What every designer & developer should know about

Mathematical Optimization

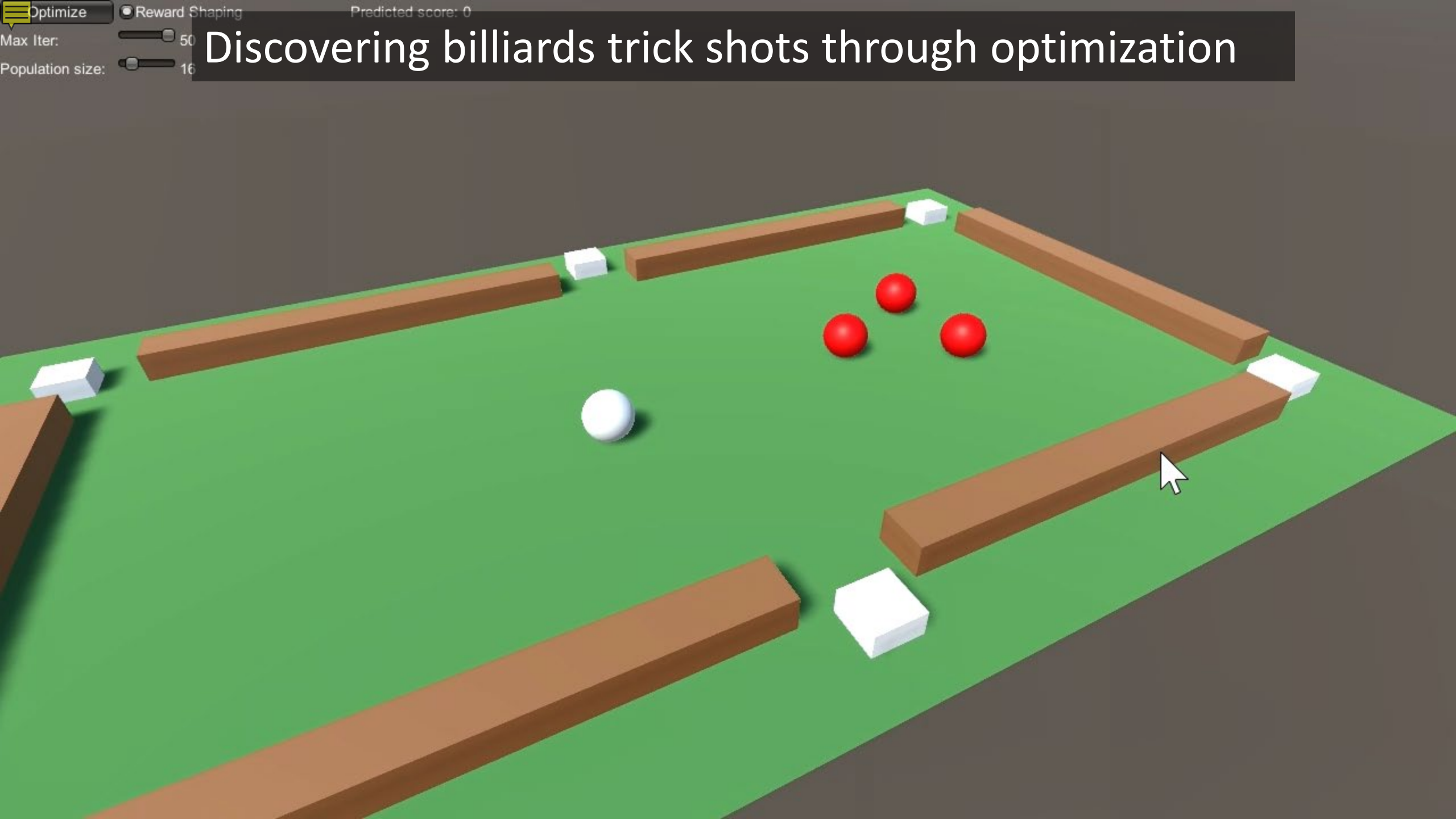
Intelligent Computational Media, spring 2024

Aalto University

Prof. Perttu Hämäläinen

perttu.hamalainen@aalto.fi

All course materials and syllabus: https://github.com/PerttuHamalainen/MediaAI


Optimize
Reward Shaping
Predicted score: 0
Max Iter:
50
Population size:
16
Discovering billiards trick shots through optimization

Synthesizing abstract art through optimization

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

Contents

- Optimization: the what and why
- Some intuitions from simple random search
- Gradient-based optimization
- Sampling-based optimization
- From continuous-valued to discrete/combinatorial 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)

Fundamentally, all AI & ML is optimization

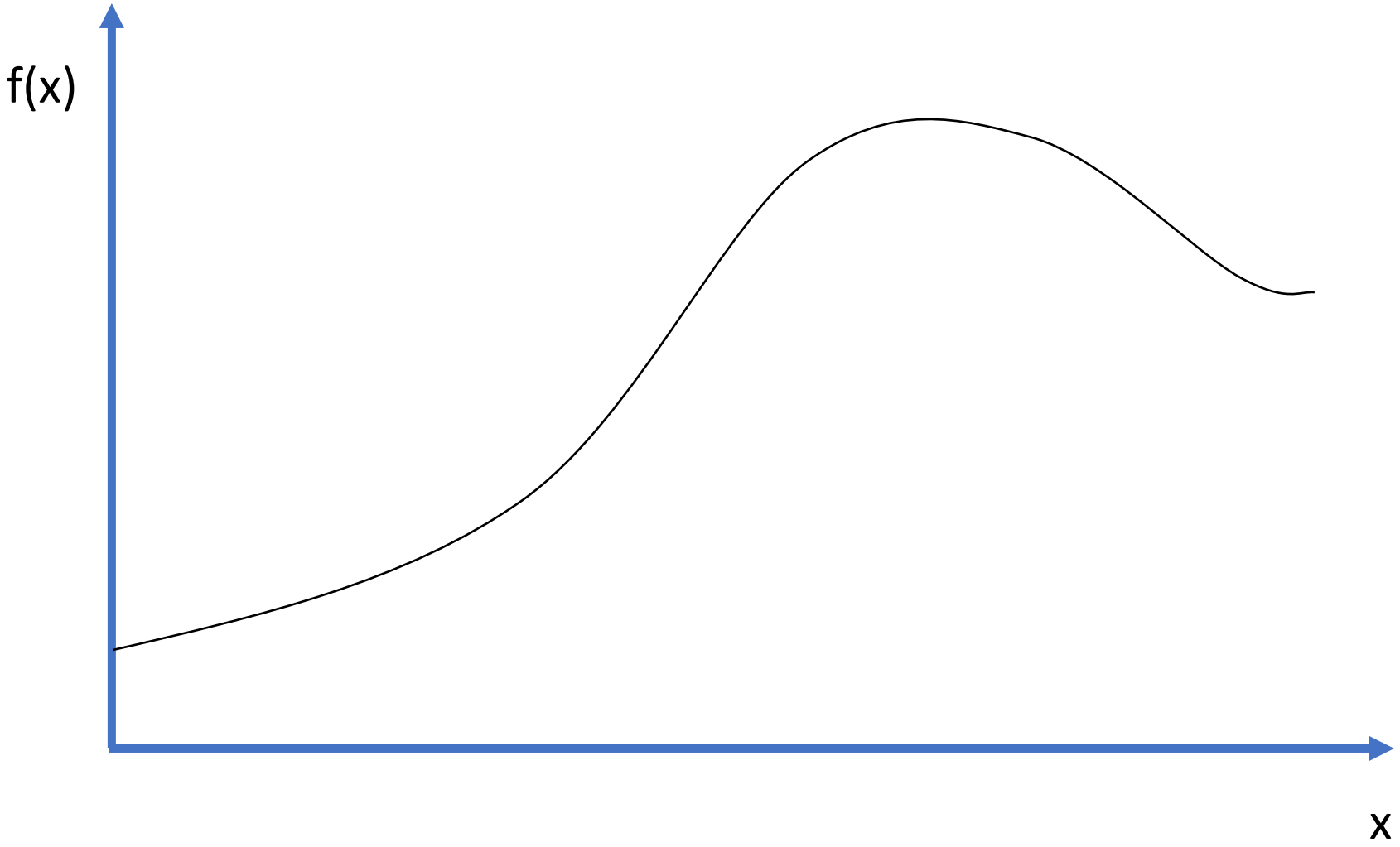
Intuition: Simple random search

The simplest optimization procedure

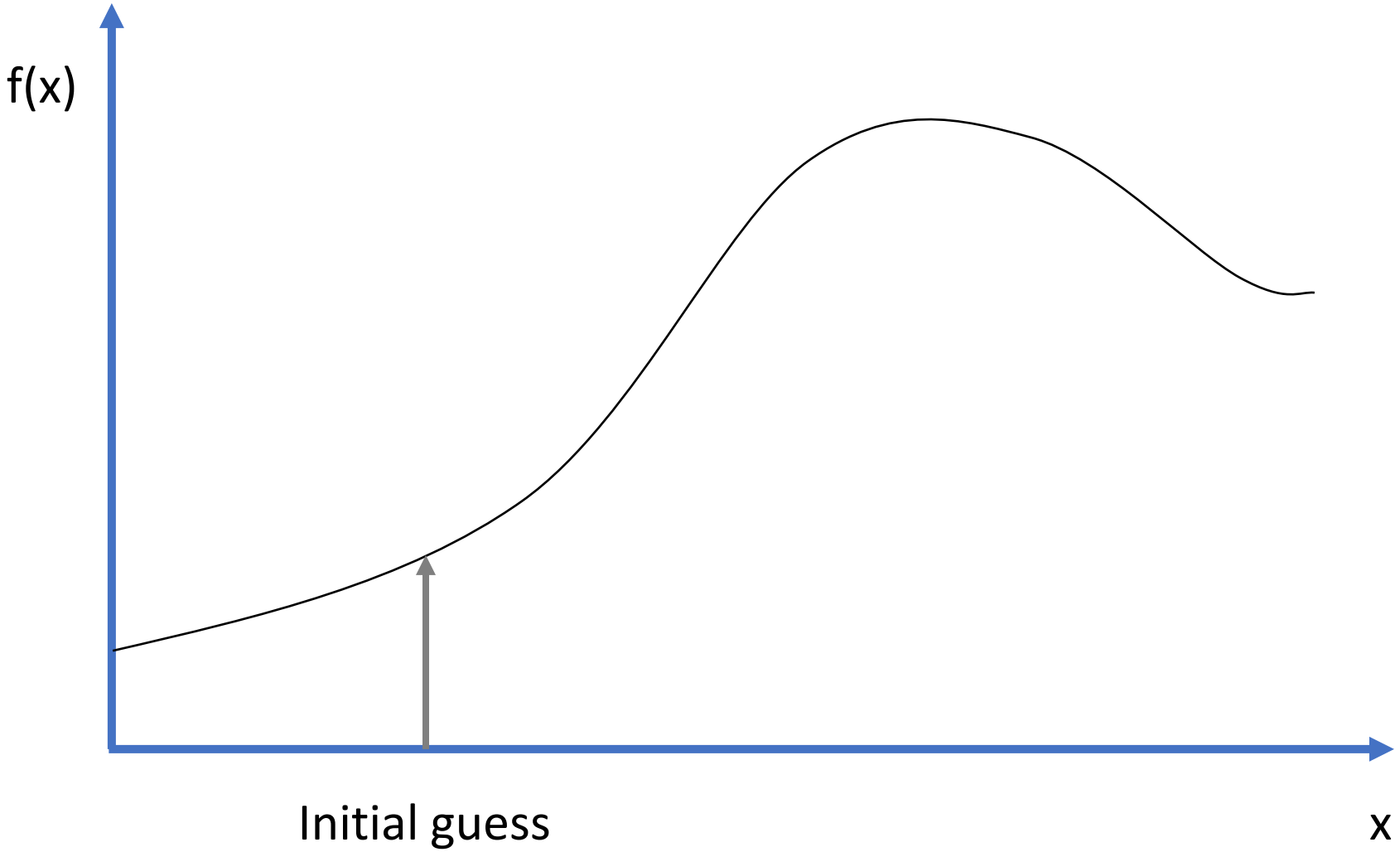
1. Initialize **x** randomly or based on some initial guess
2. Try some new $\mathbf{x}_{new}$ (typically near the current **x**)
3. If $f(\mathbf{x}_{new}) > f(\mathbf{x})$, set $\mathbf{x} = \mathbf{x}_{new}$ //assuming $f()$ is to be maximized
4. Repeat steps 2 & 3

