACT

We introduce *Adam*, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning. Some connections to related algorithms, on which *Adam* was inspired, are discussed. We also analyze the theoretical convergence properties of the algorithm and provide a regret bound on the convergence rate that is comparable to the best known results under the online convex optimization framework. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods. Finally, we discuss *AdaMax*, a variant of *Adam* based on the infinity norm.

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)


WARNING
Mathematics
Ahead

while θ_t not converged **do**

$$t \leftarrow t + 1$$

$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$$

$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

$$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$$

$$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$$

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$$

end while

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

end while

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

end while

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

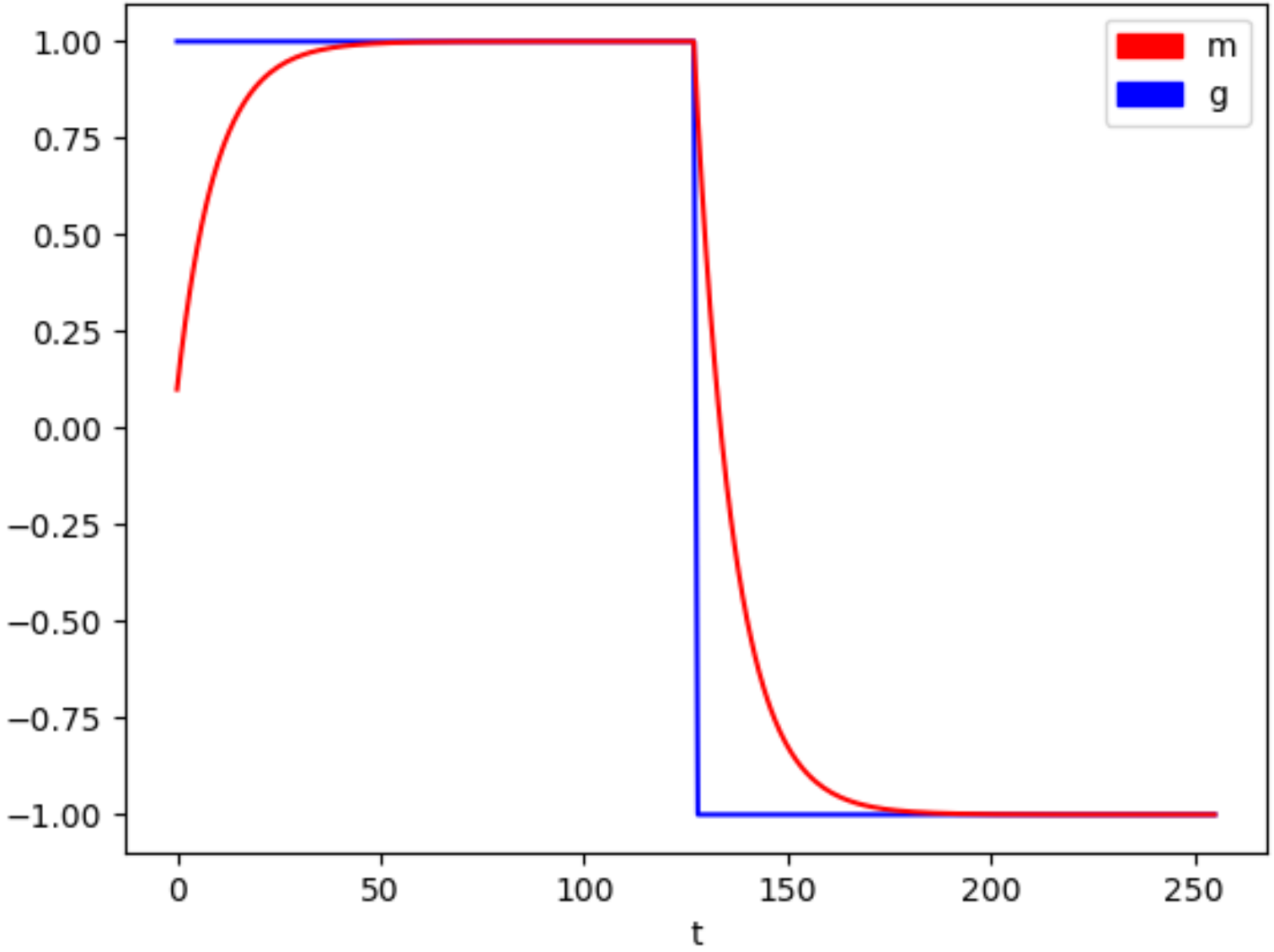
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

end while

Exponential smoothing, beta1=0.9



while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

end while

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

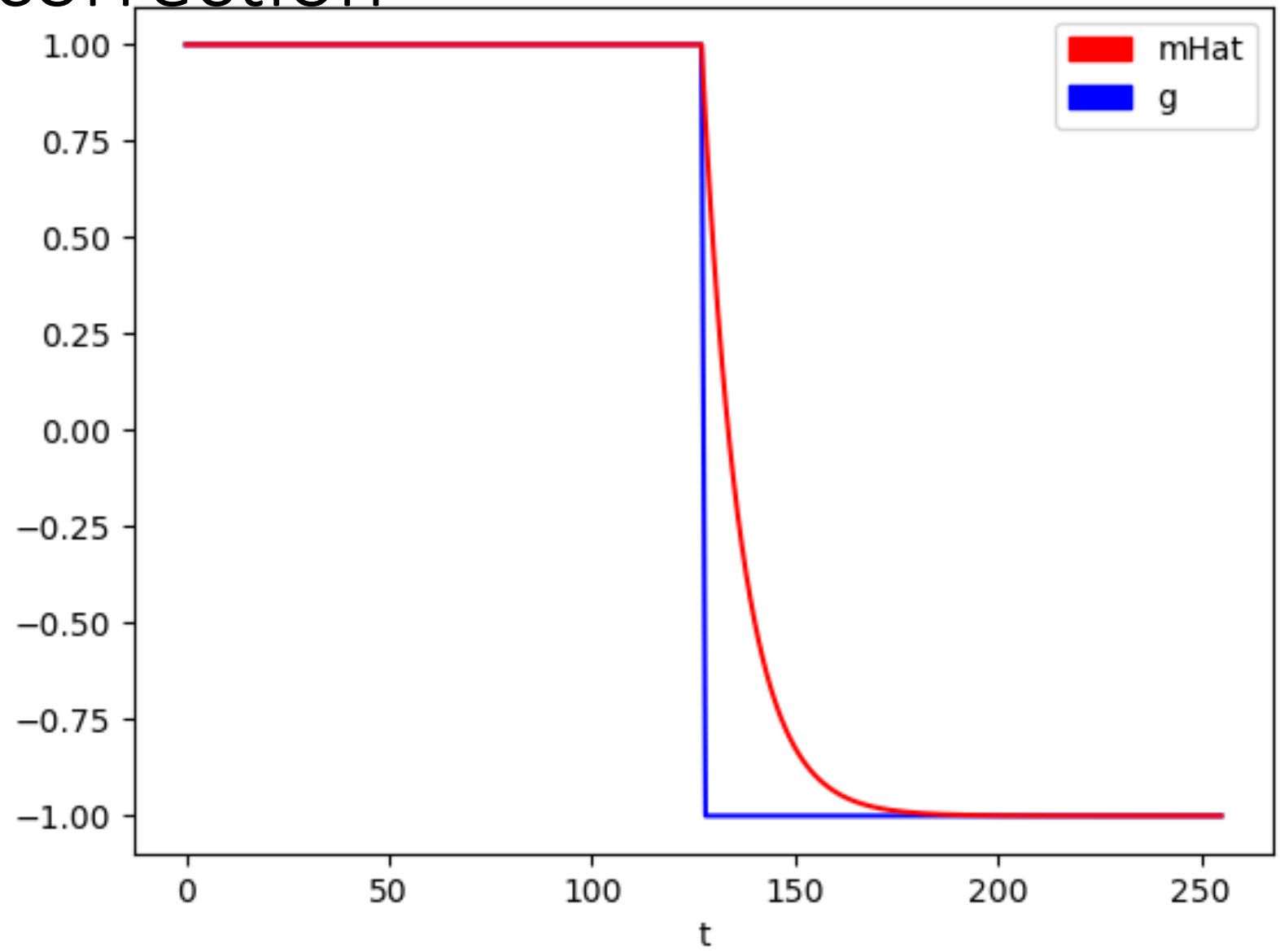
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

end while

Bias correction



while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

end while

while θ_t not converged **do**

Change of parameter values

$$\theta_t \leftarrow \theta_{t-1} - \boxed{\alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)}$$

end while

while θ_t not converged **do**

Step size

$$\theta_t \leftarrow \theta_{t-1} - \boxed{\alpha} \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$$

end while

while θ_t not converged **do**

Smoothed gradient

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \boxed{\hat{m}_t} / (\sqrt{\hat{v}_t} + \epsilon)$$

end while

while θ_t not converged **do**

Per-variable scaling

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$$

end while

while θ_t not converged **do**

Per-variable scaling

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$$

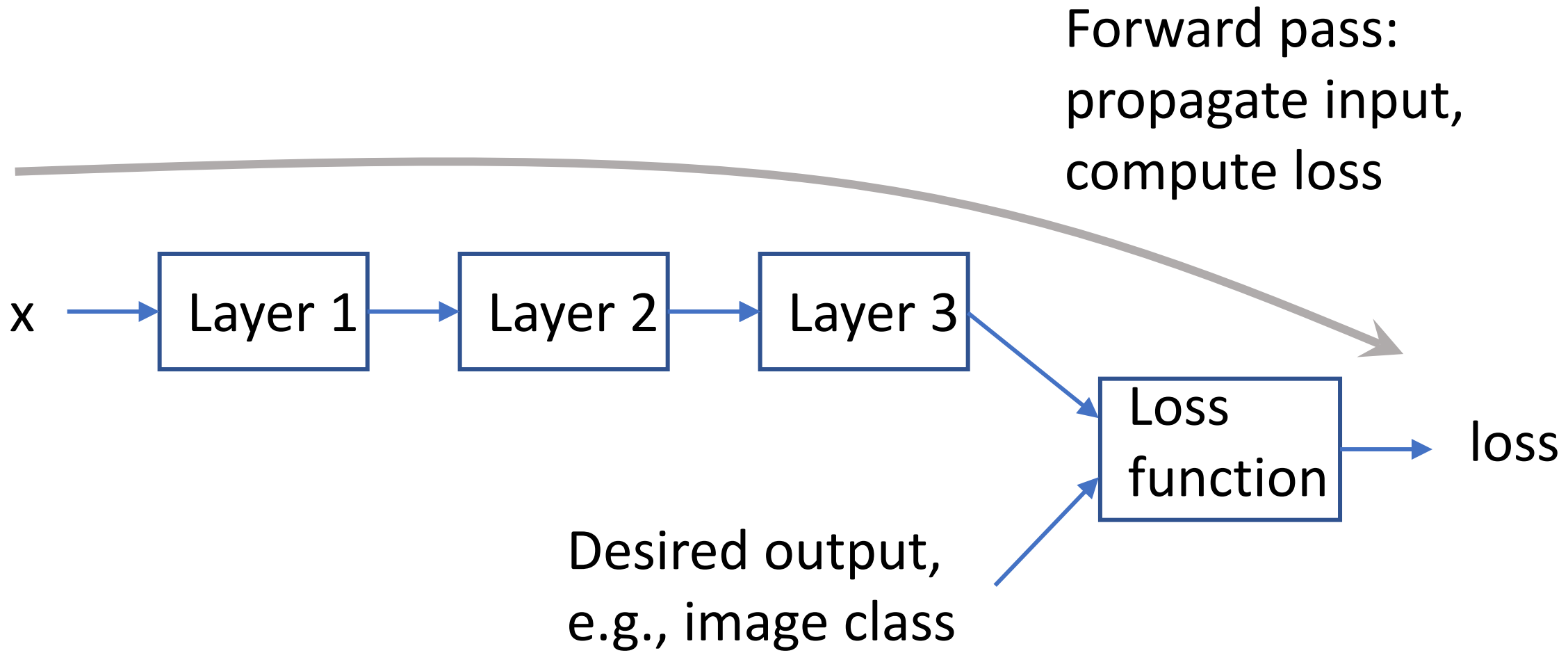
end while


WA NG
M ematics
Ahead

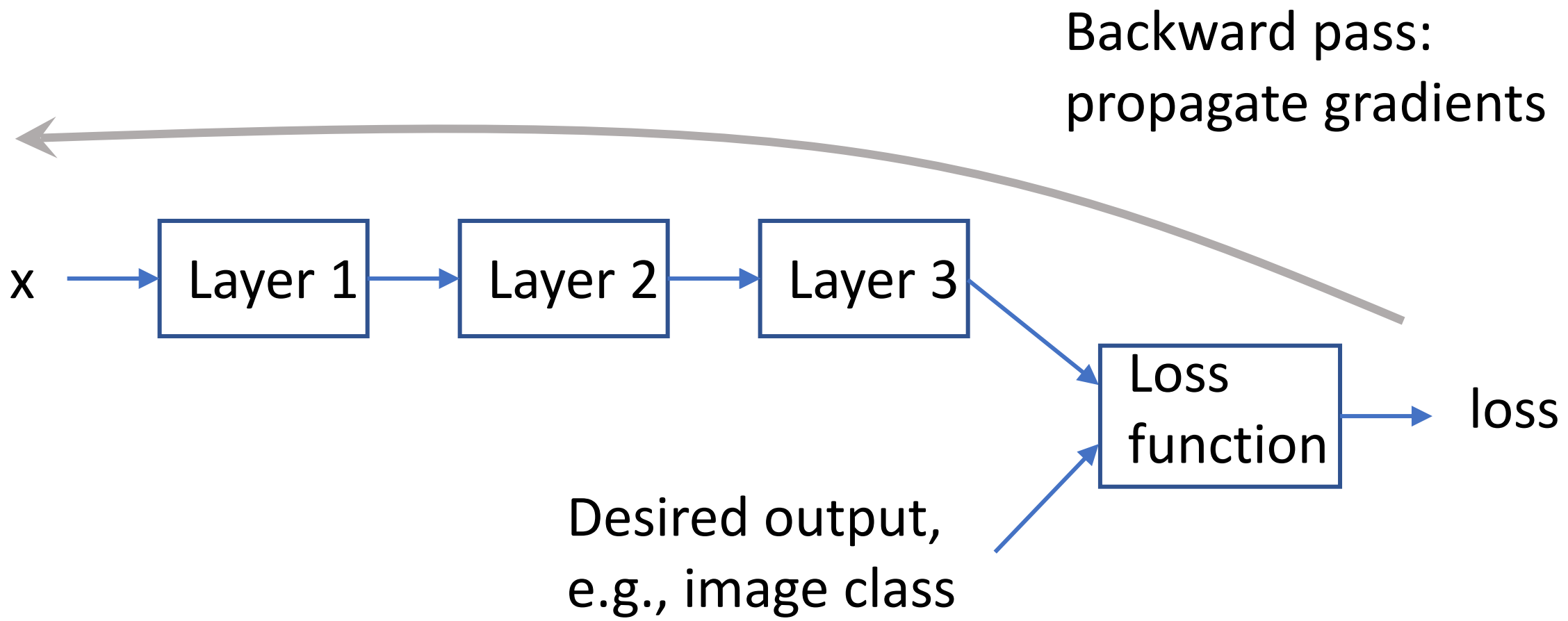
Gradients and compute graphs

- Utilizing the chain rule of calculus, Tensorflow & Pytorch allow one to compute gradients of compute graph outputs (e.g., loss function) with respect to variables (e.g., neural network weights)
- This process is called backpropagation

Backpropagation

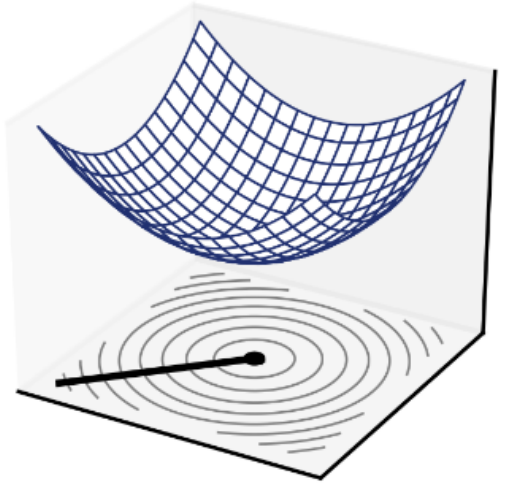


Backpropagation

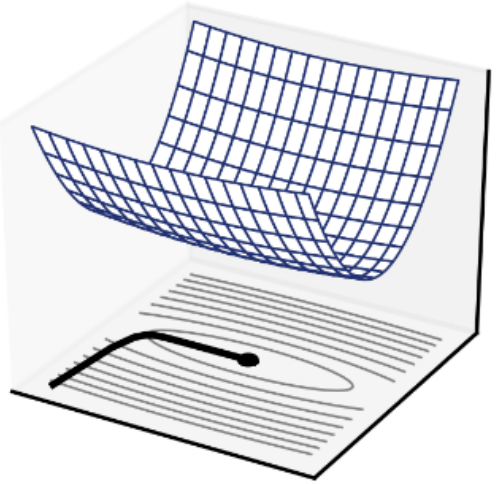


Difficulty of optimization

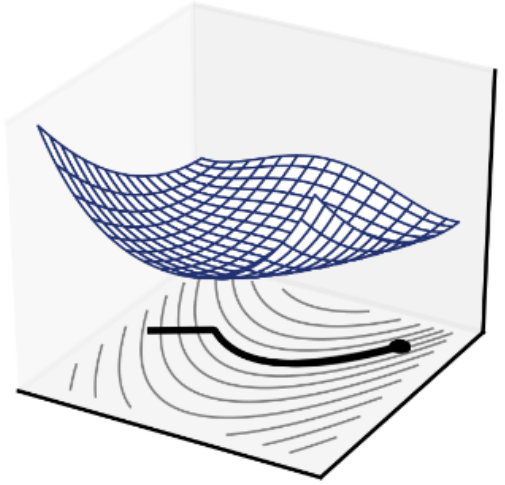
Convex,
well-conditioned



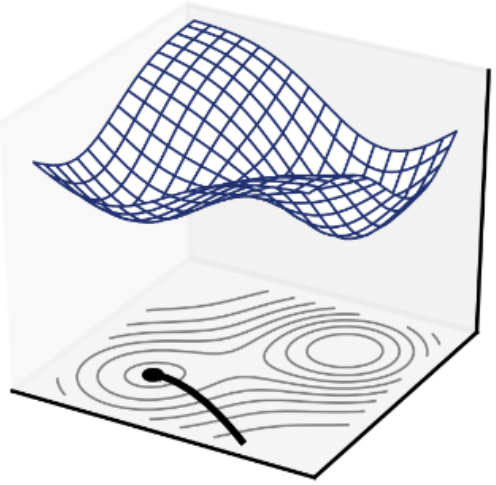
Convex,
ill-conditioned



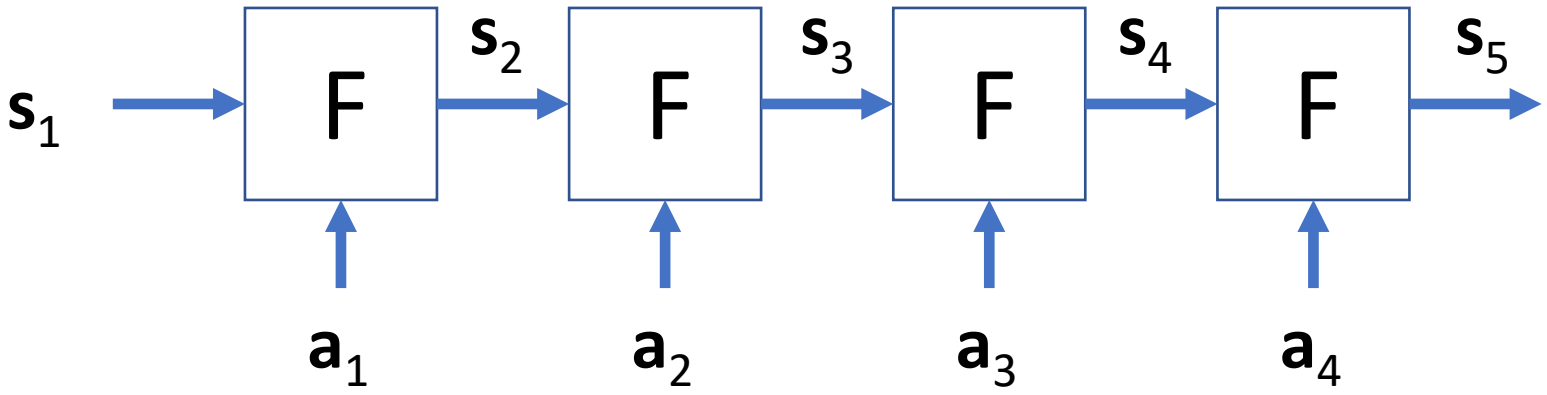
Non-convex,
unimodal



Non-convex,
multimodal



Planning through backpropagation

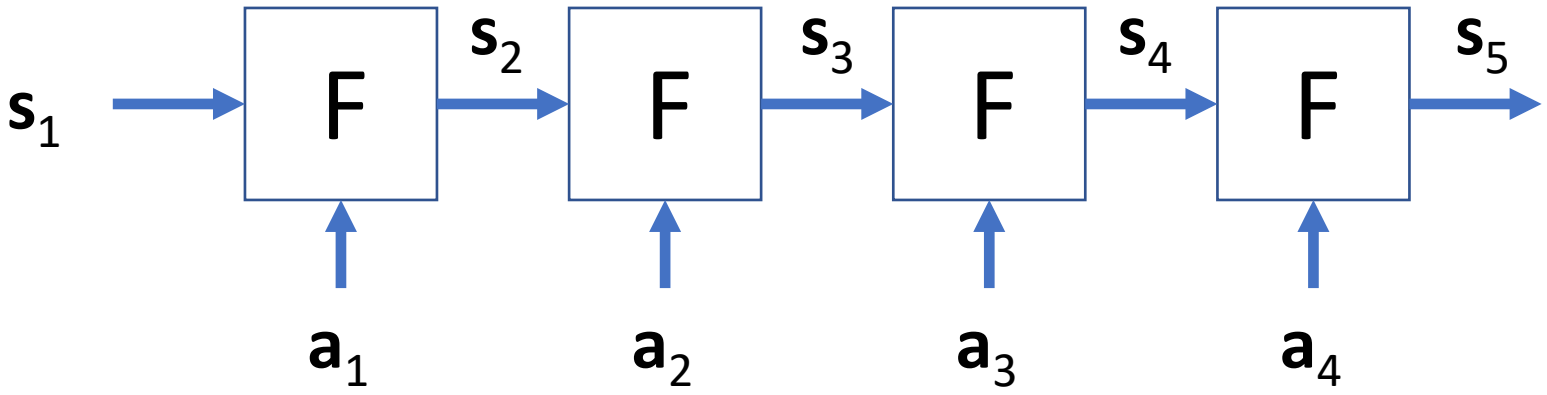


Subscripts denote simulation time

s: state, e.g., the physics simulation parameters of a robot
a: action, e.g., the robot's actuation torques
F: simulation model (code or learned neural network), $s_2=F(s_1,a_1)$

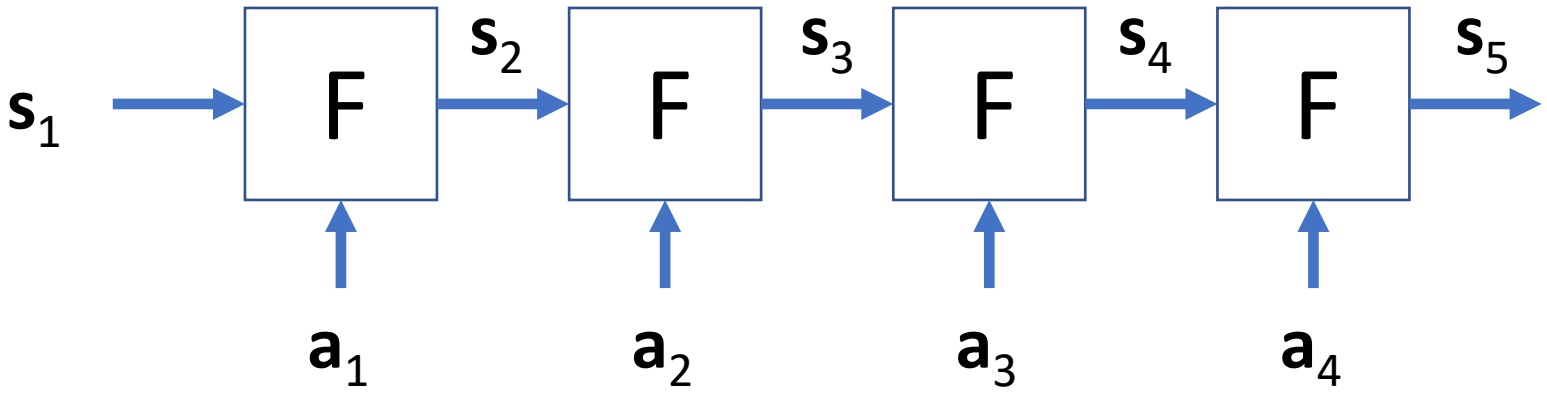
Planning through backpropagation

Objective: Maximize $\sum_i r(\mathbf{a}_i, \mathbf{s}_i)$



Planning through backpropagation

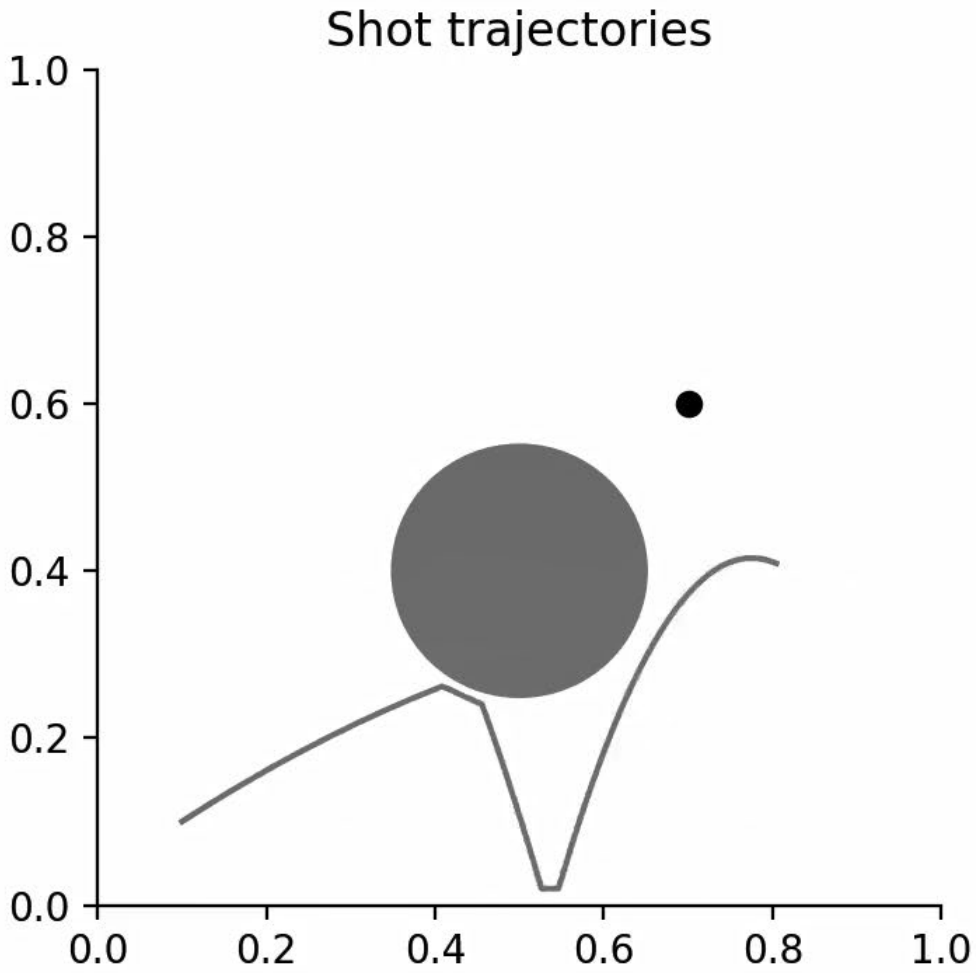
Objective: Maximize $\sum_i r(\mathbf{a}_i, \mathbf{s}_i)$

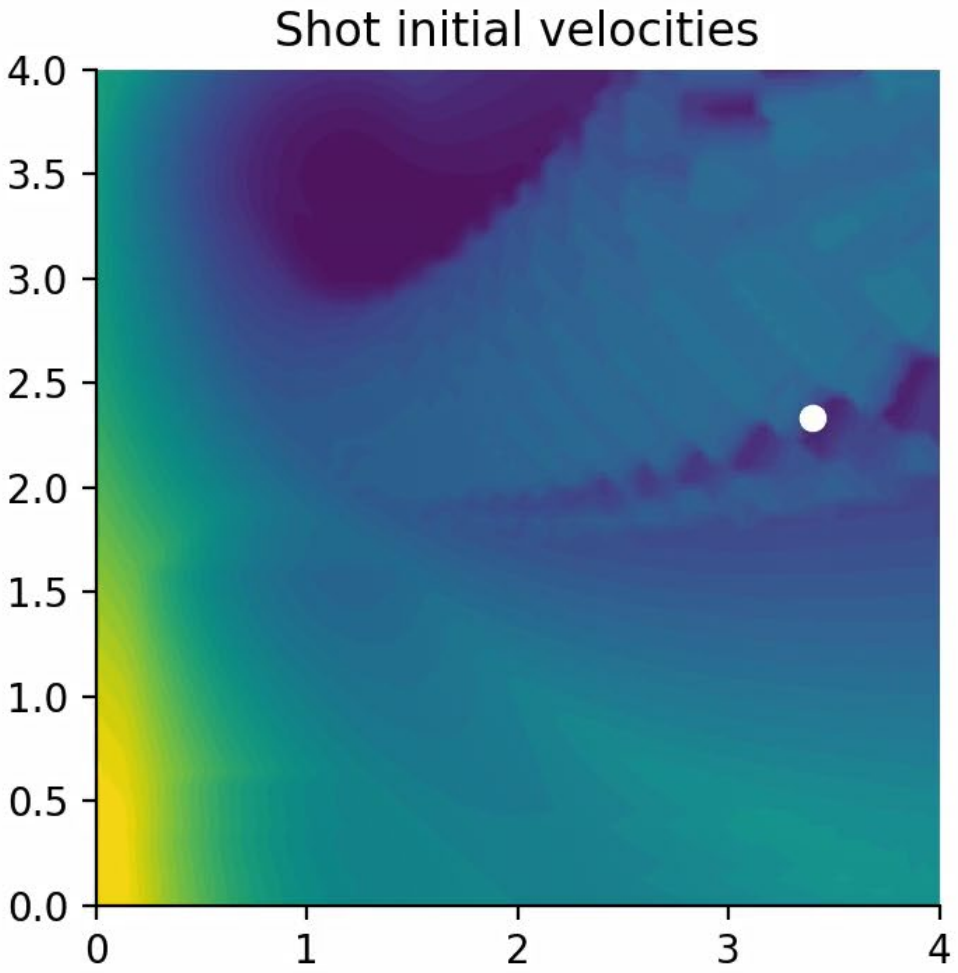


Gradient-based optimization of actions is trivial in PyTorch etc.,
if both F and $r(\mathbf{a}_i, \mathbf{s}_i)$ are differentiable

Problem: Local optima, discontinuities

Didactic example: Ballistic shot with collisions

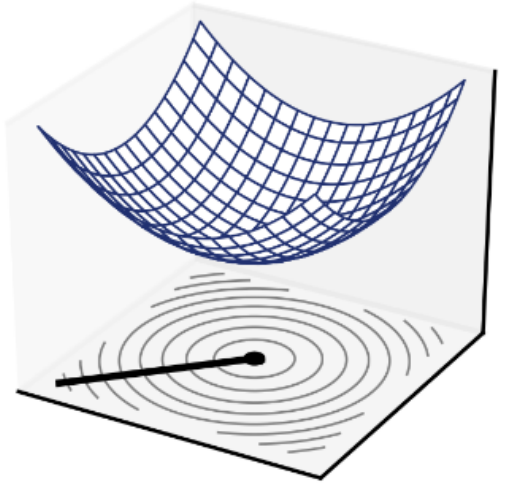




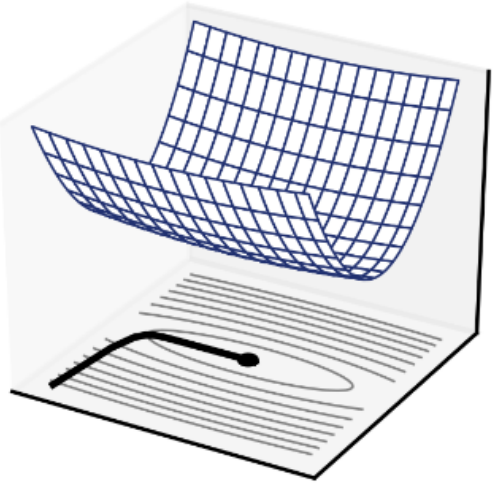
Directly optimizing initial shot vx,vy using Adam (possible because of differentiable dynamics) gets stuck in local optimum