This is similar to *simulated annealing* with specific parameters

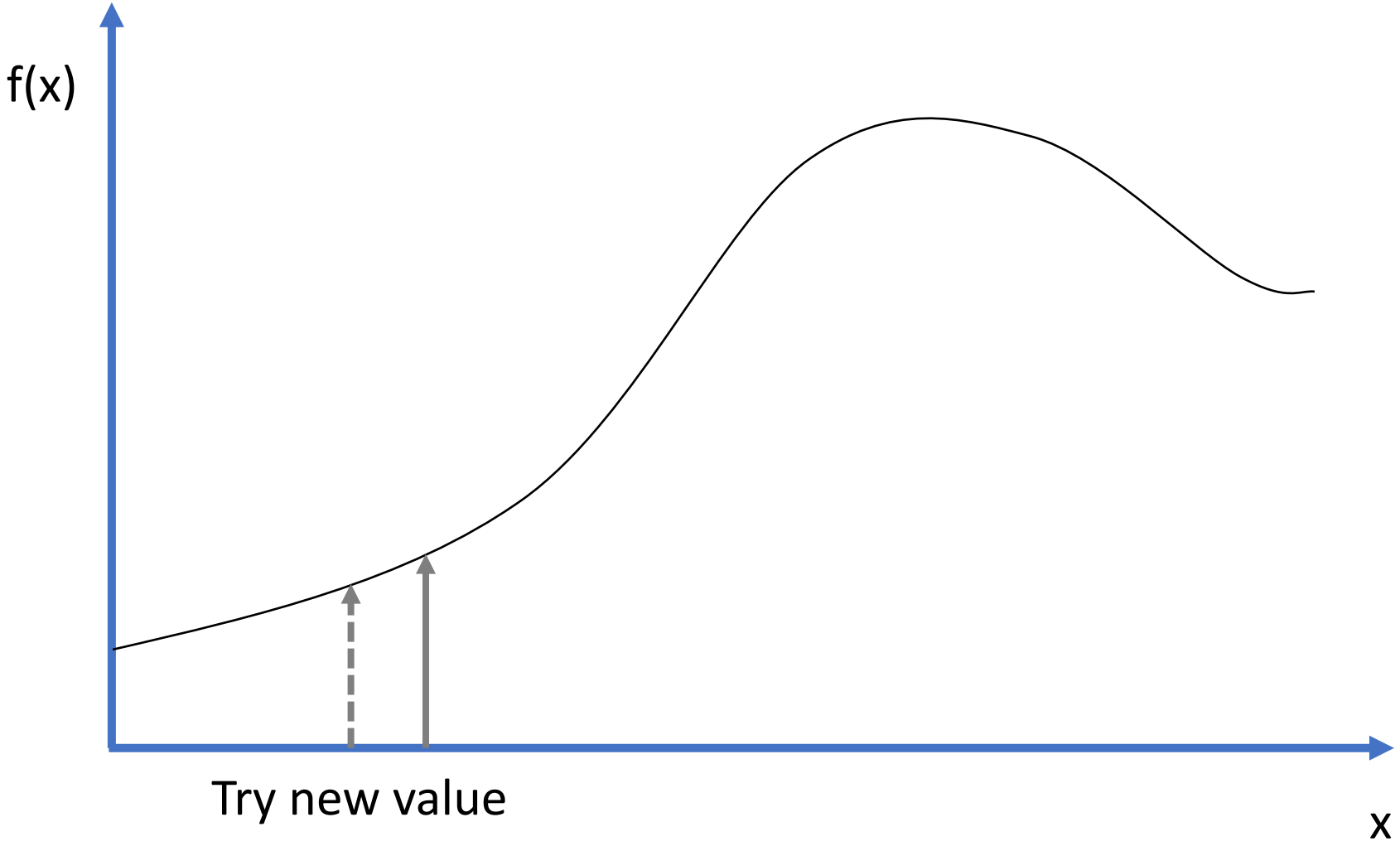
The simplest optimization procedure



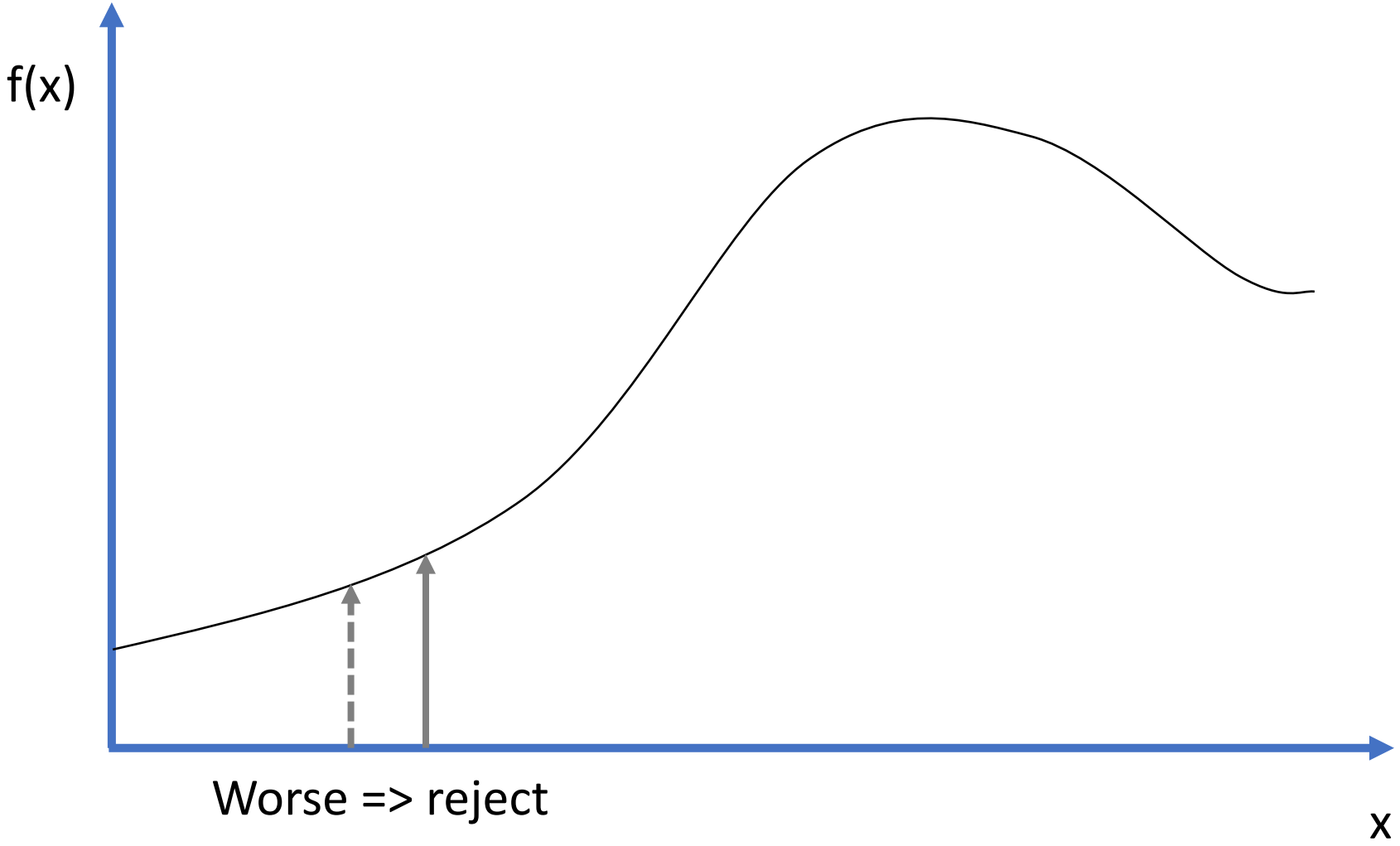
The simplest optimization procedure



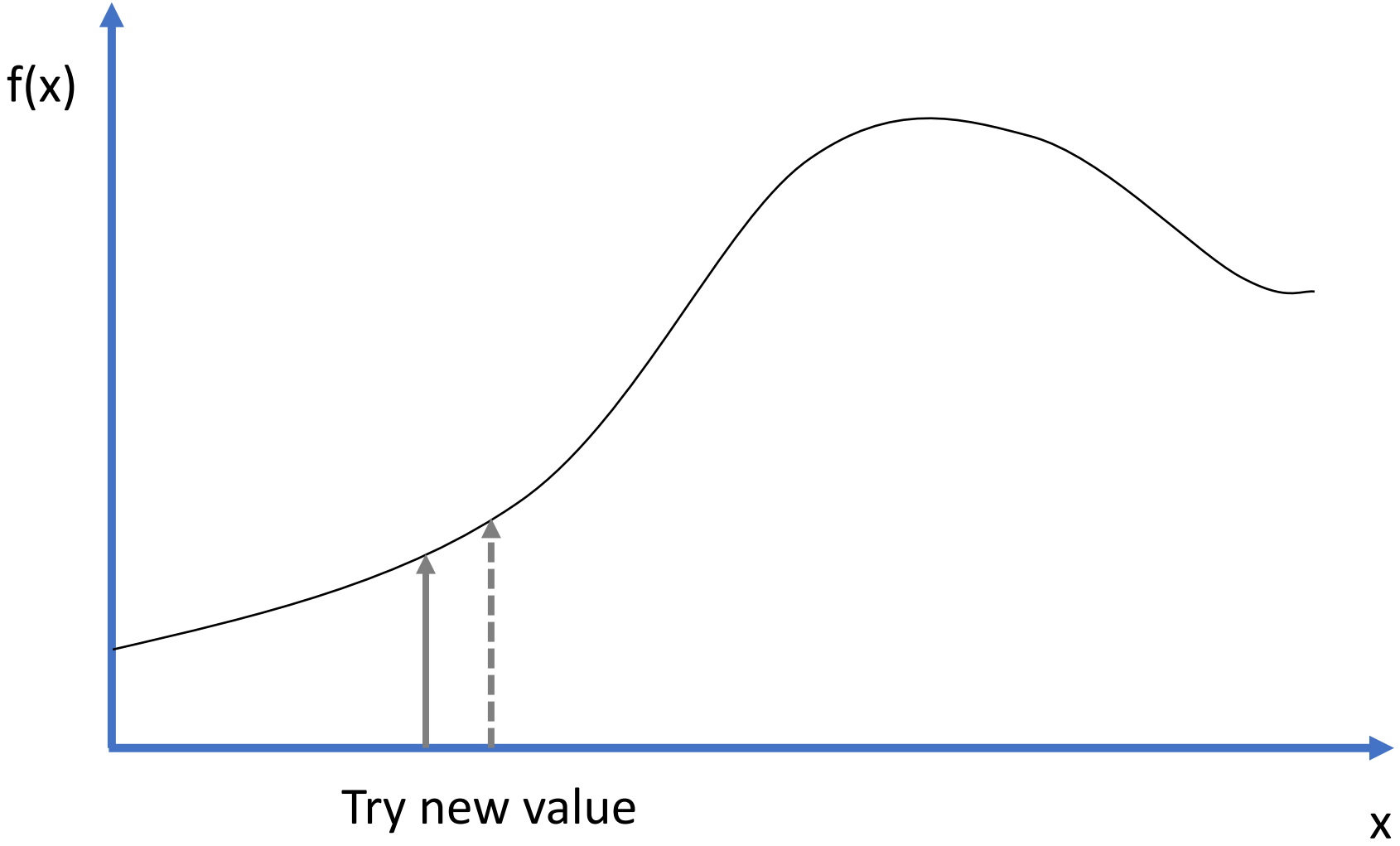
The simplest optimization procedure



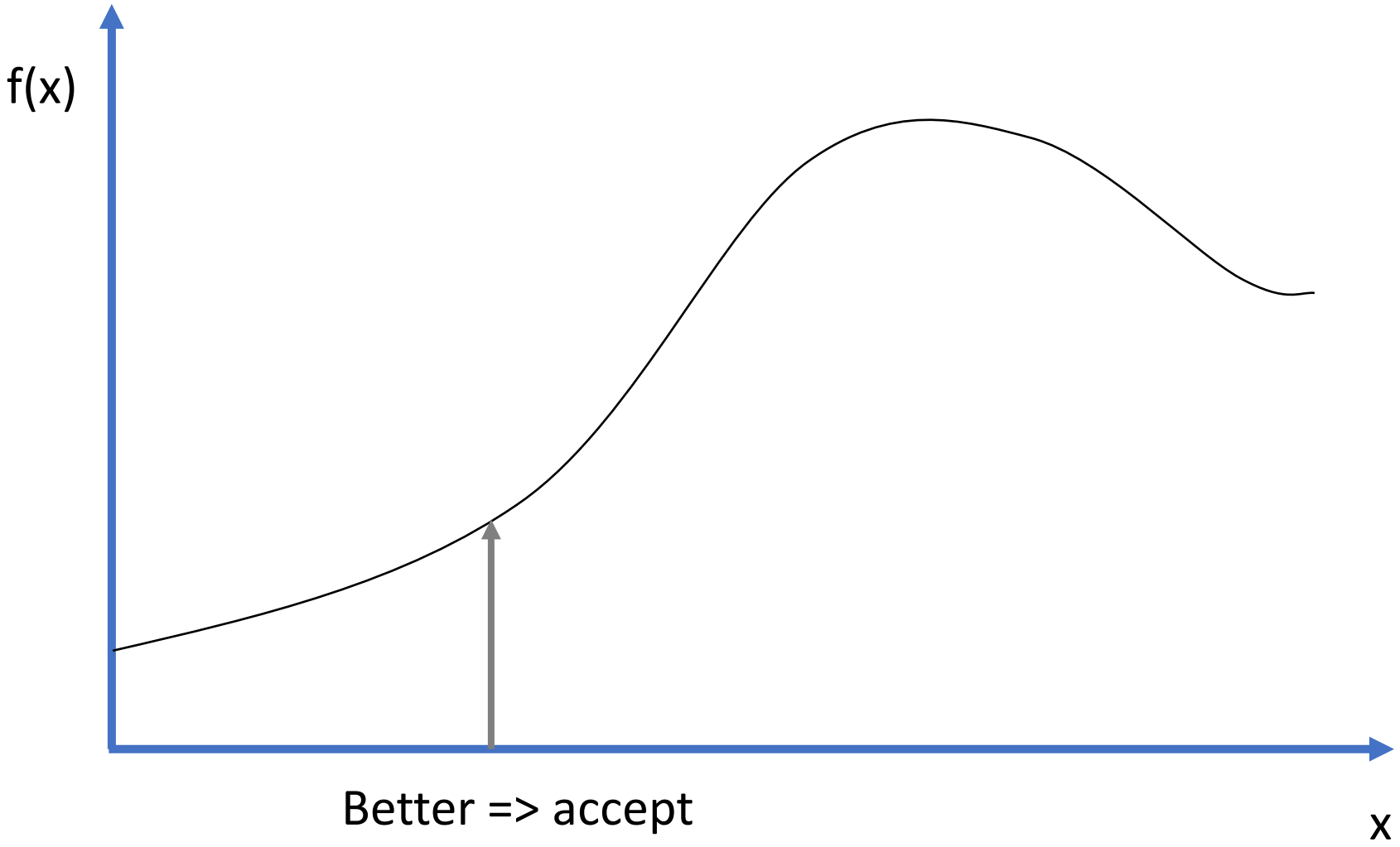
The simplest optimization procedure



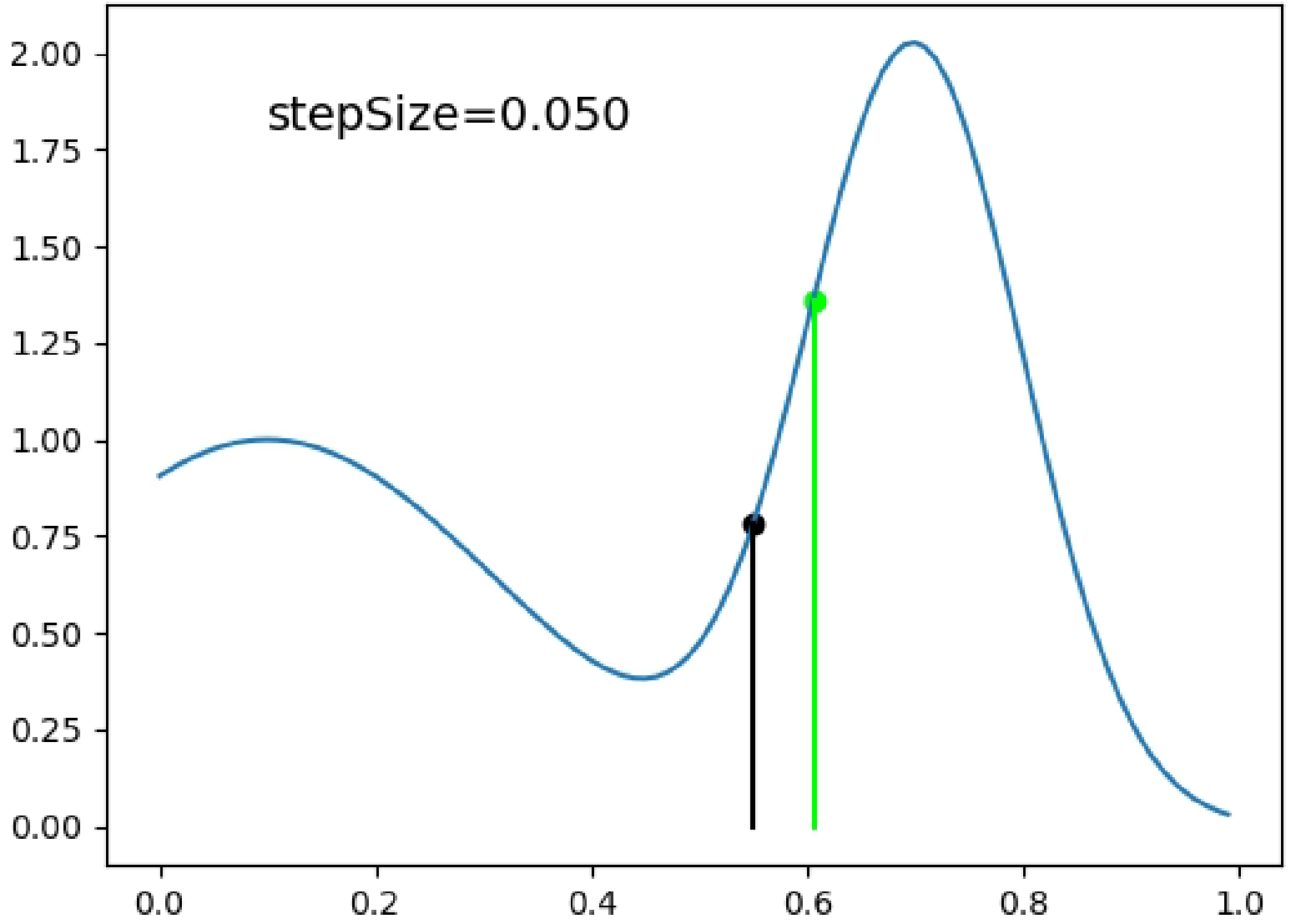
The simplest optimization procedure