Difficulty of optimization

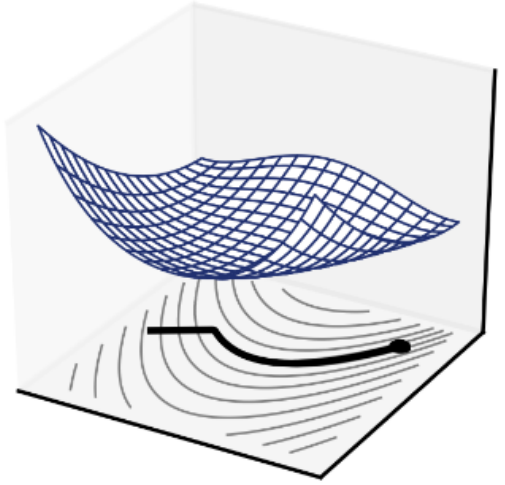
Convex,
well-conditioned



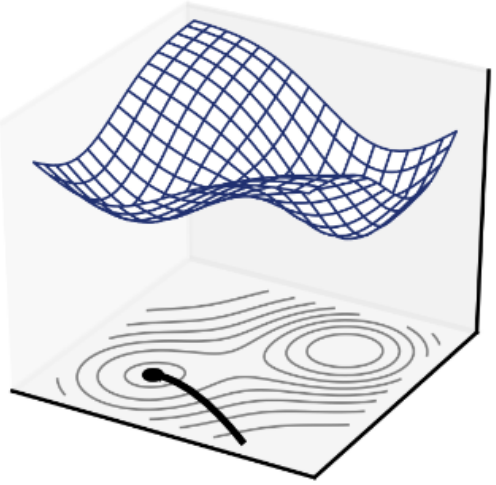
Convex,
ill-conditioned



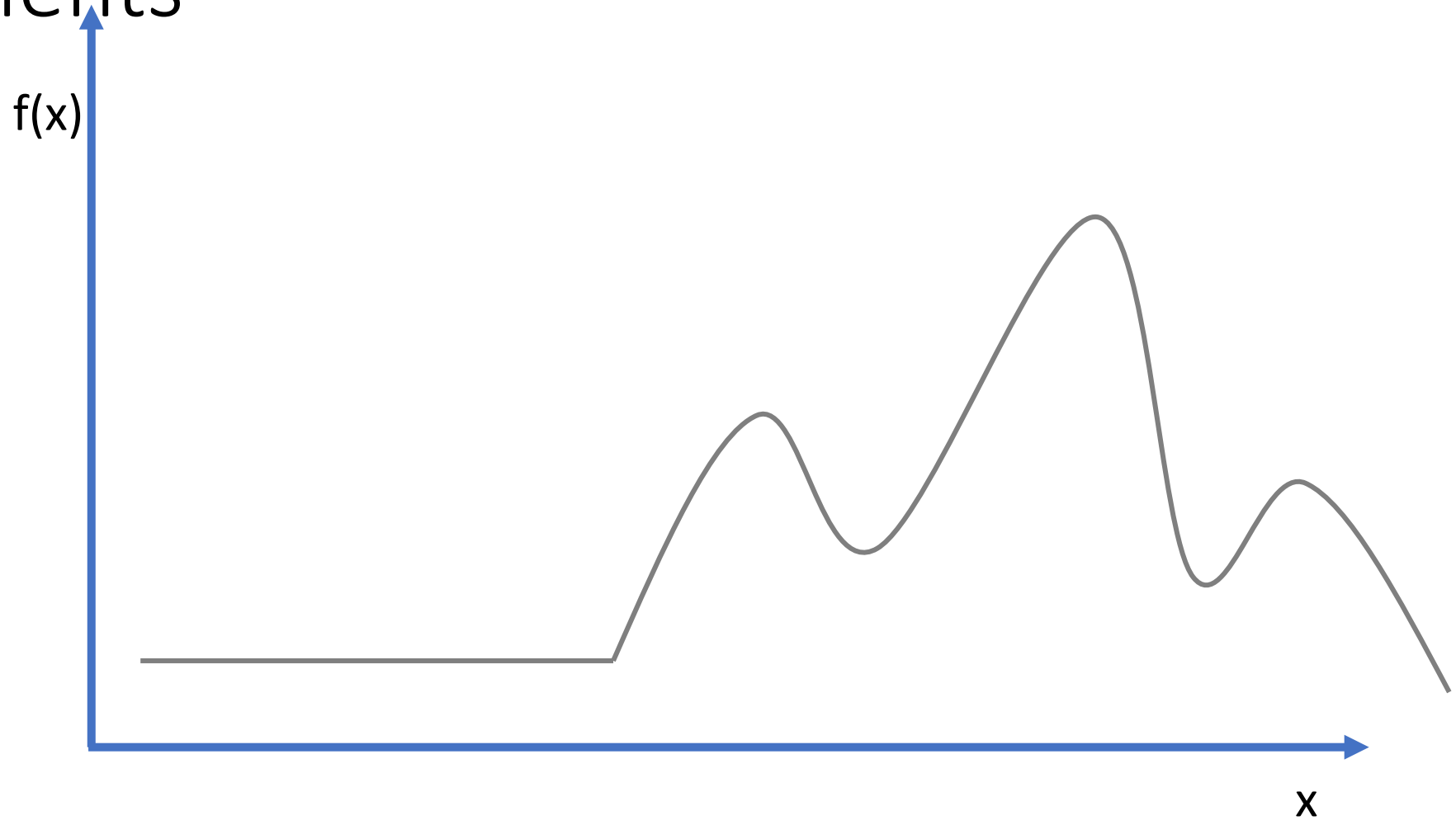
Non-convex,
unimodal



Non-convex,
multimodal

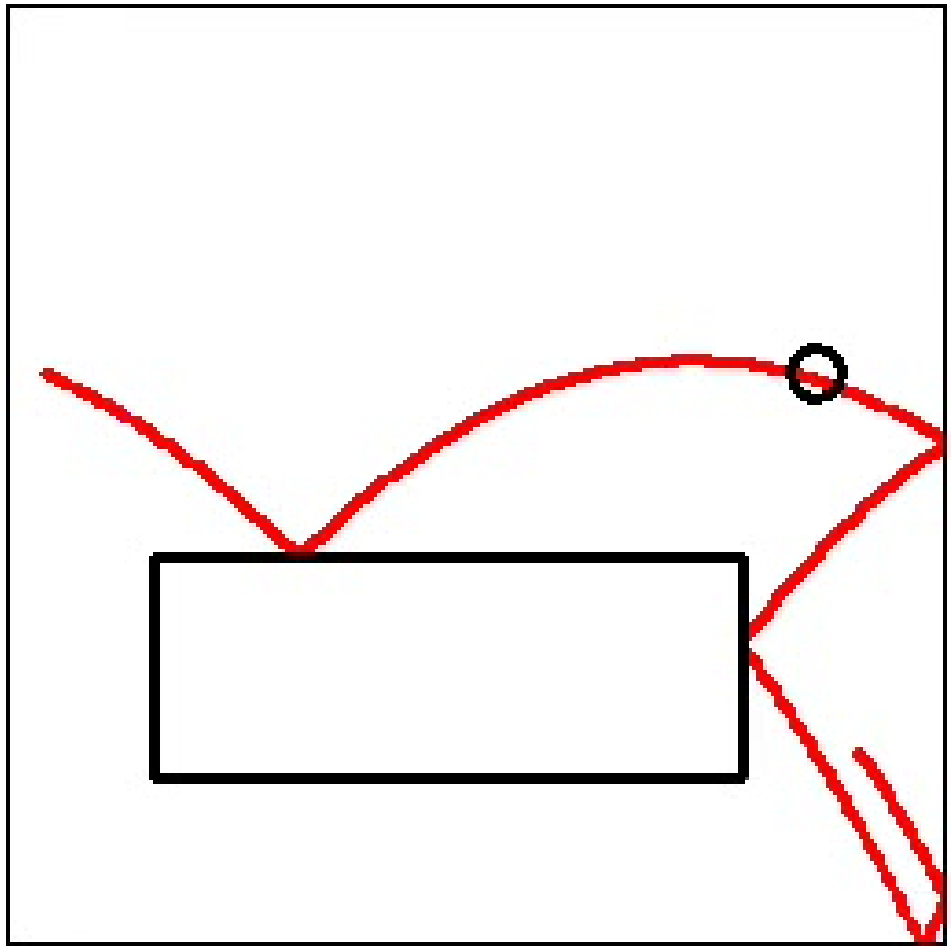


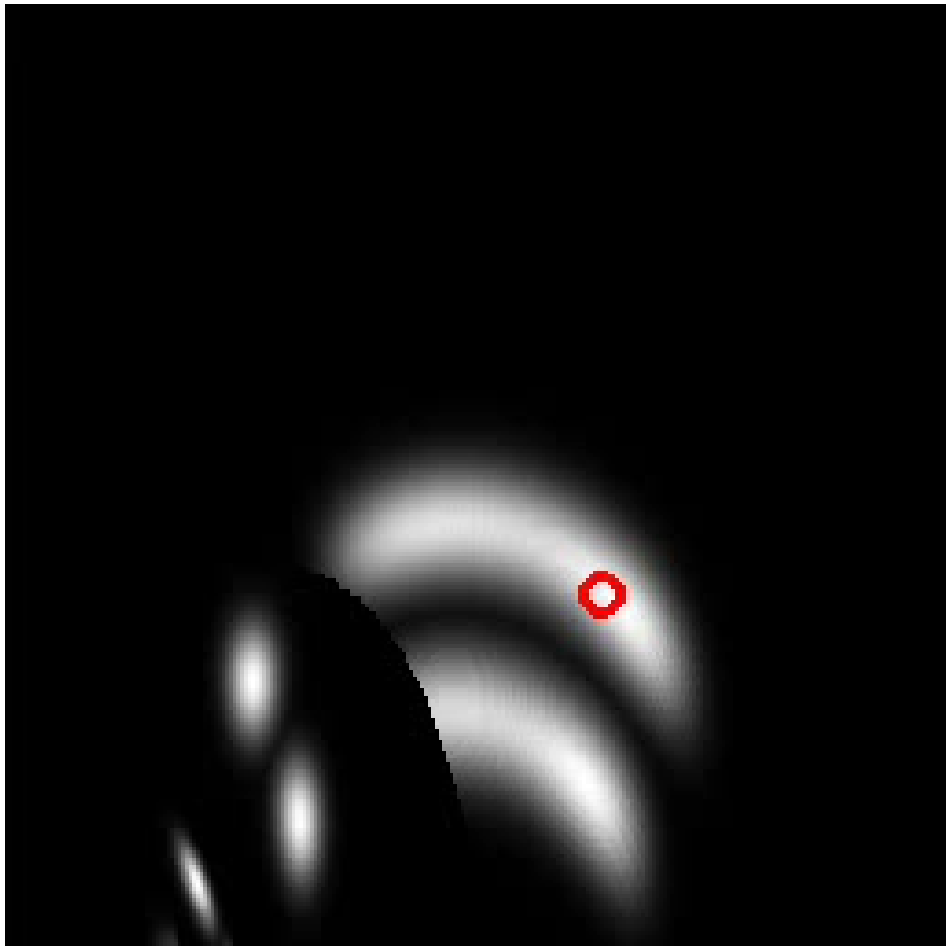
Real-life problems: multimodality, absence of gradients



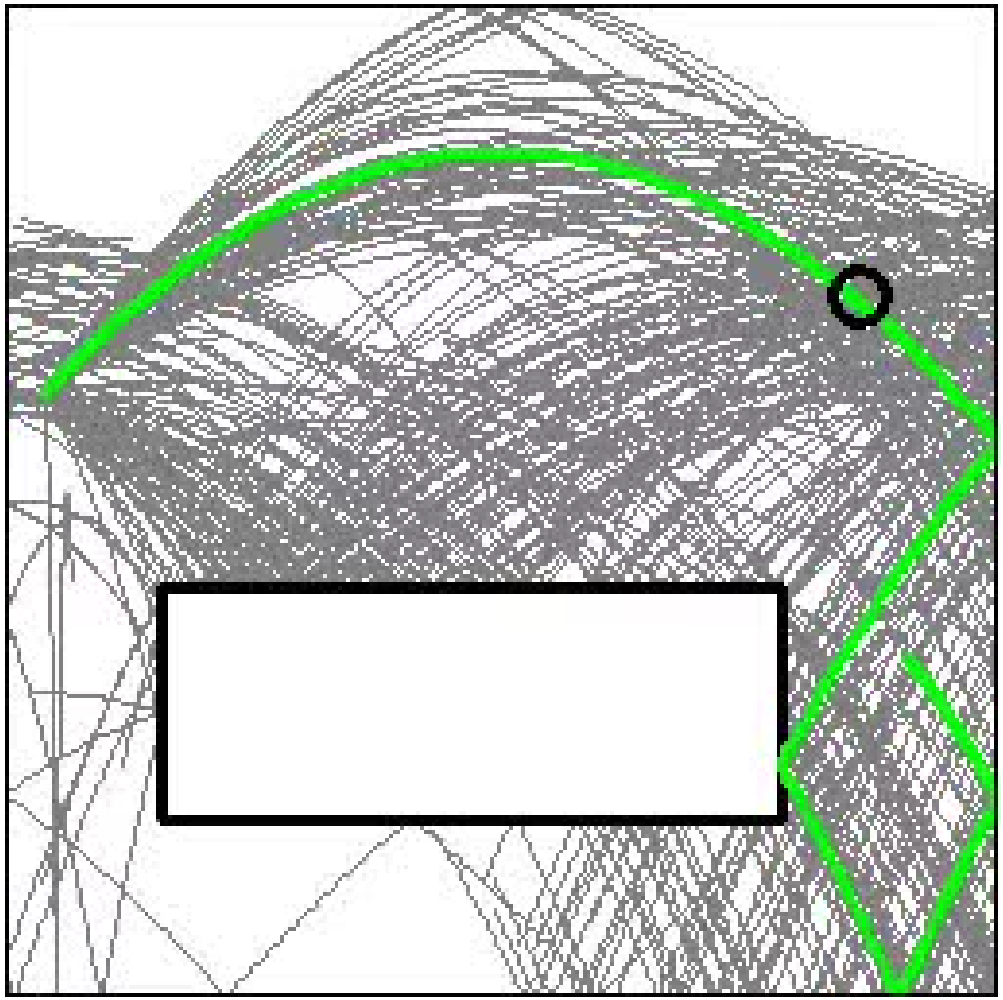
Ball trajectories

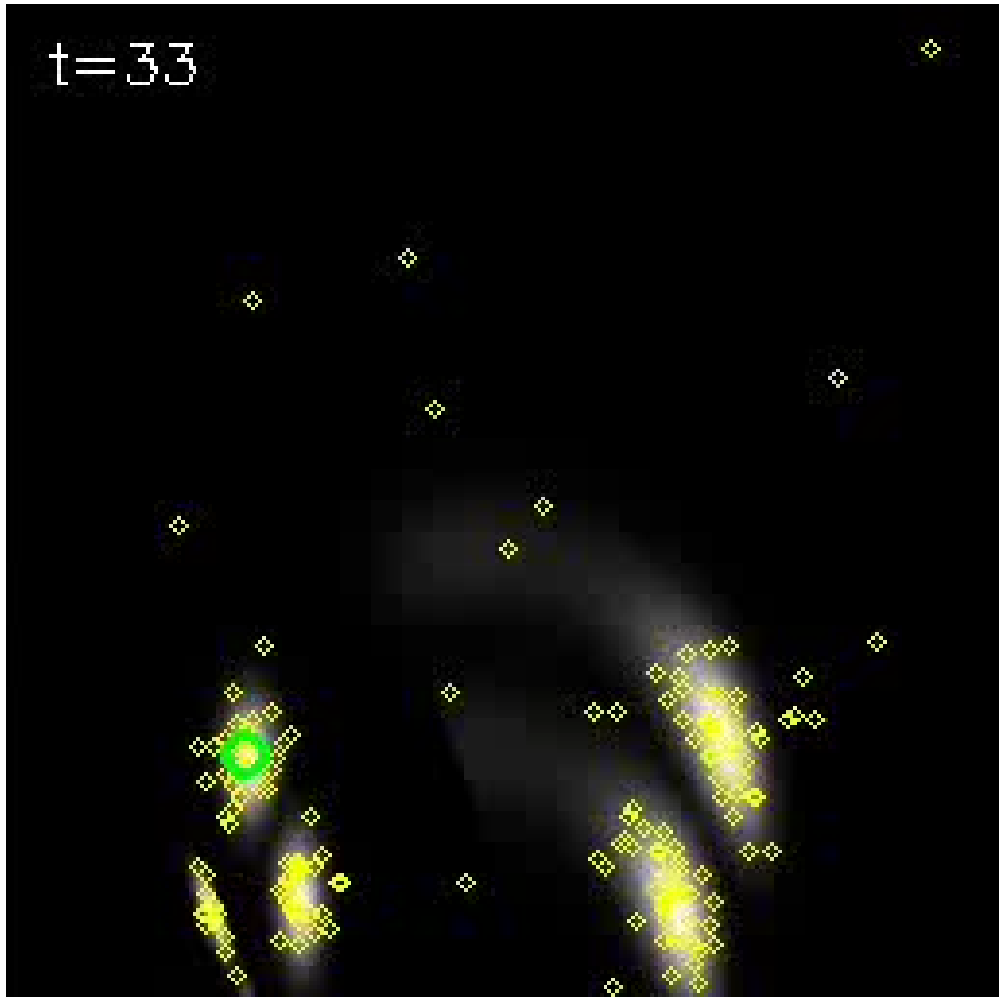
Optimization landscape





Sampling-based methods to the rescue



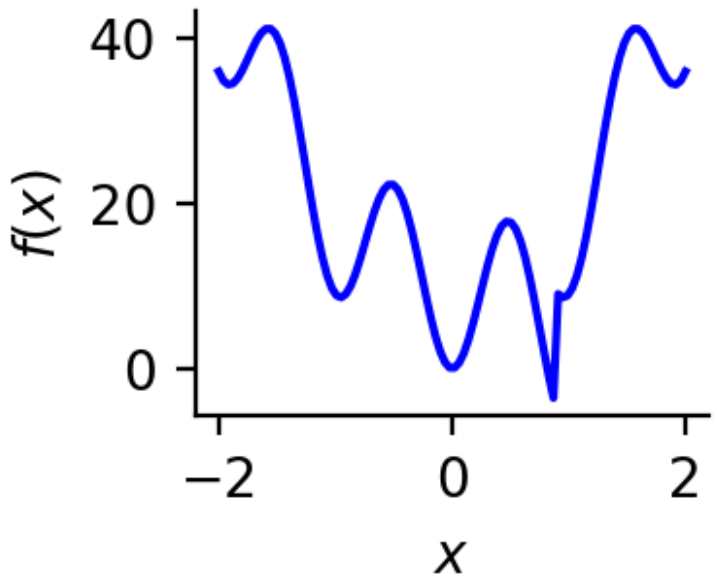


Sampling-based optimization

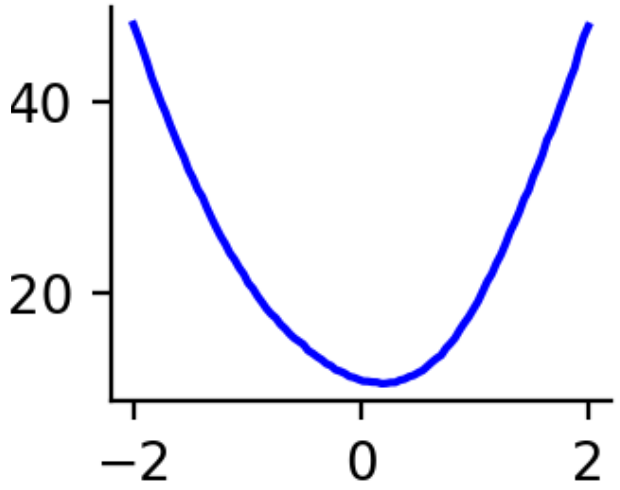
Key intuition: A multimodal function can be smoothed to make it convex

- The most common and successful multimodal optimization methods apply some form of Gaussian blurring to the objective function to avoid gradient-based optimization from getting stuck

Original $f(x)$



Blurred $f(x)$



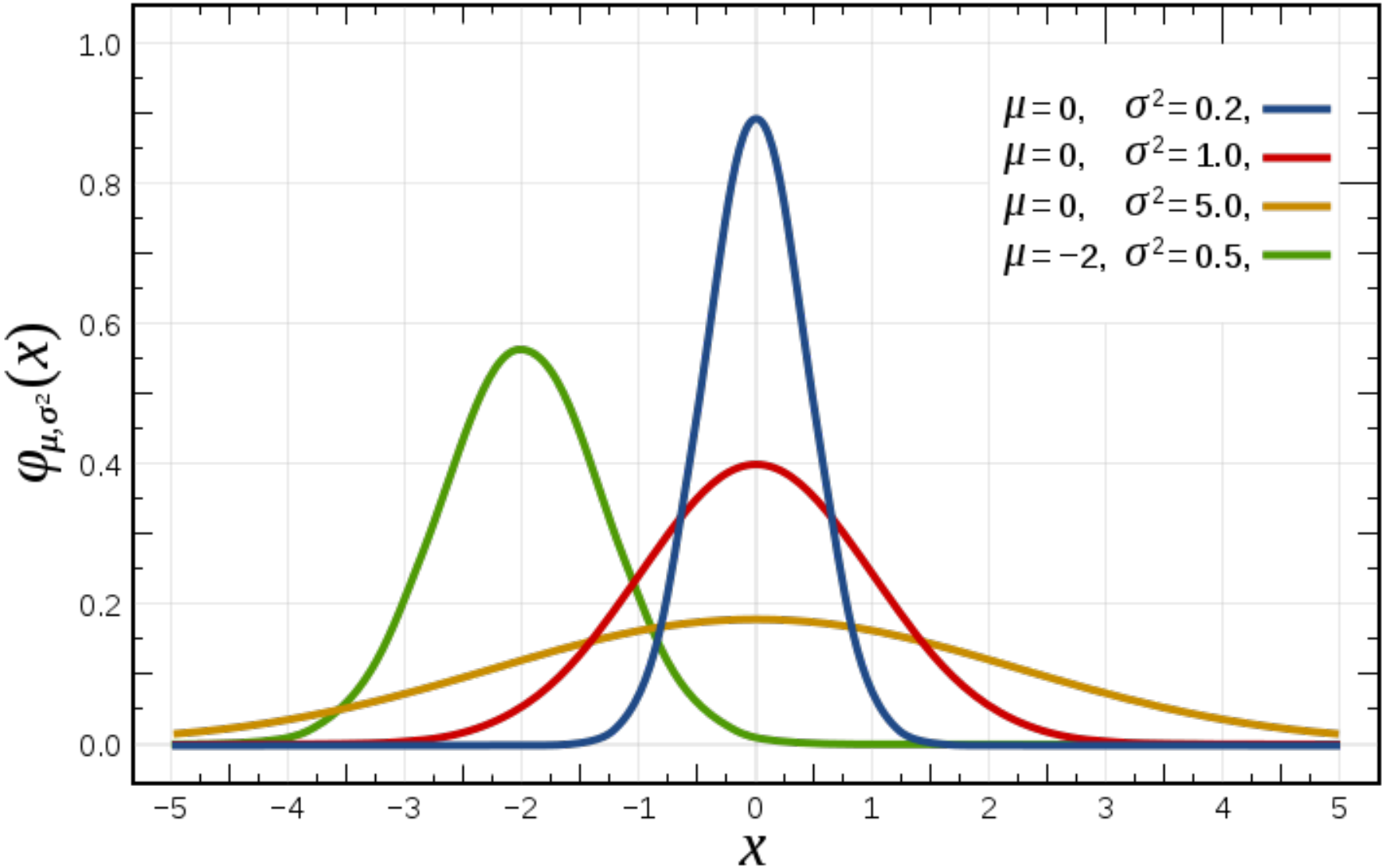
Key intuition: A multimodal function can be smoothed to make it convex

- Mathematically, optimize $\mathbb{E}_{x\sim p(x)}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x)$ instead of $f(x)$
- $p(x)$ is a normal distribution with some mean μ and standard deviation σ. The sum is over a batch of N random samples.
- ***Now, $f(\mu)$ ≈ GaussianBlur($f(x)$, σ). For σ=0, optimal μ equals the optimal x.***
- The expectation is a bound for the true minimum:

$$\min_x f(x) \leq \mathbb{E}_{x\sim p(x)}[f(x)]$$

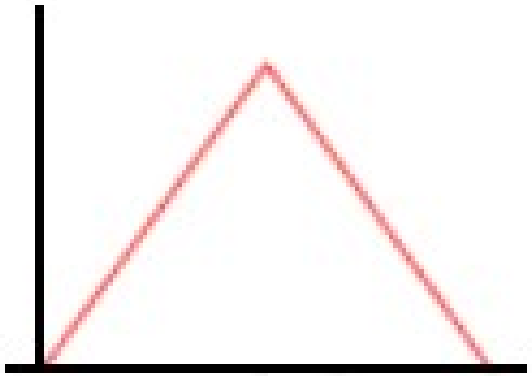
- The bound becomes tighter (E[$f(x)$] becomes closer to $f(x)$) when standard deviation approaches zero.
- Implication: To find optimal x, can optimize μ,σ. This will drive σ to 0, at which point μ is the solution.

Reminder: normal (Gaussian) distribution

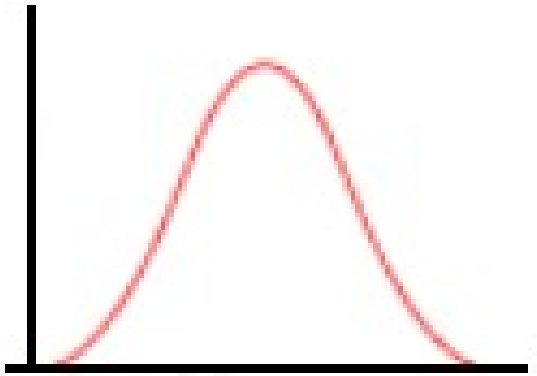


Preliminaries: normal (Gaussian) distribution

uniform
distribution
1 die

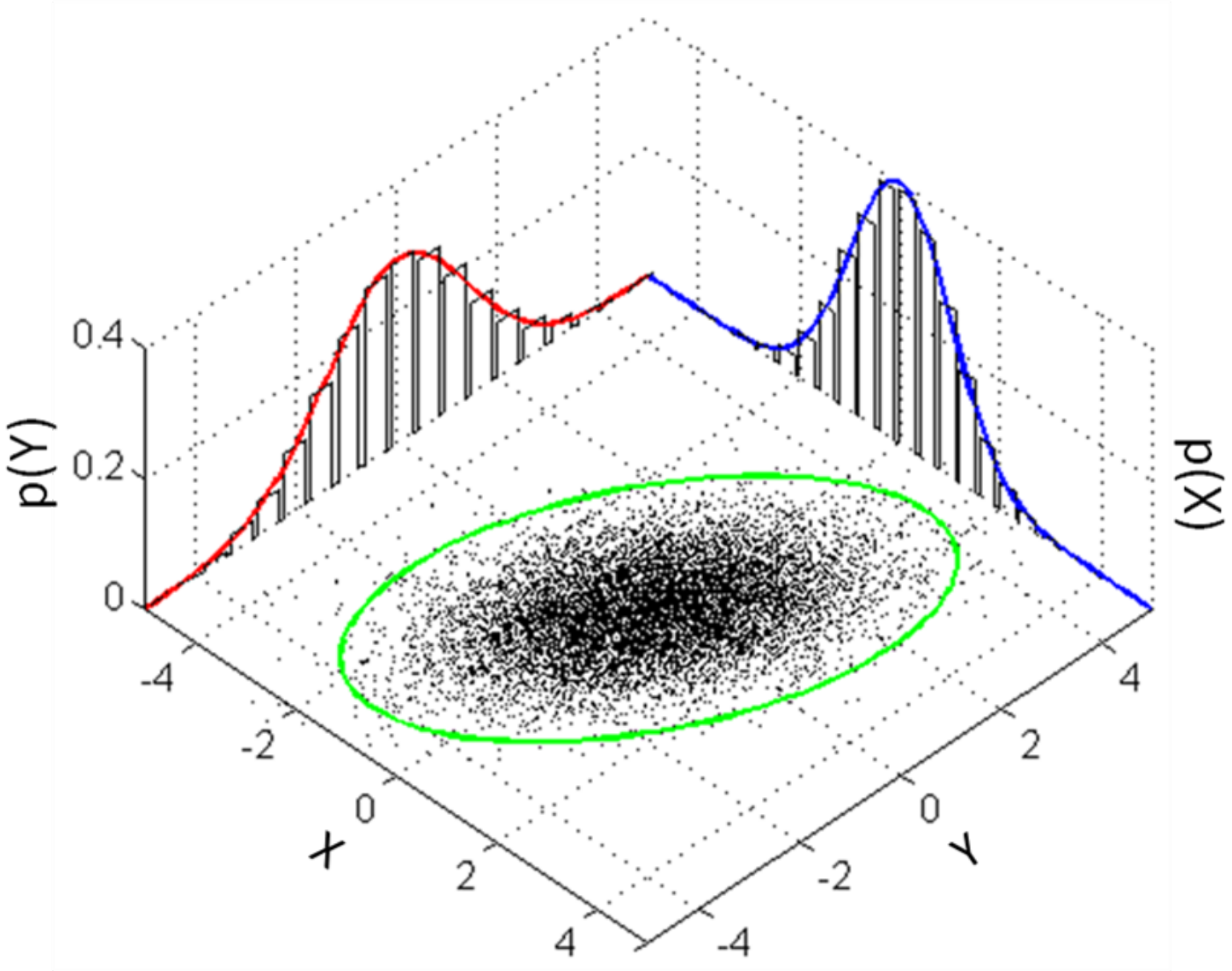


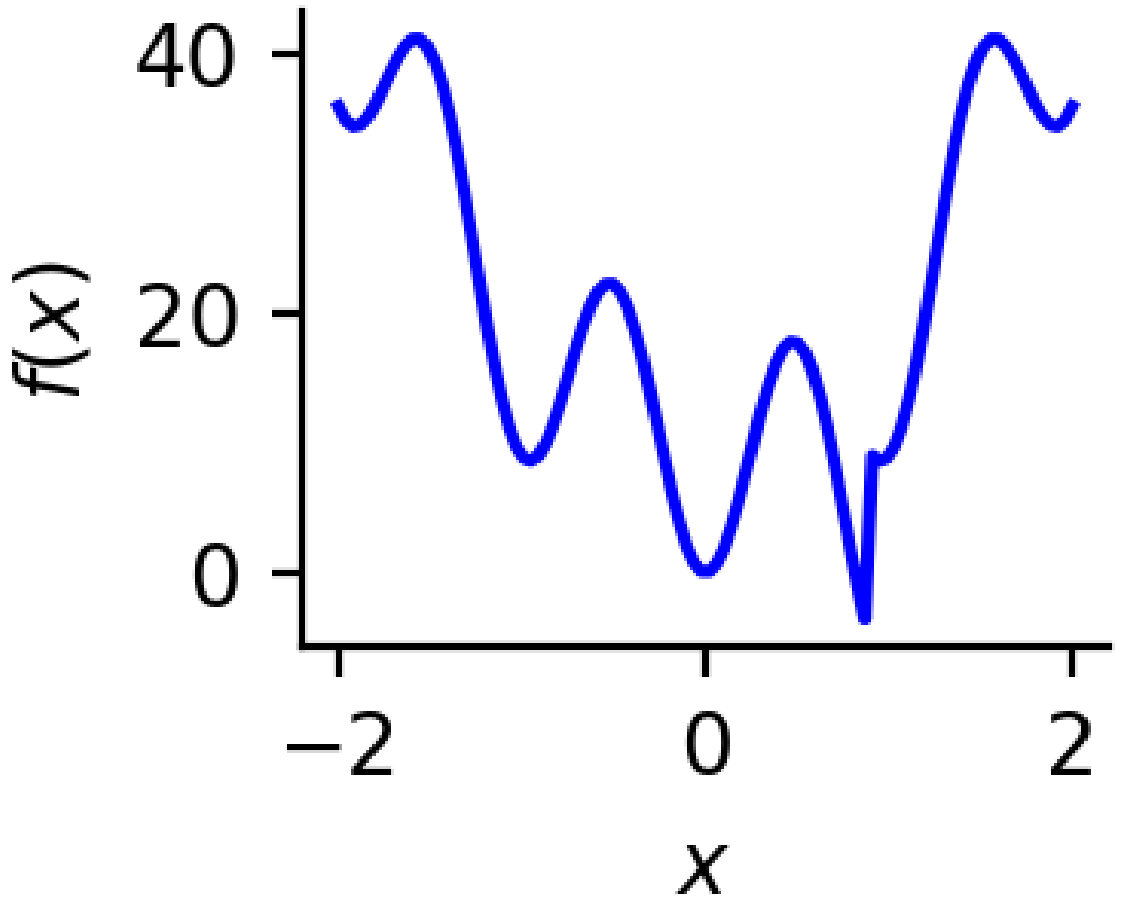
uniform sum
distribution
2 dice

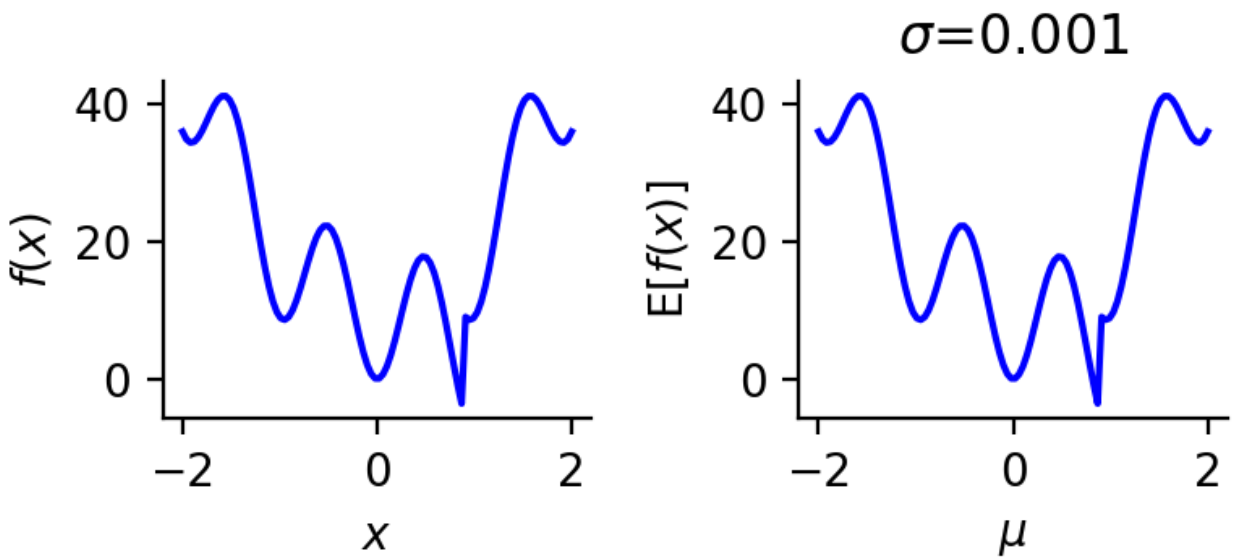


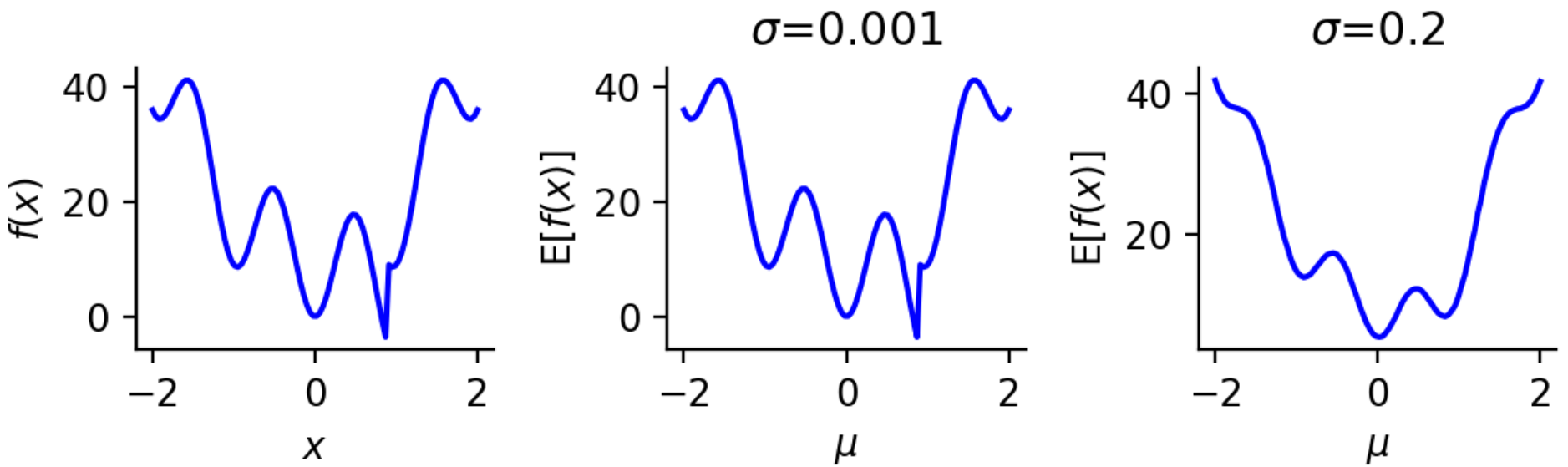
uniform sum
distribution
3 dice

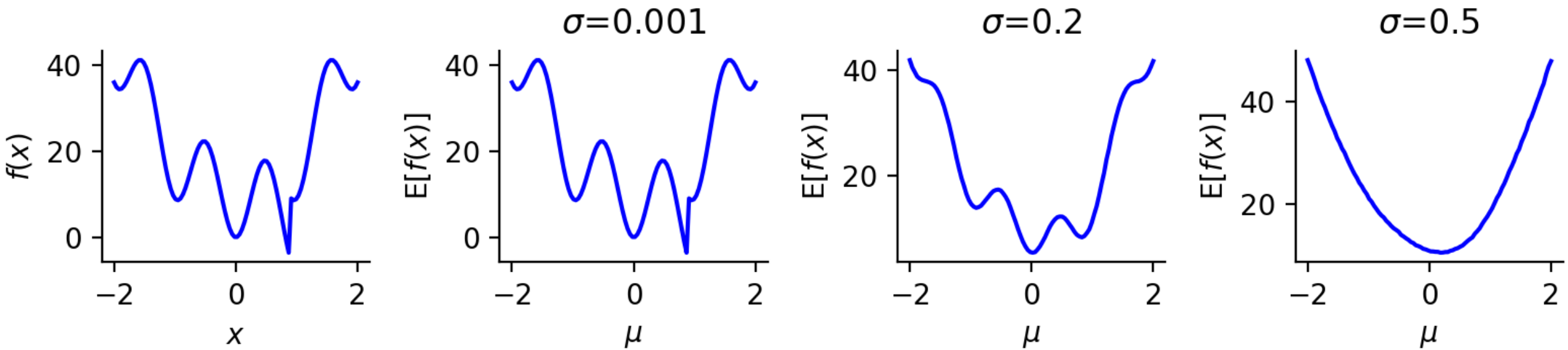
Preliminaries: normal (Gaussian) distribution



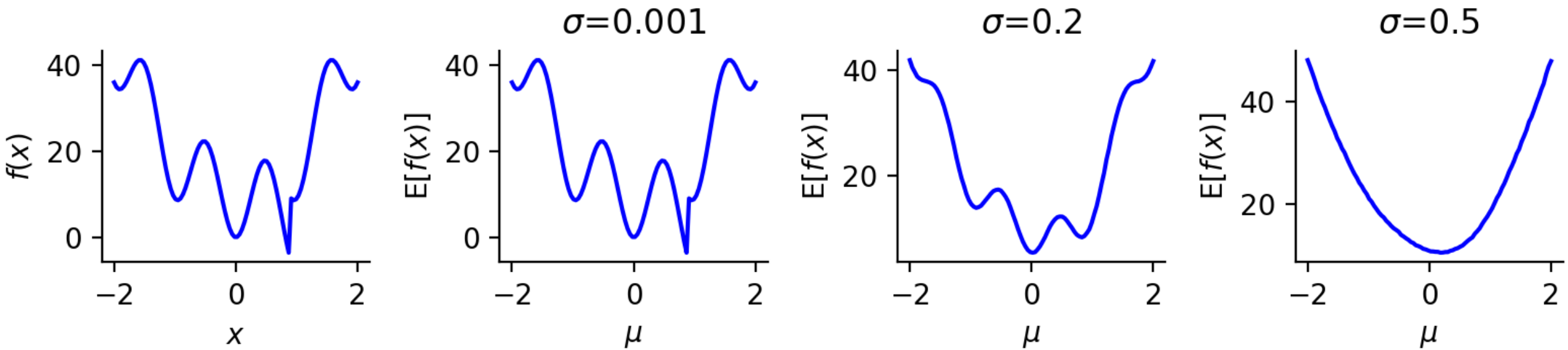

40
20
0
f(x)
−2
0
2
x


σ=0.001
40
20
0
f(x)
−2
0
2
x
E[f(x)]
μ


f(x)
40
20
0
−2
0
2
x
σ=0.001
E[f(x)]
40
20
0
−2
0
2
μ
σ=0.2
E[f(x)]
40
20
−2
0
2
μ

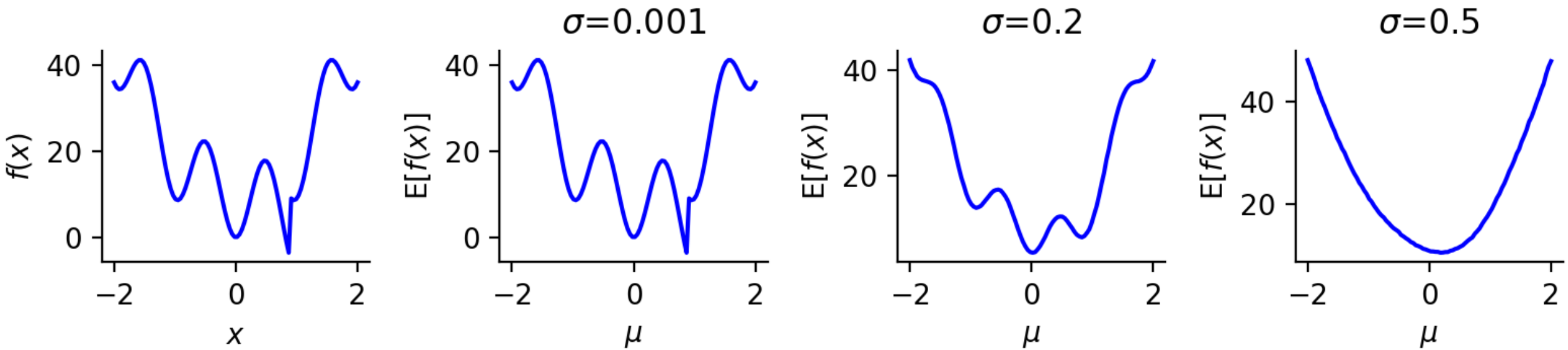

f(x)
40
20
0
−2
0
2
x
E[f(x)]
σ=0.001
μ
σ=0.2
σ=0.5

Large standard deviation makes the objective convex

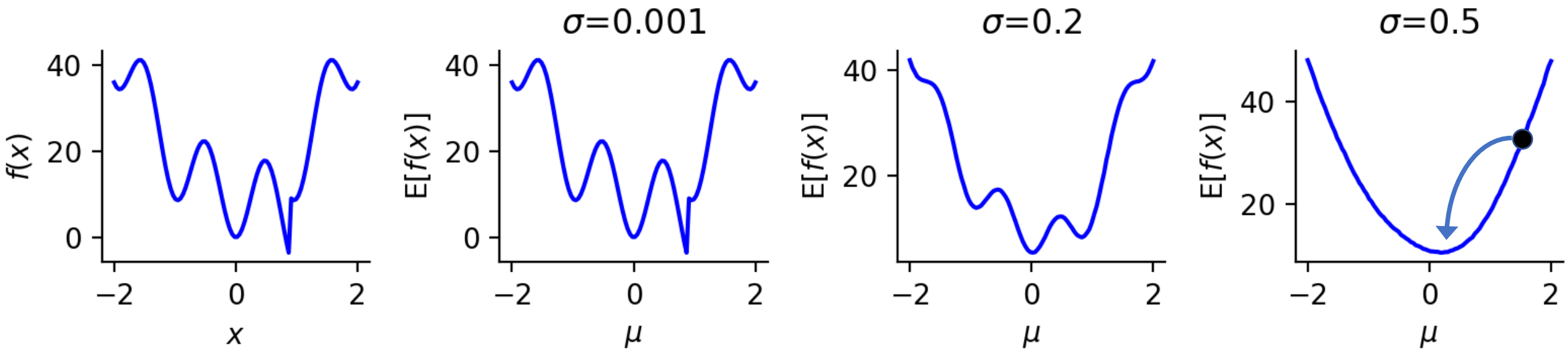


Gaussian sampling pdf = Gaussian blurring of f(x). Known to not introduce new extrema!