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

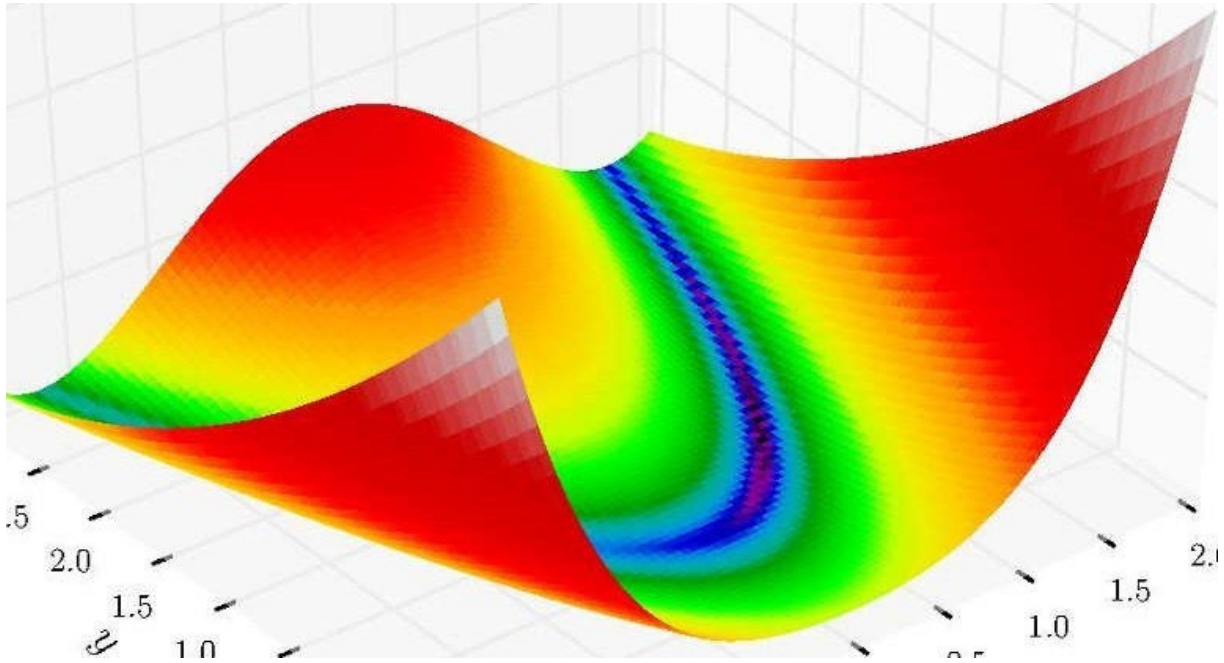
1. Initialize **x** randomly or based on some initial guess
2. Try some new $\mathbf{x}_{new}$ (typically near the current **x**)
3. If $f(\mathbf{x}_{new}) > f(\mathbf{x})$, set $\mathbf{x} = \mathbf{x}_{new}$ //assuming $f()$ is to be maximized
4. Repeat steps 2 & 3

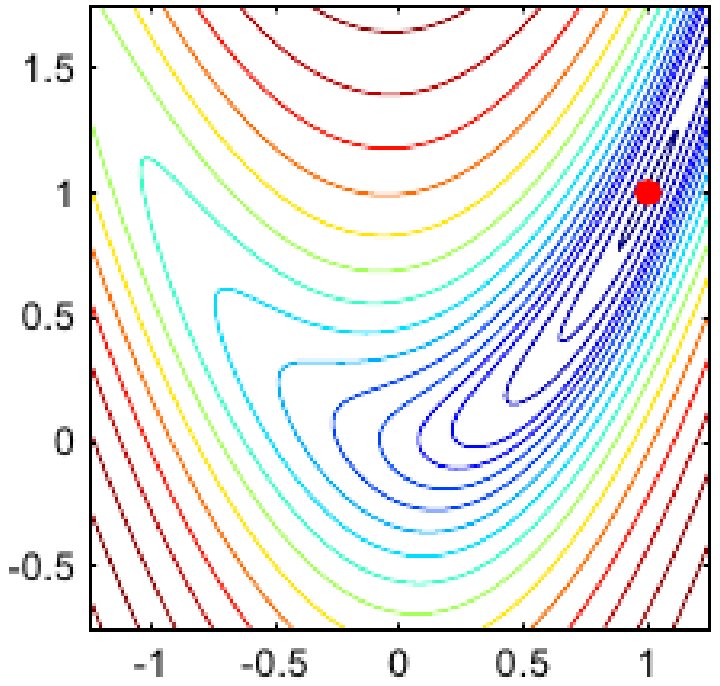
Modifications:

- 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?

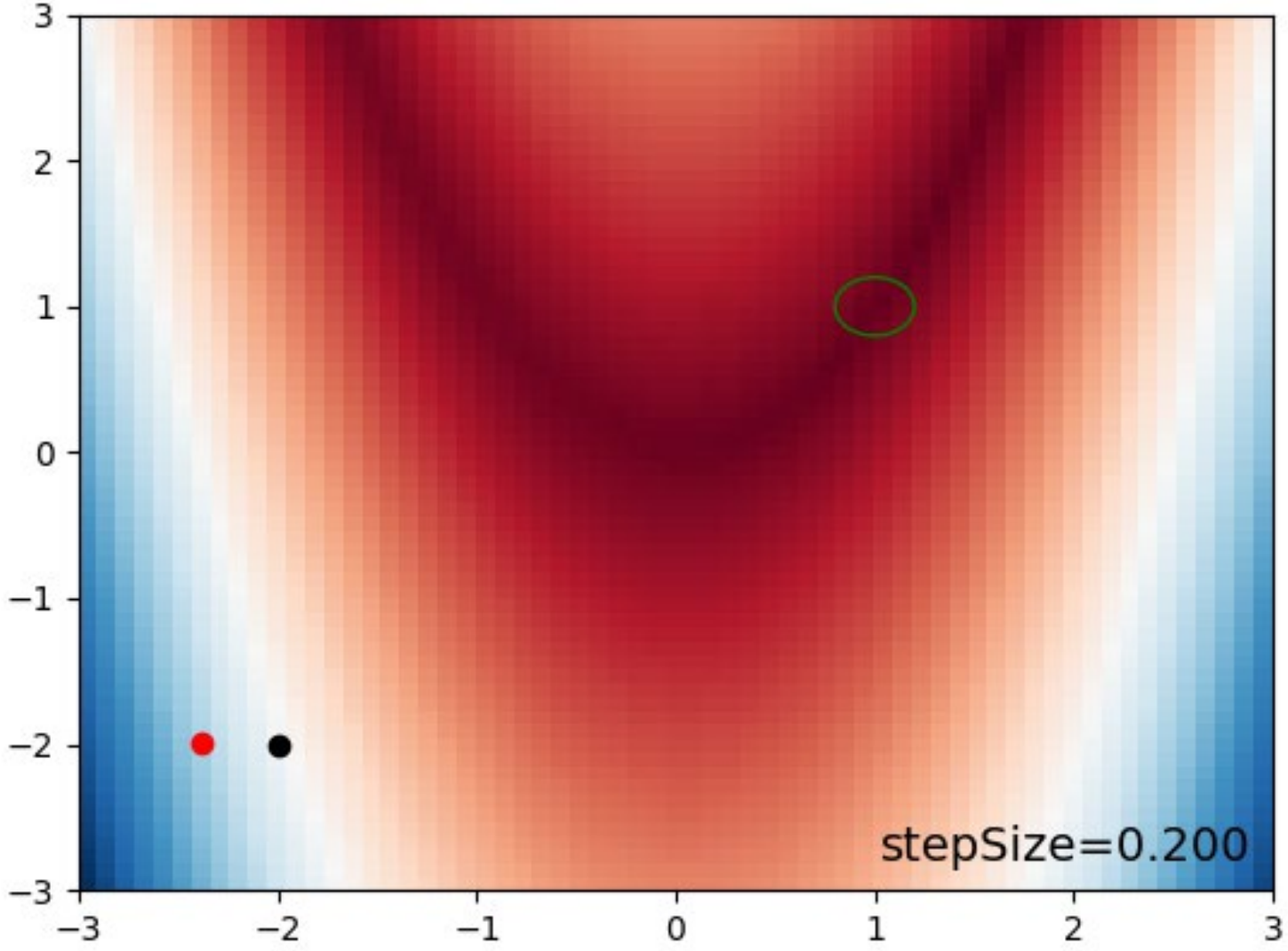
Curse of dimensionality

- 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!

For example, if one tries even just 2 values for each x_1, x_2, ..., x_N, there’s 2^N possible combinations.

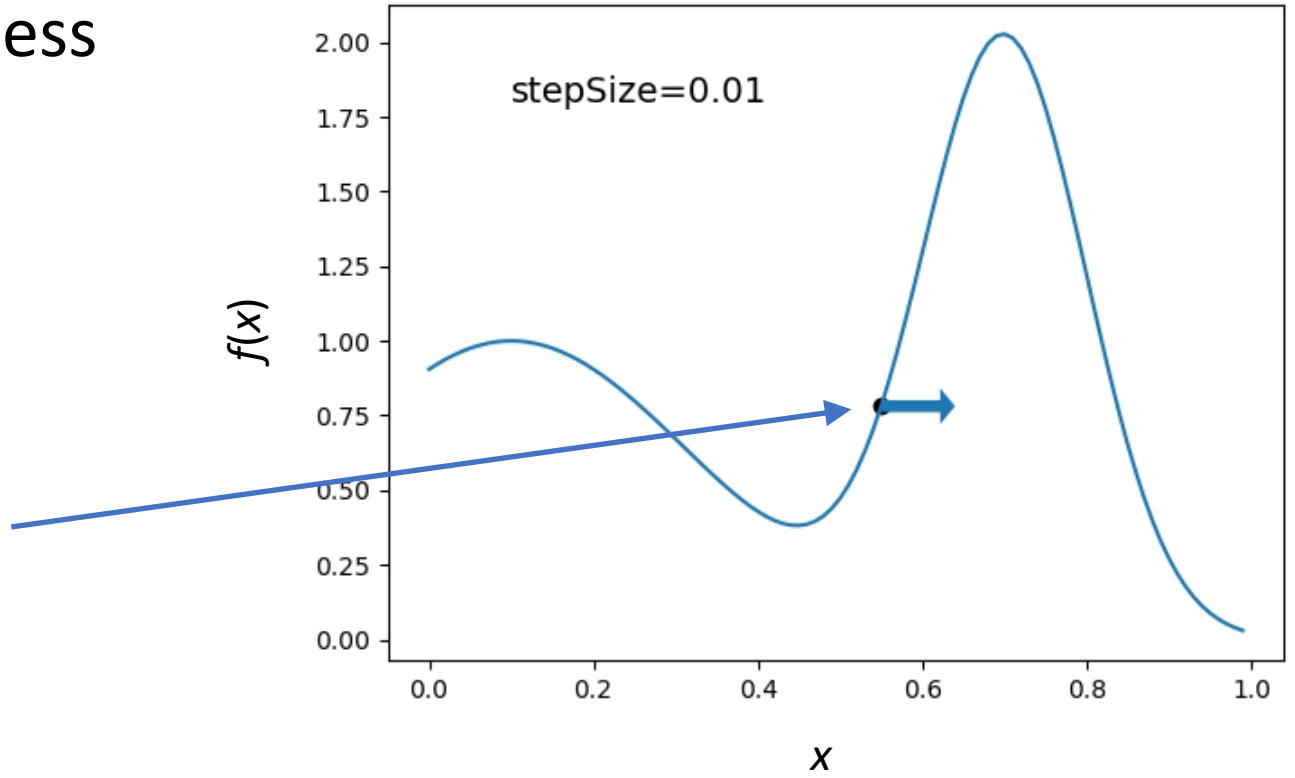
Gradient-based optimization

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:

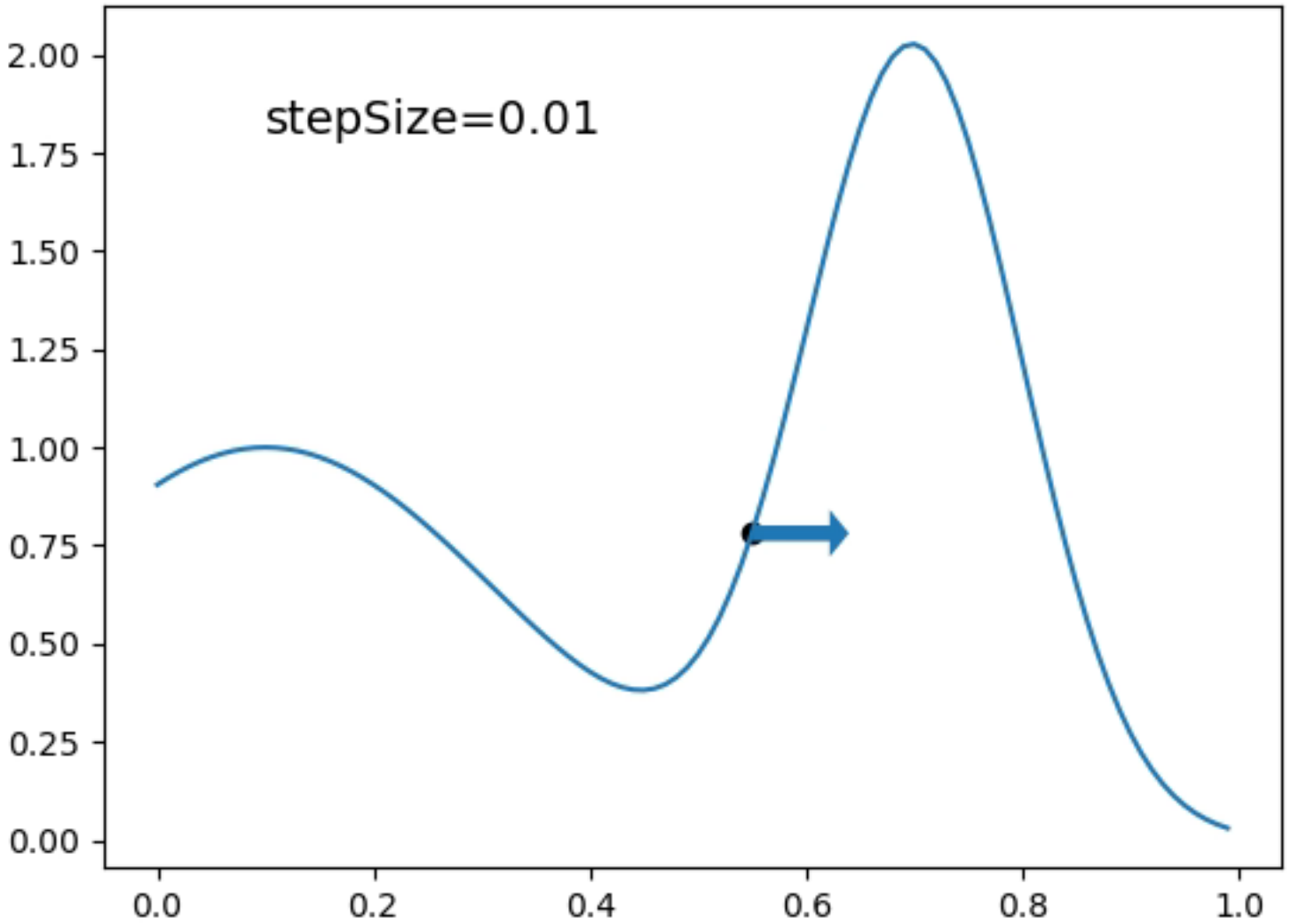
To make *f*(*x*) larger,
move *x* to the right



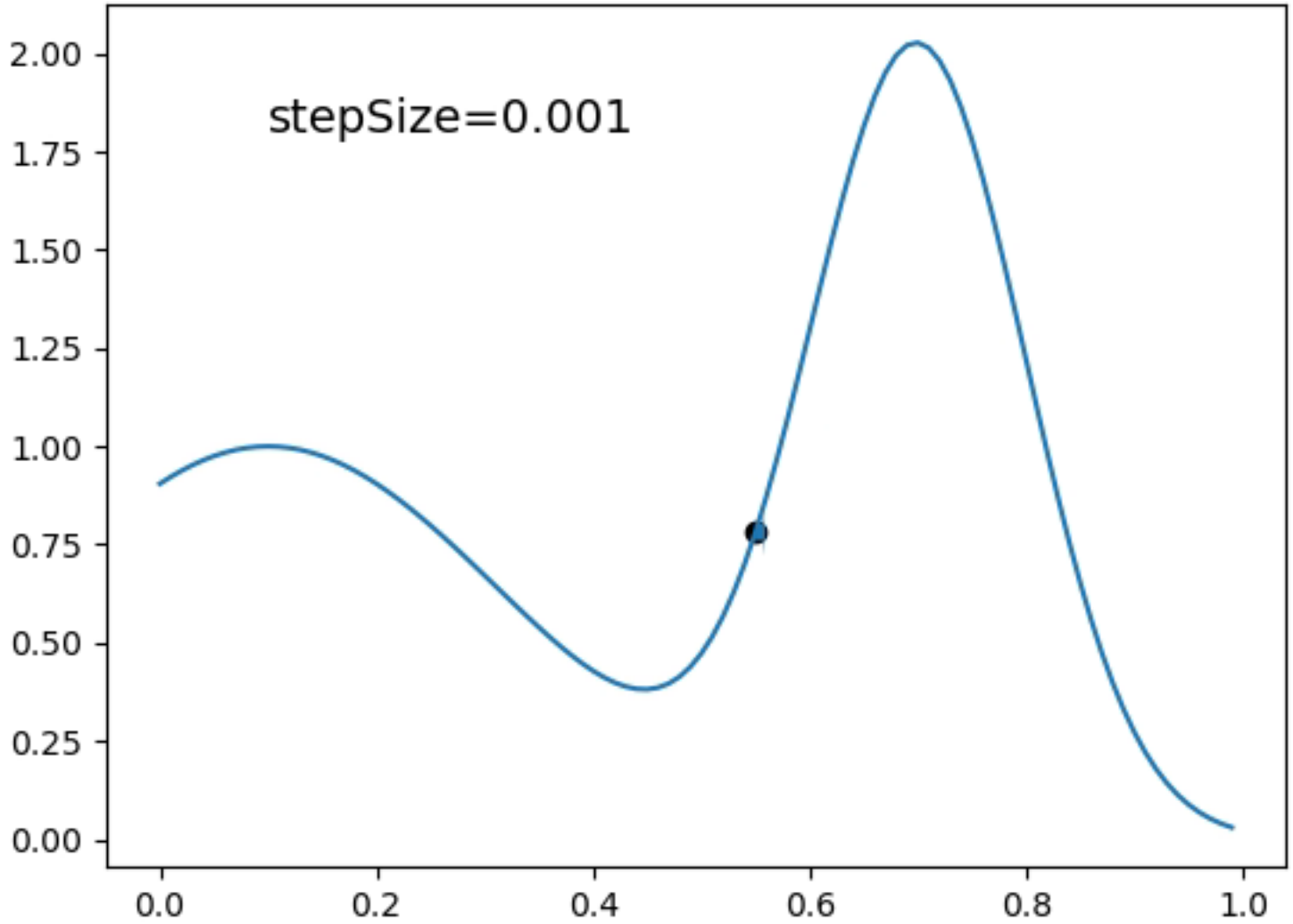
Computing the gradient

- TensorFlow, PyTorch etc. do this automatically if your objective function math is composed from the provided operations like torch.square(), torch.exp()
- 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: $\mathbf{x}_{new} = \mathbf{x} + stepSize * \nabla f(\mathbf{x})$