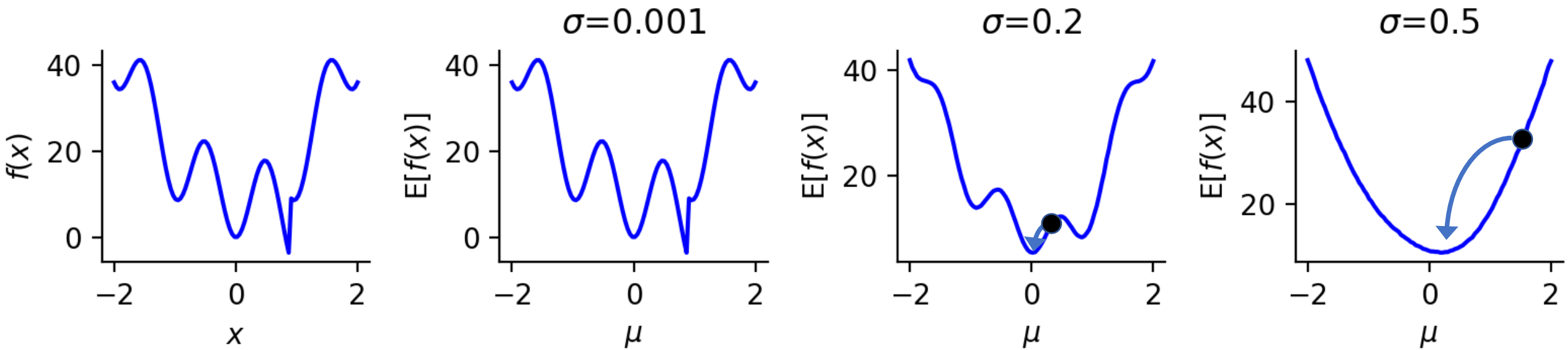
Key idea: can optimize μ with decreasing σ



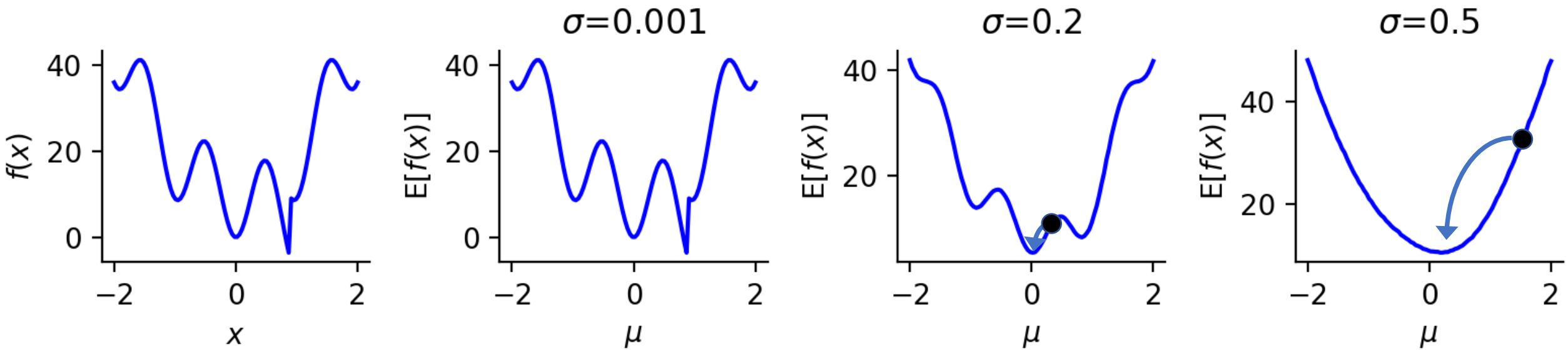
Key idea: can optimize μ with decreasing σ



Key idea: can optimize μ with decreasing σ

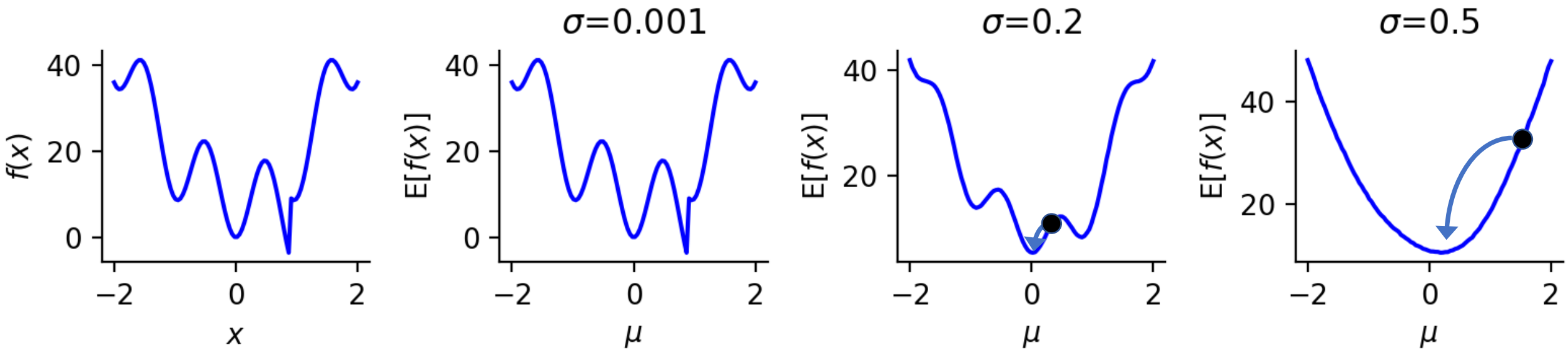


Key idea: can optimize μ with decreasing σ



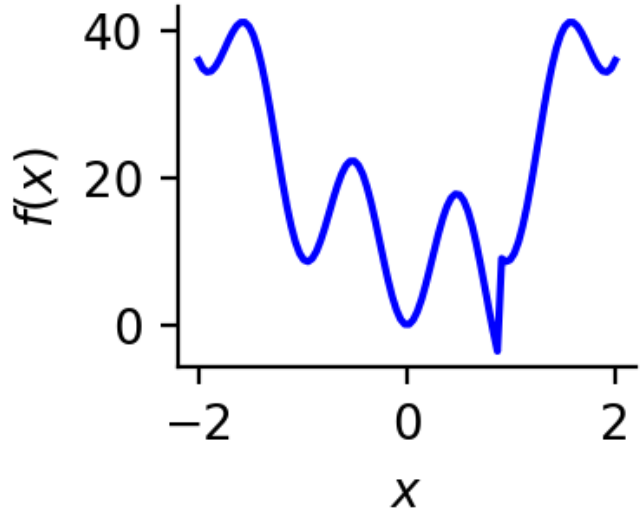
Will usually find a global optimum or a reasonably good local one.

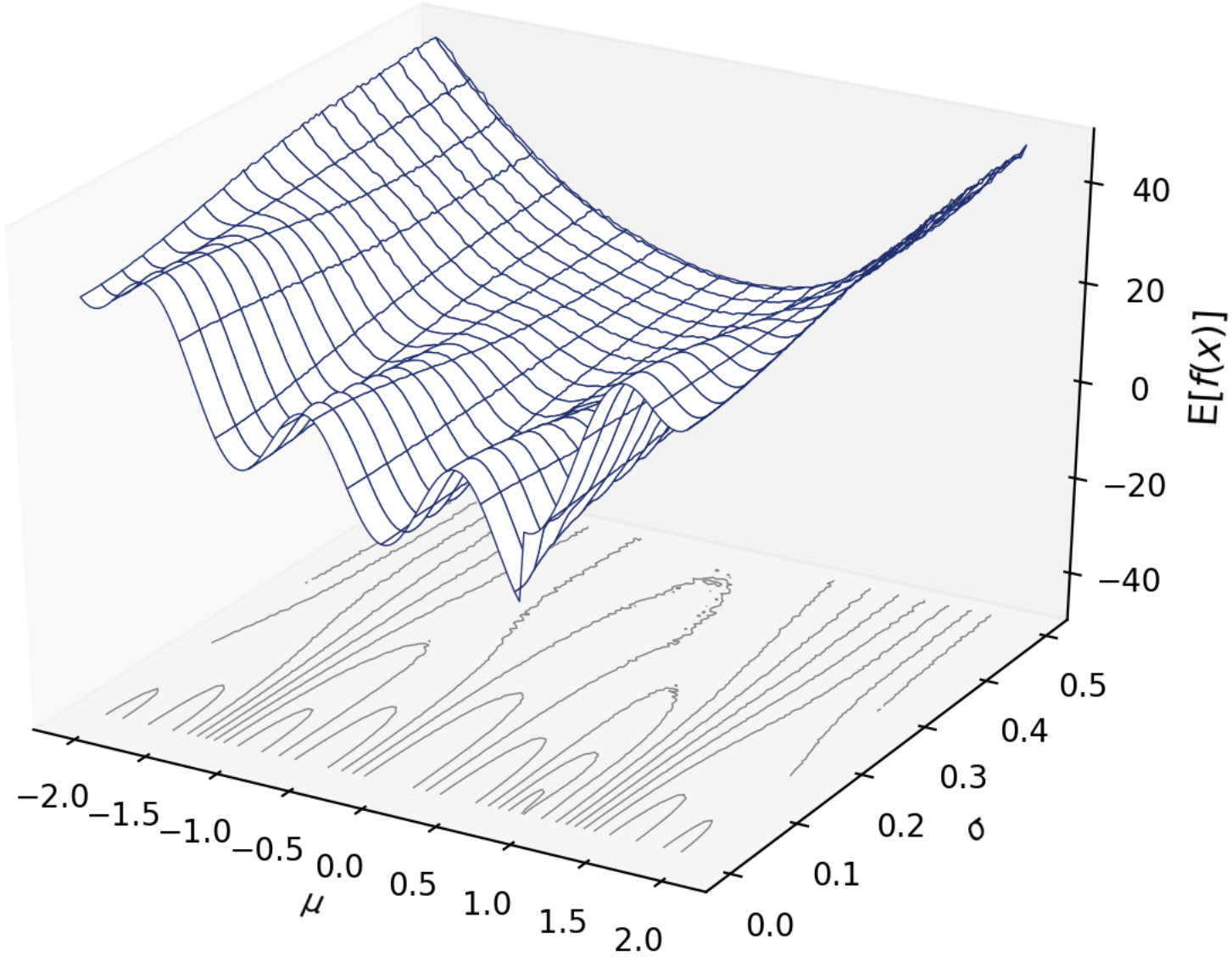
Key idea: can optimize μ with decreasing σ



How to decide the annealing schedule for σ?

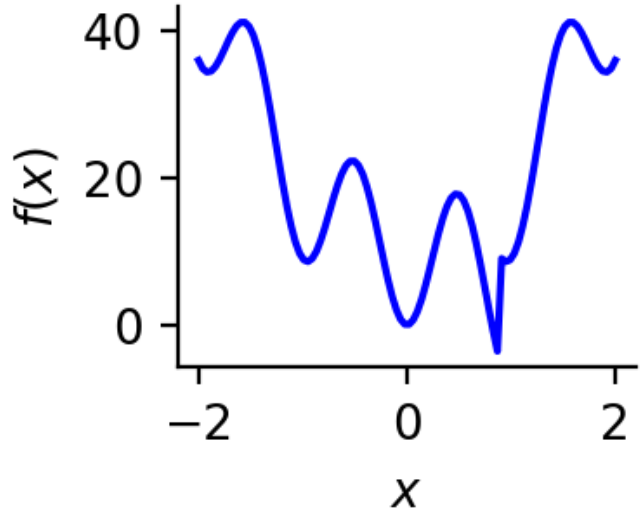
Optimize both μ, σ, start with large σ

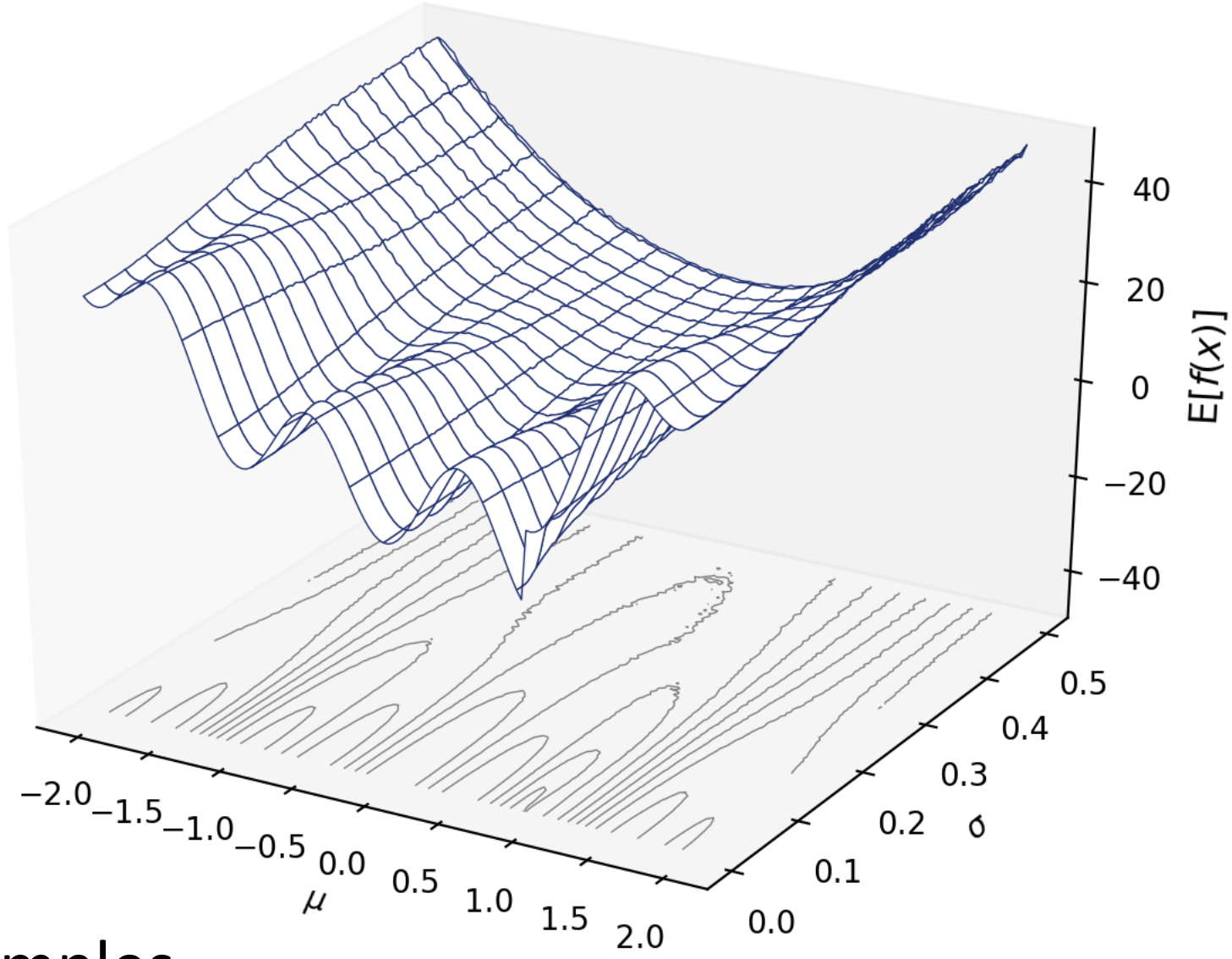




Just follow the gradient!

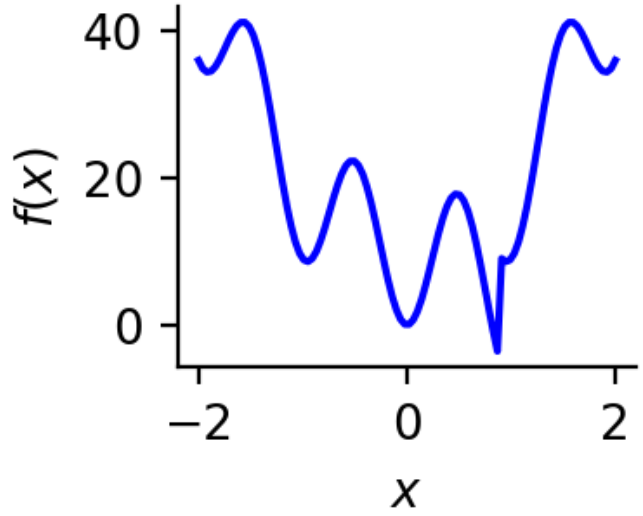
Optimize both μ, σ, start with large σ

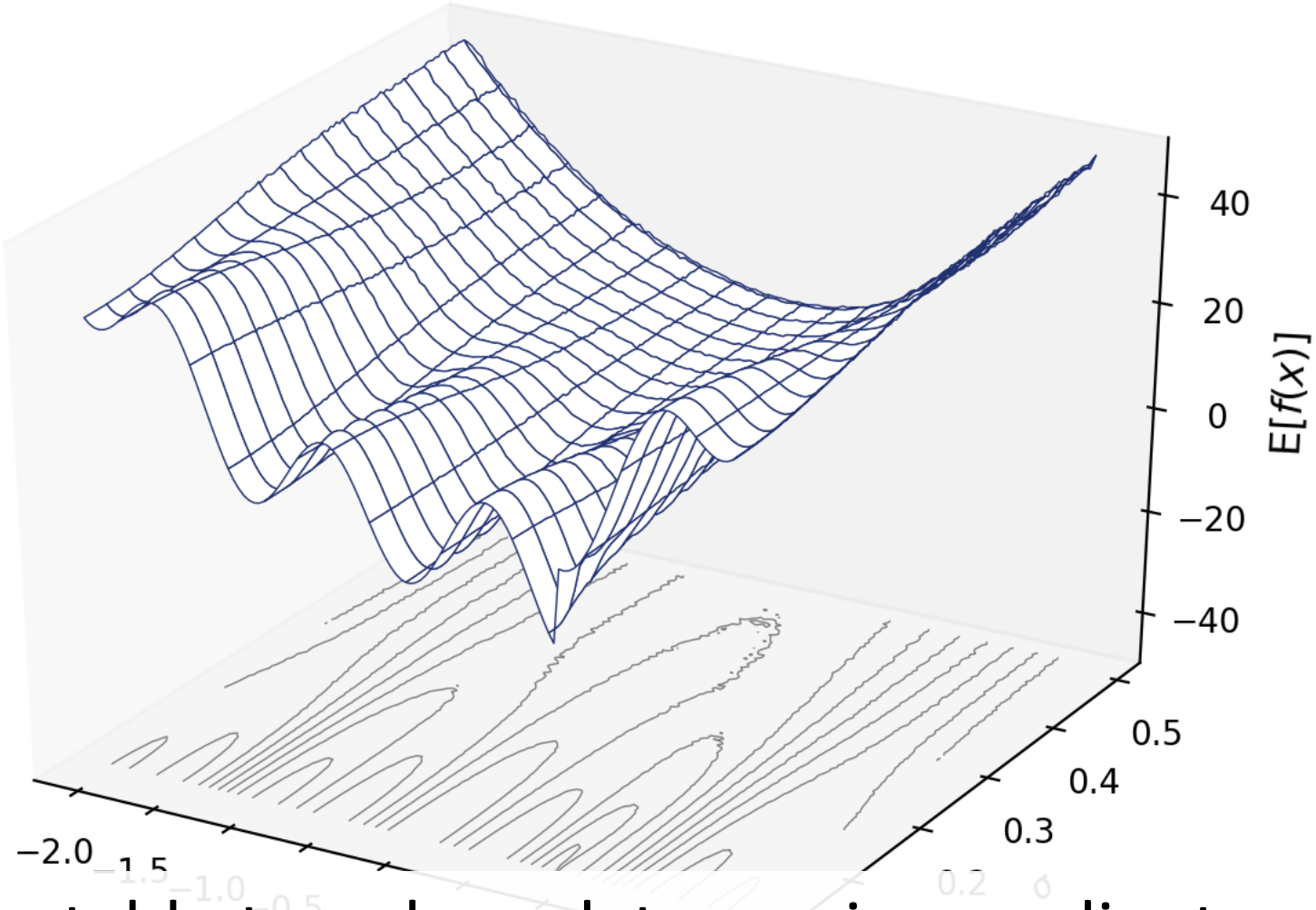




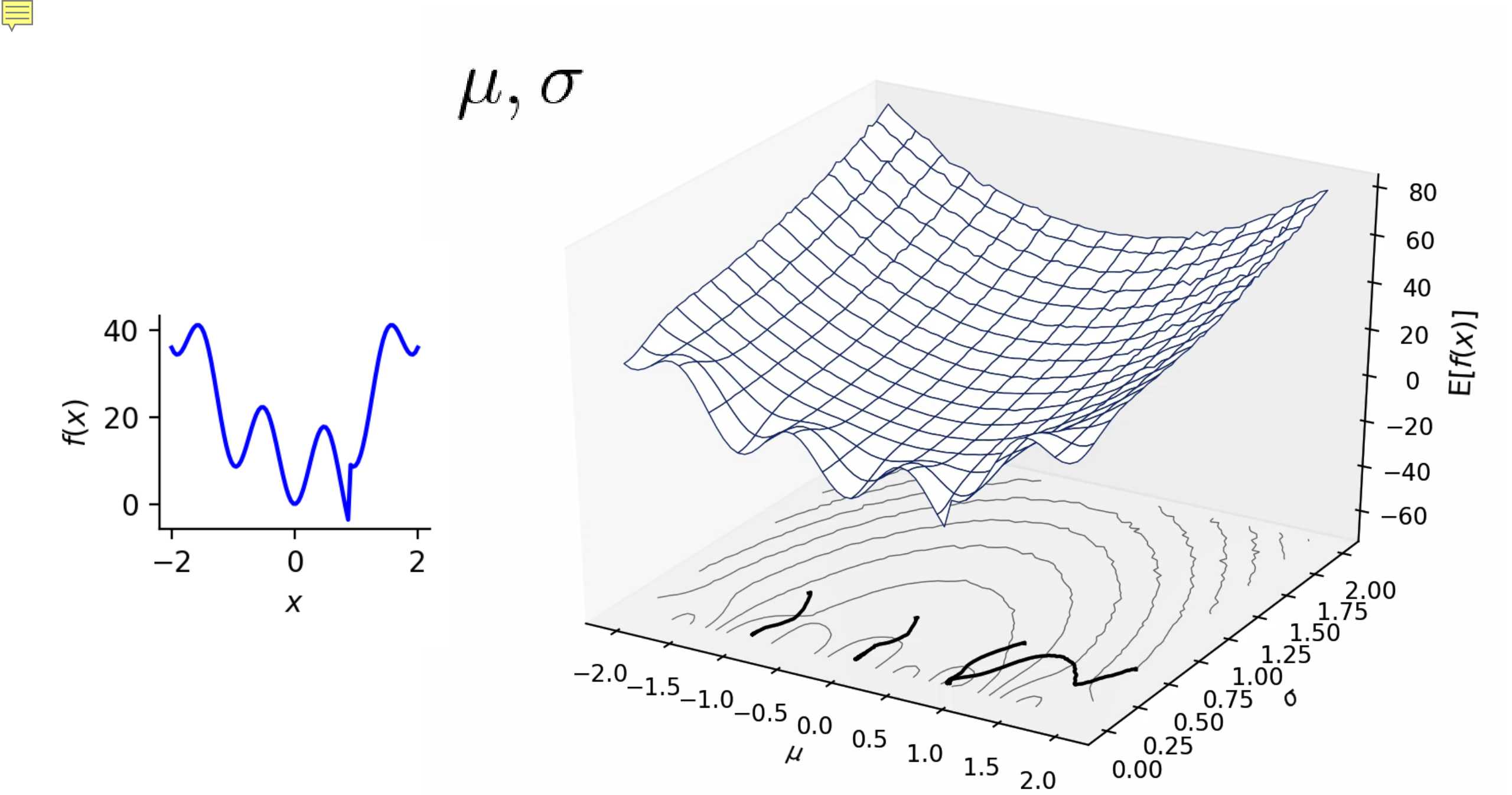
Gradient estimated from samples

Optimize both μ, σ, start with large σ





In practice, it may be more stable to only update μ using gradient, and linearly or exponentially drop σ towards zero.



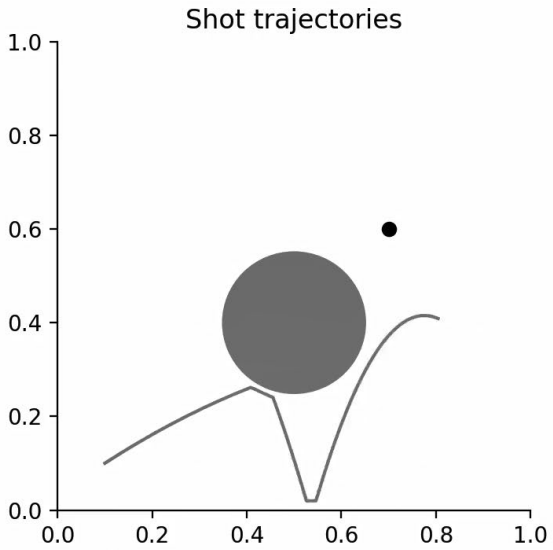
Video: Optimizing with different initial μ, σ. Large sigma: all runs converge to same optimum, independent of initial μ.

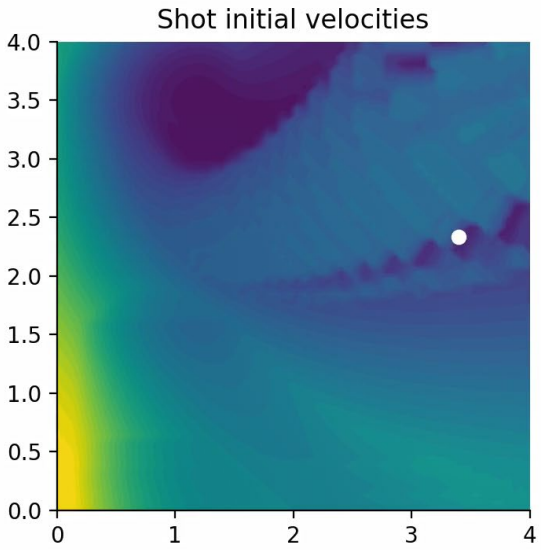
Typical optimization interface

```
shotVels=optimizer.ask()
fvals,trajectories=simulateBatch(shotVels)
loss=optimizer.tell(fvals)
```

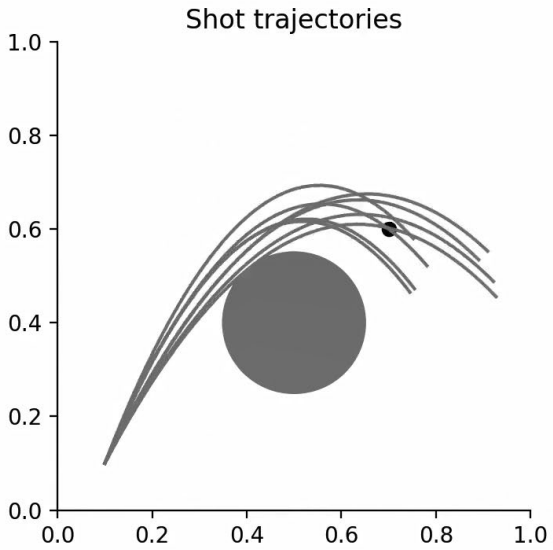
Didactic example: Ballistic shot with collisions

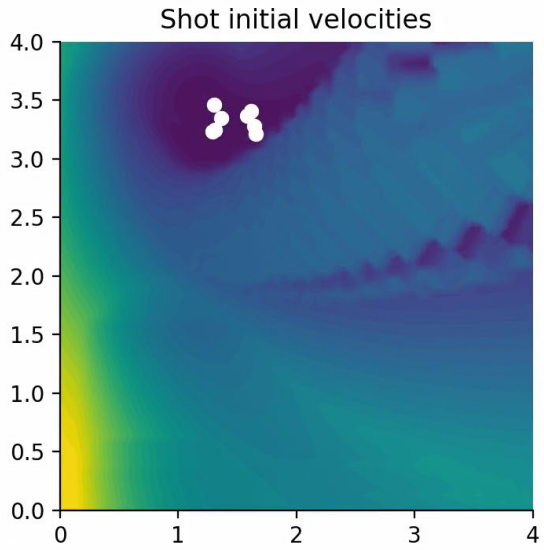
Adam (optimize f(x))





Adam (optimize E[$f(x)$])





How to estimate the gradient from samples?


WARNING
Mathematics
Ahead

Gradient estimators

Score function estimator, Variational Optimization, REINFORCE (pretty much all RL):

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[f(\mathbf{x})] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i)\nabla_\theta \log p_\theta(\mathbf{x}_i)$$

$$\theta = [\mu, \sigma]$$

Gradient estimators

Score function estimator, Variational Optimization, REINFORCE (pretty much all RL):

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[f(\mathbf{x})] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i)\nabla_\theta \log p_\theta(\mathbf{x}_i)$$

Mean over *N* samples,
$\mathbf{x}_i$ sampled from $p_\theta(\mathbf{x})$

Gradient estimators

Score function estimator, Variational Optimization, REINFORCE (pretty much all RL):

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[f(\mathbf{x})] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i)\nabla_\theta \log p_\theta(\mathbf{x}_i)$$

Which way to change θ to increase the probability of sampled $\mathbf{x}_i$

Gradient estimators

Score function estimator, Variational Optimization, REINFORCE (pretty much all RL):

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[f(\mathbf{x})] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i)\nabla_\theta \log p_\theta(\mathbf{x}_i)$$

Increase the probability of positive $f(\mathbf{x})$, decrease the probability of negative $f(\mathbf{x})$

Gradient estimators

Score function estimator, Variational Optimization, REINFORCE (pretty much all RL):

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[f(\mathbf{x})] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i)\nabla_\theta \log p_\theta(\mathbf{x}_i)$$

Increase the probability of positive $f(\mathbf{x})$, decrease the probability of negative $f(\mathbf{x})$

See slide notes for links to more details, e.g., the derivation of this identity through the so-called log derivative trick.

Gradient estimators

Reparameterization trick:

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[f(\mathbf{x})] = \nabla_\theta \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[f(\mathbf{x}(\mathbf{z},\theta))]$$

Gradient estimators

Reparameterization trick:

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[f(\mathbf{x})] = \nabla_\theta \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[f(\mathbf{x}(\mathbf{z},\theta))] \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta f(\mathbf{x}_i(\mathbf{z}_i,\theta))$$

Compute gradient at all sampled $\mathbf{x}_i$ and average over the batch

Gradient estimators

Reparameterization trick:

$$\nabla_\theta \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[f(\mathbf{x})] = \nabla_\theta \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[f(\mathbf{x}(\mathbf{z},\theta))] \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta f(\mathbf{x}_i(\mathbf{z}_i,\theta))$$

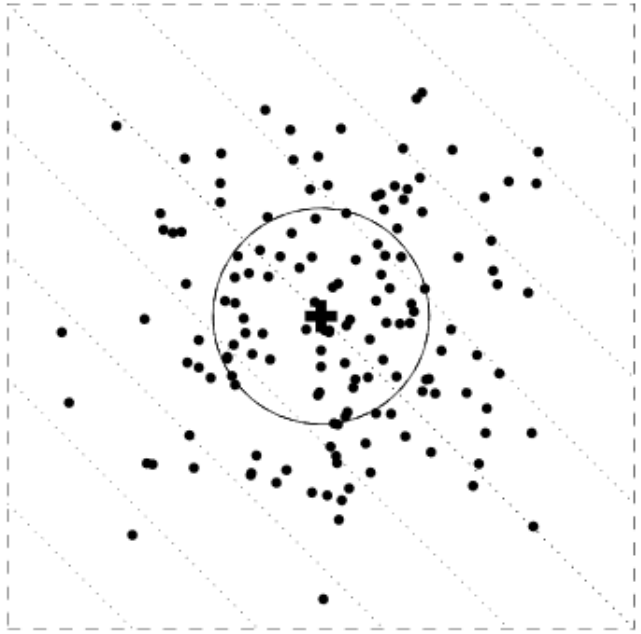
$$\theta = [\mu, \sigma], \quad \mathbf{z} \sim N(\mathbf{z}; 0, \mathbf{I}), \quad \mathbf{x} = \mu + \sigma\mathbf{z}$$

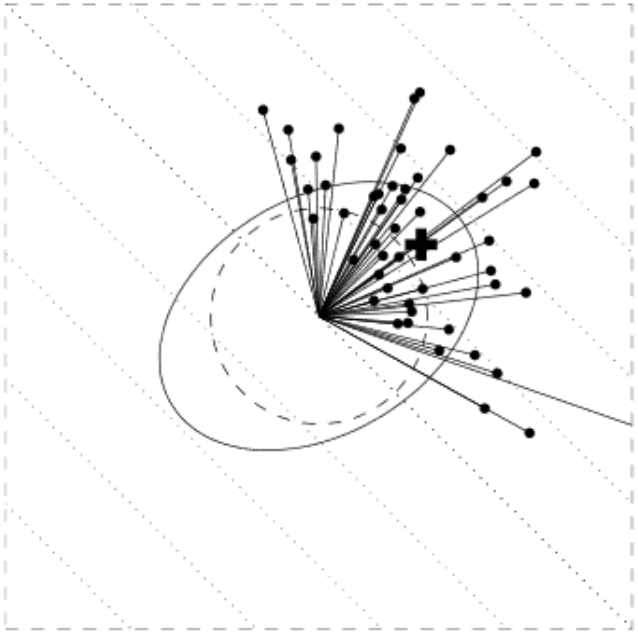

WA NG
M ematics
Ahead

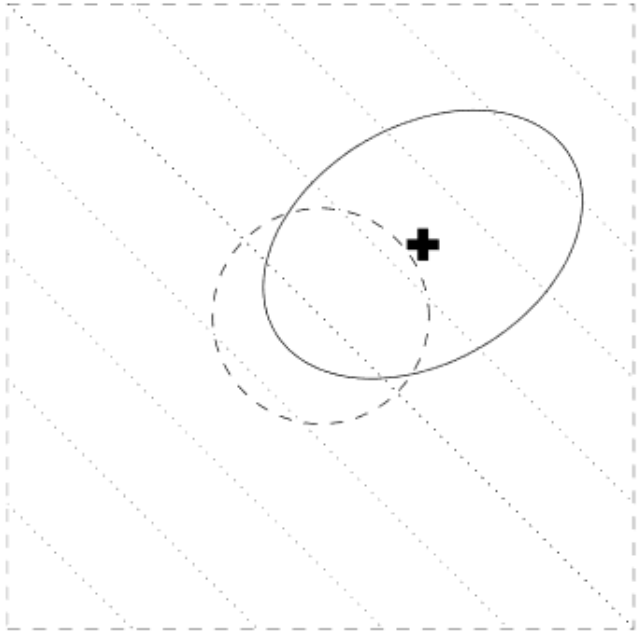
Practical algorithms

- Taking a single gradient step for each batch of samples can be expensive
- Proximal Policy Optimization (PPO): take multiple steps for each batch. This can be unstable => limit the difference between the old and new distributions or policies
- Covariance Matrix Adaptation Evolution Strategy (CMA-ES): convert $f(x)$ into weights, e.g., sort and use the ranks so that the best samples in a batch have largest weights. Then fit the normal distribution to the remaining samples. In practice, quite similar to PPO

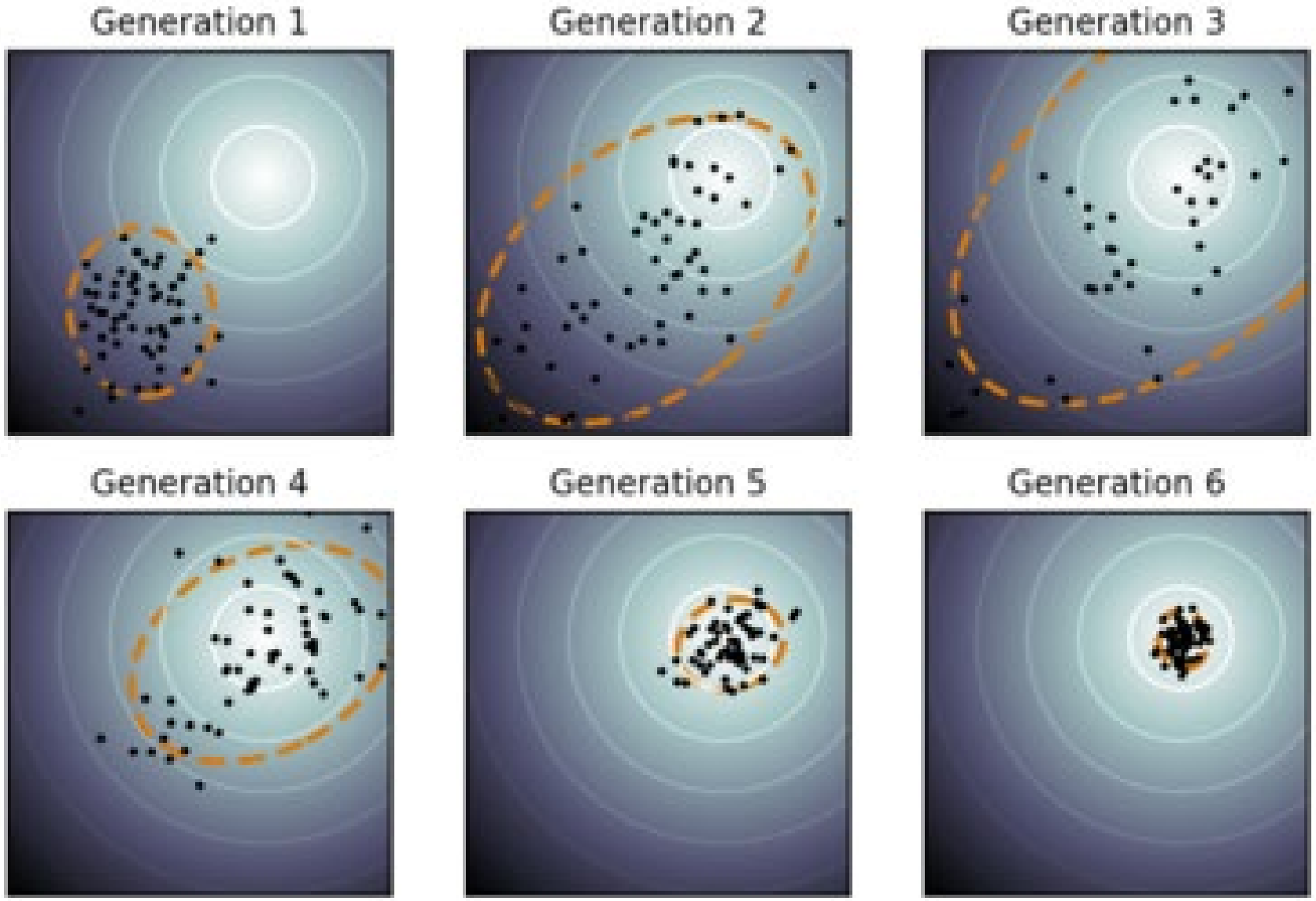
CMA-ES in a nutshell



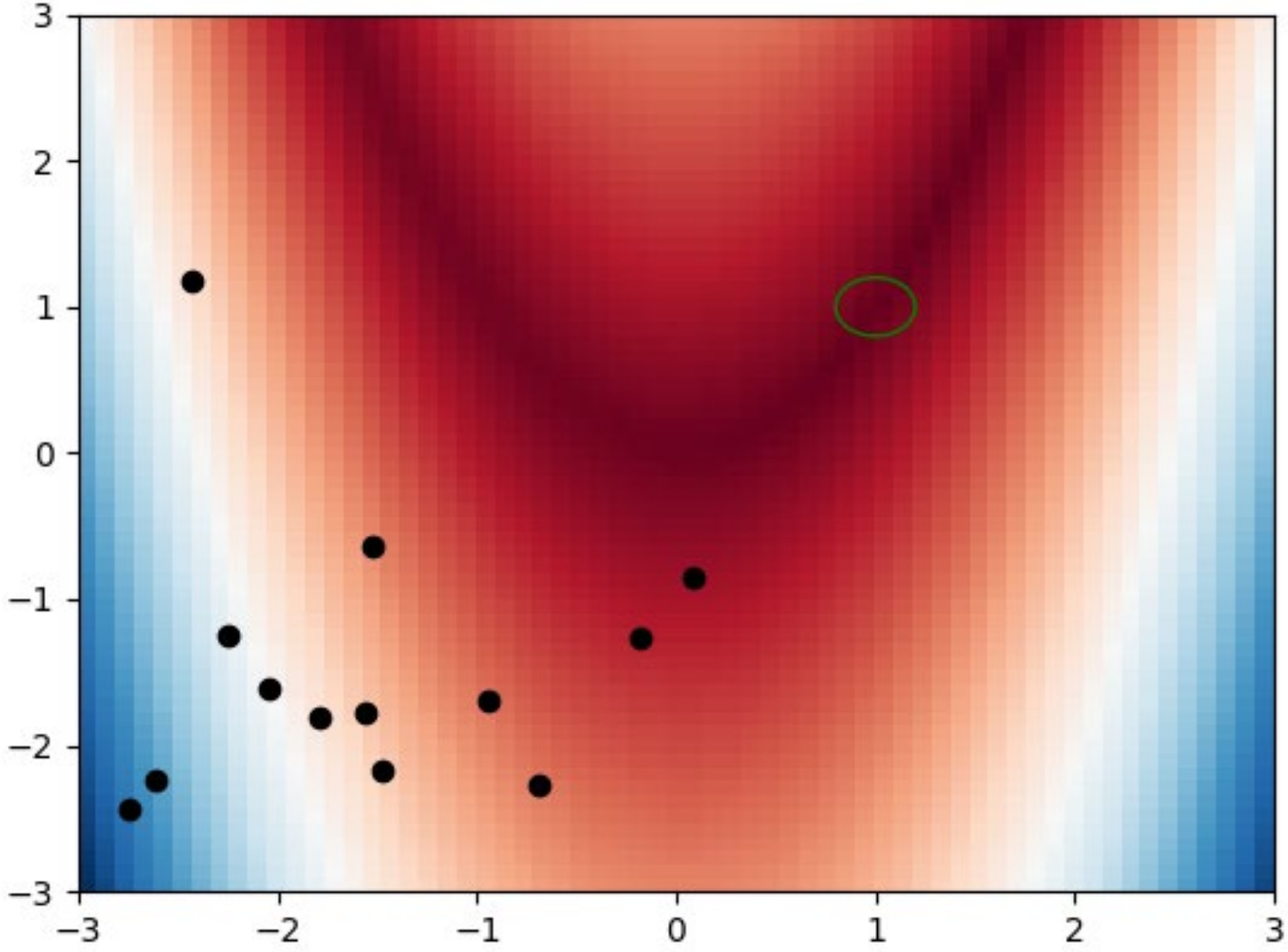


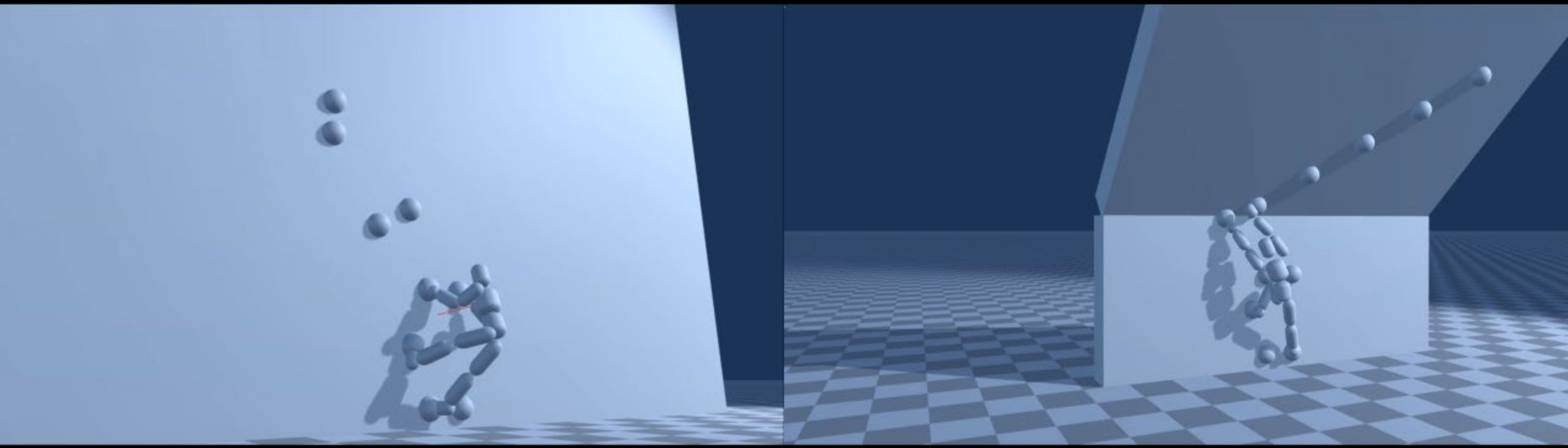


CMA-ES (Hansen & Ostenmeier 2001)



CMA-ES





The CMA Evolution Strategy: A Tutorial

Nikolaus Hansen
Inria
Research centre Saclay–Île-de-France
Université Paris-Saclay, LRI

Contents

Limited-Memory Matrix Adaptation for Large Scale Black-box Optimization

Ilya Loshchilov
Research Group on Machine Learning
for Automated Algorithm Design
University of Freiburg, Germany
ilya.loshchilov@gmail.com

Tobias Glasmachers
Institut für Neuroinformatik
Ruhr-Universität Bochum, Germany
tobias.glasmachers@ini.rub.de

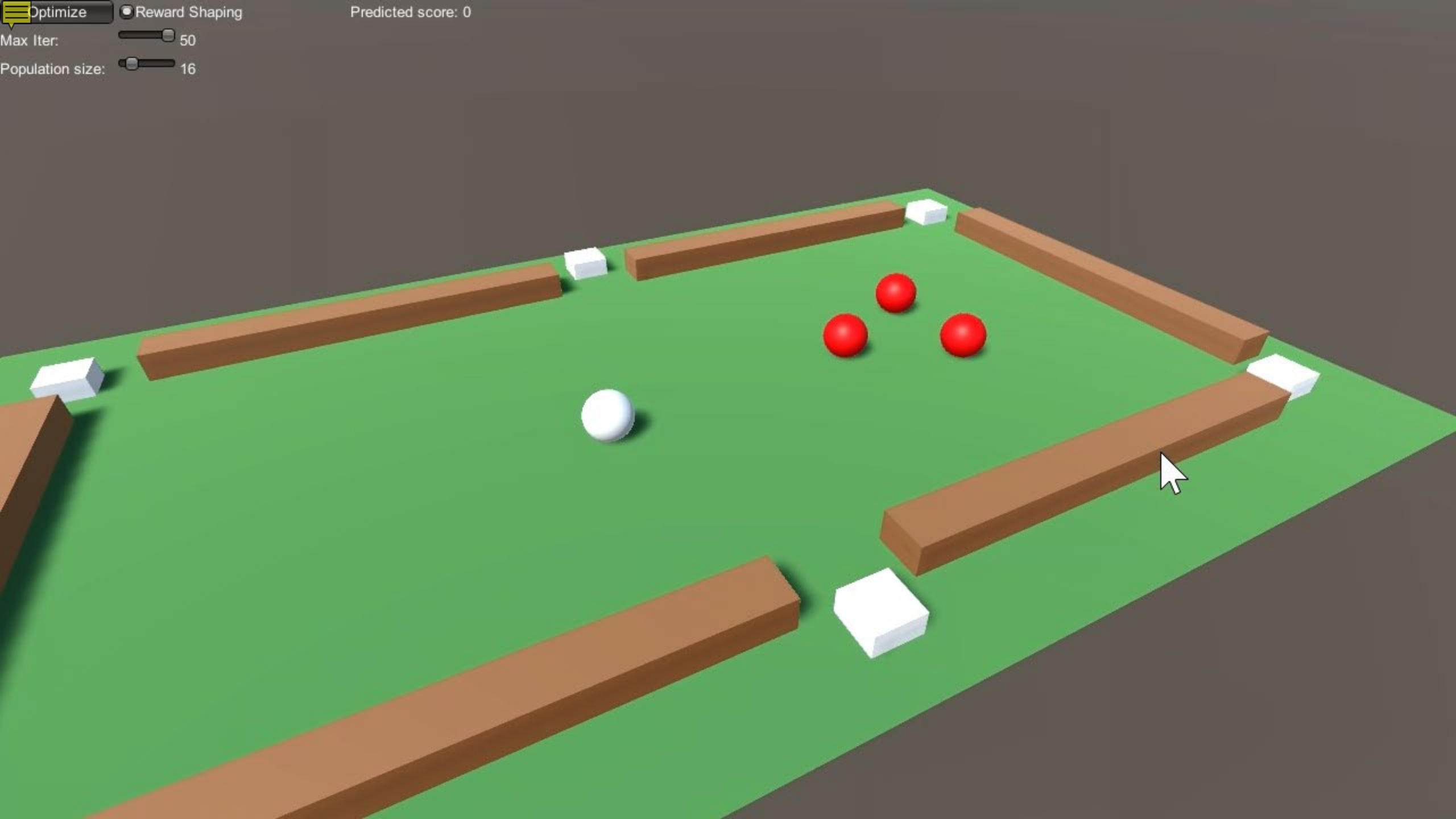
Hans-Georg Beyer
Research Center Process and Product Engineering
Vorarlberg University of Applied Sciences, Dornbirn, Austria
hans-georg.beyer@fhv.at

Abstract

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a popular method to deal with nonconvex and/or stochastic optimization problems when the gradient information is not available. Being based on the CMA-ES, the recently proposed Matrix Adaptation Evolution Strategy (MA-ES) provides a rather surprising result that the covariance matrix and all associated operations (e.g., potentially unstable eigendecomposition) can be replaced in the CMA-ES by a updated transformation matrix without any loss of performance. In order to further simplify MA-ES and reduce its $\mathcal{O}(n^2)$ time and storage complexity to $\mathcal{O}(n\log(n))$, we present the Limited-Memory Matrix Adaptation Evolution Strategy (LM-MA-ES) for efficient zeroth order large-scale optimization. The algorithm demonstrates state-of-the-art performance on a set of established large-scale benchmarks. We explore the algorithm on the problem of generating adversarial inputs for a (non-smooth) random forest classifier, demonstrating a surprising vulnerability of the classifier.

LM-MA-ES (Loschilov et al. 2017)

- Limited memory variant which scales to millions of variables, O(N log(N))
- LM-MA-ES works even in generating adversarial images, something that had previously only been done with gradient-based optimization
- Unity implementation provided in the course repository

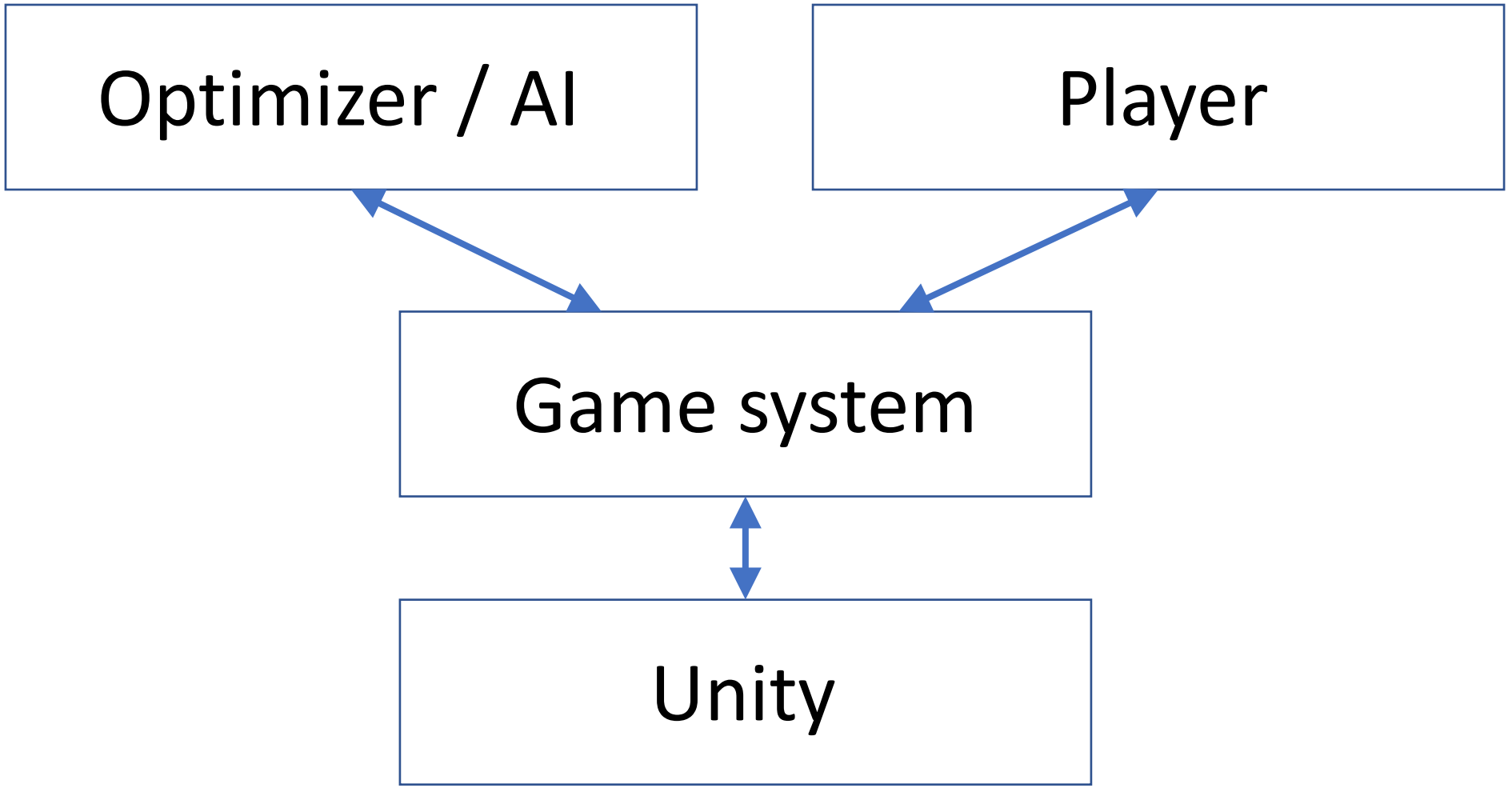

Optimize
Reward Shaping
Predicted score: 0
Max Iter:
50
Population size:
16

(LM-)MAES Unity interface

```
public interface IMAES
{
    int recommendedPopulationSize(int dim);
    void init(int dim, int populationSize, double[] initialMean, float init
    void generateSamples(OptimizationSample[] samples, int nInitialGuesses
    double update(OptimizationSample[] samples);
    double[] getBest();
    double getBestObjectiveFuncValue();
    void optimize(Func<double[], int, double> objectiveFunc, int maxIter);
}
```

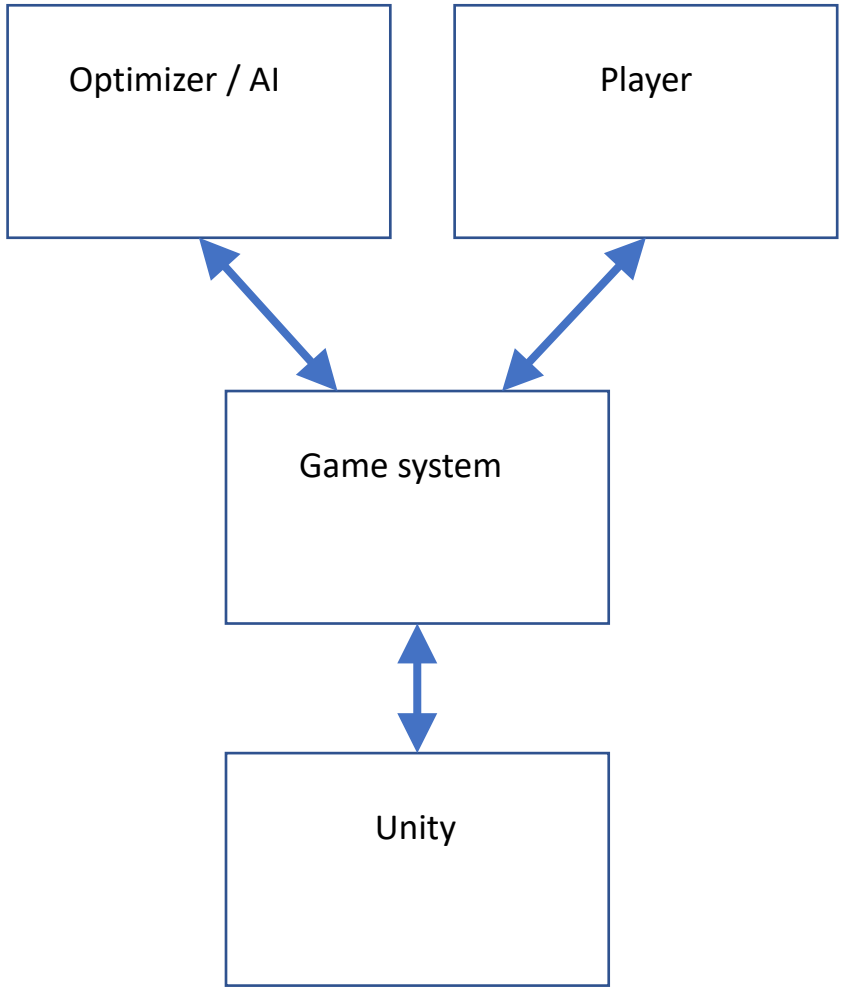
Plain C# implementation, no math or machine learning libraries needed.

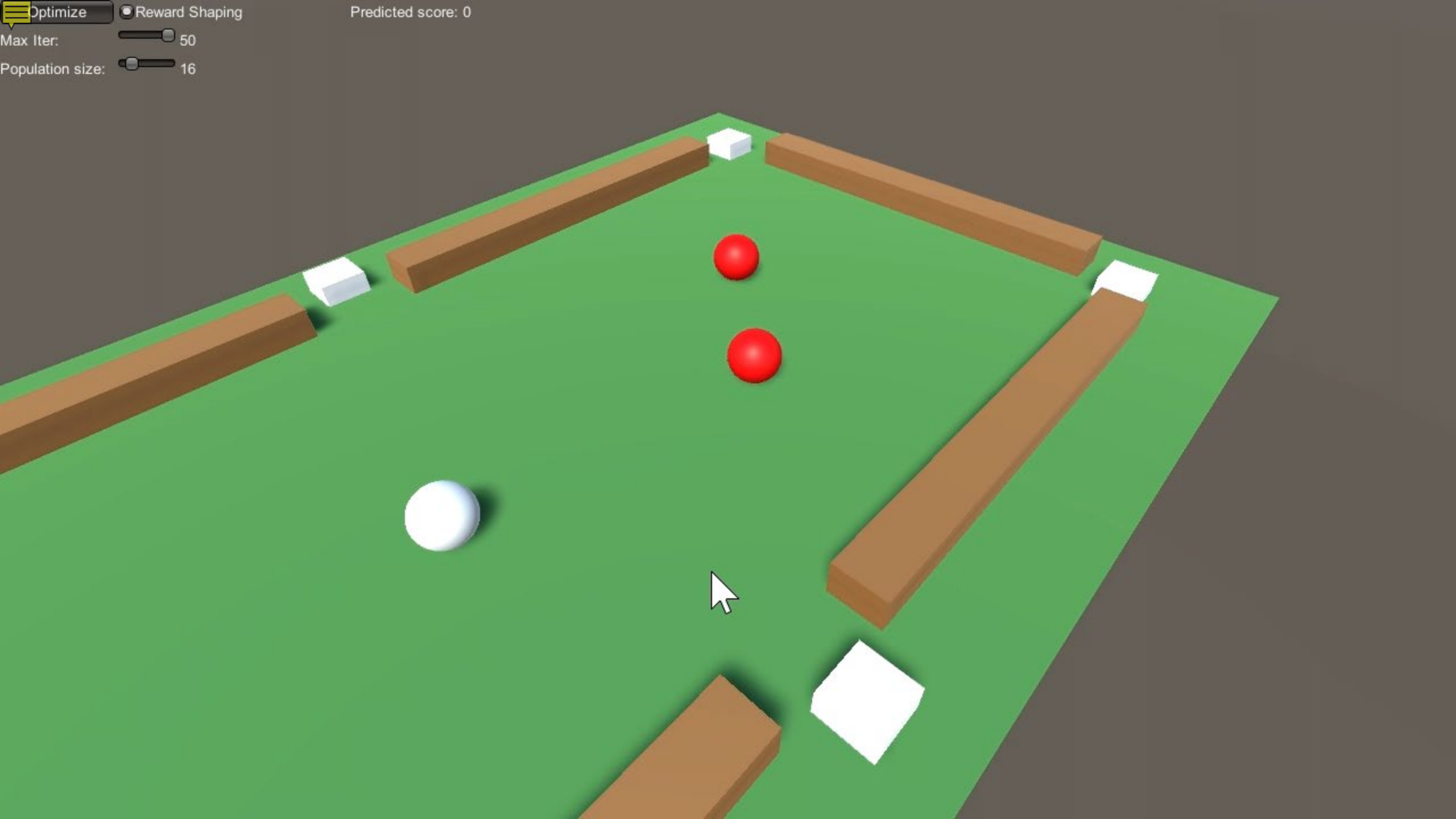
Design pattern: AI and Player as game system controllers



Game system

- Implement actions
- Simulate actions speculatively (for evaluating $f(\mathbf{x})$ without affecting system state)
- Save & restore state (needed for the above)
- The speculative simulation may also be useful for the player!




Optimize
Reward Shaping
Predicted score: 0
Max Iter:
50
Population size:
16

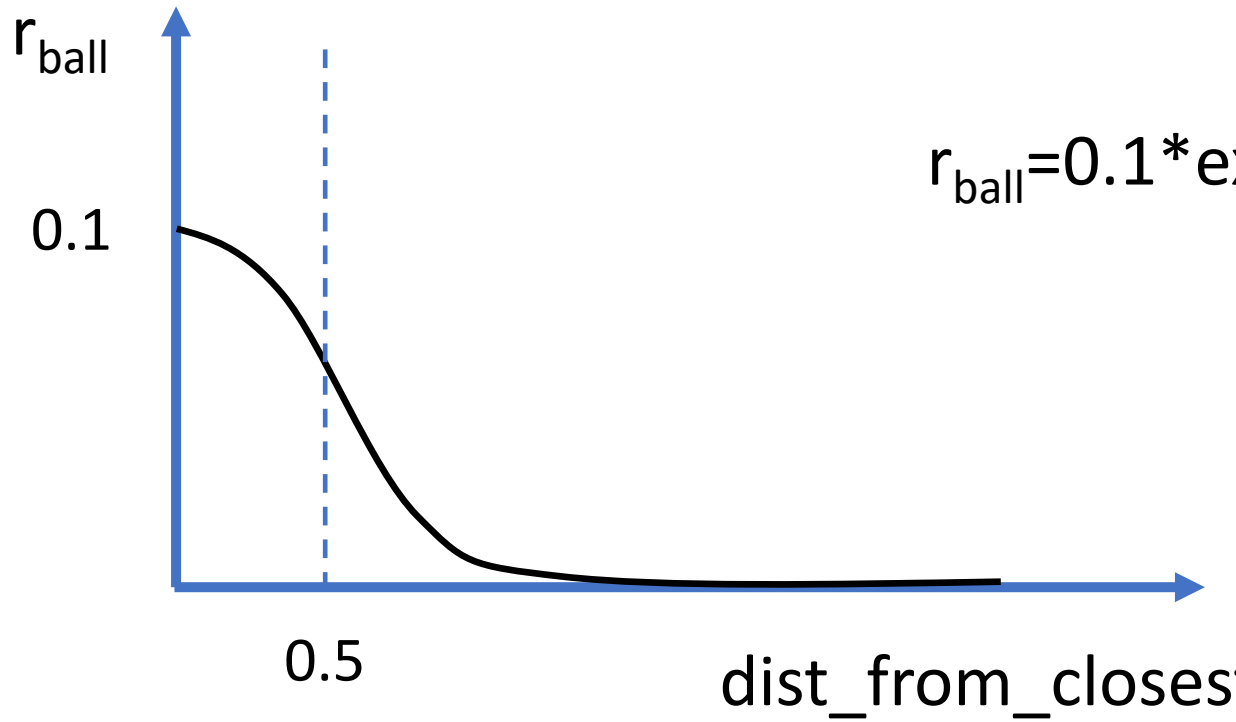
IntelligentPool code walkthrough...

Key things to remember in sampling-based optimization

- Larger sample batches (CMA-ES population size) => less gradient noise. Harder problems require more samples
- Reward design: avoid flat $f(\mathbf{x})$ with no gradients, e.g., simply giving a +1 score for each pocketed billiards ball
- Reward shaping, proxy objective: give a small bonus for balls that are close to pockets at the end of the shot

Reward shaping

- Simply giving a +1 score for each pocketed ball => mostly flat $f(\mathbf{x})$
- Reward shaping, proxy objective: give a small bonus for balls that are close to pockets at the end of the shot



$$r_{ball}=0.1*\exp(-0.5*dist\_from\_closest^2/0.5^2)$$

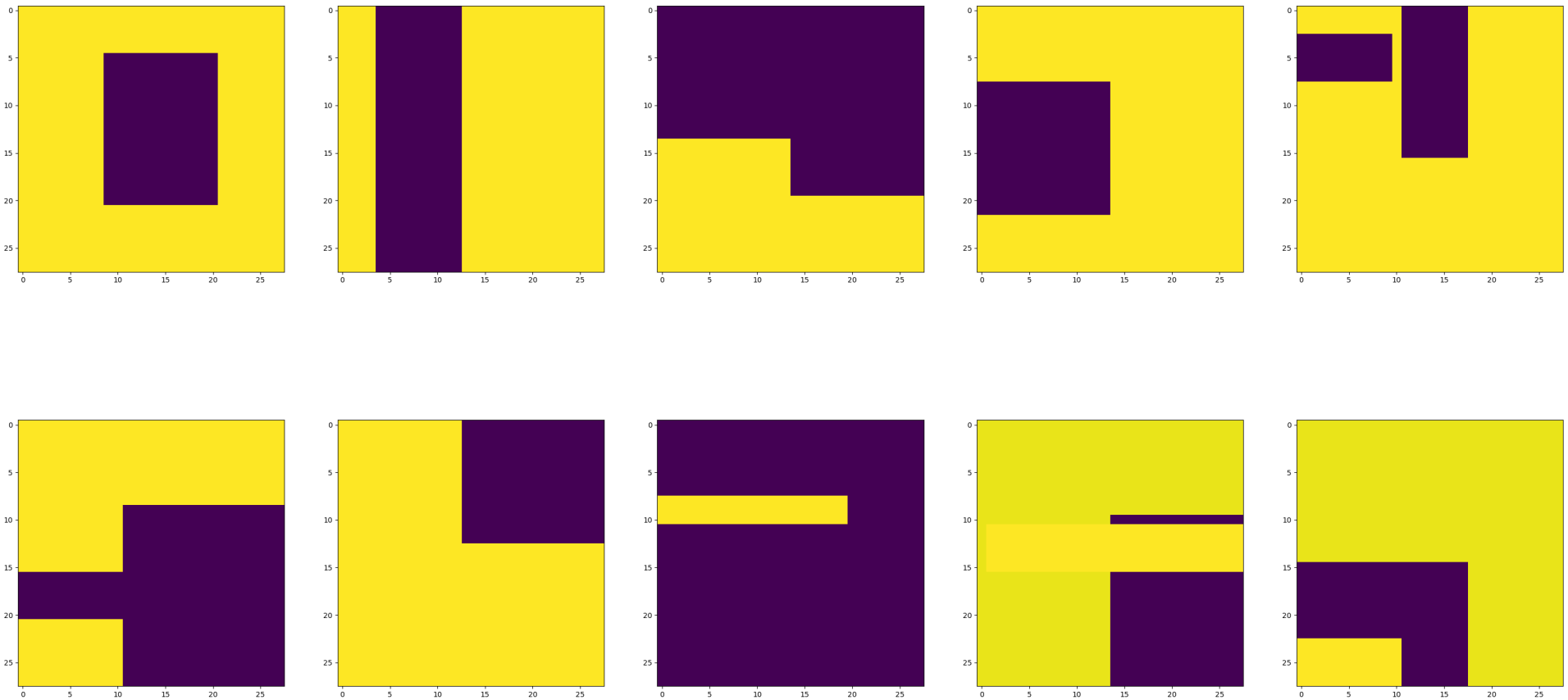
Parallel computing

- Sampling-based methods are usually trivially parallelizable
- Threading overhead negligible if $f(\mathbf{x})$ evaluation is costly
- However, physics simulators typically not thread-safe
- Unity: physics simulation automatically parallelized if groups of objects not interacting (not used in the billiards demo) => create copies of the scene that are placed well apart

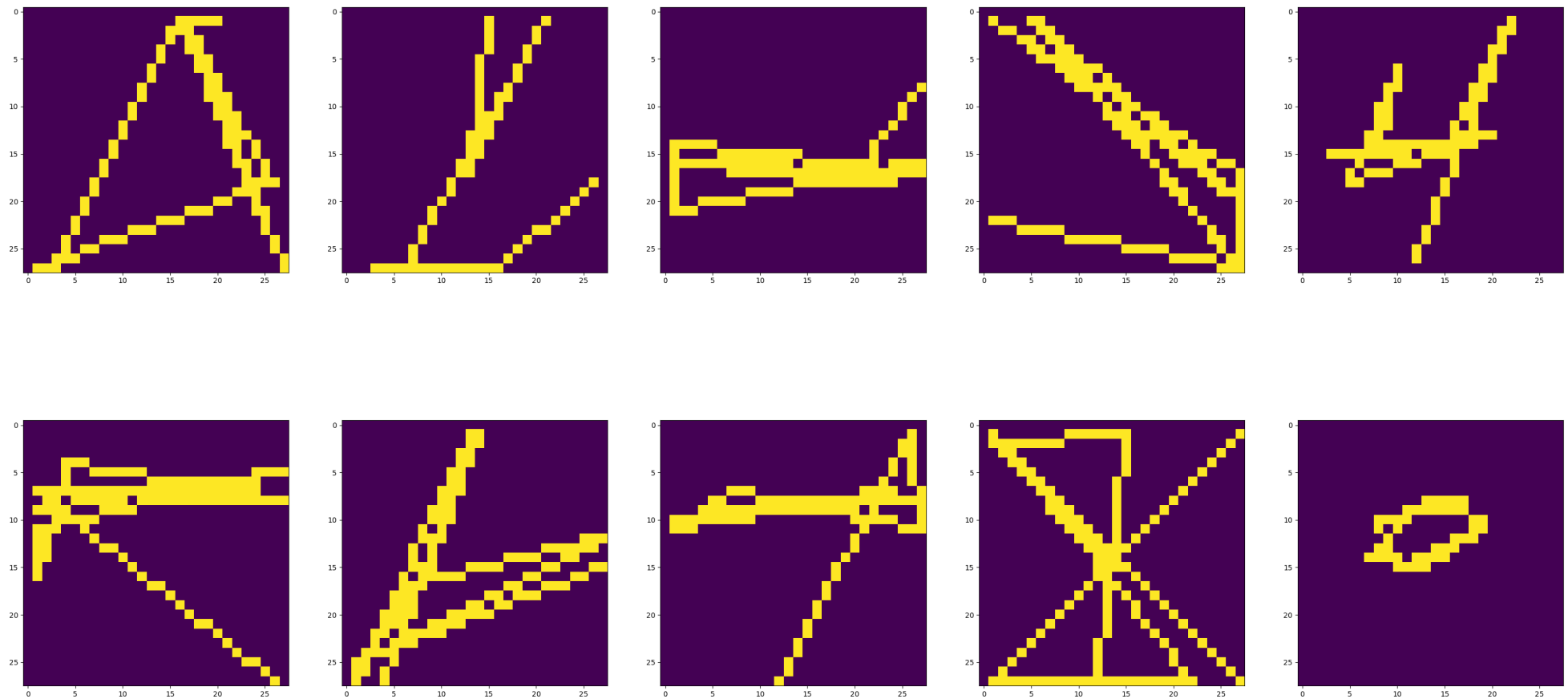
Exercise: CMA-ES for Abstract Adversarial Images

- **x** contains the parameters of a painting composed of primitives like rectangles (parameters: corner coordinates, color)
- $f(\mathbf{x})$ is the target class probability of a convolutional image classifier network, similar to what we trained in the day 1 exercise.
- Results tell us what the network finds relevant, sometimes aesthetically interesting. Could one produce high-quality abstract art with this method? Shadow puppetry with posed animation characters?
- Bonus: optimize the adversarial images pixel-by-pixel using Tensorflow's Adam
- Jupyter notebooks: CMA-ES_Art.ipynb, AdversarialMNIST.ipynb