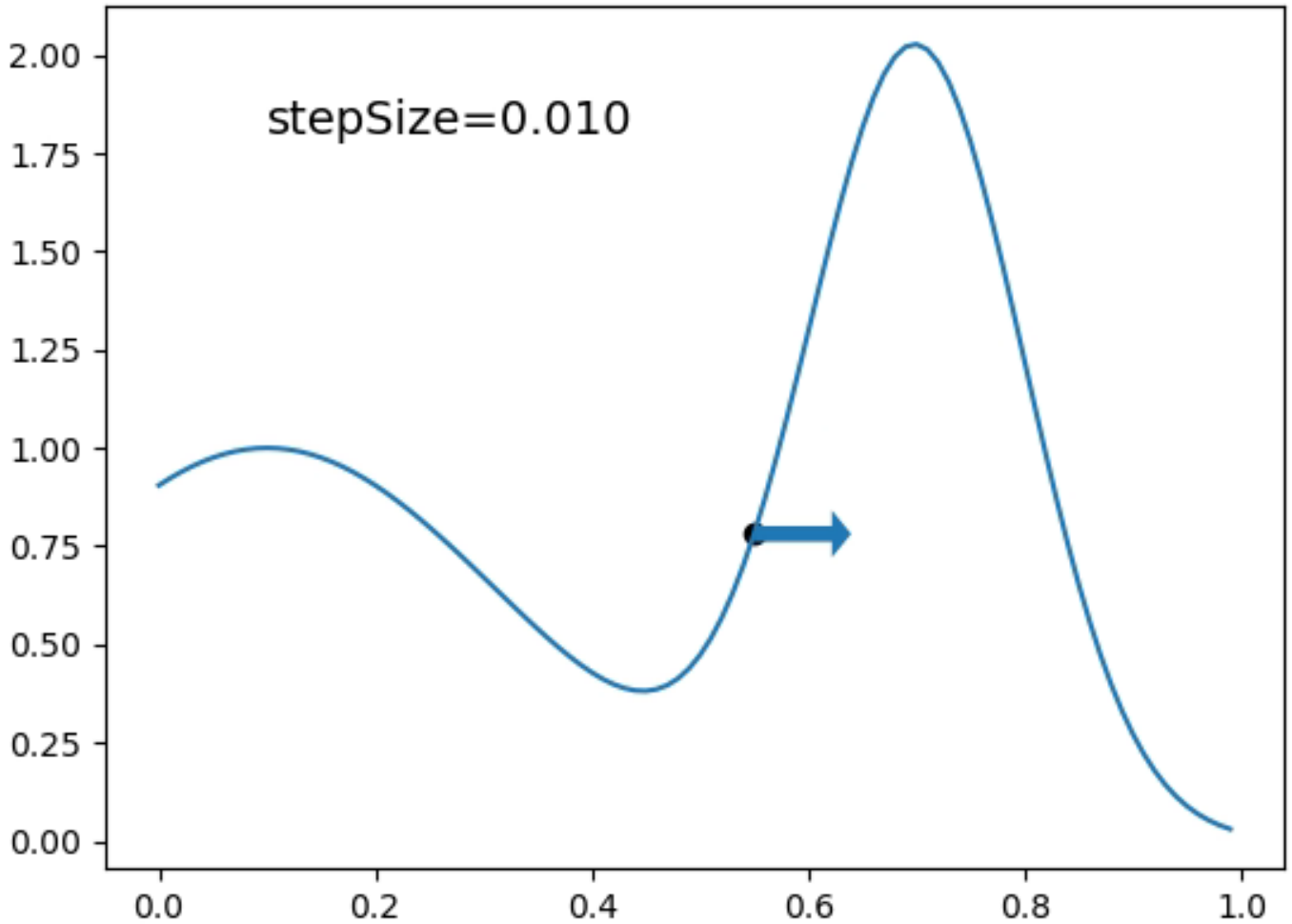
Smaller stepsize: no oscillations, but slow convergence



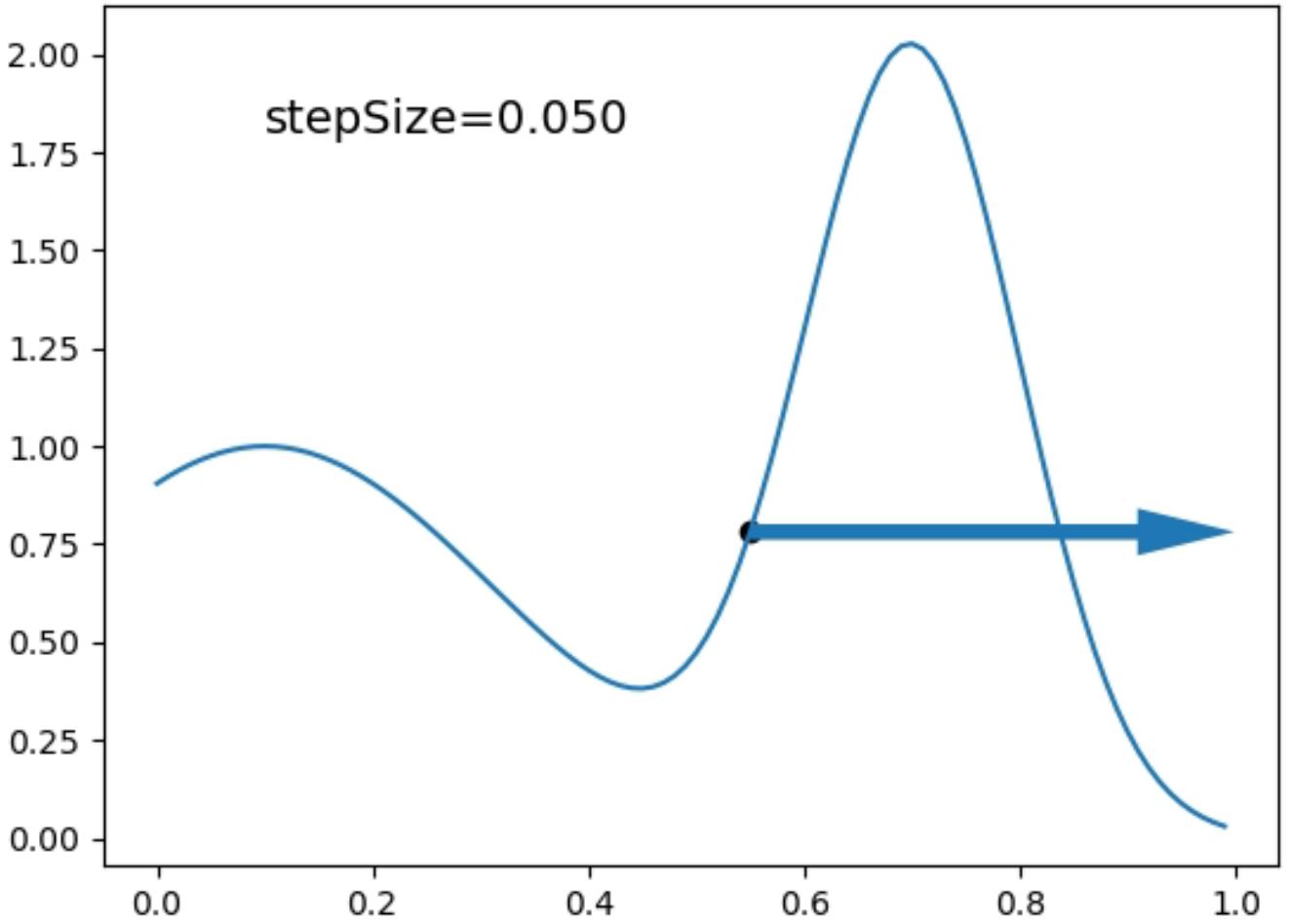
Speeding up convergence

- Step size decay
- Gradient clipping
- Momentum

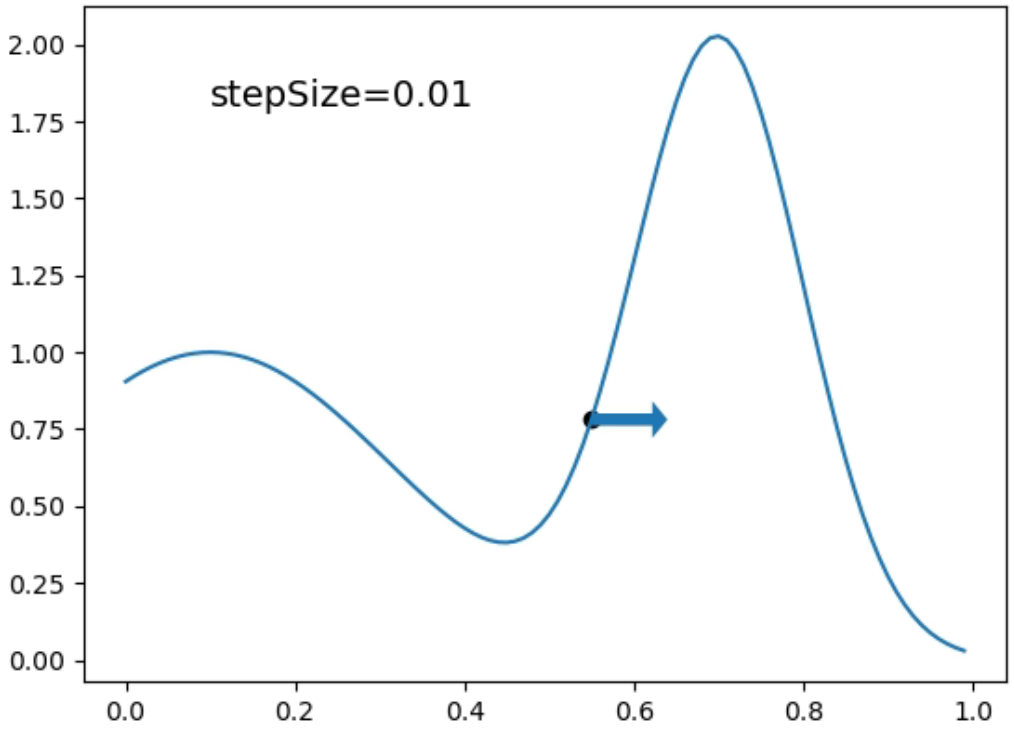
Step size decay

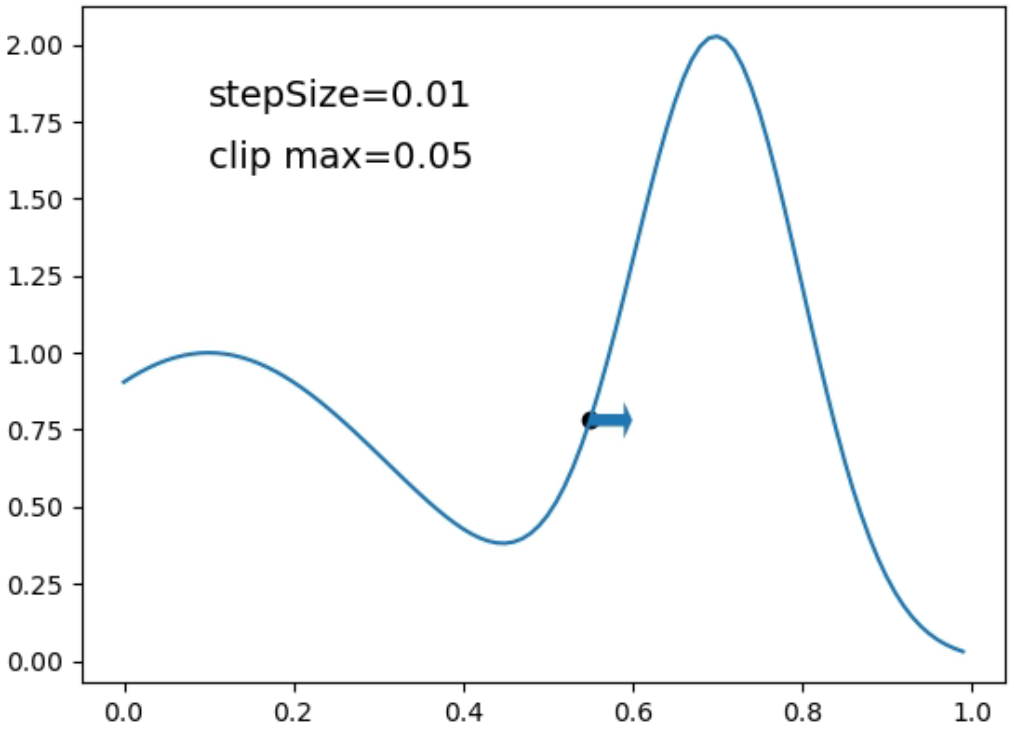


Step size decav



Gradient clipping

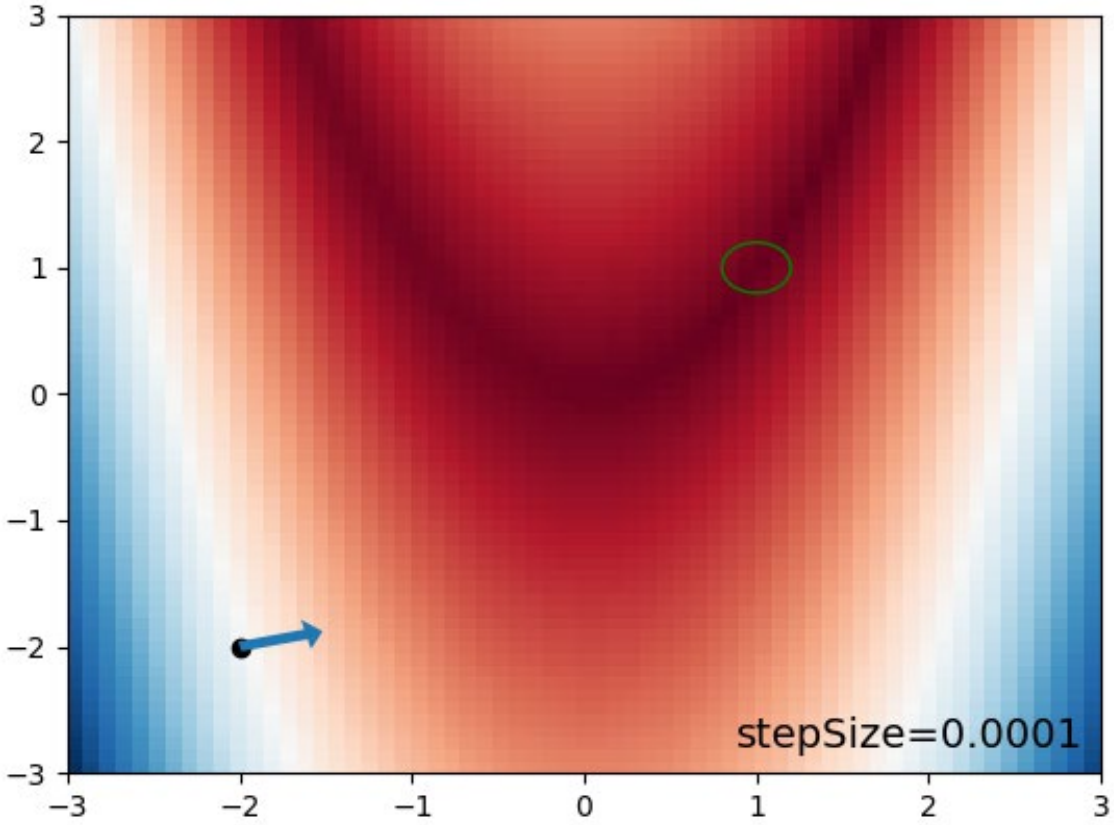




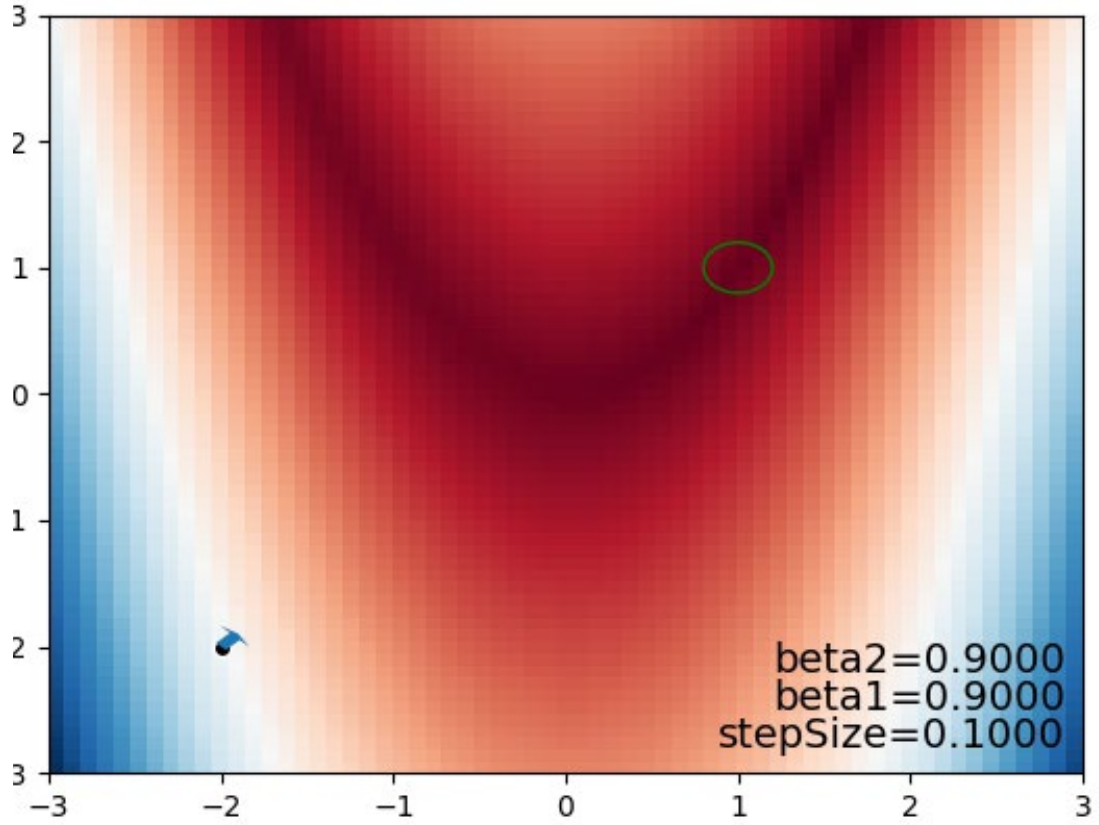
Momentum

- Gradient interpreted as acceleration instead of velocity
- Can be used together with gradient clipping
- **Adam** is a modern momentum-based method with some clever additional tricks. A good default choice for optimizing neural networks

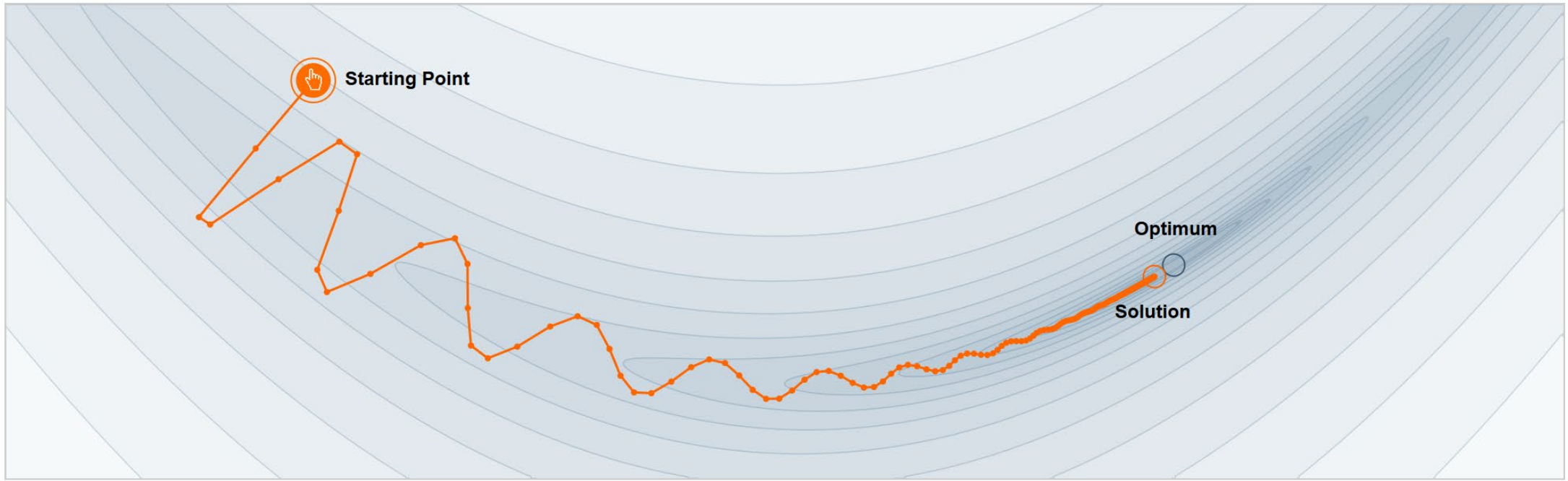
Without momentum



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

https://distill.pub/2017/momentum/

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.


WARNING
Mathematics
Ahead

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)

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**

$t \leftarrow t + 1$

Count iterations

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

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

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

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

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

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

end while

while θ_t not converged **do**

$t \leftarrow t + 1$

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

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

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

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

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

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

end while

while θ_t not converged **do**

$t \leftarrow t + 1$

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

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

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

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

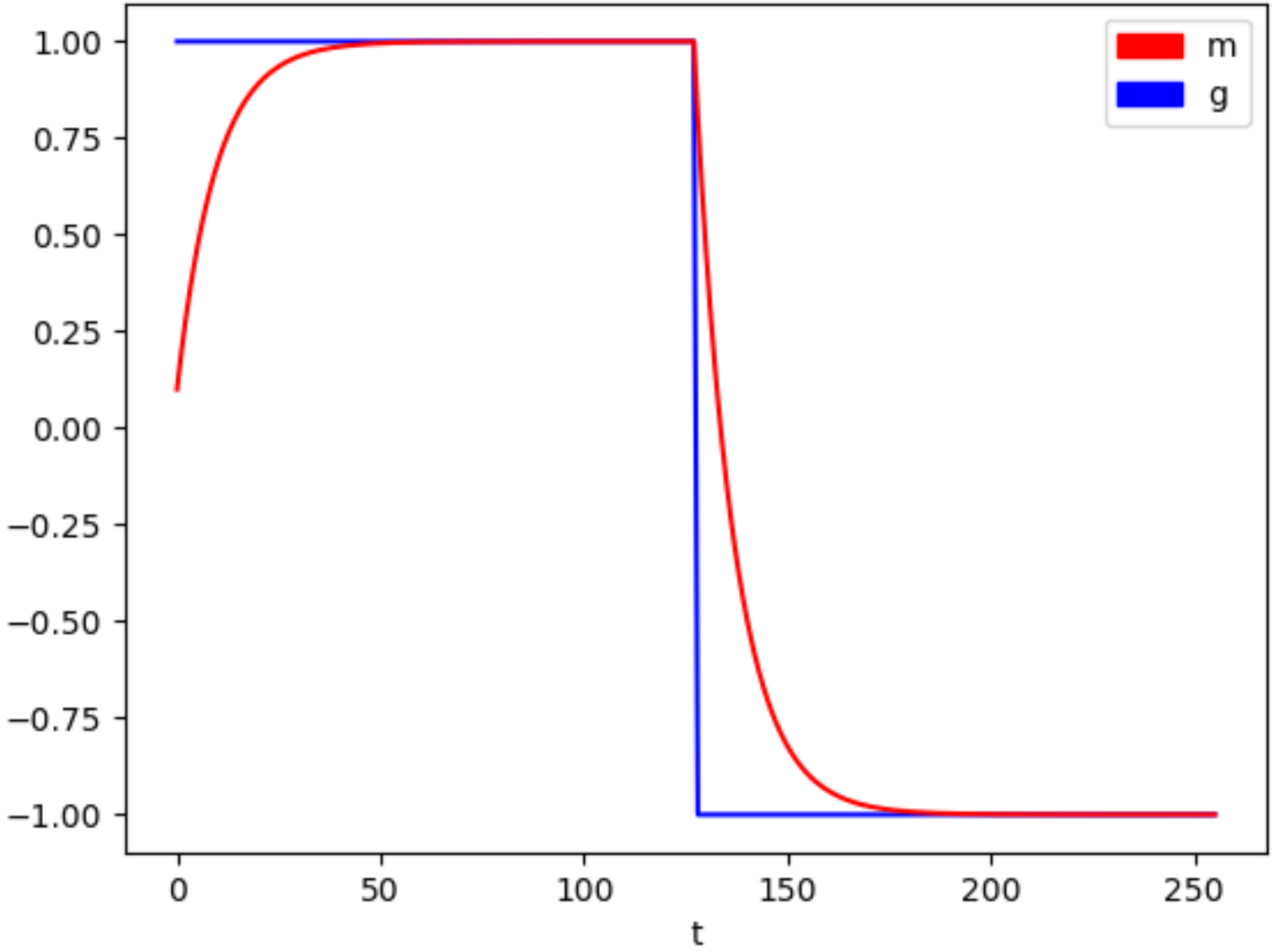
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