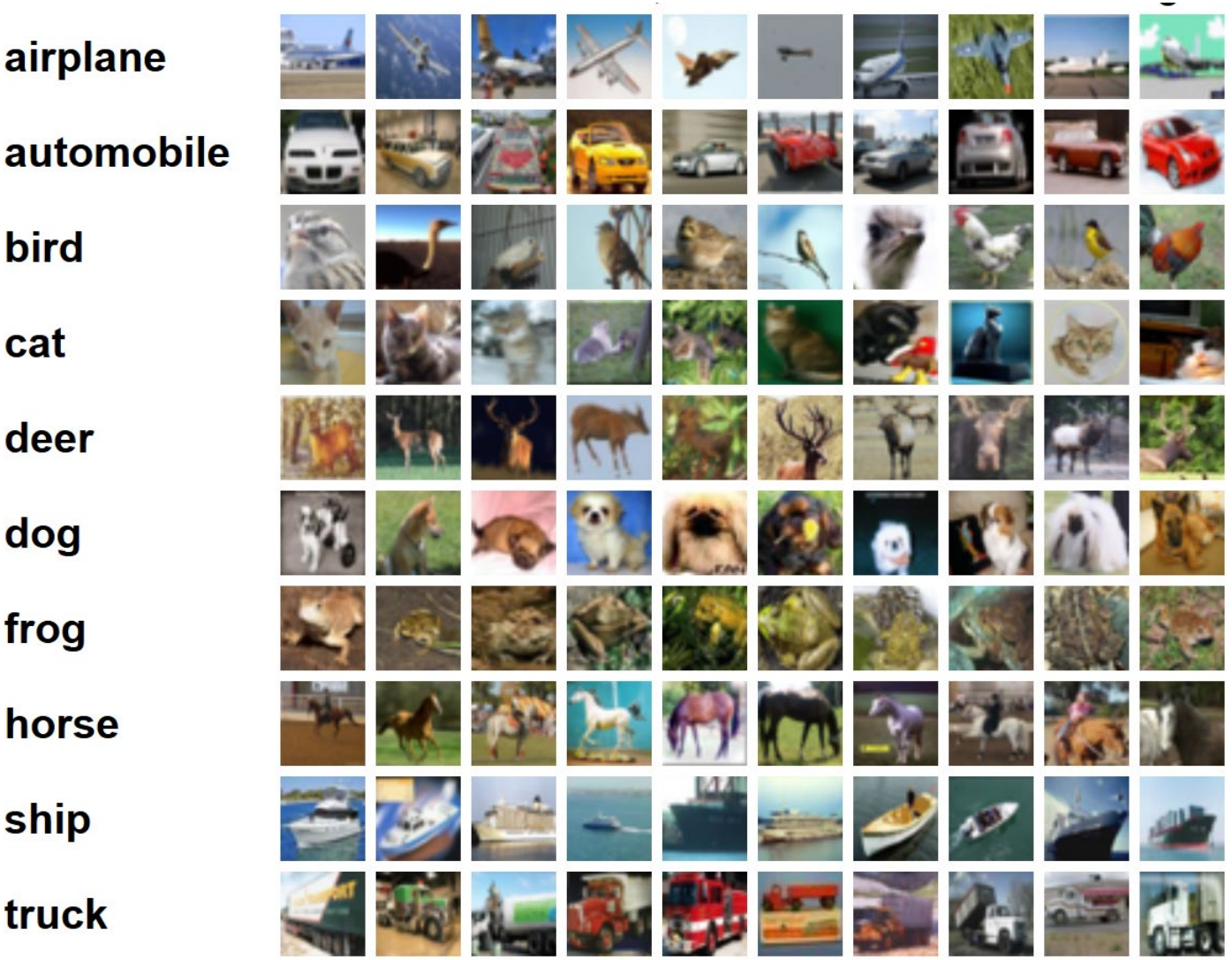
MNIST using 3 rectangles



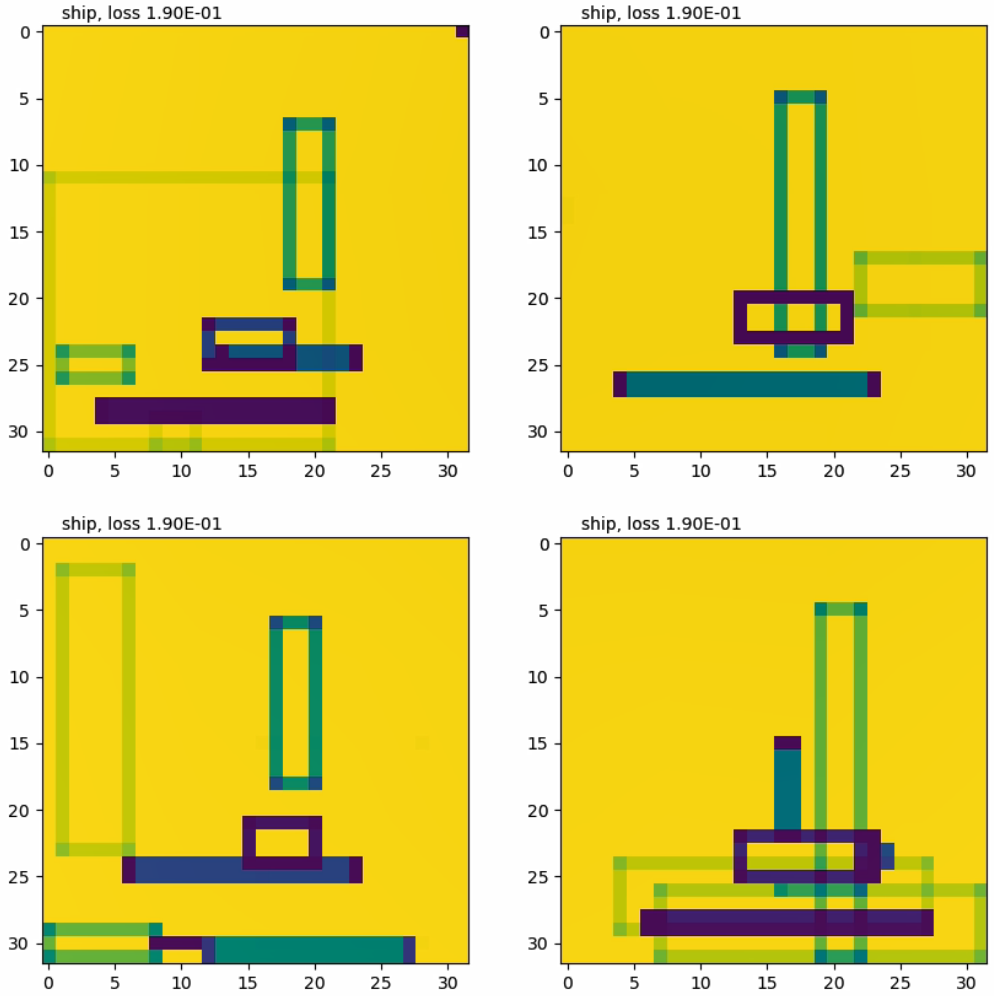
MNIST using 7 lines



CIFAR-10 dataset



Optimizing CIFAR-10 images



Same idea with a bigger neural network and more sophisticated drawing primitives

Perception Engines: cello, cabbage, hammerhead shark, iron, tick, starfish, binoculars, measuring cup, blow dryer, and jack-o-lantern

https://medium.com/@tom_25234/synthetic-abstractions-8f0e8f69f390

Discrete/combinatorial optimization

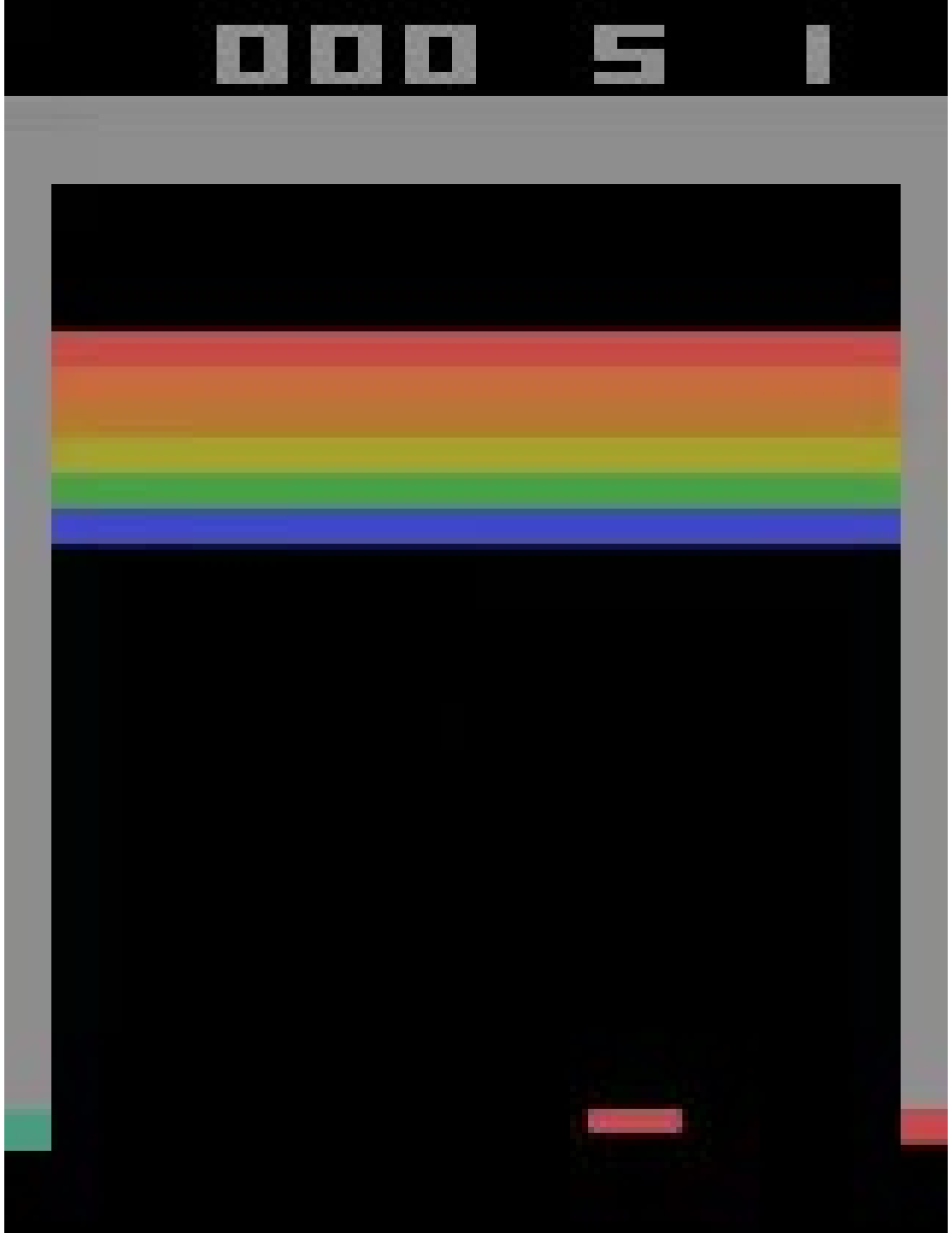
Discrete-valued optimization

- Q: How to compute gradient of $f(x)$ with respect to discrete variables, e.g., which button to press when controlling a game character?
- A: You can't, as per the definition of gradient.
- However, optimizing E[f(x)] works just as well if x is sampled from a discrete distribution that defines the probabilities of each button
- Again, even if f(x) is not smooth, the expectation can be smooth, and act as a bound that can be minimized to also minimize the true objective.

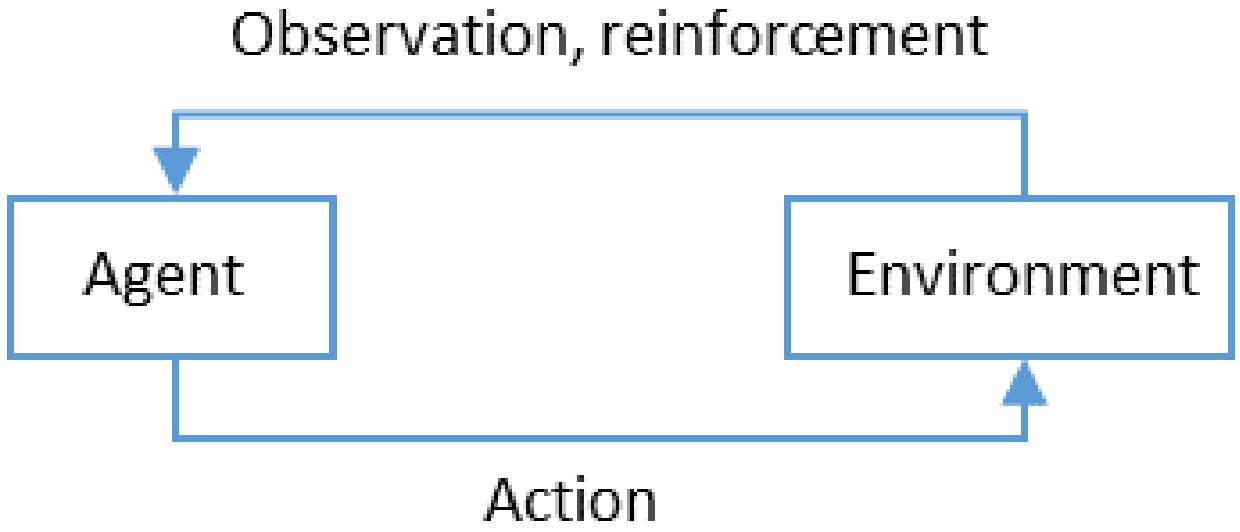
(Deep) Reinforcement learning

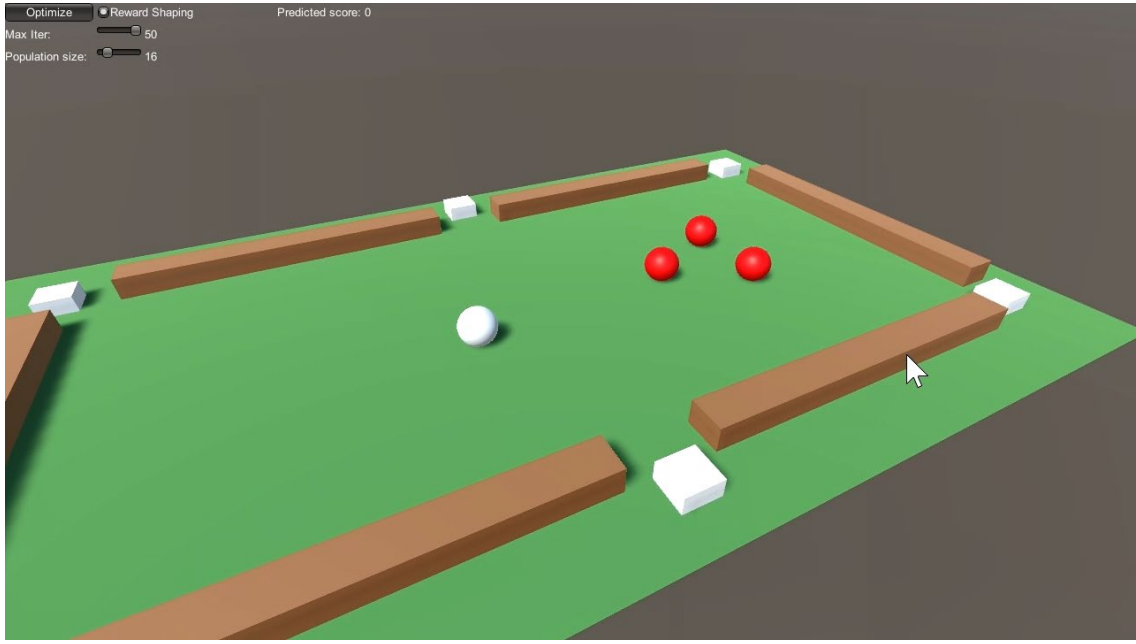
Mnih et al. 2015:

Human-level control of Atari games using Deep Reinforcement Learning (DRL)

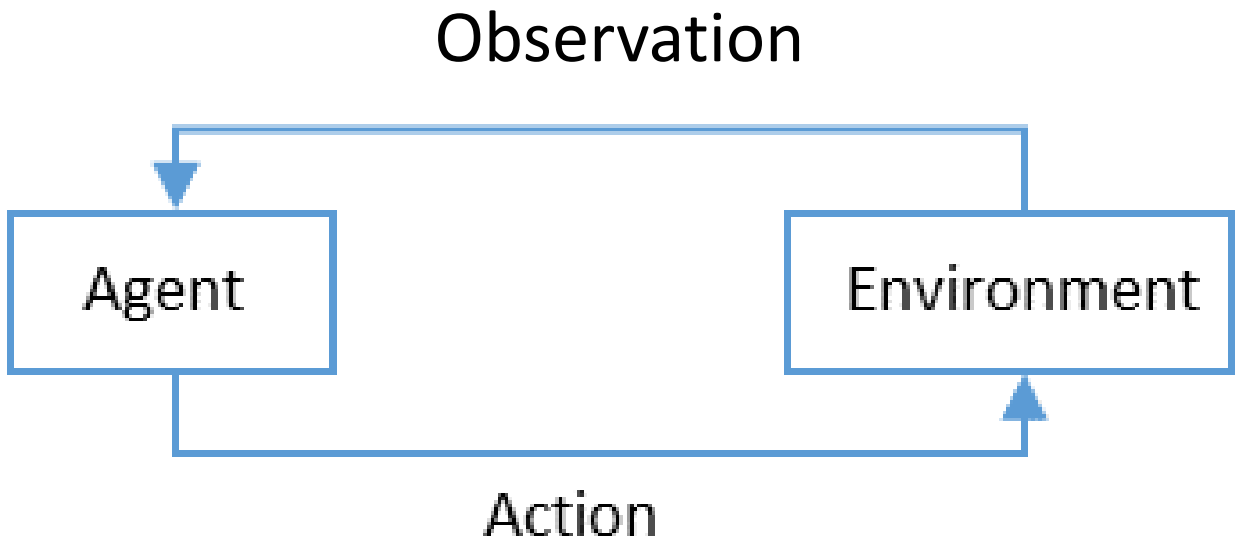


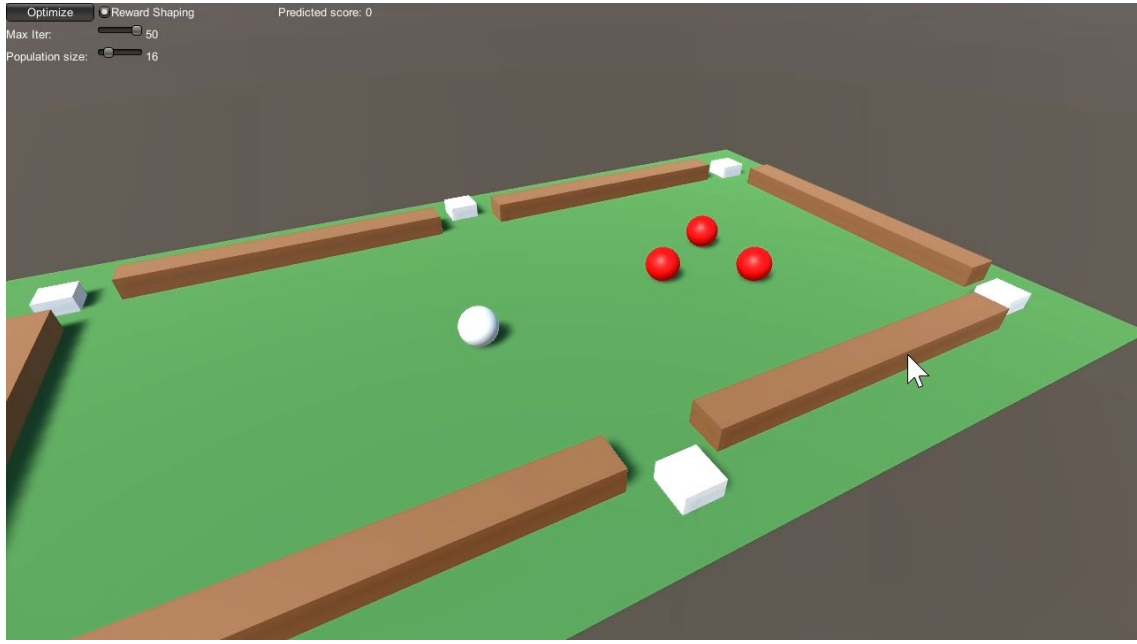
Real life: Countless variations of the same problems





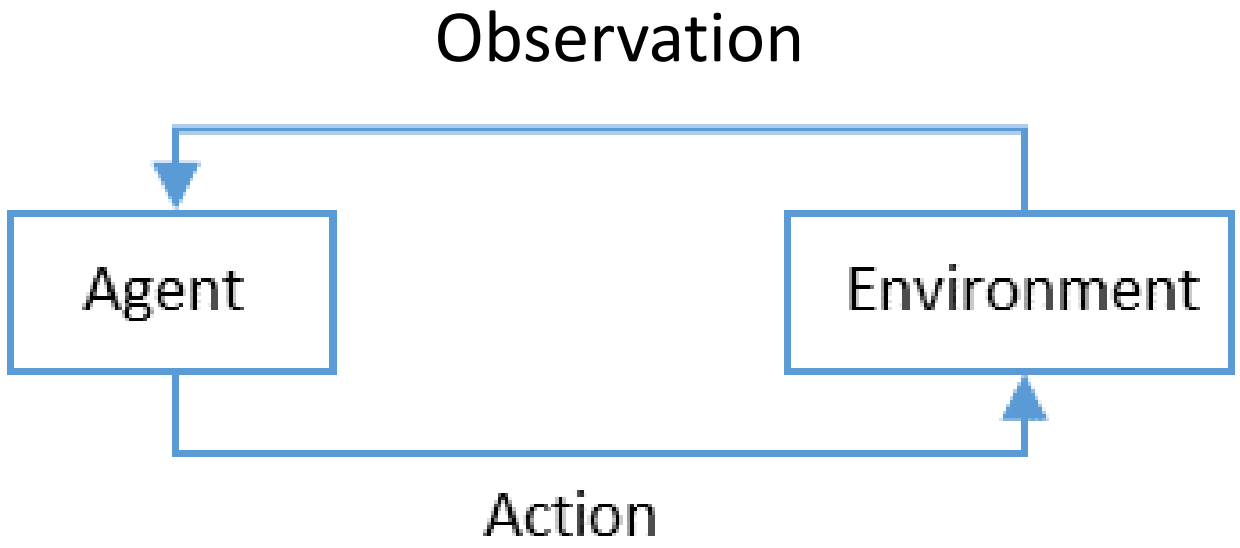
Real life: Countless variations of the same problems

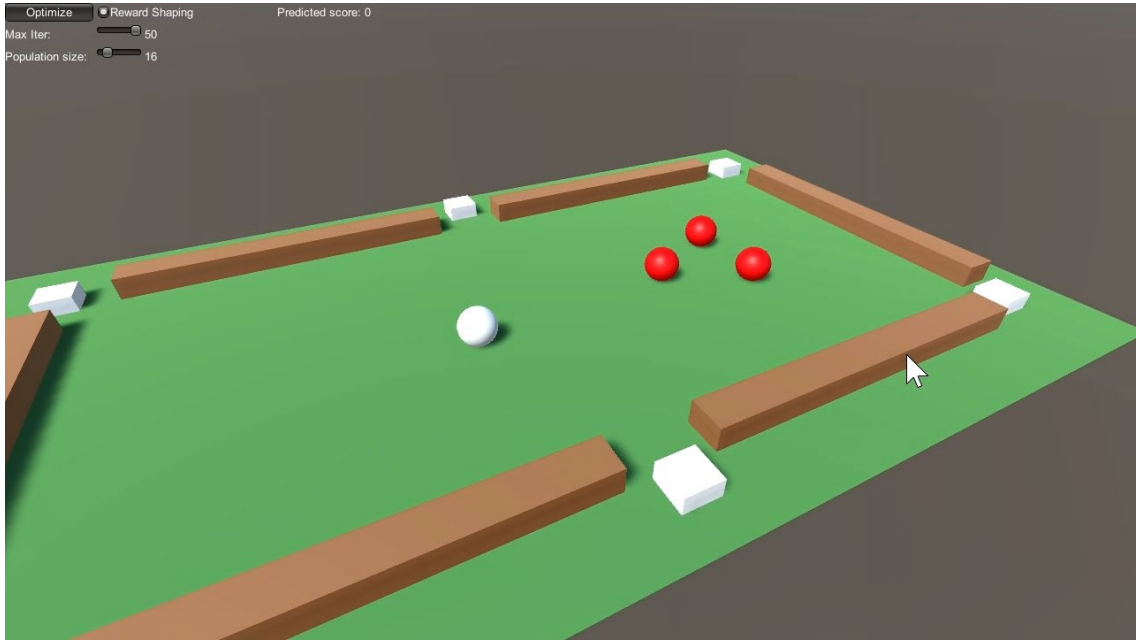




Observation: configuration of balls, i.e., problem definition for optimizing action

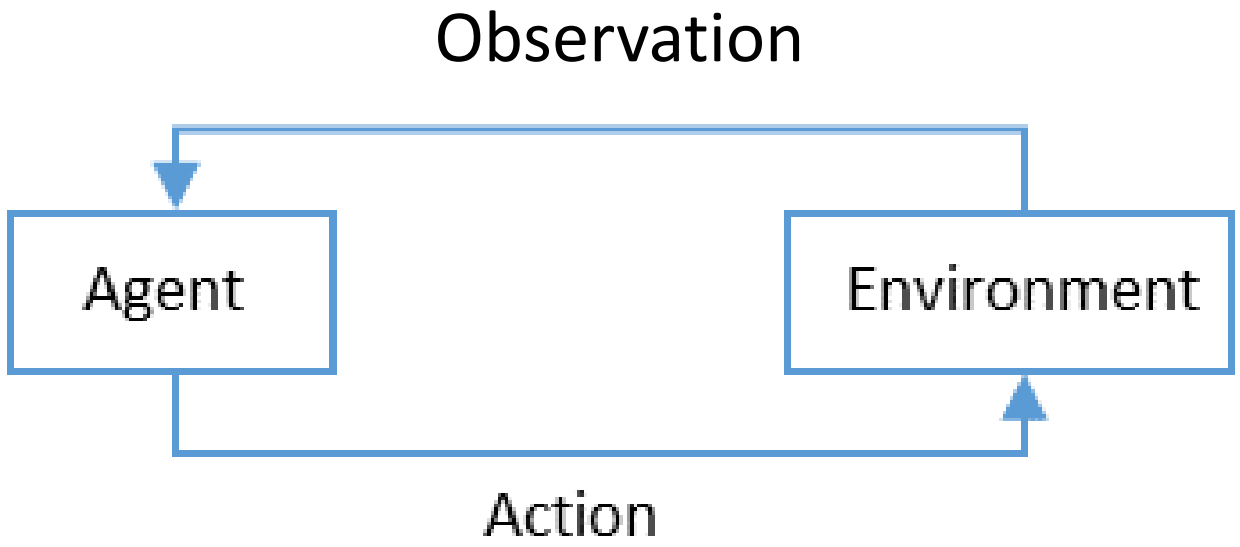
Real life: Countless variations of the same problems

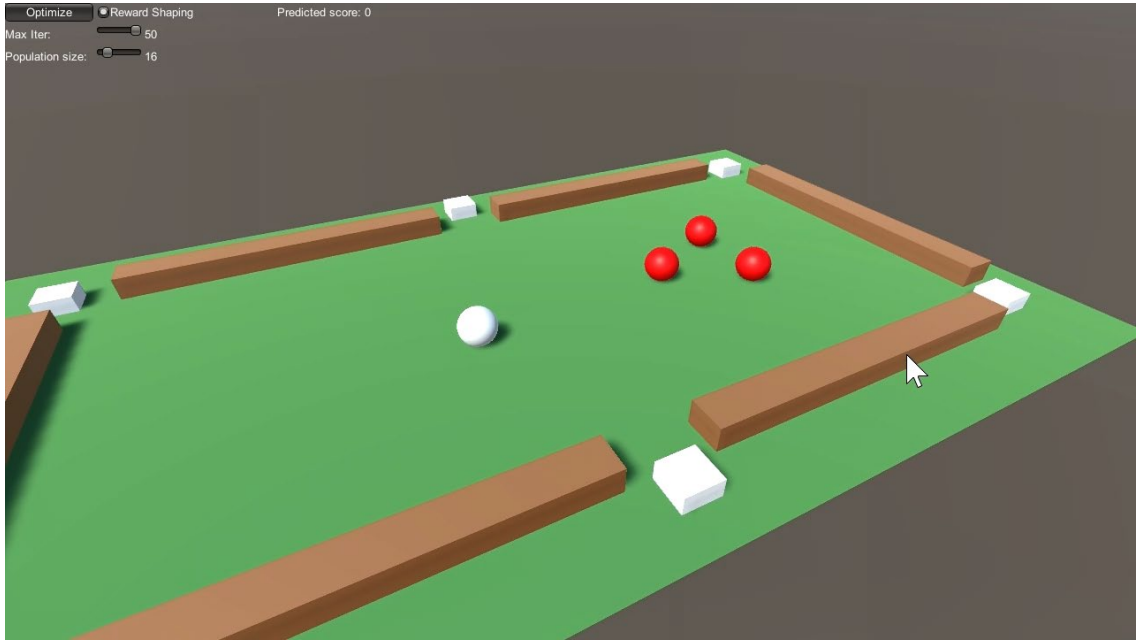




DRL goal: Instead of optimizing a single action **a**, optimize the parameters **θ** of a policy function $\mathbf{a}=\pi_{\boldsymbol{\theta}}(\mathbf{o})$ that gives the best action for any observation **o**.

Real life: Countless variations of the same problems

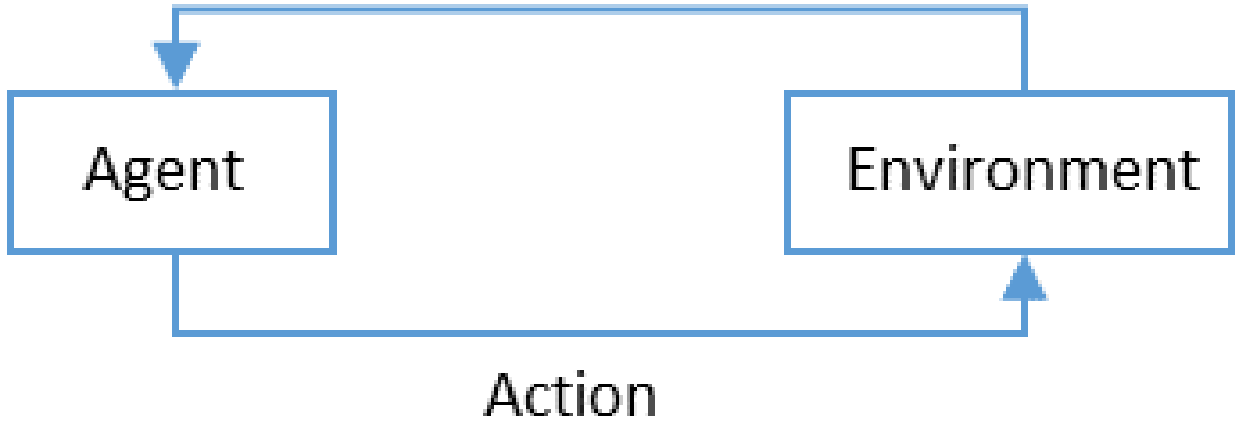


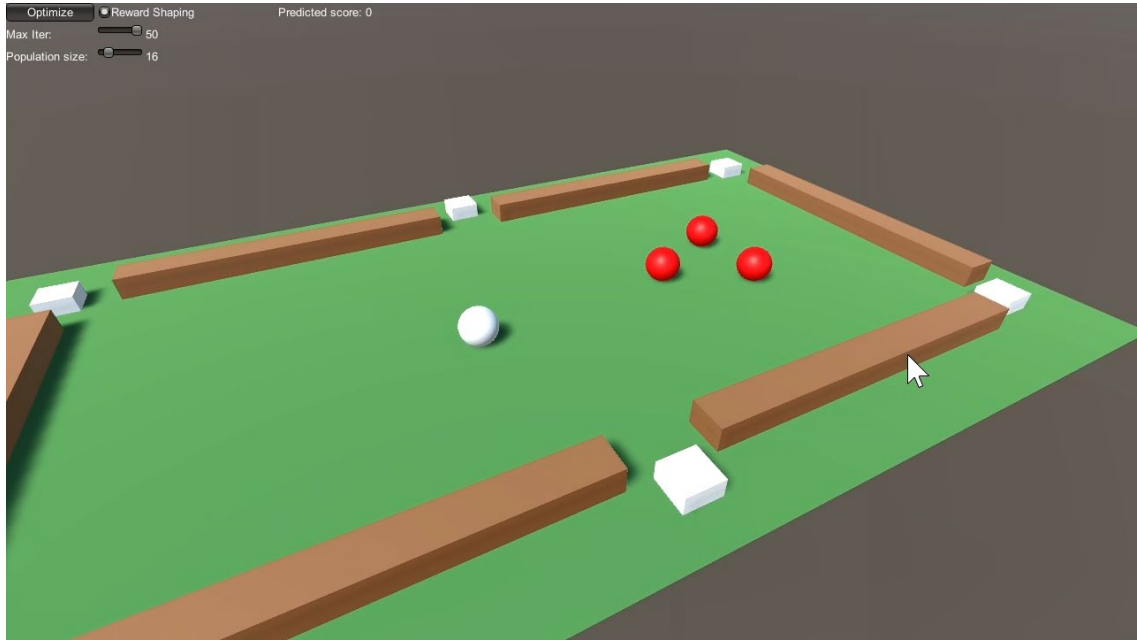


Essentially: Explore/sample different actions in different situations, memorize the best actions, and interpolate/extrapolate to be able to act in new situations.

Real life: Countless variations of the same problems

Observation and reward



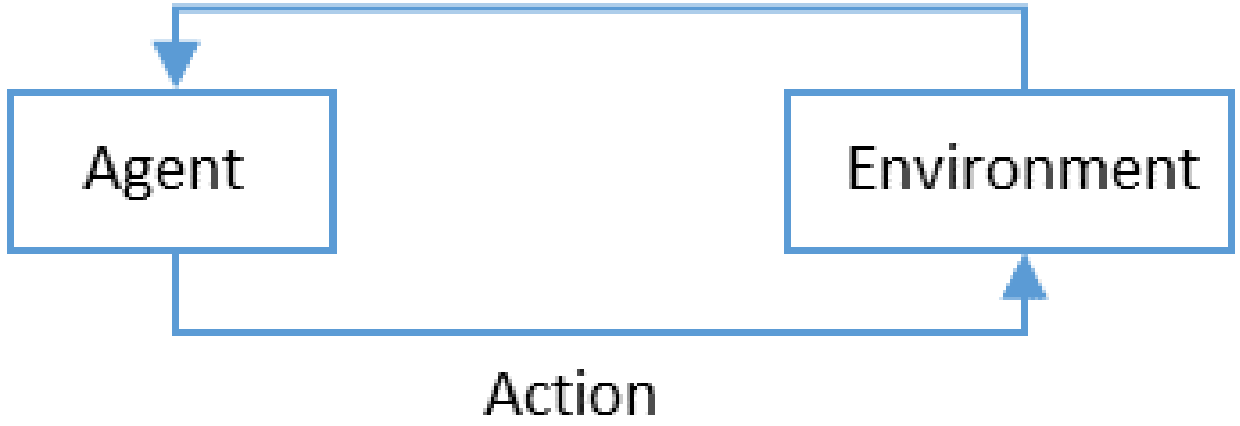


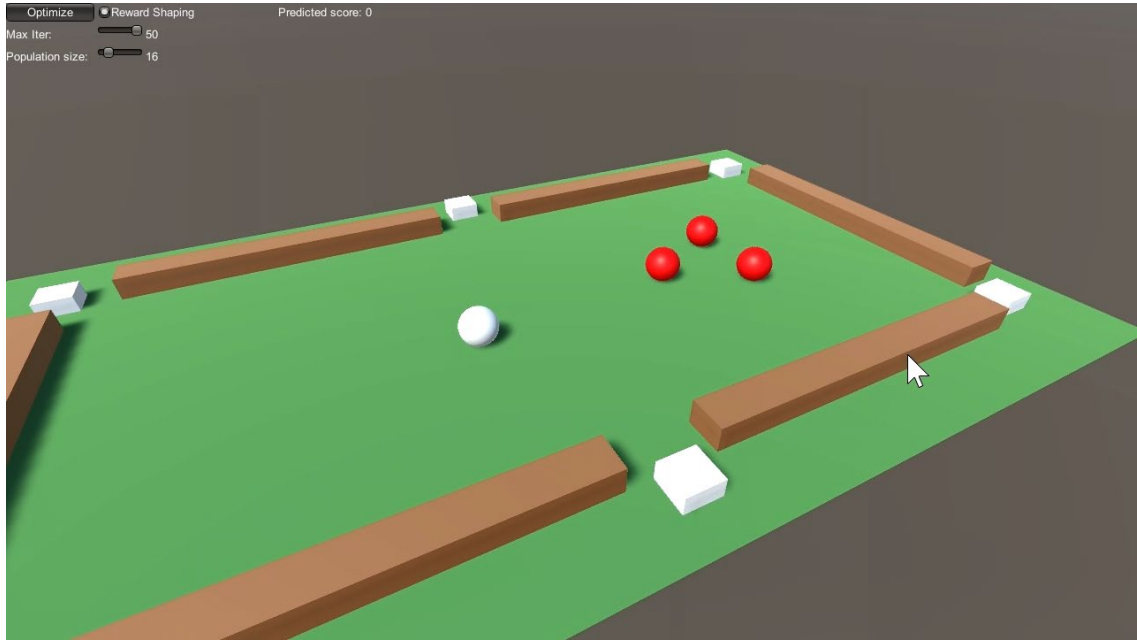
Optimization objective: Maximize the expected sum of (future) rewards:

$$\mathbb{E}[\sum_t \gamma^t r(\mathbf{a}_t, \mathbf{o}_t)]$$

Real life: Countless variations of the same problems

Observation and reward





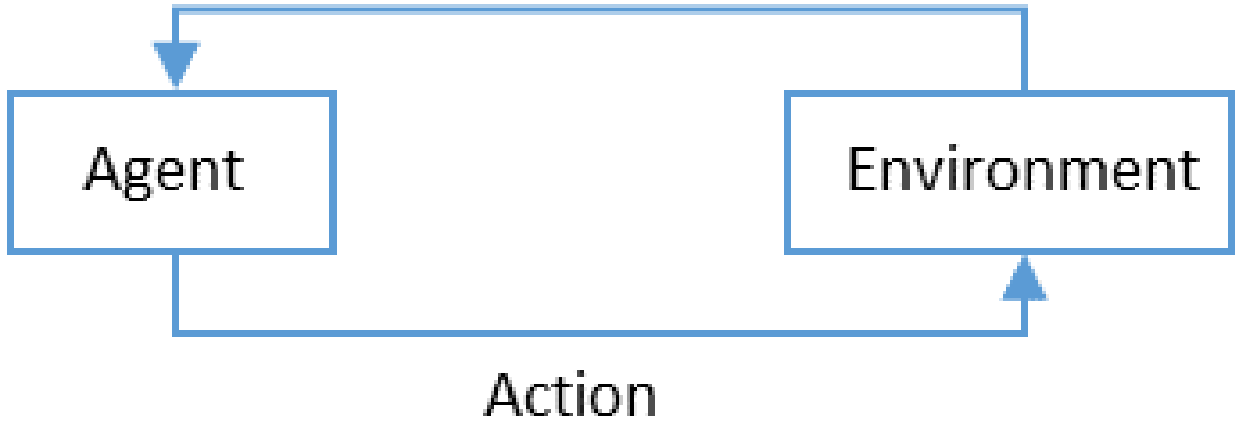
Optimization objective: Maximize the expected sum of (future) rewards:

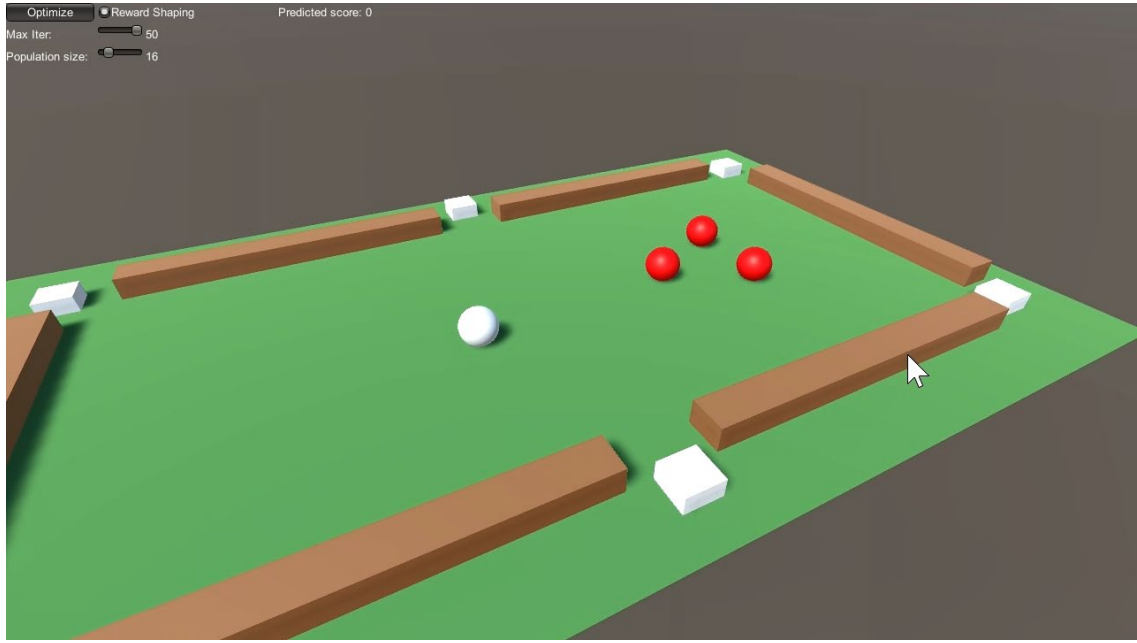
$$\mathbb{E}[\sum_{t} \gamma^t r(\mathbf{a}_t, \mathbf{o}_t)]$$