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

end while

Smoothing of the gradient over time

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

Exponentially smoothed squared gradient
(an estimate of gradient magnitude)

while θ_t not converged **do**

$t \leftarrow t + 1$

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

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

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

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

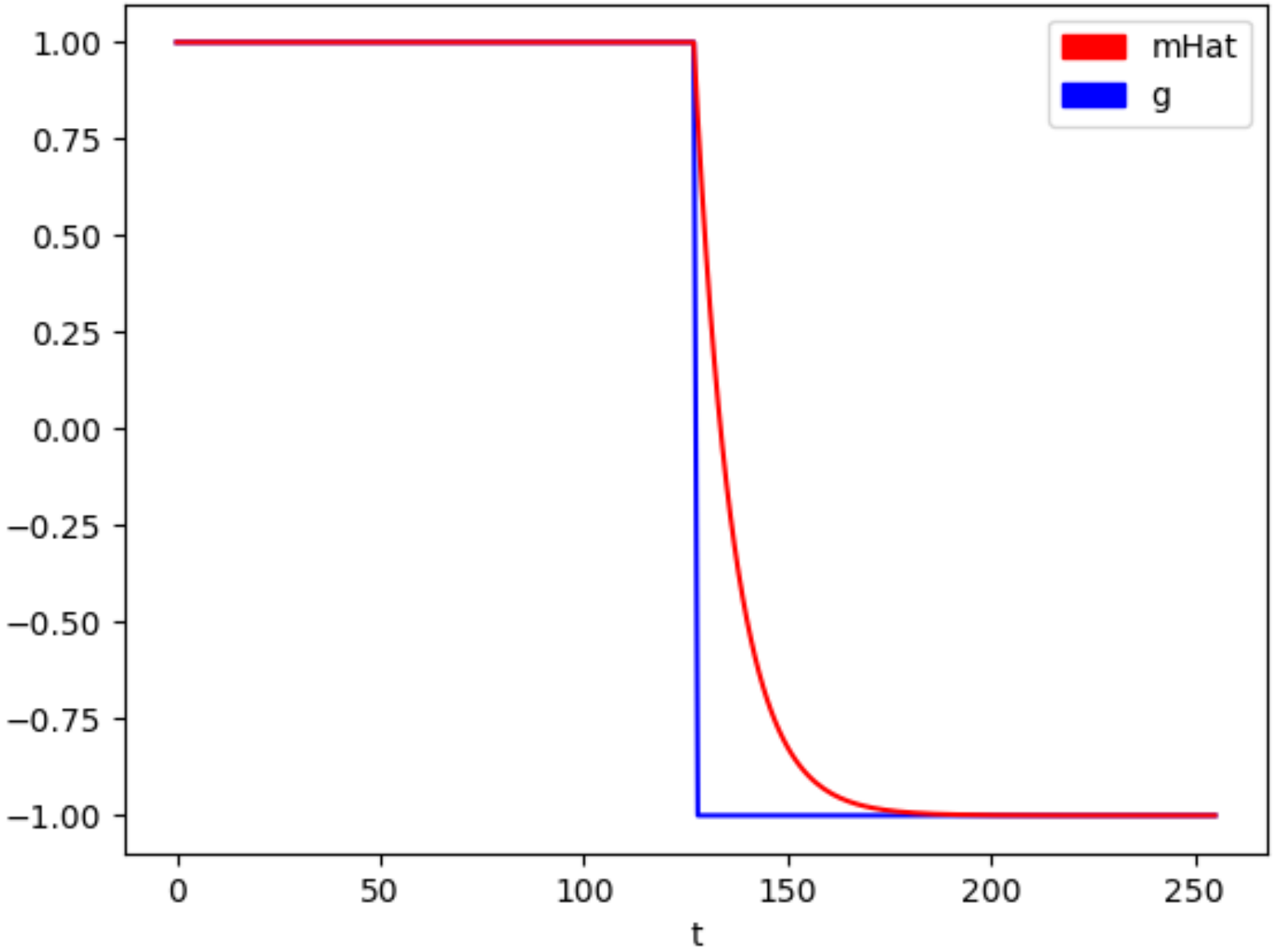
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

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

end while

Remove initial bias in exponential smoothing

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

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

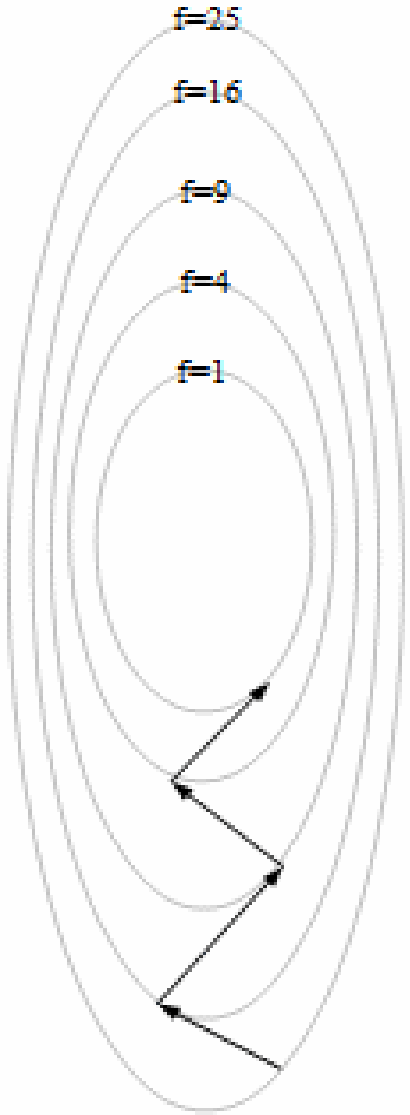
end while

Why scaling matters: Gradient is orthogonal (perpendicular) to the objective function isocontours

$f(\mathbf{x})$ is constant along the isocontour.

Here, the optimum is in the middle. The isocontours are elongated and gradient-based optimization zigzags, making slow progress.

Could one somehow remove the elongation?

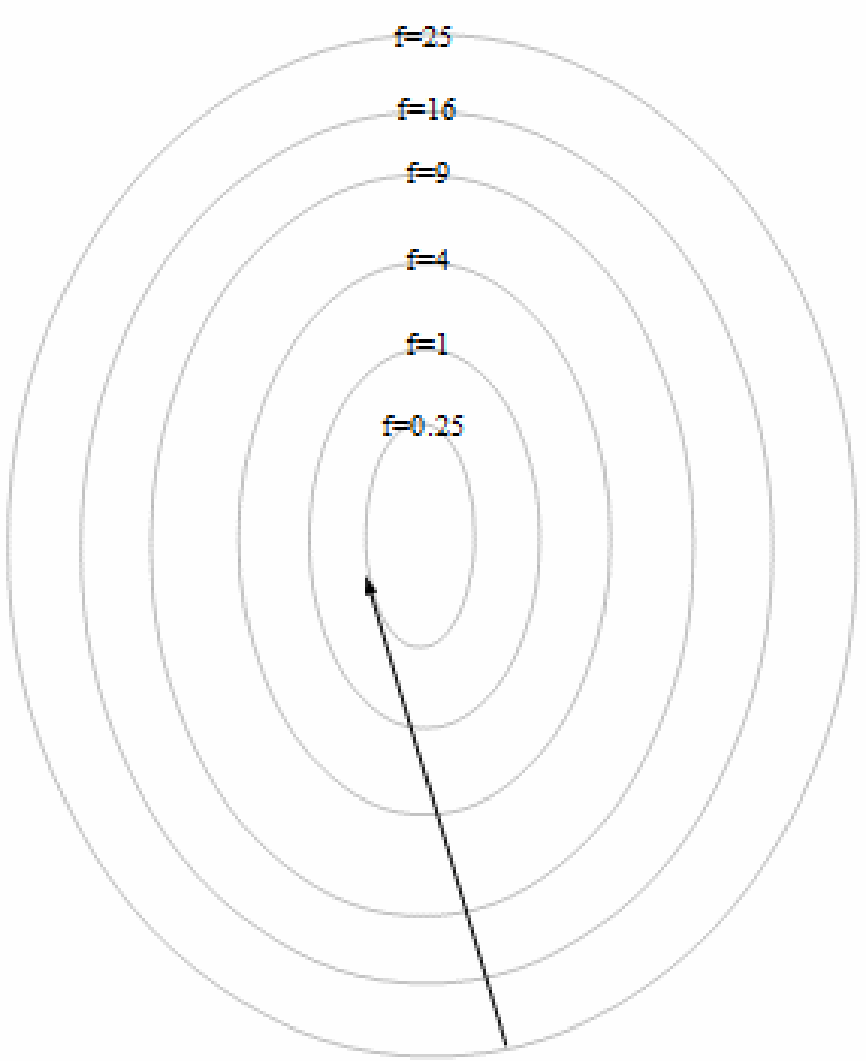


Preconditioning: Scaling the optimization landscape to make the isocontours more spherical

Instead of optimizing $f(\mathbf{x})$, optimize preconditioned f(**x**').

x'=A**x**, where the preconditioning matrix A implements scaling and rotation.

Now, the gradient points more accurately towards the optimum.



Good preconditioning.
Rapid progress towards solution.

while θ_t not converged **do**

Adam's gradient scaling is a clever way to implement adaptive preconditioning with minimal memory and compute overhead.