Expectation: We're interested in the average behavior over all actions and observations

Real life: Countless variations of the same problems

Observation and reward





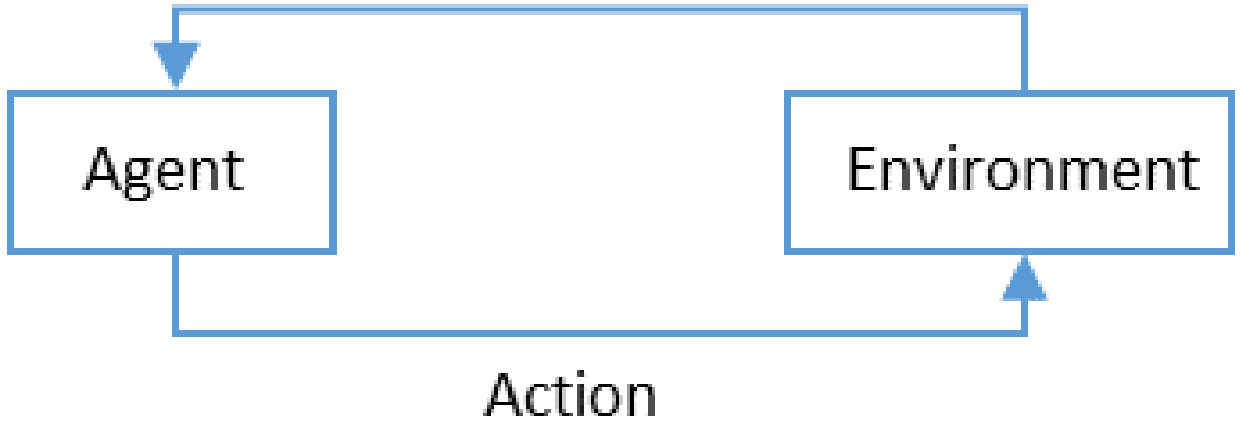
Optimization objective: Maximize the expected sum of (future) rewards:

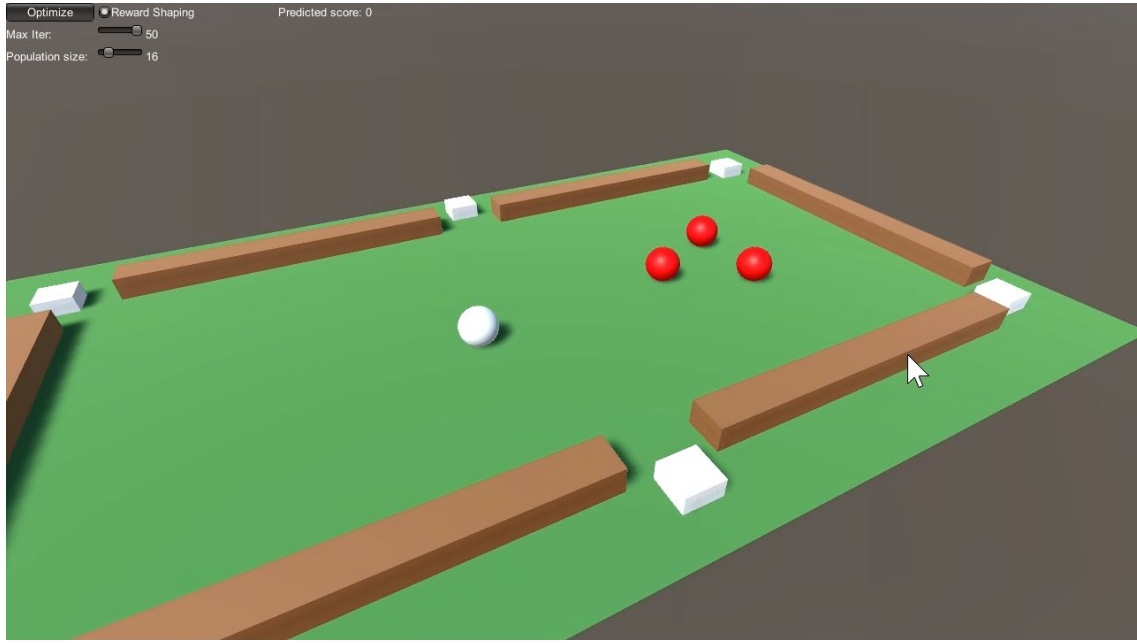
$$\mathbb{E}[\sum_{t} \gamma^{t} r(\mathbf{a}_t, \mathbf{o}_t)]$$

Time. 0 means “current” time and other values are in the future

Real life: Countless variations of the same problems

Observation and reward





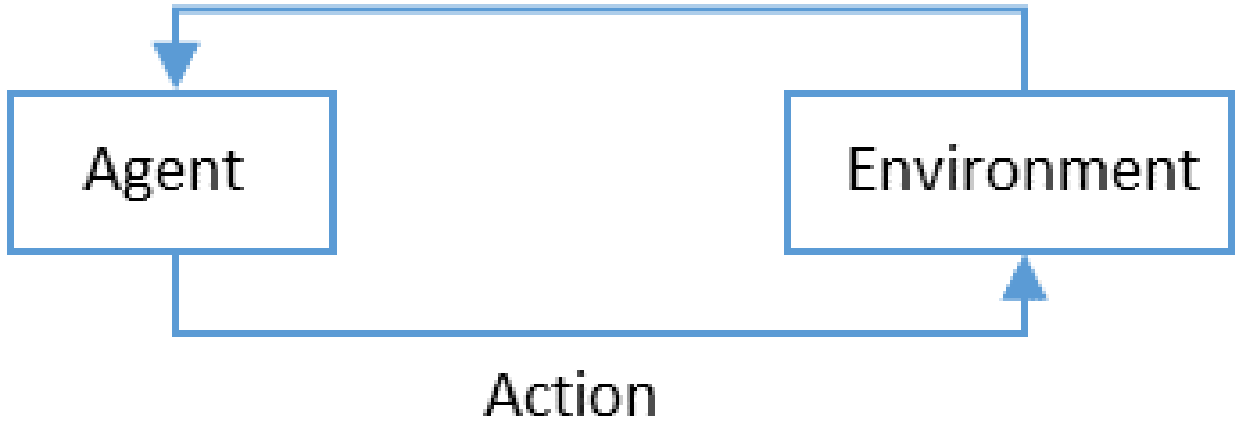
Optimization objective: Maximize the expected sum of (future) rewards:

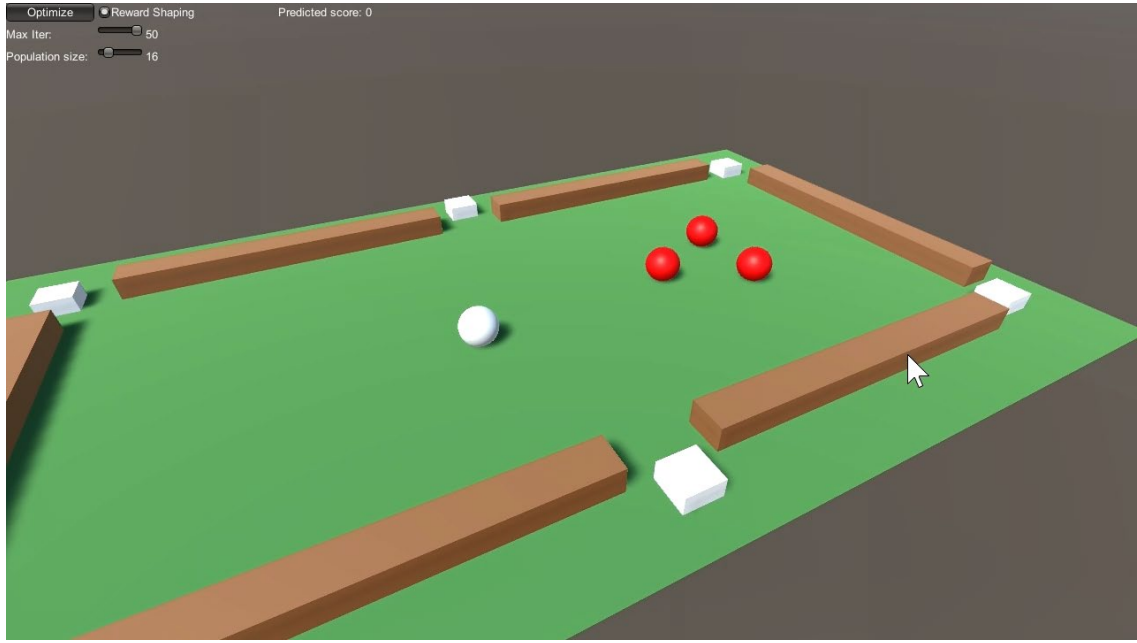
$$\mathbb{E}\left[\sum_t \gamma^t r(\mathbf{a}_t, \mathbf{o}_t)\right]$$

Reward for action $\mathbf{a}_t$ and observation $\mathbf{o}_t$ at time t

Real life: Countless variations of the same problems

Observation and reward





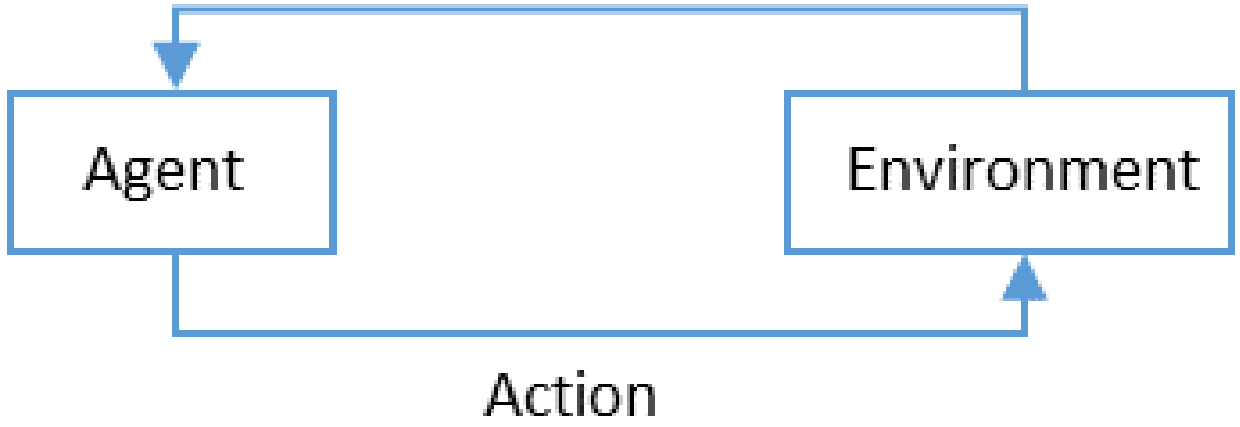
Optimization objective: Maximize the expected sum of (future) rewards:

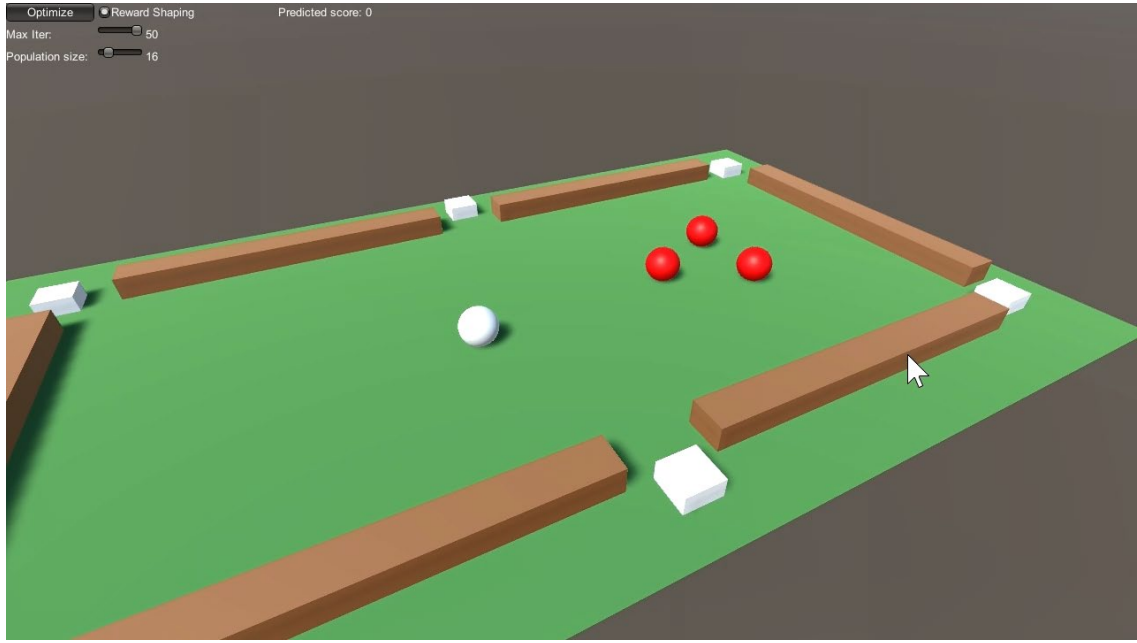
$$\mathbb{E}\left[\sum_t \gamma^t r(\mathbf{a}_t, \mathbf{o}_t)\right]$$

Sum over time: Consider not just a single action but future actions too

Real life: Countless variations of the same problems

Observation and reward



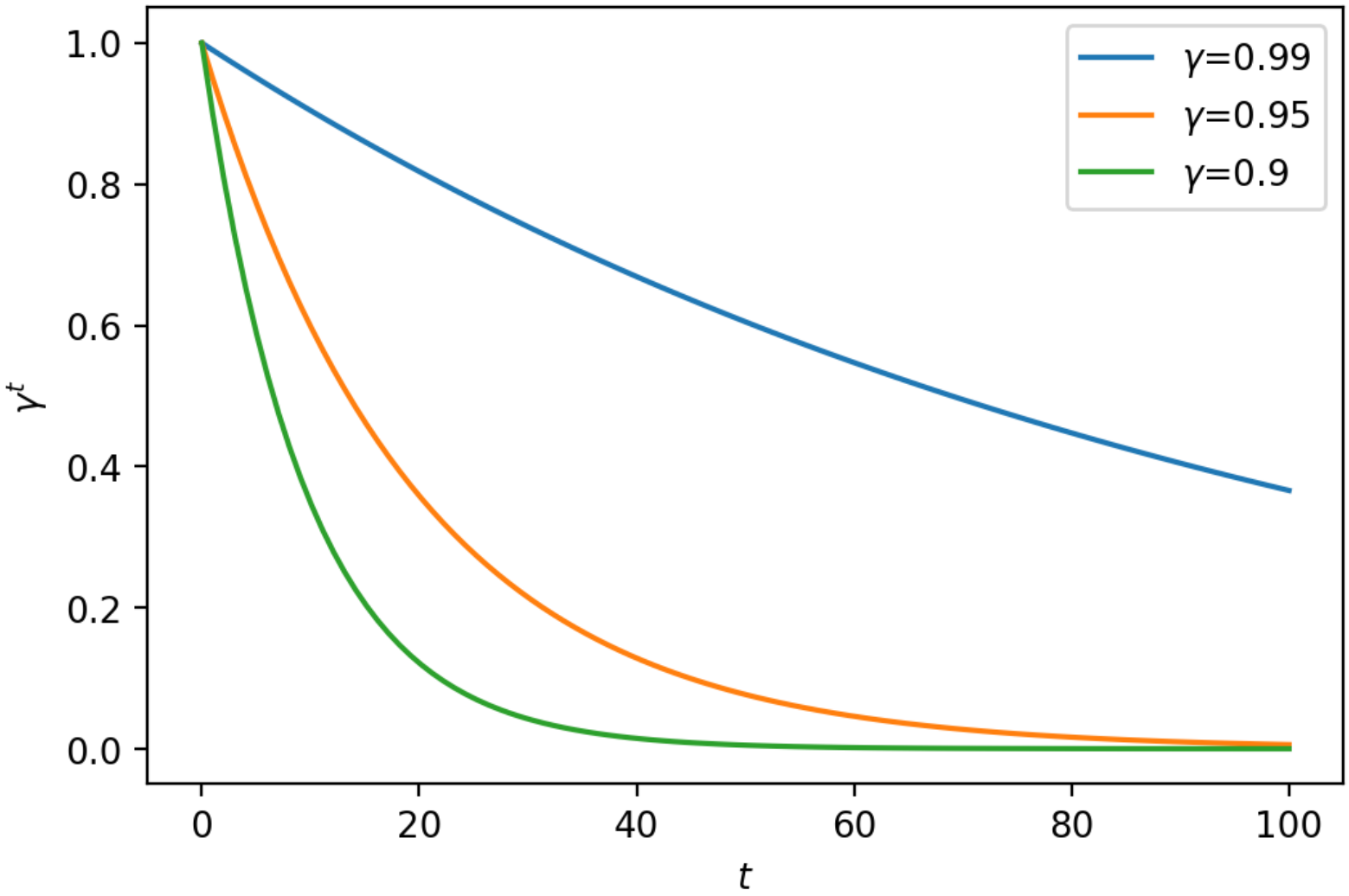


Optimization objective: Maximize the expected sum of (future) rewards:

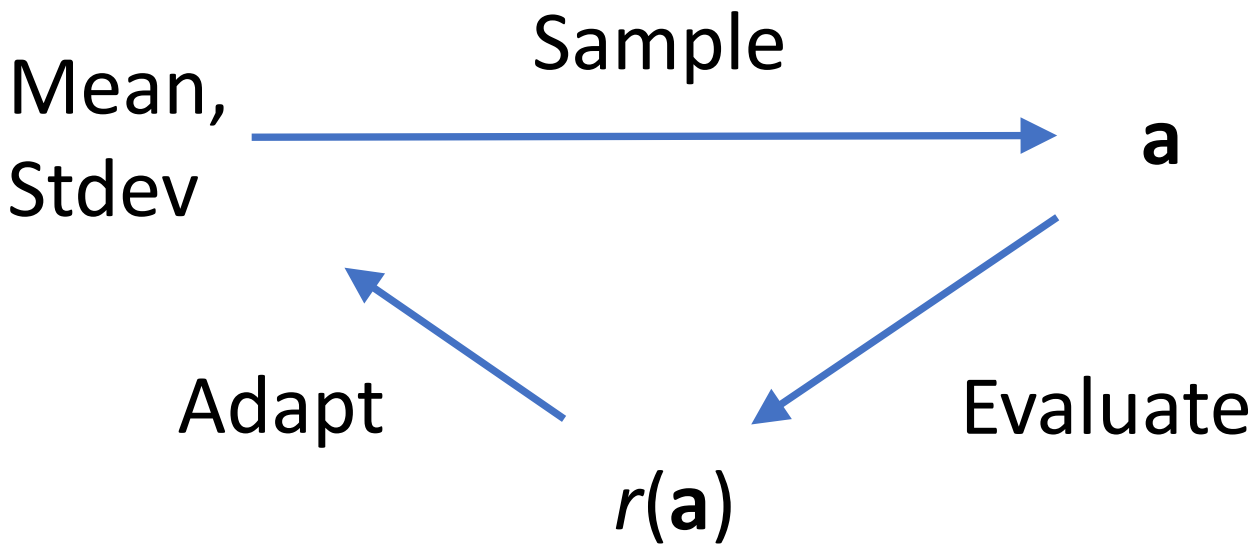
$$\mathbb{E}[\sum_t \gamma^t r(\mathbf{a}_t, \mathbf{o}_t)]$$

Discount factor

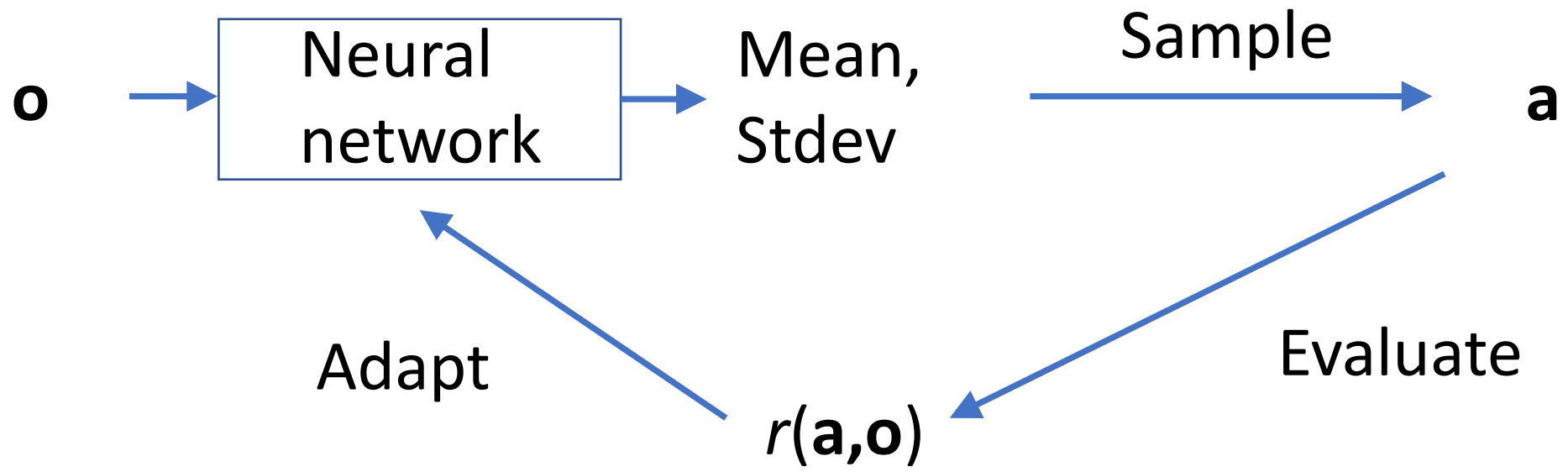
Small discount factor: Immediate rewards matter more



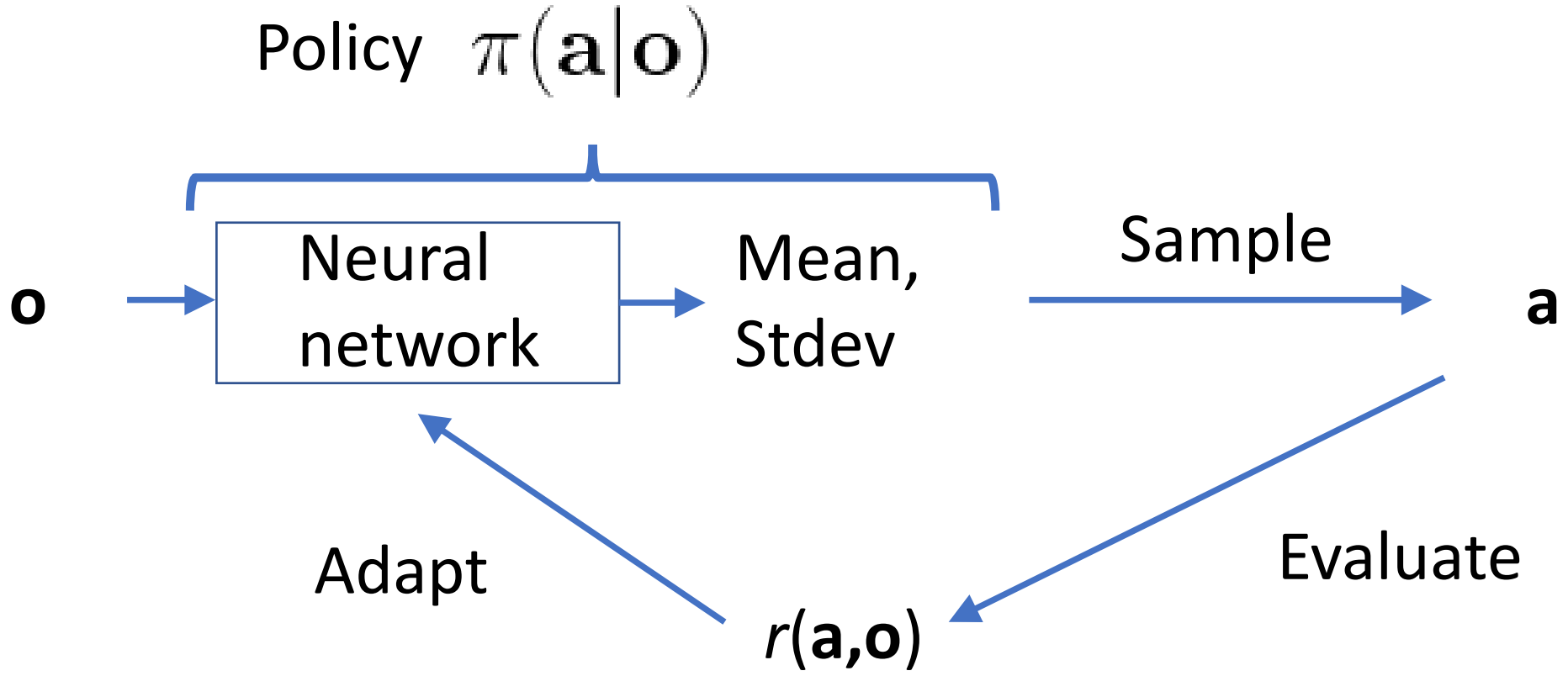
Optimizing actions independent of state (e.g., CMA-ES)



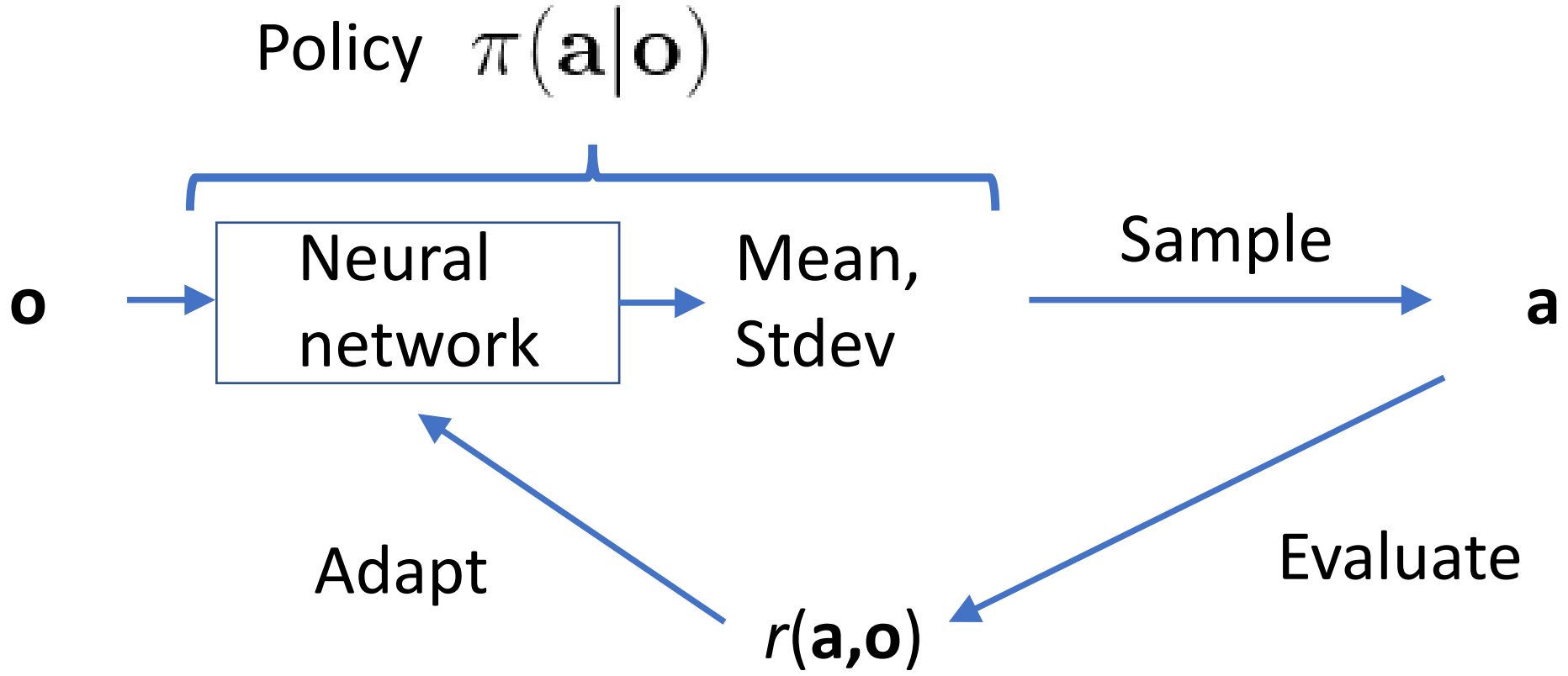
DRL: The action distribution depends on the observation (e.g., configuration of billiards balls)



DRL: The action distribution depends on the observation (e.g., configuration of billiards balls)



DRL: The action distribution depends on the observation (e.g., configuration of billiards balls)



JULY 20, 2017

Proximal Policy Optimization

We're releasing a new class of reinforcement learning algorithms, **Proximal Policy Optimization (PPO)**, which perform comparably or better than state-of-the-art approaches while being much simpler to implement and tune. PPO has become the default reinforcement learning algorithm at OpenAI because of its ease of use and good performance.

VIEW ON GITHUB

VIEW ON ARXIV

READ MORE

PPO and other episodic Reinforcement Learning methods as pseudocode

Until iteration simulation budget exhausted:

Sample initial state **s**, observed as **o**

Until terminal state encountered or time limit:

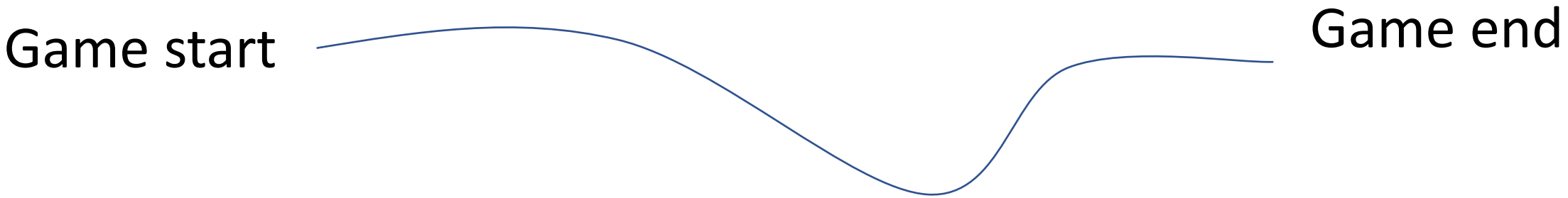
Sample action **a** according to policy

Execute action (simulate world model)

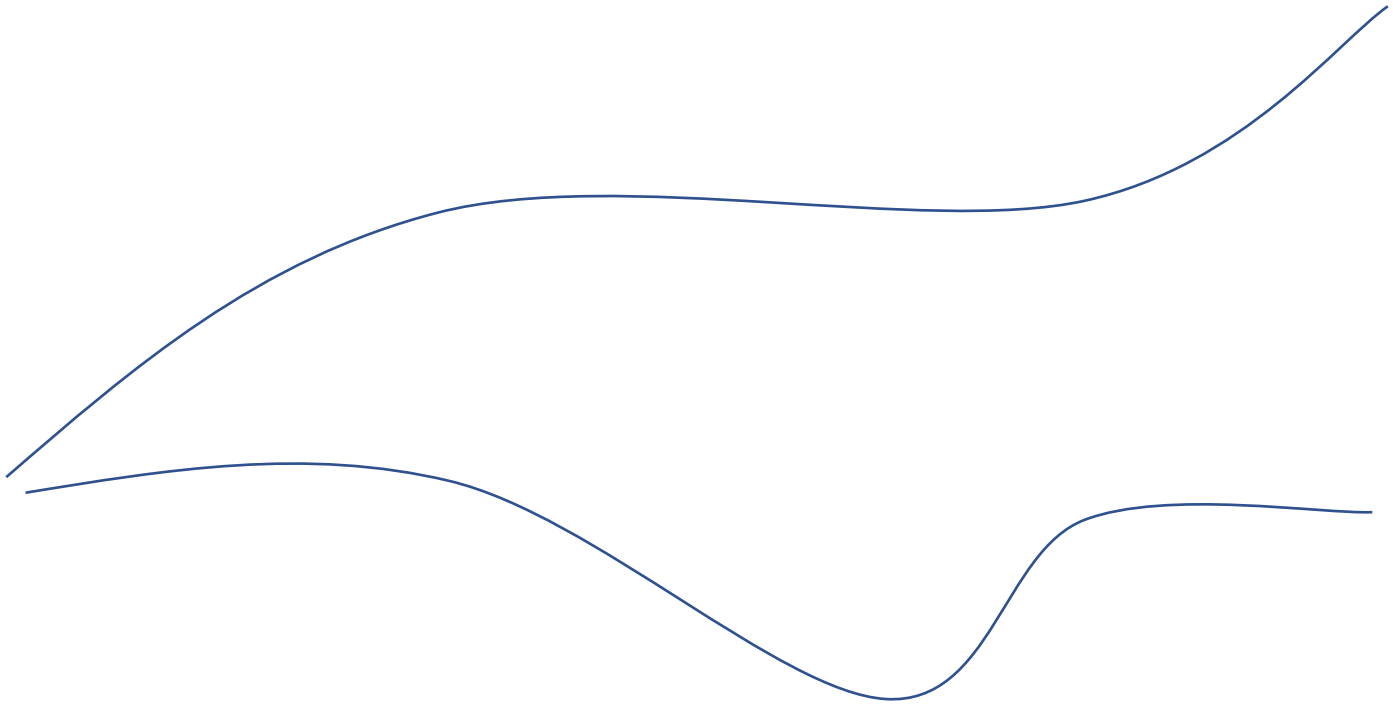
Observe new **o**’ and reward *r*

One episode

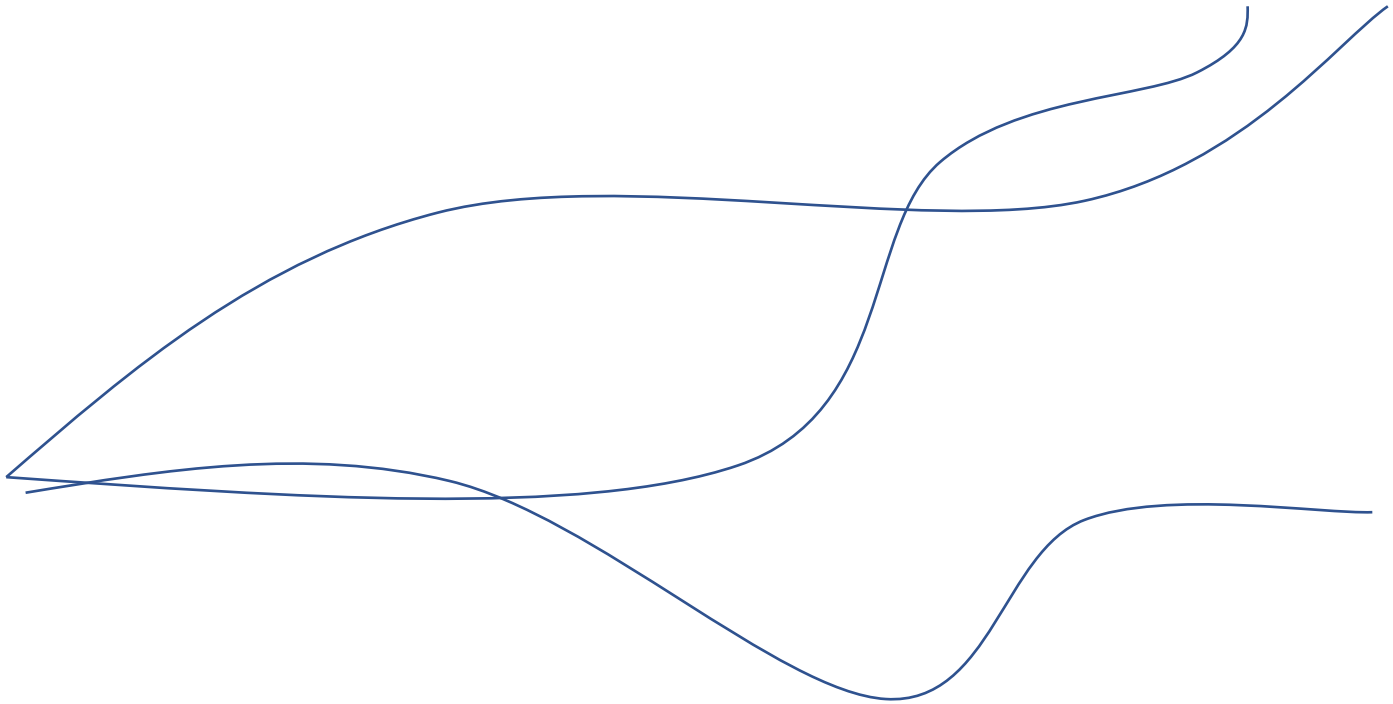
Update the policy based on the collected experience tuples [**o**,**a**,**o**’,*r*]

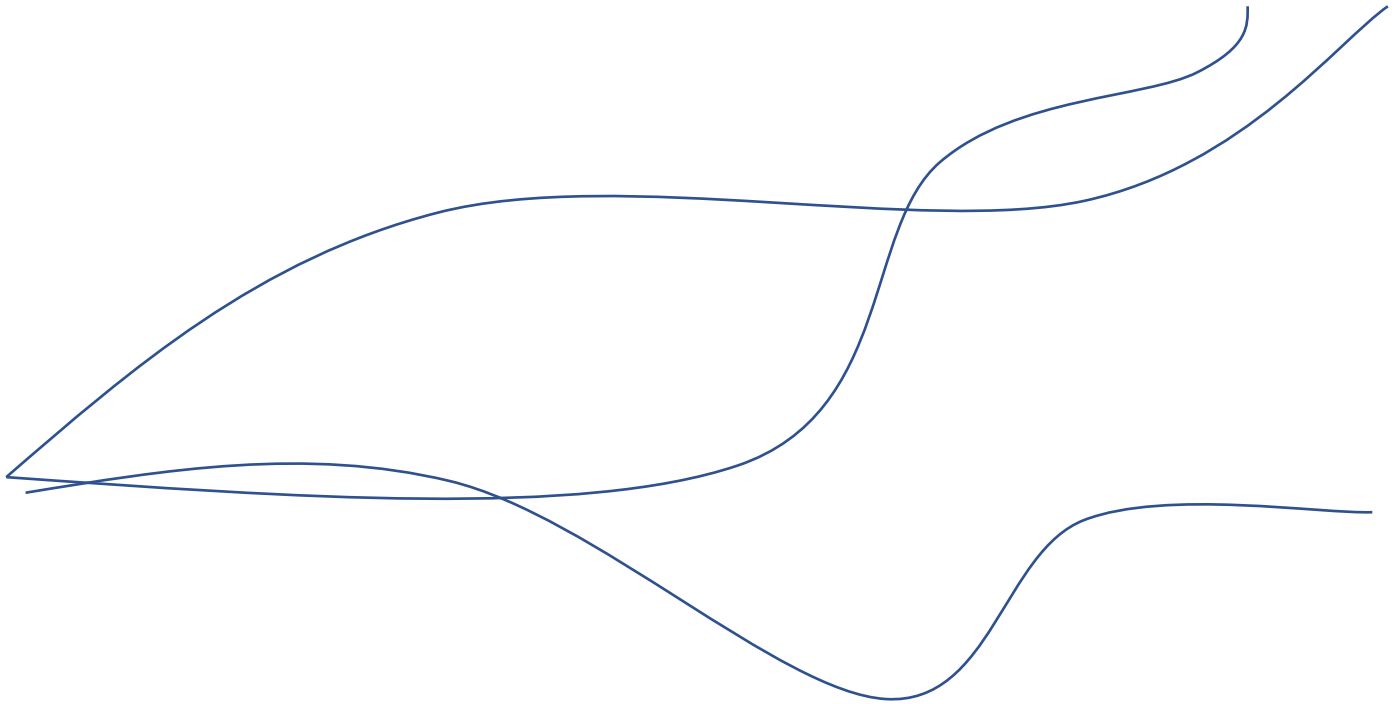

Game start
Game end

Game start

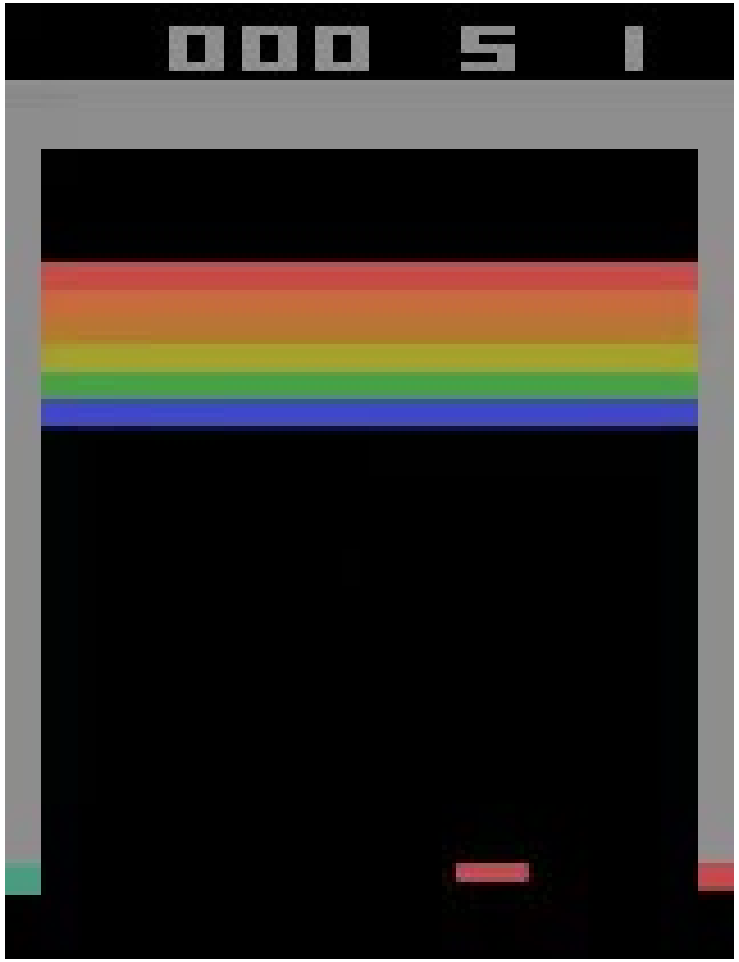


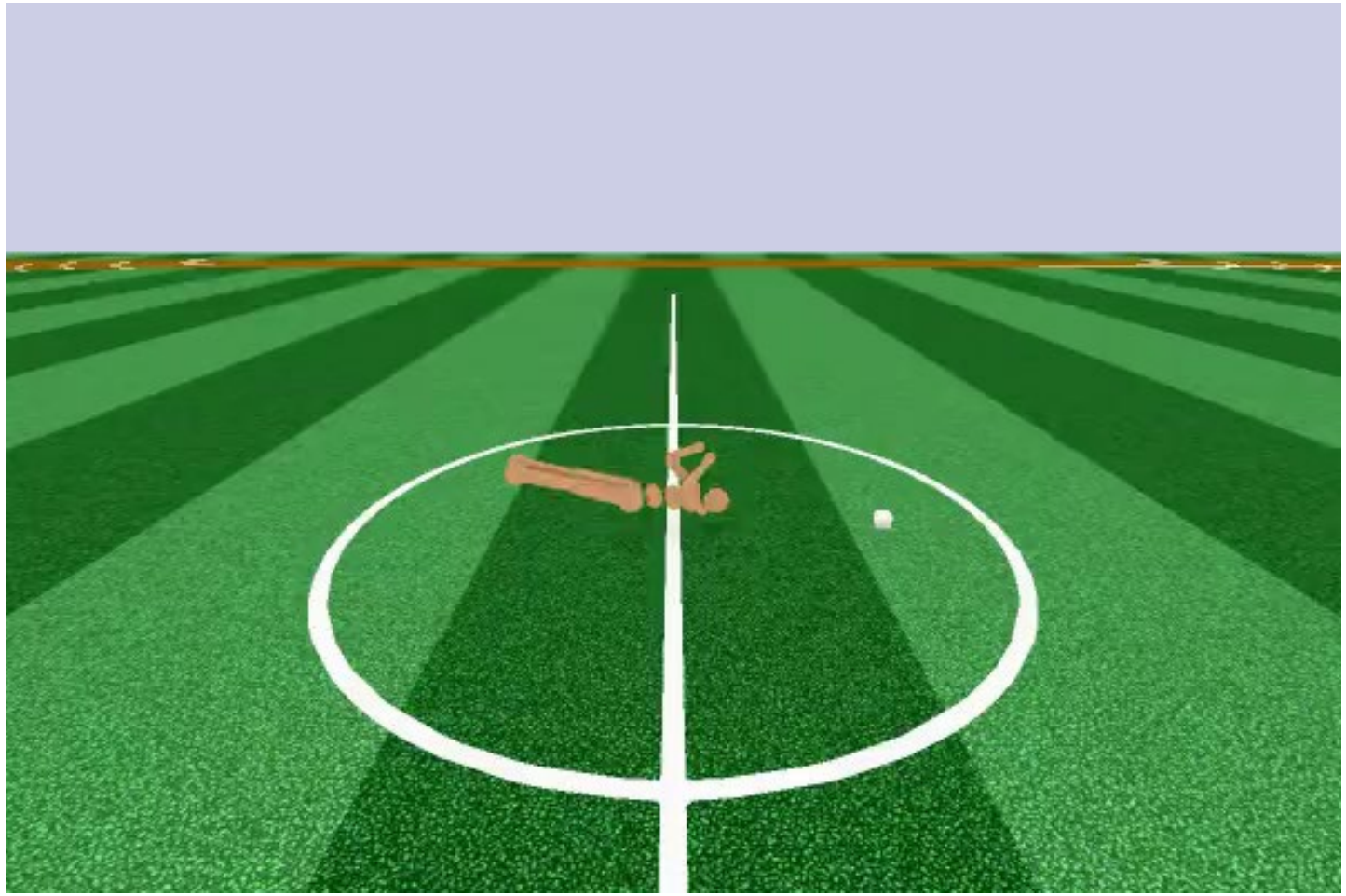
Game start



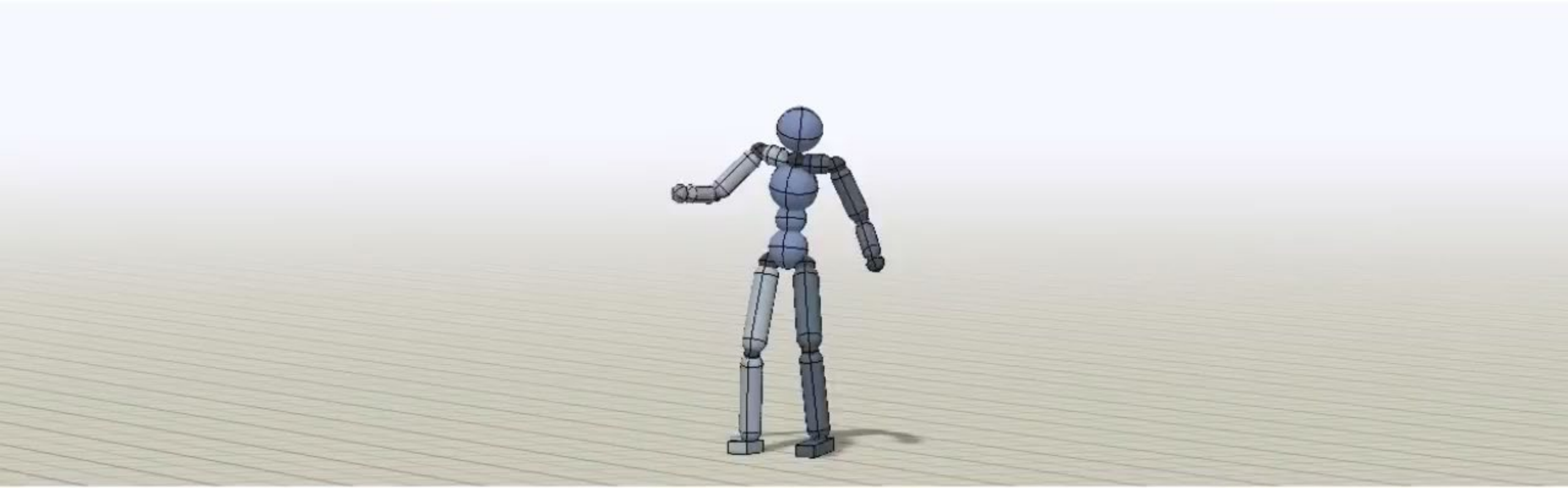


When enough episodes collected, use all the [**o**,**a**,**o**’,*r*] to compute the gradient


000 5 1



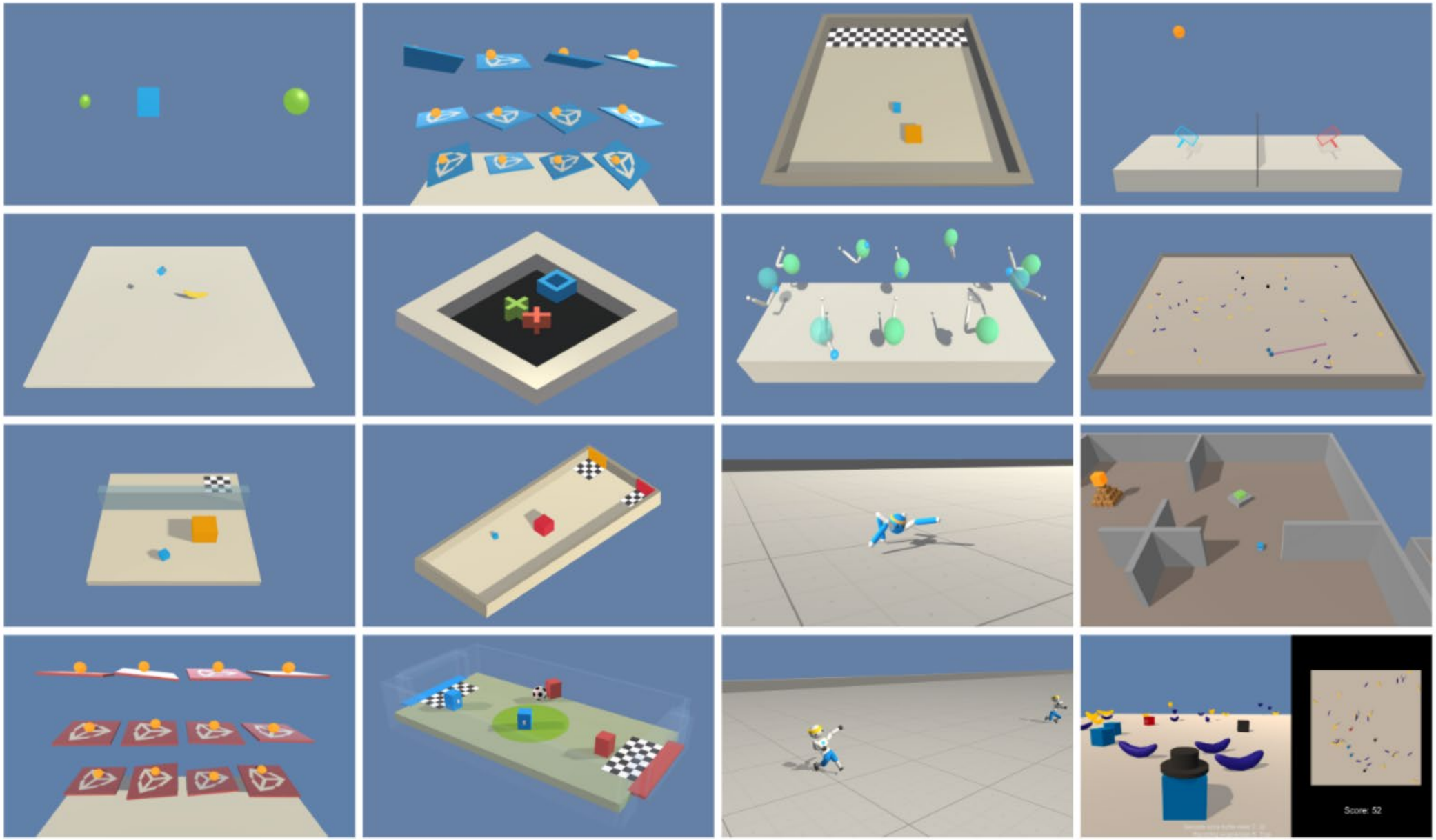
DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills



Xue Bin Peng[1], Pieter Abbeel[1], Sergey Levine[1], Michiel van de Panne[2]

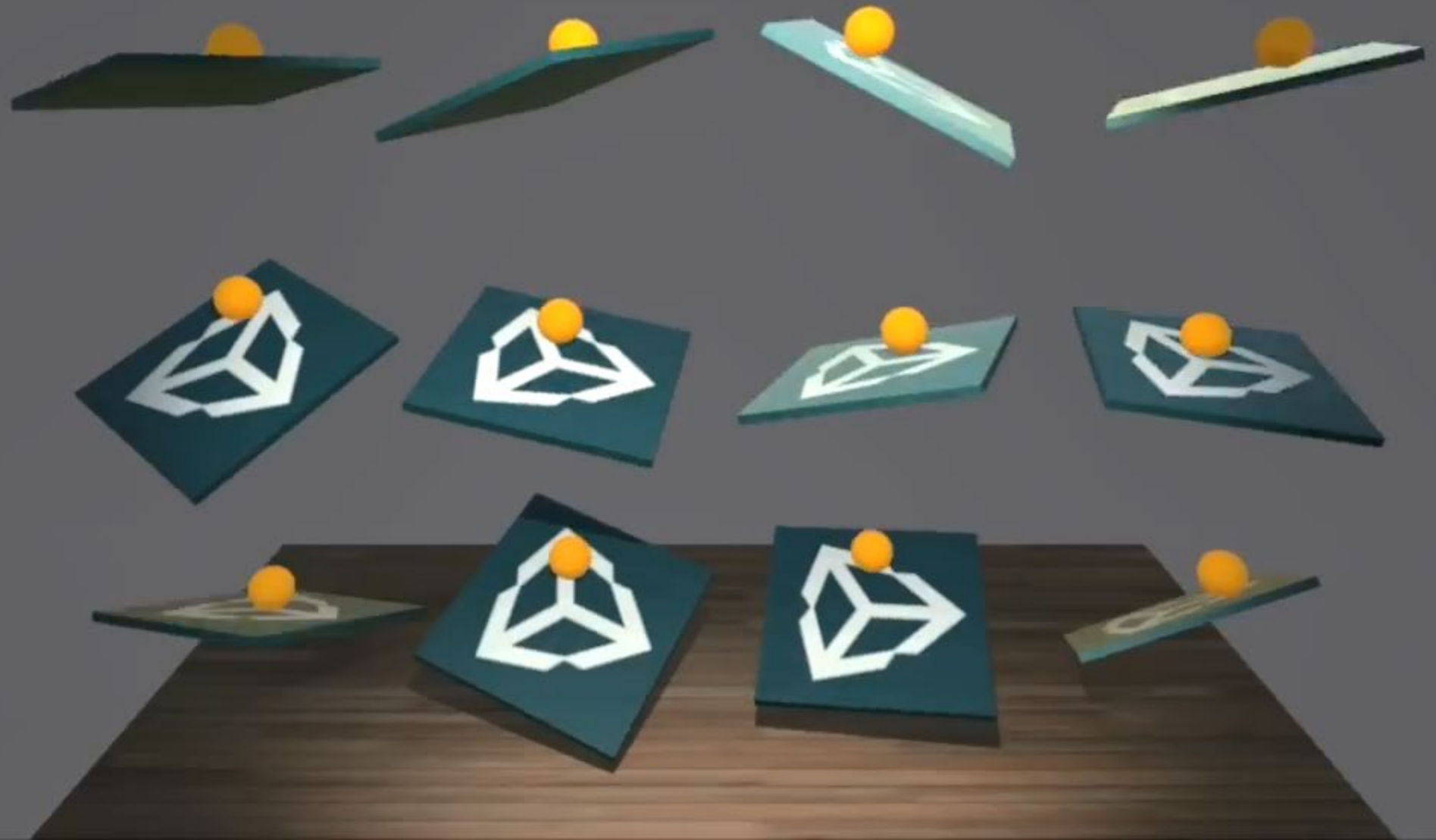
[1] University of California Berkeley

[2] University of British Columbia



Unity ML-Agents Toolkit (Beta)

Unity Machine Learning Agents, PPO



https://openai.com/blog/emergent-tool-use/

SEPTEMBER 17, 2019 • 9 MINUTE READ

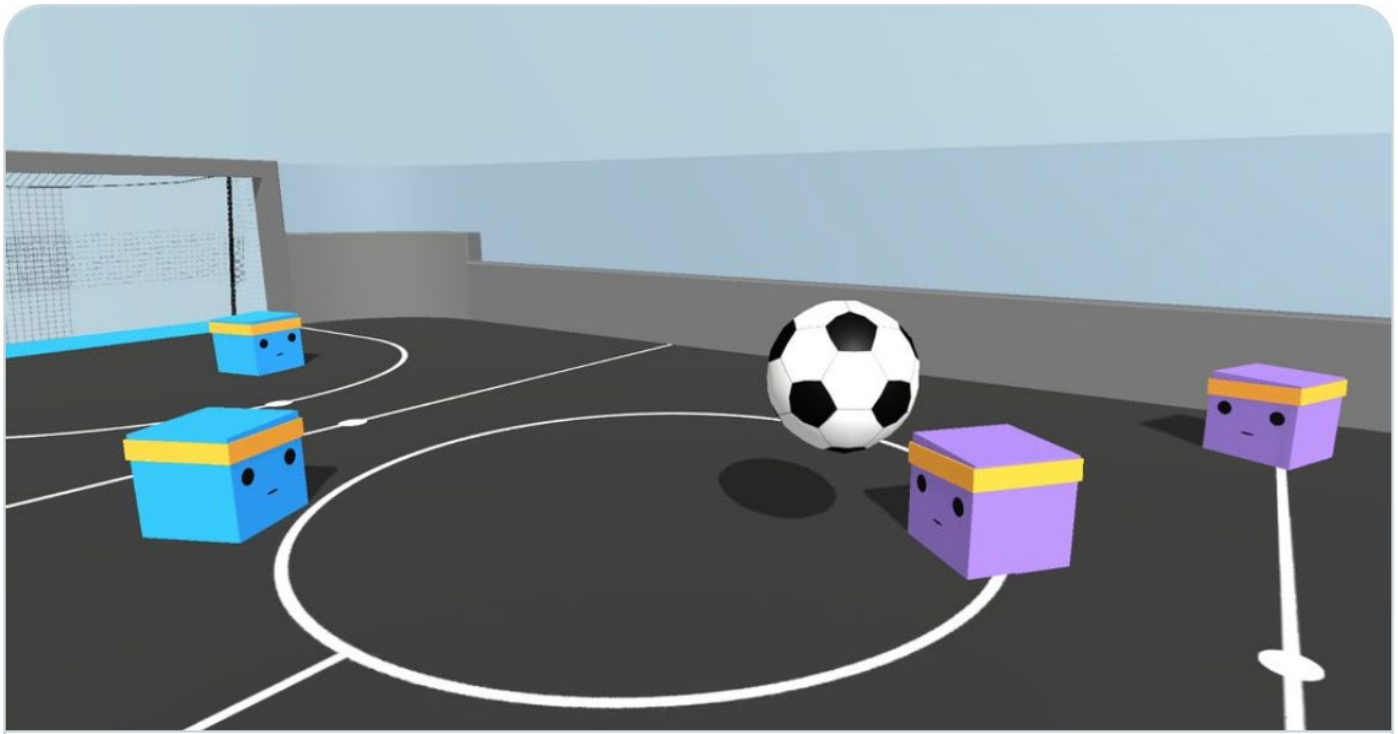
Emergent Tool Use from Multi-Agent Interaction

We've observed agents discovering progressively more complex tool use while playing a simple game of hide-and-seek. Through training in our new simulated hide-and-seek environment, agents build a series of six distinct strategies and counterstrategies, some of which we did not know our environment supported. The self-supervised emergent complexity in this

Unity @unity3d · Feb 28

Score! New with ML-Agents v0.14: self-play and the ability to train competitive agents in adversarial games! #mlagents

Play around with it in our soccer demo environment. ⚽



Training intelligent adversaries using self-play with ML-Agents - Unity T...

In the latest release of the ML-Agents Toolkit (v0.14), we have added a self-play feature that provides the capability to train competitive agents...

blogs.unity3d.com

https://blogs.unity3d.com/2020/02/28/training-intelligent-adversaries-using-self-play-with-ml-agents/

Some methods to try

- PPO (Schulman et al. 2017). Works for both continuous actions (e.g., robot joint torques) and discrete actions (e.g., gamepad button presses)
- Soft Actor Critic (Haarnoja et al. 2018): Only for continuous actions.

Both PPO and SAC are available in Unity ML Agents.

SAC can be more efficient, but may require more fine-tuning of parameters.

In Python: OpenAI Gym

```
import gym
env = gym.make("CartPole-v1")
observation = env.reset()
for _ in range(1000):
  env.render()
  action = env.action_space.sample() # your agent here (this takes random actions)
  observation, reward, done, info = env.step(action)

  if done:
    observation = env.reset()
env.close()
```

Tonic: A Deep Reinforcement Learning Library for Fast Prototyping and Benchmarking

Fabio Pardo
Imperial College London
f.pardo@imperial.ac.uk

Abstract

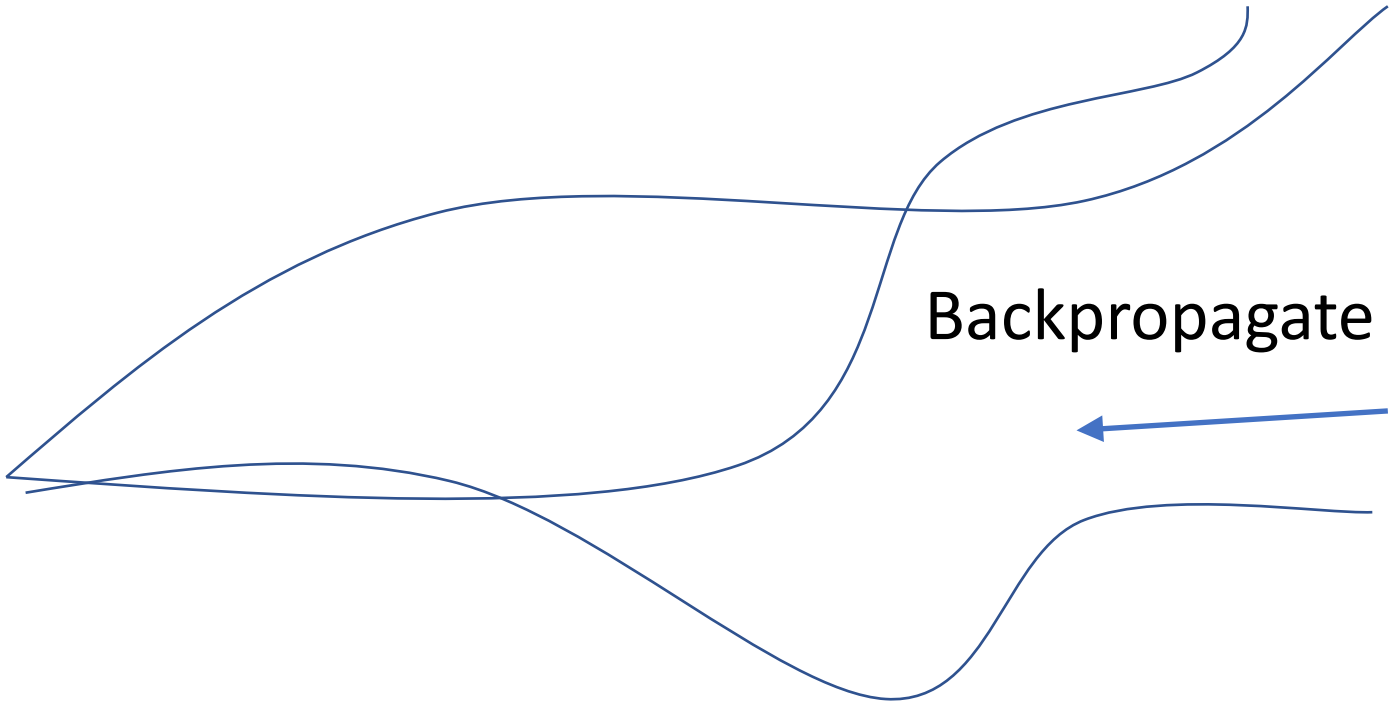
Deep reinforcement learning has been one of the fastest growing fields of machine learning over the past years and numerous libraries have been open sourced to support research. However, most codebases have a steep learning curve or limited flexibility that do not satisfy a need for fast prototyping in fundamental research. This paper introduces Tonic, a Python library allowing researchers to quickly implement new ideas and measure their importance by providing: 1) a collection of configurable modules such as exploration strategies, replays, neural networks, and updaters 2) a collection of baseline agents: A2C, TRPO, PPO, MPO, DDPG, D4PG, TD3 and SAC built with these modules 3) support for the two most popular deep learning frameworks: TensorFlow 2 and PyTorch 4) support for the three most popular sets of continuous-control environments: OpenAI Gym, DeepMind Control Suite and PyBullet 5) a large-scale benchmark of the baseline agents on 70 continuous-control tasks 6) scripts to experiment in a reproducible way, plot results, and play with trained agents.

https://github.com/fabiopardo/tonic


WARNING
Mathematics
Ahead

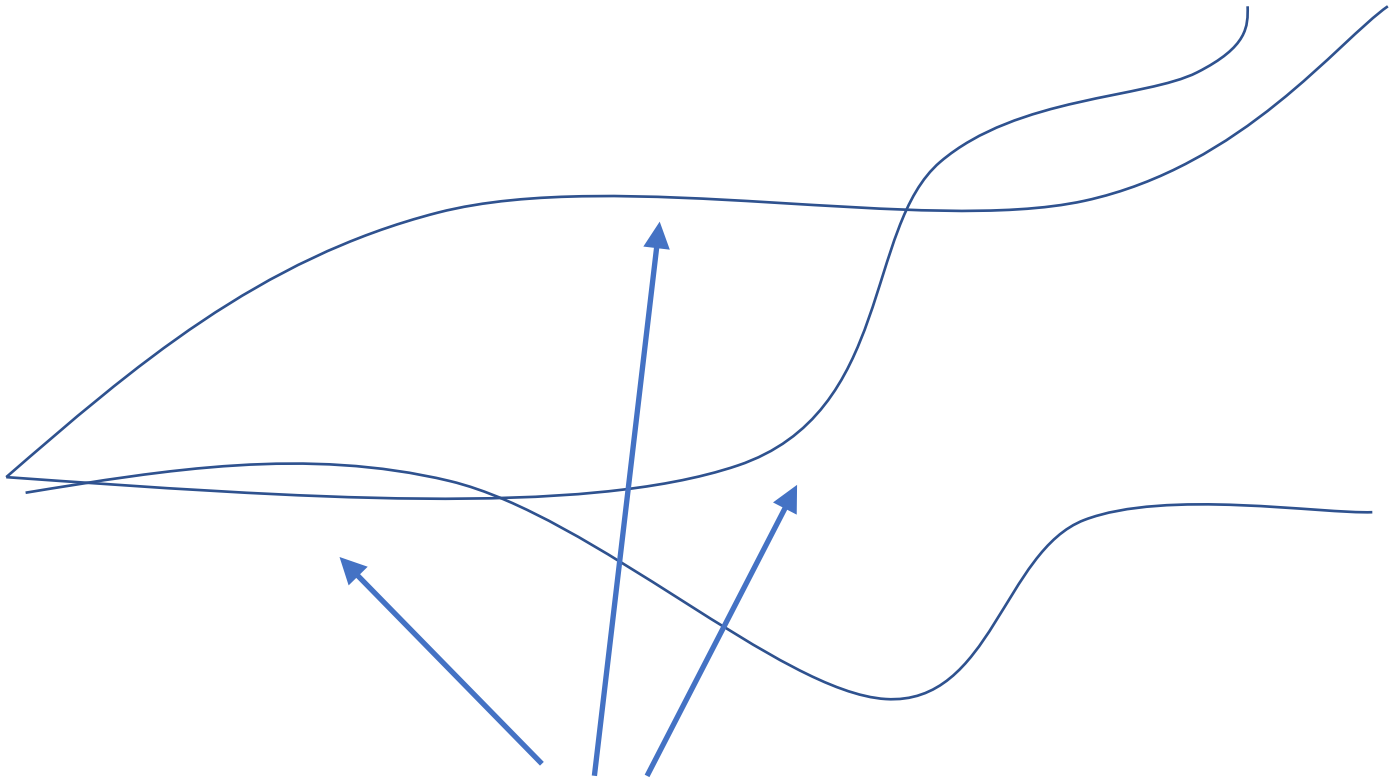
Markov Decision Process (MDP)

- A term very often encountered if you read RL papers
- A formal way to define a behavior optimization problem
- Defined by: The states **s**, actions **a**, a reward function $r(\mathbf{s},\mathbf{a})$, and the discount factor γ
- (Deep) Reinforcement Learning is a tool that one can apply once a problem has been formulated as an MDP => one only needs to define/implement the **s**,**a**,r,γ
- If the full state is not observed, the formulation is a Partially Observable Markov Decision Process (PO-MDP), where **s** is replaced by **o**.
- In PO-MDP, the agent typically needs to observe a sequence of multiple timeteps to be able to infer the unobserved state and act accordingly.



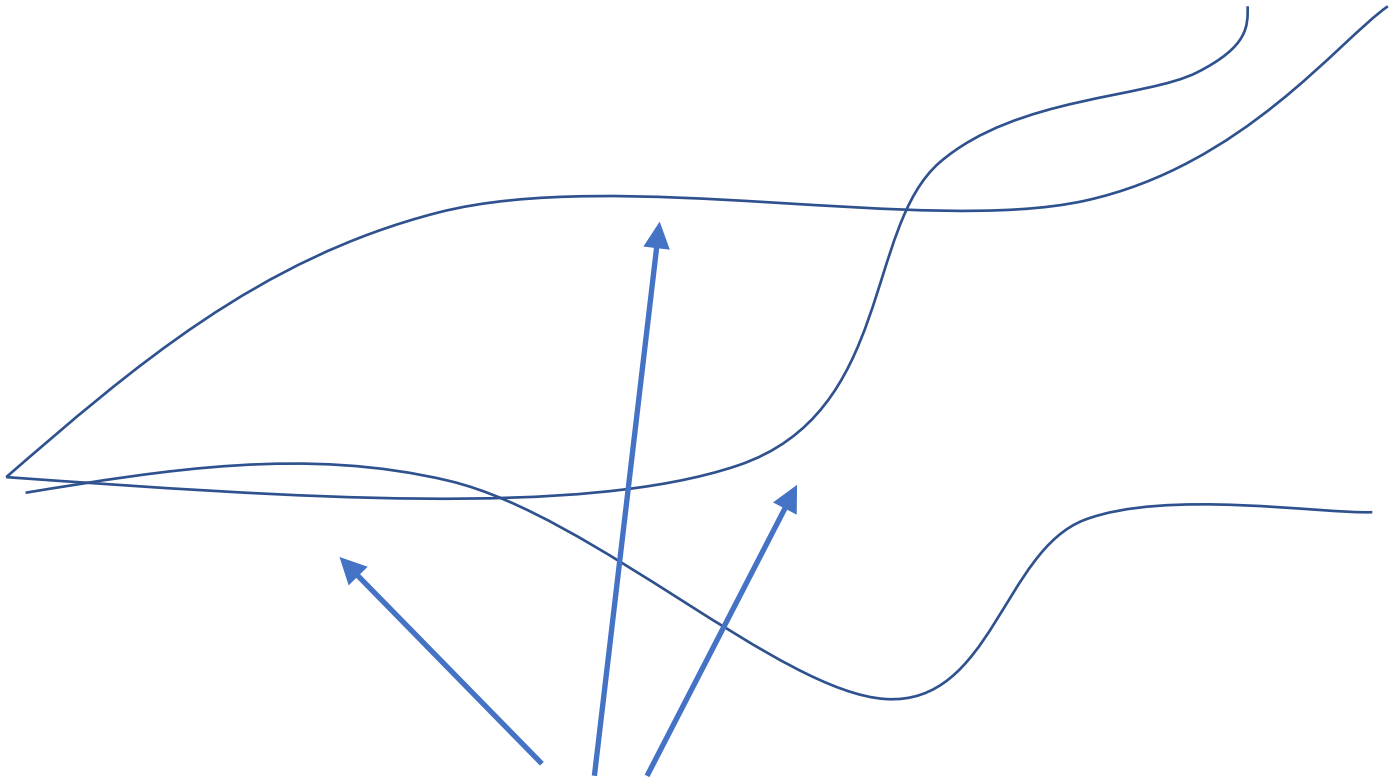
$$R(\mathbf{a},\mathbf{s}) = r(\mathbf{a},\mathbf{s}) + \gamma\, R(\mathbf{a},\mathbf{s}')$$

Game start



Train a value function V(**s**)=E[R(**s**)] predictor network with the returns for each **s** in the episode trajectories.

Game start



Compute advantages as A(**a**,**s**)=Q(**a**,**s**)-V(**s**),
Q(**a**,**s**)=r(**a**,**s**)+ γ V(**s**') (Bellman recursion)

Training loss

$$\mathcal{L} = -\frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) \log \pi_{\theta}(\mathbf{a}_i|\mathbf{s}_i)$$

Training loss

$$\mathcal{L} = -\frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)$$

Training loss

Average over all samples in minibatch

$$\mathcal{L} = -\frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)$$

Training loss

Minimizing the loss = maximizing the average

$$\mathcal{L} = -\frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i)\log \pi_{\theta}(\mathbf{a}_i|\mathbf{s}_i)$$

Training loss

Probability of action $\mathbf{a}_i$ in state $\mathbf{s}_i$

$$\mathcal{L} = -\frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)$$

Training loss

Parameters of the policy neural network, i.e., the optimized variables

$$\mathcal{L} = -\frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) \log \pi_{\theta}(\mathbf{a}_i|\mathbf{s}_i)$$

Training loss

The advantage values, treated as constants
(computed based on the iteration's experience data)

$$\mathcal{L} = -\frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)$$

Training loss

Maximize this = maximize the probability of actions with positive advantages

$$\mathcal{L} = -\frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i)\log \pi_{\theta}(\mathbf{a}_i|\mathbf{s}_i)$$

Gaussian policy with identity covariance matrix

$$\mathcal{L} = \frac{1}{M} \sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) ||\mathbf{a}_i - \mu_\theta(\mathbf{s})||^2$$

Training with advantage-weighted samples

The sampling mean output by the policy network.

$$\mathcal{L} = \frac{1}{M} \sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) ||\mathbf{a}_i - \mu_\theta(\mathbf{s})||^2$$

Training with advantage-weighted samples

Neural network parameters that we optimize

$$\mathcal{L} = \frac{1}{M} \sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) ||\mathbf{a}_i - \mu_{\theta}(\mathbf{s})||^2$$

Training with advantage-weighted samples

Squared distance between the policy and actions

$$\mathcal{L} = \frac{1}{M} \sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) \boxed{||\mathbf{a}_i - \mu_\theta(\mathbf{s})||^2}$$

Training with advantage-weighted samples

Advantages as weights:
minimize the distance to
positive-advantage actions

$$\mathcal{L} = \frac{1}{M} \sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) ||\mathbf{a}_i - \mu_\theta(\mathbf{s})||^2$$

Updating both mean and variance

$$\mathcal{L} = \frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) \sum_j \left[\frac{(a_{i,j} - \mu_{j;\theta}(\mathbf{s}_i))^2}{c_{j;\theta}(\mathbf{s}_i)} + 0.5 \log c_{j;\theta}(\mathbf{s}_i) \right]$$


WA NG
M ematics
Ahead

How to use DRL

- Implement a game or simulation, wrapped inside the OpenAI Gym Env interface in Python, or Unity ML's agent interface in C#
- Define how simulation state is mapped to a fixed length observation vector
- Define how the episode initial state is sampled when the DRL optimizer calls env.reset()
- Define what causes the episode to end (typically, timeout or failure/termination)
- Define a mapping from a fixed length action vector into simulation actions, e.g., physics simulation torques or simulated gamepad button presses
- Define a reward function
- Optional: define a curriculum, i.e., start training with some easier problem, then modify it as training progresses (pedagogy can matter even more than when training humans…)
- Test training with some DRL algorithm

Typical problems: Normalization

- Make sure that the reward function is in range 0…1 or close to it
- Make sure the observations are in range -1…1 or have mean 0 and standard deviation 1
- Reason: Neural network optimization is highly sensitive to initialization. The DRL networks are typically initialized assuming this kind of normalized rewards and observations, and the algorithms may fail otherwise.

Typical problems: Early termination and negative rewards

- Episode termination when the agent fails (e.g., walking agent falls down) can reduce gradient noise and speed up training, compared to running every episode until a timeout
- However, if the agent can receive negative rewards, it may optimize by failing as fast as possible to avoid the negative rewards
- Solution: Make sure your rewards are positive. Print out the minimum reward encountered during training. Add a constant “alive bonus” to all rewards if the minimum is negative.

Typical problems: Zero or noisy gradient

- DRL can only learn if the random exploration manages to discover good actions (i.e., high rewards). Otherwise, there's no gradient to follow.
- Debug: Watch what happens at the beginning of the training. Are any of the random actions meaningful?
- Solution 1: Increase the amount of collected experience per algorithm iteration (more likely that at least some actions are good)
- Solution 2: Make sure your reward function is informative and not just zero most of the time. If possible, reduce gamma to reduce the noise in summing together the future rewards
- Solution 3: Design your actions so that they are more likely to produce rewards. For example, the policy could output navigation waypoints driven towards using some other controller, instead of the policy outputting raw steering actions
- Solution 4: Use a curiosity bonus that encourages exploration even with zero rewards (implemented in Unity ML, easy to toggle)

RAIVO

YOU'RE THE TRAINER

YOUR ROBOT IS THE FIGHTER

THE WORLD IS YOUR ARENA

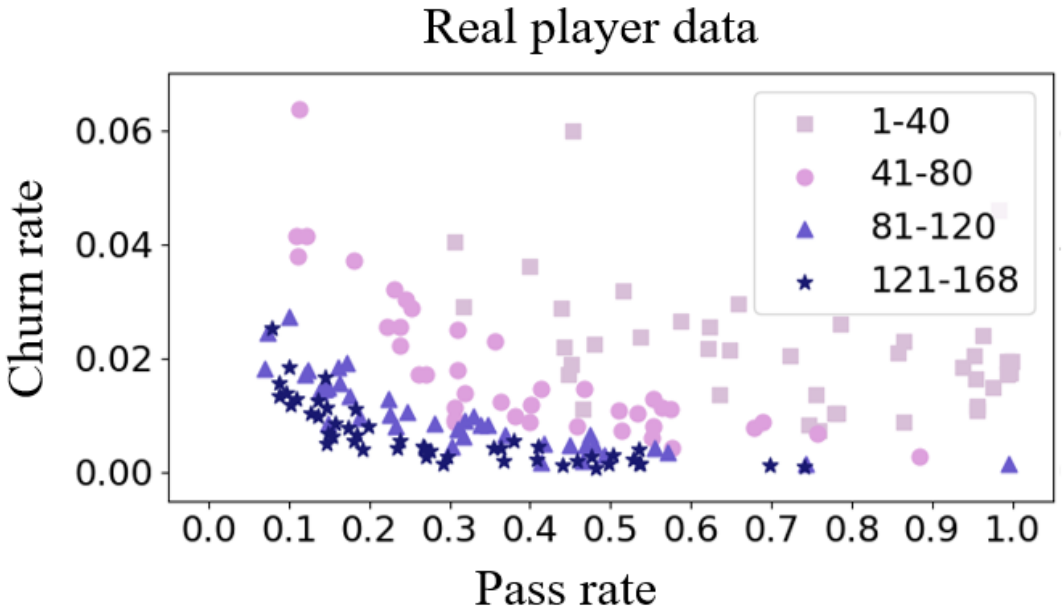
WELCOME TO RAIVO

MACHINE LEARNING GAMIFIED

https://www.utoposgames.com/raivo

Predicting player behavior and experience using DRL agents

- An Aalto-Rovio collaboration: https://twitter.com/perttu_h/status/1302536834002628608?s=20
- We use DRL agents to predict pass rate (i.e., level difficulty, a crucial part of game experience) and churn rate (behavioral metric, how many players quit at each level)



Forward search methods

Selected mode:Getting up

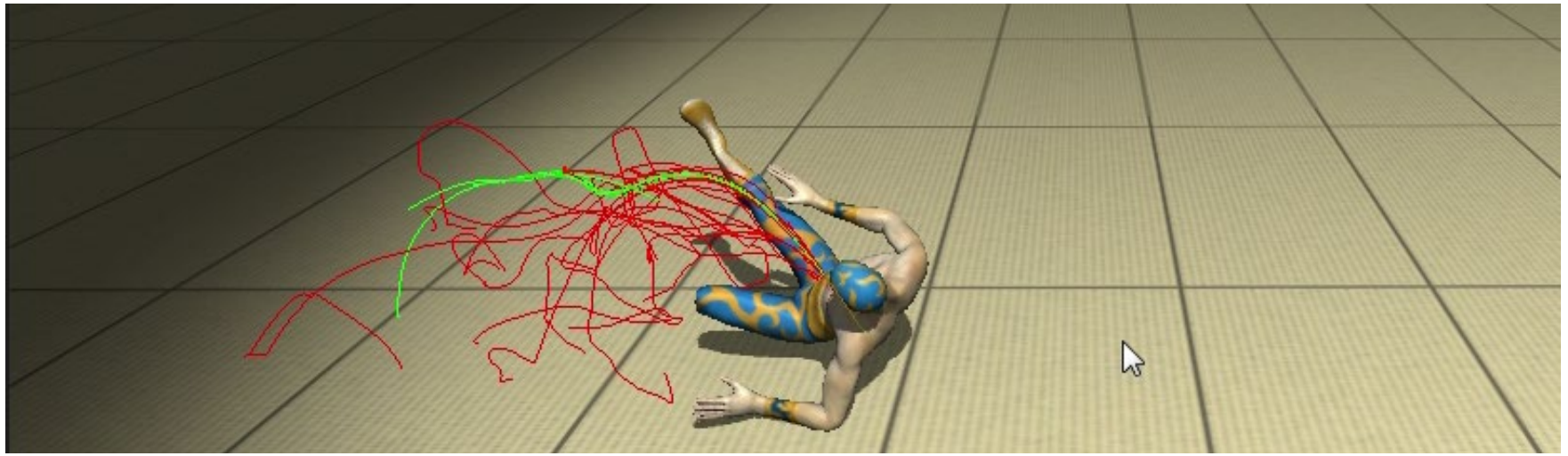
Online Motion Synthesis using Sequential Monte Carlo

SIGGRAPH 2014

Live demo

Controlling an agent through forward search

1. Simulate multiple action strategies up to a planning horizon or termination. Run simulations in parallel threads if possible
2. Step the ”master simulation” forward using the best strategy
3. Repeat from the resulting state, possibly informing the forward simulations based on previous results. Planning horizon is shifted one time step forward (rolling horizon, receding horizon)



Deep RL vs. Forward Search

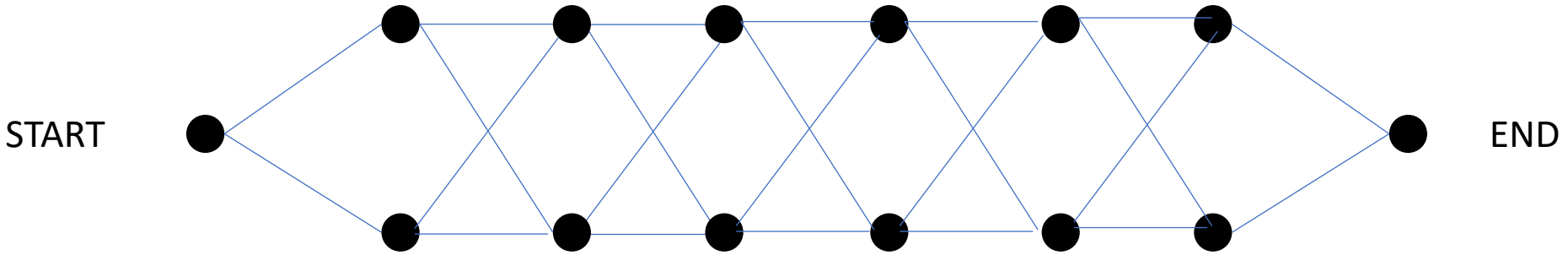
Deep RL:
+ Once trained, very computationally efficient
- Training can be very slow and may not converge

Search methods:
+ Get good results fast, without training neural networks
- Requires forward simulation (many times faster than real-time simulation)
- Requires capability to save and load simulation state

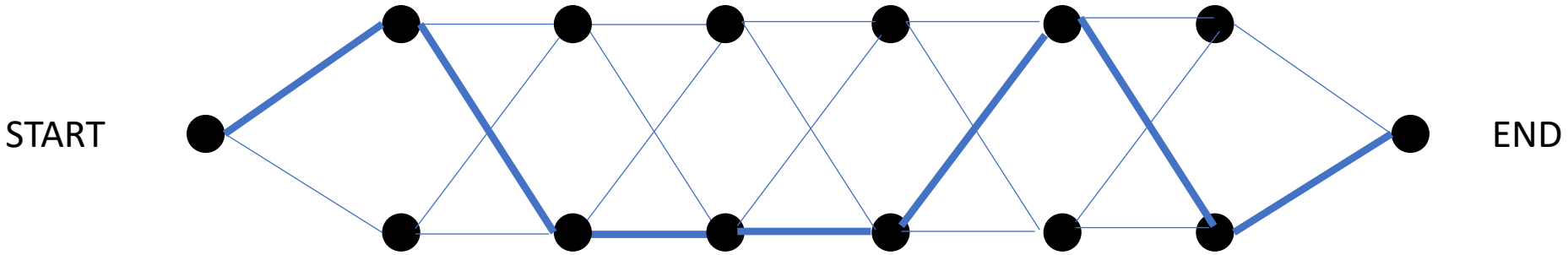
Combining search & neural networks can allow adjusting the tradeoffs.

Why is searching for action sequences hard?

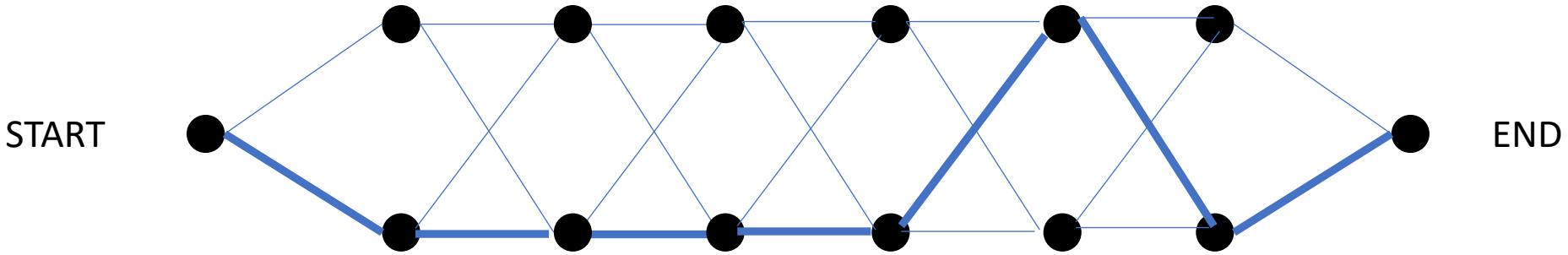
Curse of dimensionality



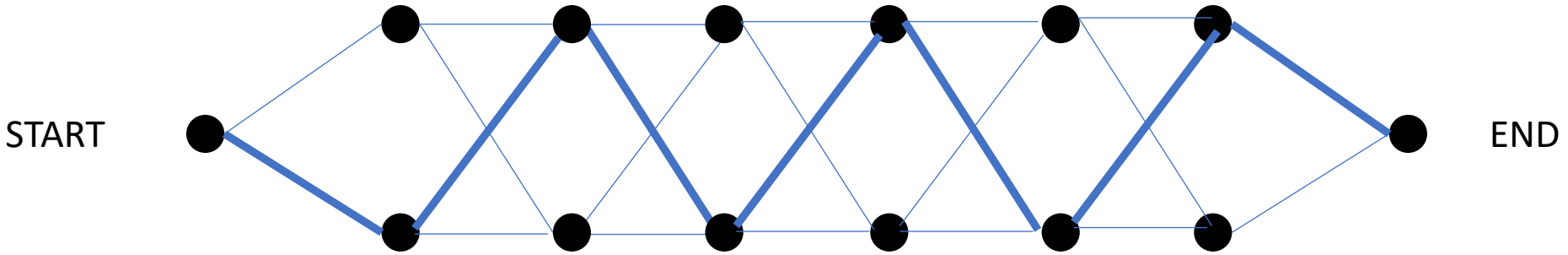
Curse of dimensionality



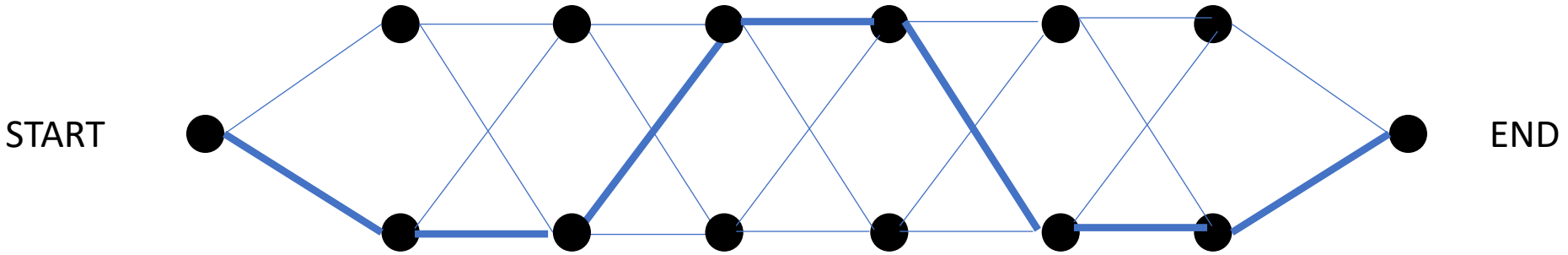
Curse of dimensionality



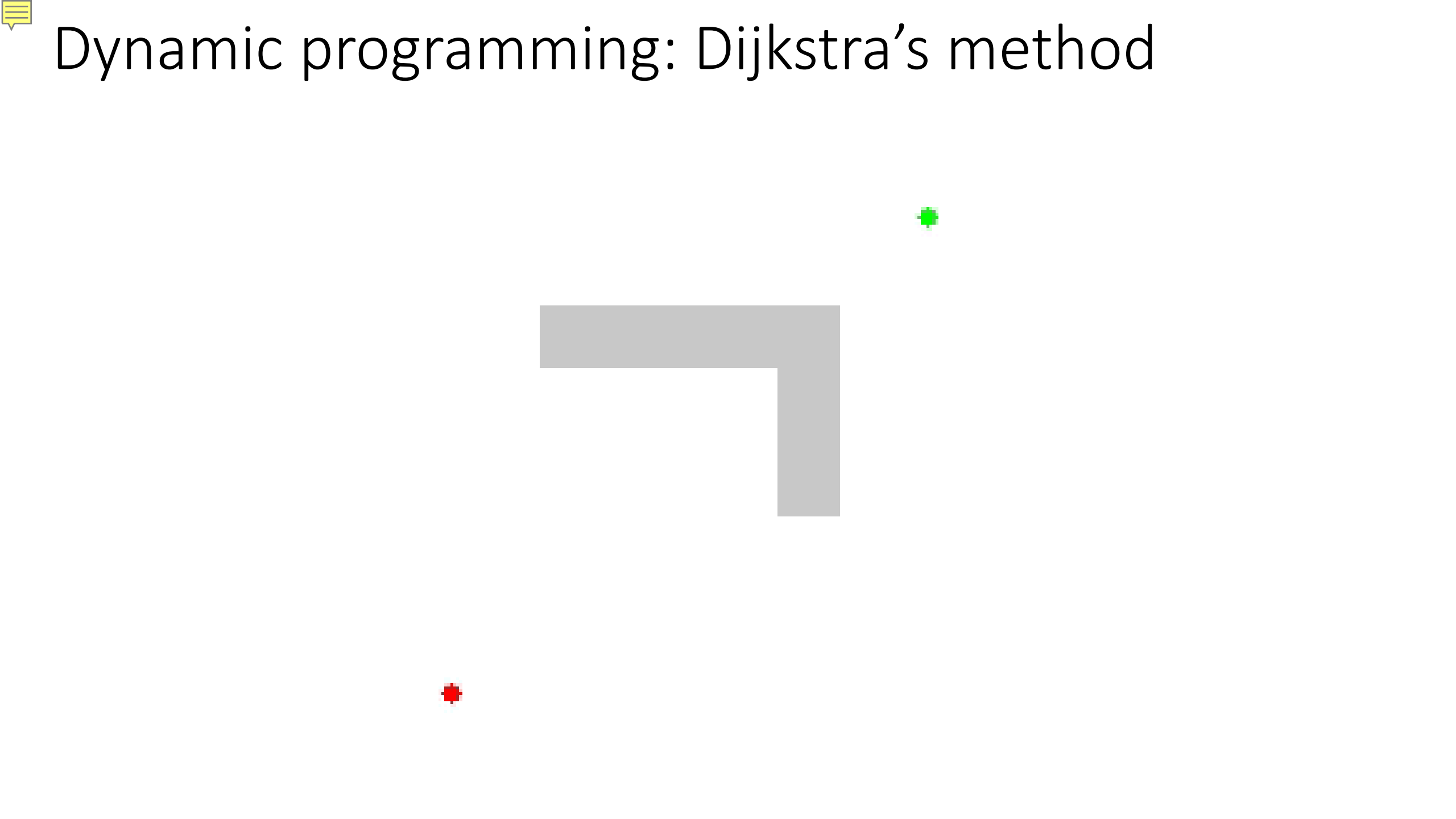
Curse of dimensionality



Curse of dimensionality



Dynamic programming: Dijkstra’s method

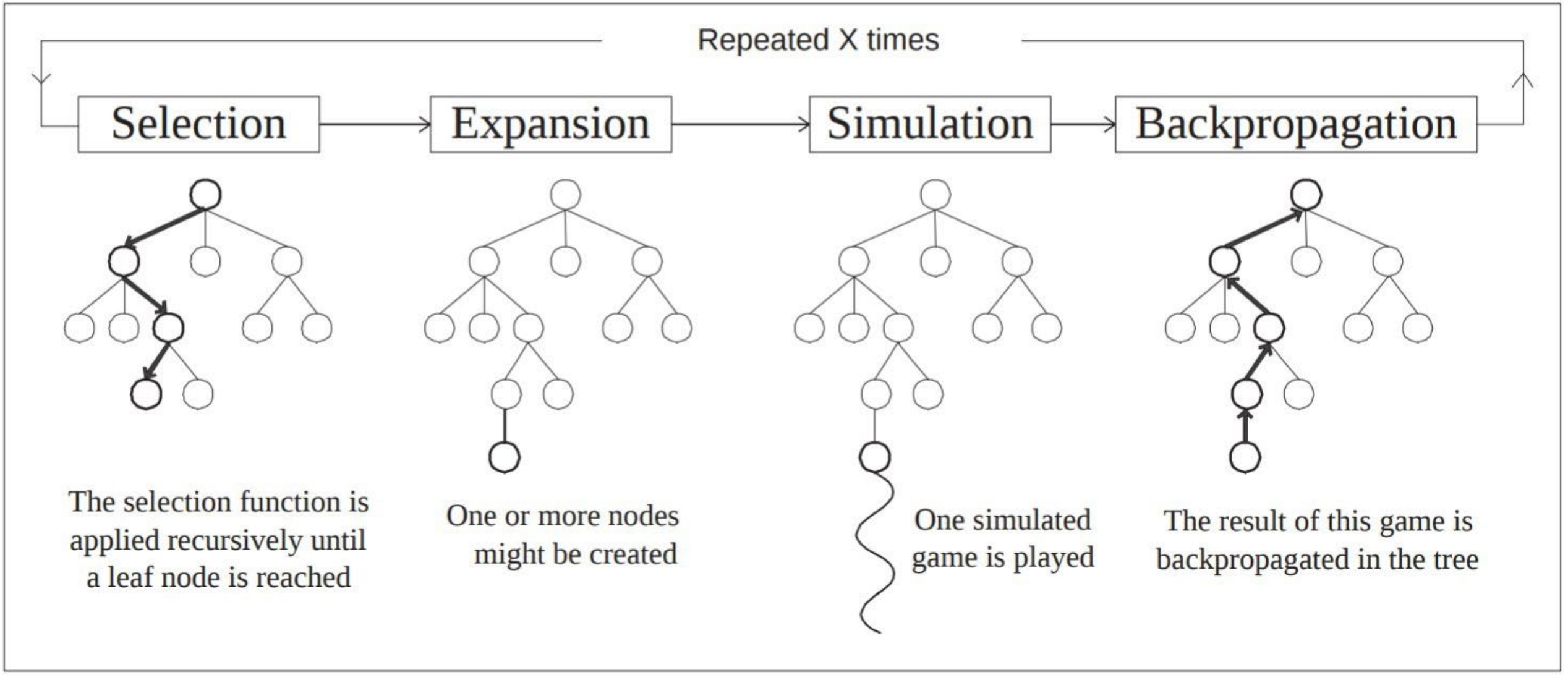


From Dijkstra to A*

Why is it still hard?

- In 2d or 3d, this is trivial
- Unity’s NavMesh implements pathfinding, no need to code yourself
- However, the search process becomes very slow with dozens or hundreds of possible actions
- A* heuristics hard to design for problems beyond pathfinding

Monte Carlo Tree Search to the rescue



Node selection: Maximize the Upper Confidence Bound

$$\overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

Node selection: Maximize the Upper Confidence Bound

$$\overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

Utility of node *j*, i.e., mean of subtree rewards

Node selection: Maximize the Upper Confidence Bound

$$\overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

How many times this node has been selected

Node selection: Maximize the Upper Confidence Bound

$$\overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

How many times the parent node has been visited

Node selection: Maximize the Upper Confidence Bound

$$\overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

Tuning parameter, adjusts the balance between *exploration* and *exploitation*

Node selection: Maximize the Upper Confidence Bound

$$\overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

Exploitation:
maximize utility
(e.g., average number of wins)

Node selection: Maximize the Upper Confidence Bound

$$\overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

Exploration:

High for nodes with a small number of visits n_j


[~ Mario AI Benchmark ~ 0.1.9]
DIFFICULTY: 03
SEED: 123456
TYPE: Overground(0)
LENGTH: 2 of 320
HEIGHT: 2 of 15
×00
×00
×00
Agent: MCTSAgent
PRESSED KEYS:
R>>
ALL KILLS: 0
by Fire: 0
by Shell: 0
by Stomp: 0
TIME
399
FPS:
42
RUN

REPORT

A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play

David Silver[1,2,*,†], Thomas Hubert[1,*], Julian Schrittwieser[1,*], Ioannis Antonoglou[1], Matthew Lai[1], Arthur Guez[1], Marc Lancto…
+ See all authors and affiliations

Science 07 Dec 2018:
Vol. 362, Issue 6419, pp. 1140-1144
DOI: 10.1126/science.aar6404

One program to rule them all

Computers can beat humans at increasingly complex games, including chess and Go. However, these programs are typically constructed for a particular game, exploiting its properties, such as the symmetries of the board on which it is played. Silver *et al.* developed a program called AlphaZero, which taught itself to play Go, chess, and shogi (a Japanese version of chess) (see the Editorial, and the Perspective by Campbell). AlphaZero managed to beat state-of-the-art programs specializing in these three games. The ability of AlphaZero to adapt to various game rules is a notable step toward achieving a general game-playing system.

Science, this issue p. **1140**; see also pp. **1087** and **1118**

Science
Vol 362, Issue 6419
07 December 2018

A Survey of Monte Carlo Tree Search Methods

Cameron Browne, *Member, IEEE,* Edward Powley, *Member, IEEE,* Daniel Whitehouse, *Member, IEEE,* Simon Lucas, *Senior Member, IEEE,* Peter I. Cowling, *Member, IEEE,* Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton

Abstract—Monte Carlo Tree Search (MCTS) is a recently proposed search method that combines the precision of tree search with the generality of random sampling. It has received considerable interest due to its spectacular success in the difficult problem of computer Go, but has also proved beneficial in a range of other domains. This paper is a survey of the literature to date, intended to provide a snapshot of the state of the art after the first five years of MCTS research. We outline the core algorithm's derivation, impart some structure on the many variations and enhancements that have been proposed, and summarise the results from the key game and non-game domains to which MCTS methods have been applied. A number of open research questions indicate that the field is ripe for future work.

Index Terms—Monte Carlo Tree Search (MCTS), Upper Confidence Bounds (UCB), Upper Confidence Bounds for Trees (UCT), Bandit-based methods, Artificial Intelligence (AI), Game search, Computer Go.

1 Introduction

Monte Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results. It has already had a profound impact on Artificial Intelligence (AI) approaches for domains that can be represented as trees of sequential decisions, particularly games and planning problems.

In the five years since MCTS was first described, it has become the focus of much AI research. Spurred on by some prolific achievements in the challenging task of computer Go, researchers are now in the pro-

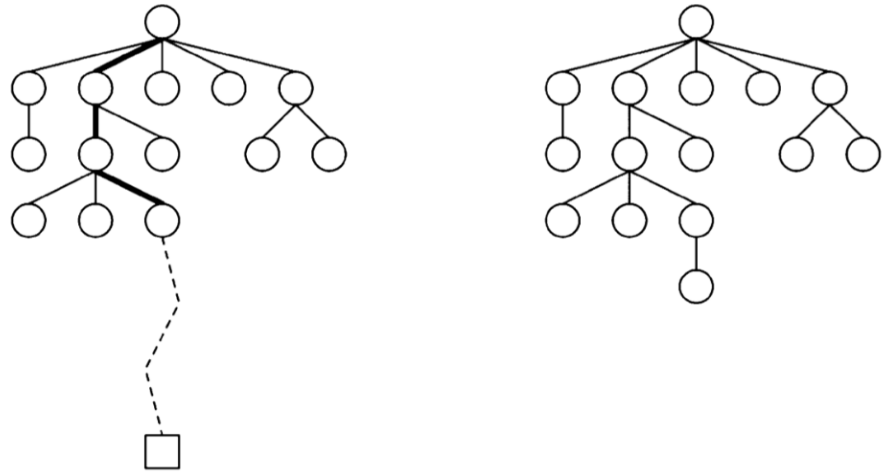


Fig. 1. The basic MCTS process [17].

Monte Carlo Tree Search resources

The best visualizations I've found:

https://int8.io/monte-carlo-tree-search-beginners-guide/

Game AI Book section on MCTS (page 45): http://gameaibook.org/book.pdf

Colab notebook: MCTS for OpenAI Gym environments

https://colab.research.google.com/github/yandexdataschool/Practical_RL/blob/spring19/week10_planning/seminar_MCTS.ipynb

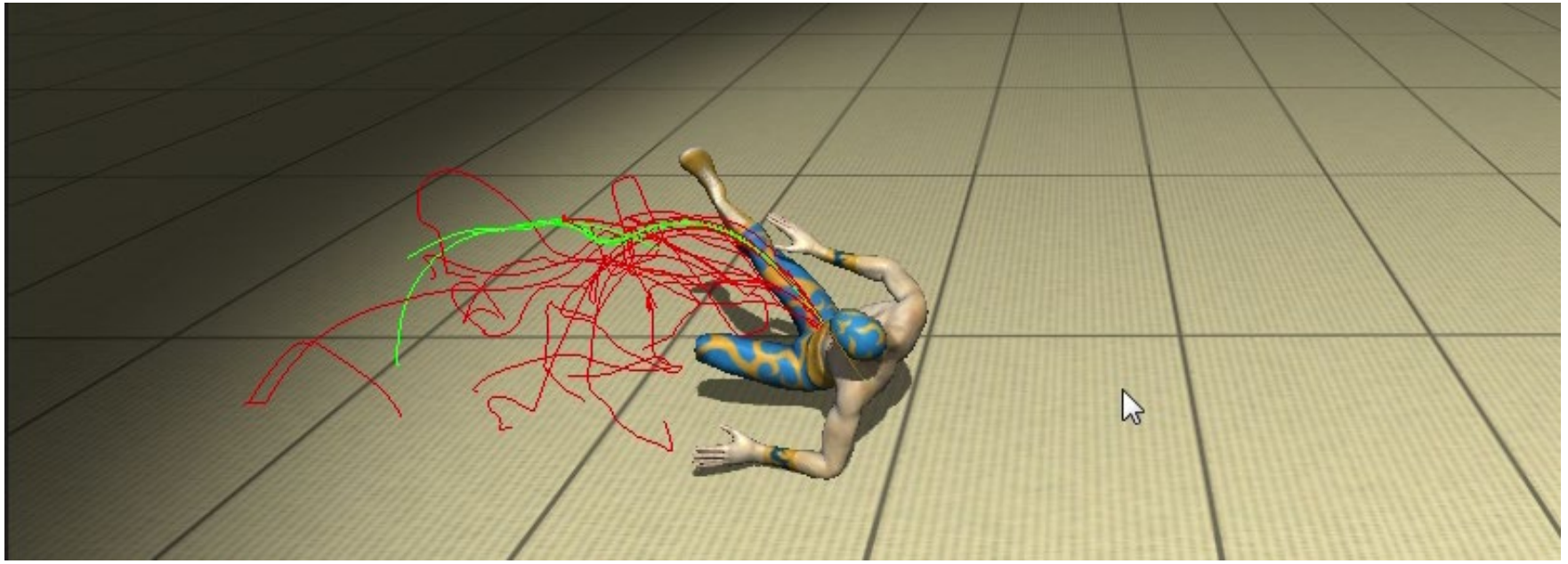
MCTS as a Deep Reinforcement Learning alternative

- Both find solutions to a Markov Decision Process defined by states, actions, and rewards
- MCTS benefit: usually gets results faster, but requires the ability to save and restore simulation state
  - IMPORTANT: You can quickly iterate on the MDP design, e.g., what kinds of rewards to use, without always waiting for a day for RL training
- MCTS drawback: requires more CPU than simply querying a policy network for an action based on the current observation.
- Recommended use: If you can save & restore state and have discrete actions, start with MCTS. Switch to RL when you're sure that your reward function produces good results.

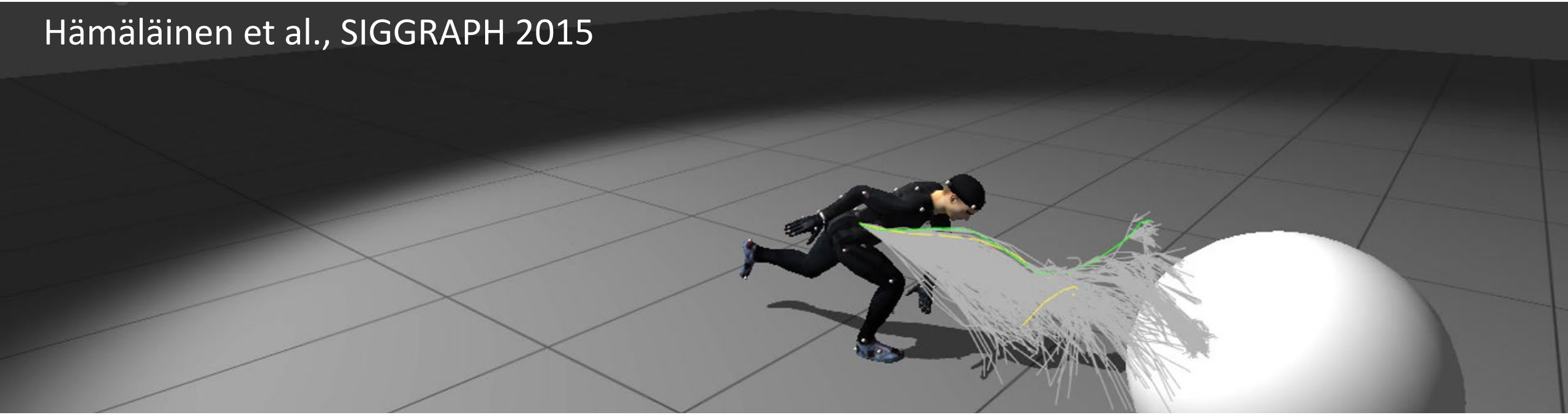
What about continuous actions?

Our SIGGRAPH 2014 work: no tree search

- Simply run a number of rollouts from the current state
- Search space reduced by encoding action sequences as pose splines
- Take action following the best rollout
- The distribution of rollout actions is adapted



Control Particle Belief Propagation (C-PBP)



$$\mathcal{P}(\mathbf{z}) \propto \left(\prod_{k} \psi_k(\mathbf{z}_k) \right) \left(\prod_{k=1}^{K} \Psi_{\text{fwd}}(\mathbf{z}_{k-1}, \mathbf{z}_k) \right) \left(\prod_{k=0}^{K-1} \Psi_{\text{bwd}}(\mathbf{z}_{k+1}, \mathbf{z}_k) \right)$$

FDI-MCTS (Rajamäki & Hämäläinen, 2017)

- Simple MCTS variant for fixed-horizon continuous control. Simplification of C-PBP, no cumbersome math.
- Combines deep neural networks and tree search
- Benefit: Very fast convergence, humanoid walking in less than a minute on a single computer
- Drawback: Requires forward simulation and the resulting policy not always as robust as, e.g., with PPO


Simulation time 00:00: 0 (frame 0), total computing time 0.08 s
Total training simulation steps: 2304
Number of trajectories: 64
Controller update: 84 ms,
Planning horizon: 1.2 seconds
Pruning threshold 20000.0
state cost 1085.57, current control cost 4.35, end state cost 334.36
Rajamäki & Hämäläinen, SCA 2017

FDI-MCTS algorithm overview

1. Simulate a number of trajectories forward
   - Trajectories utilize different types of information
   - One trajectory repeats the best trajectory of the previous frame
   - Some trajectories: neural network policy + noise
   - Some trajectories: find nearest memorized states, use their stored actions + noise
   - Some trajectories: random actions
2. During the forward simulation, terminate worst trajectories, and reassign their simulation resources by forking the best trajectories
3. After forward simulation, step the master simulation forward with the first action of the best trajectory.
4. Memorize the state and action taken
5. In a background thread, constantly retrain the neural network policy with the memorized states and actions. Also update the nearest neighbor search data structure (a decision forest)

Recent continuous control MCTS

- Moerland, Thomas M., et al. "A0C: Alpha zero in continuous action space." *arXiv preprint arXiv:1805.09613* (2018).

Predictive Physics Simulation in Game Mechanics
ACM CHI PLAY 2017

Perttu Hämäläinen (Aalto University)
Xiaoxiao Ma (Aalto University)
Jari Takatalo (Aalto University)
Julian Togelius (NYU Tandon School of Engineering)

Appendix: Extra material

Neuroevolution

- Apply an evolutionary optimization method like CMA-ES directly to policy network parameters θ (neural network weights & biases)
- Some algorithms like NEAT optimize both network parameters and network architecture
- Optimization objective $f(\theta)$ = sum of rewards over episode

One iteration:

1. Sample parameters $\theta_1, \ldots, \theta_N$ for N neural networks
2. Run 1 or more episode with each network, compute $f(\theta)$
3. Update the sampling distribution

Geijtenbeek 2013 (CMA-ES neuroevolution)

Flexible Muscle-Based Locomotion for Bipedal Creatures

SIGGRAPH ASIA 2013

Thomas Geijtenbeek
Michiel van de Panne
Frank van der Stappen

Papers on neuroevolution

Such, Felipe Petroski, et al. "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning." *arXiv preprint arXiv:1712.06567* (2017)

Salimans, Tim, et al. "Evolution strategies as a scalable alternative to reinforcement learning." *arXiv preprint arXiv:1703.03864* (2017).

Optimizing action sequences: Summary

Deep RL:

+ Produces a policy network that can compute actions very efficiently

- Training can be very slow

Neuroevolution:

+ Produces a policy network that can compute actions very efficiently

+ Simple to implement: only policy network, no training

- With large policy networks, can be even slower than Deep RL and/or requires extra tricks that make implementation more complex

Forward search methods like MCTS:

+ Find good actions fast, without training or optimizing neural networks

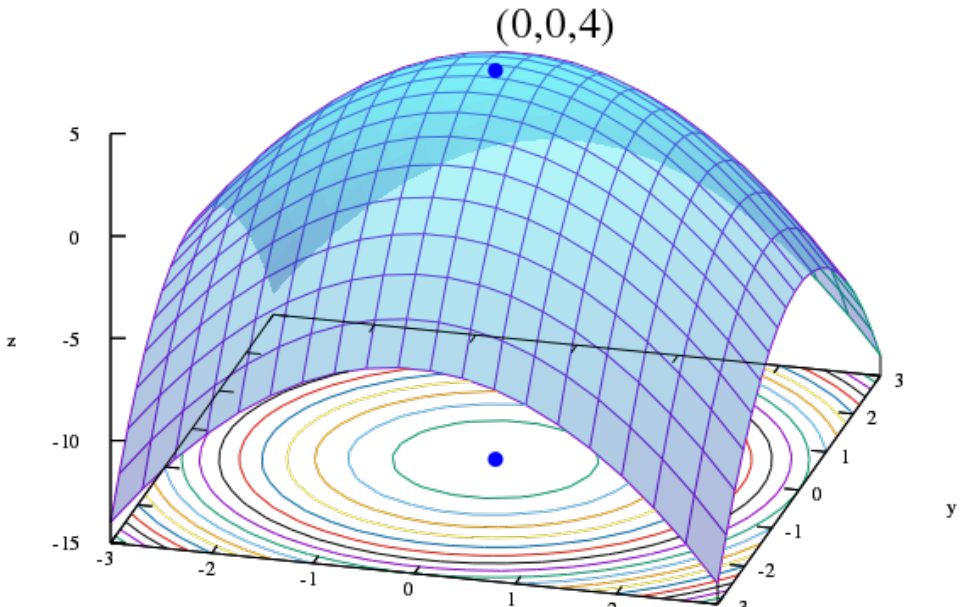
- Requires forward simulation (many times faster than real-time simulation)

- Requires capability to save and load game or simulation state

- Seems to be hard to implement in a clean, generic, and easy-to-use manner in Unity. Challenge to students with Unity experience: Can you figure out a way?

Curvature

- Gradient descend: form a 1st order Taylor expansion, i.e., linear approximation of $f(\mathbf{x})$, use that to determine search direction
- Newton's method: a 2nd order model using 2nd derivatives
- If $f(\mathbf{x})$ is quadratic, model fits perfectly and directly gives the optimum
- An analogue of Newton's method for action sequences: Differential Dynamic Programming (DDP)



Curvature

- Newton's method requires the Hessian matrix of second derivatives, requires N^2 function evaluations and memory
- BFGS, L-BFGS: approximate Hessian indirectly
- If $f(\mathbf{x})$ is sum of squares: Gauss-Newton, Levenberg-Marquardt.
- An analogue of Gauss-Newton for action sequences: Iterative Linear Quadratic Gaussian (ILQG, e.g., Tassa et al. 2012)