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

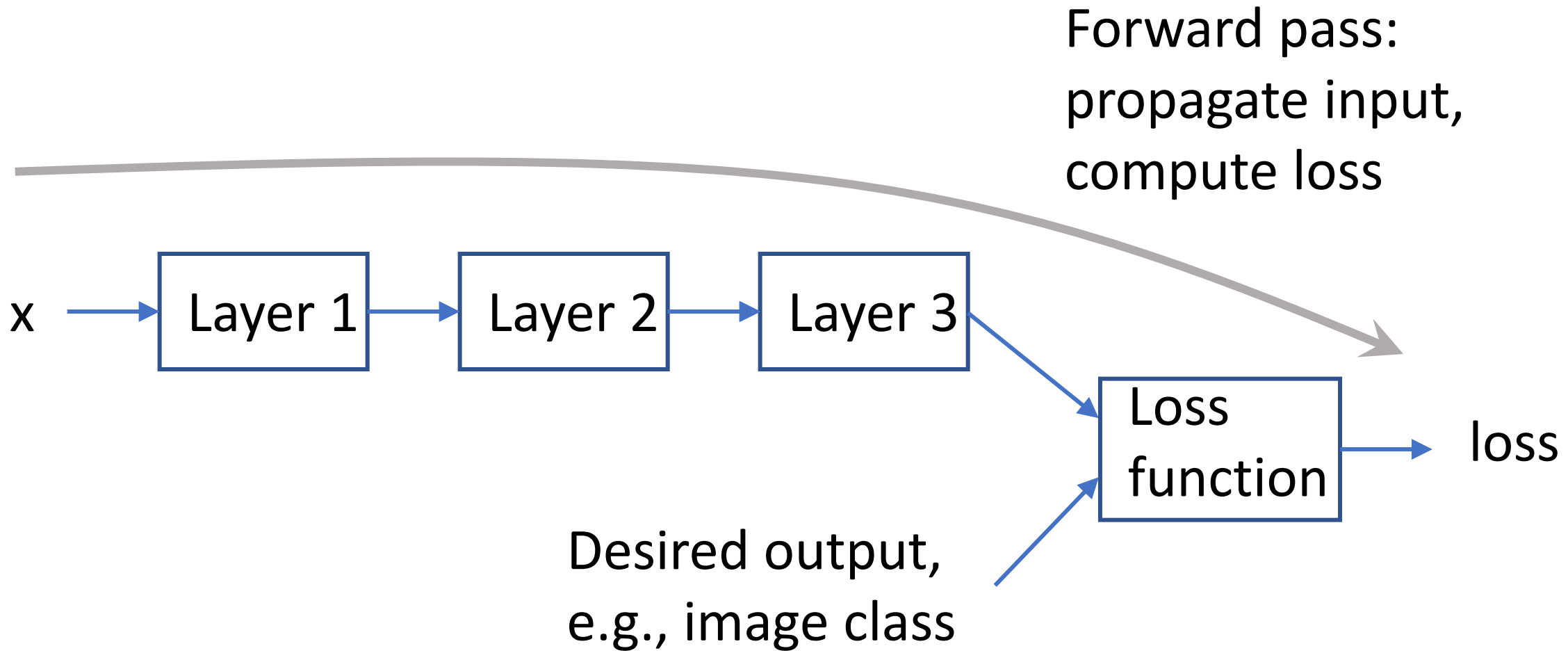
end while


WA NG
M ematics
Ahead

Gradients and neural networks

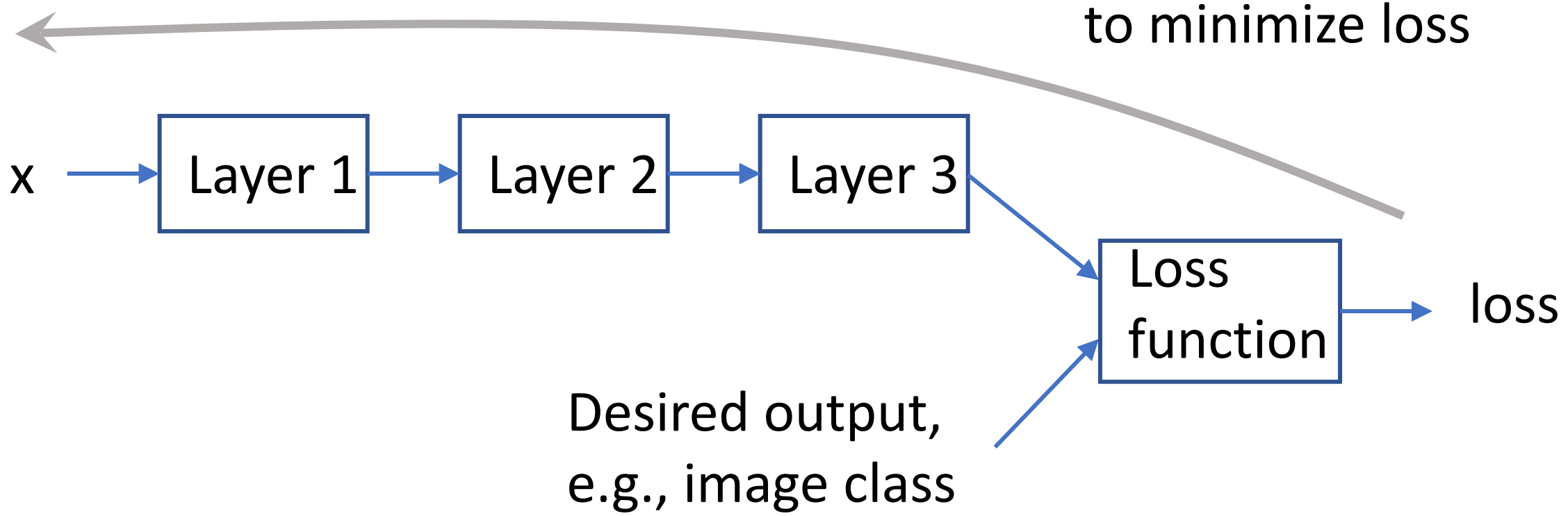
- Utilizing the chain rule of calculus, Tensorflow & Pytorch allow one to compute gradients of neural network outputs (e.g., loss function) with respect to variables (e.g., neural network weights)
- This process is called backpropagation

Backpropagation

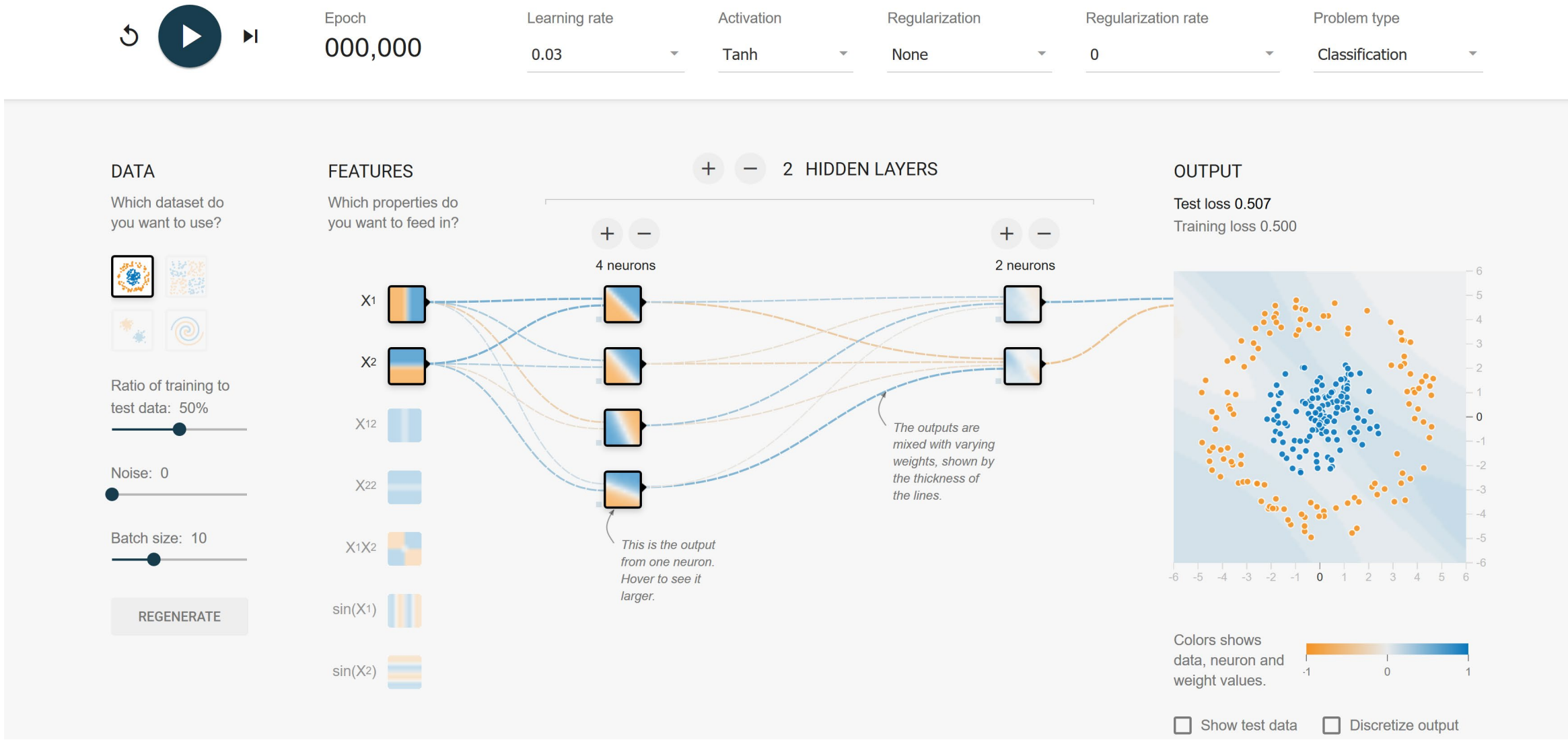


Backpropagation

Backward pass:
propagate gradients
and adjust layer params
to minimize loss



See it in action: https://playground.tensorflow.org



CLIPDraw: Backpropagation adjusts network inputs instead of network parameters

"A drawing of a cat".

"Horse eating a cupcake".

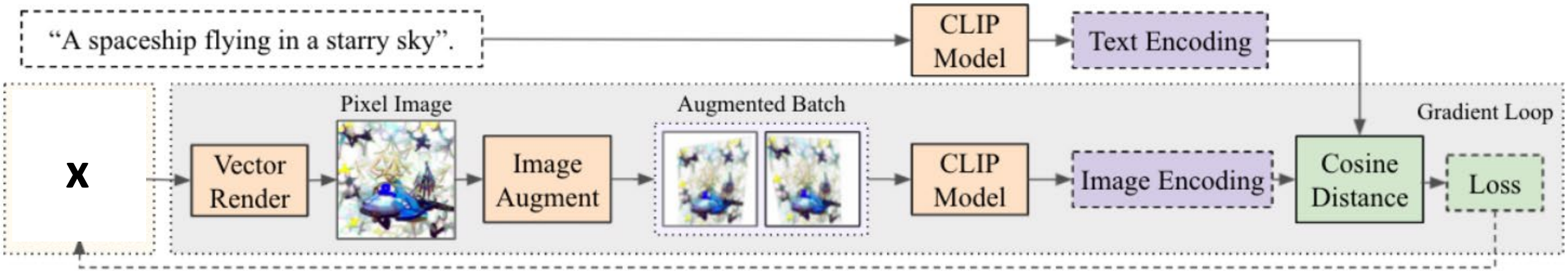
"A 3D rendering of a temple".

"Family vacation to Walt Disney World".

"Self".

https://arxiv.org/pdf/2106.14843.pdf
https://colab.research.google.com/github/kvfrans/clipdraw/blob/main/clipdraw.ipynb

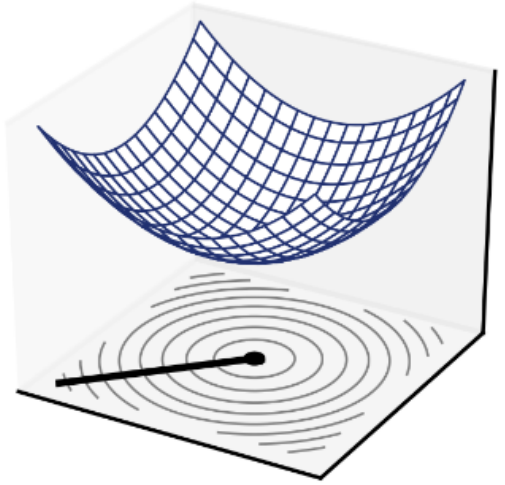
CLIPDraw: Backpropagation adjusts network inputs instead of network parameters



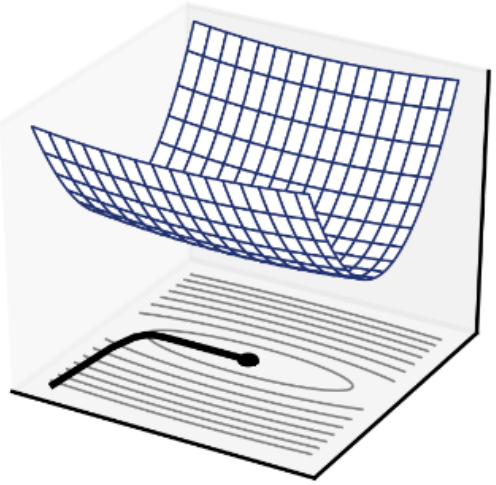
1. Use a differentiable Bezier curve renderer to draw curves described by parameters **x**
2. Compute CLIP encoding of both the drawing and text description
3. Compute the cosine distance between the encodings
4. Backpropagate gradients to adjust **x** so that the distance is minimized

Difficulty of optimization

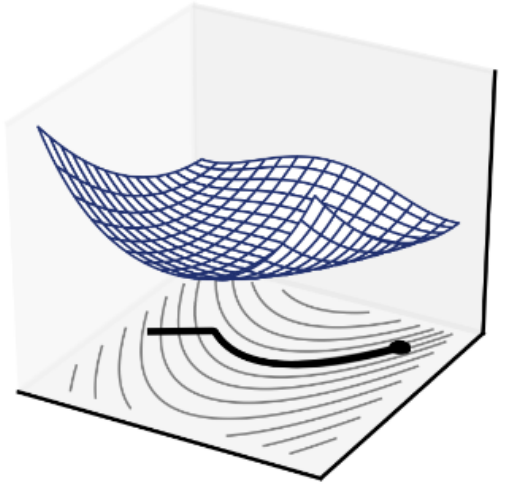
Convex,
well-conditioned



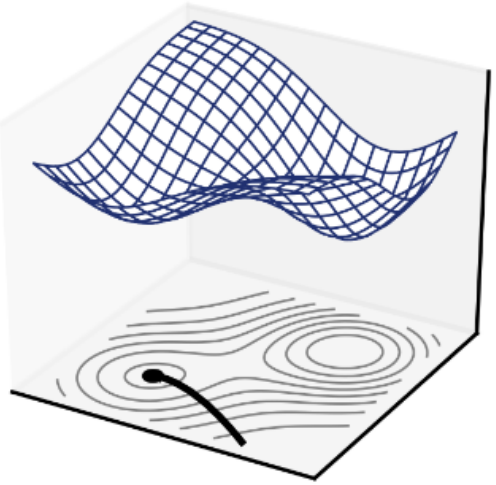
Convex,
ill-conditioned



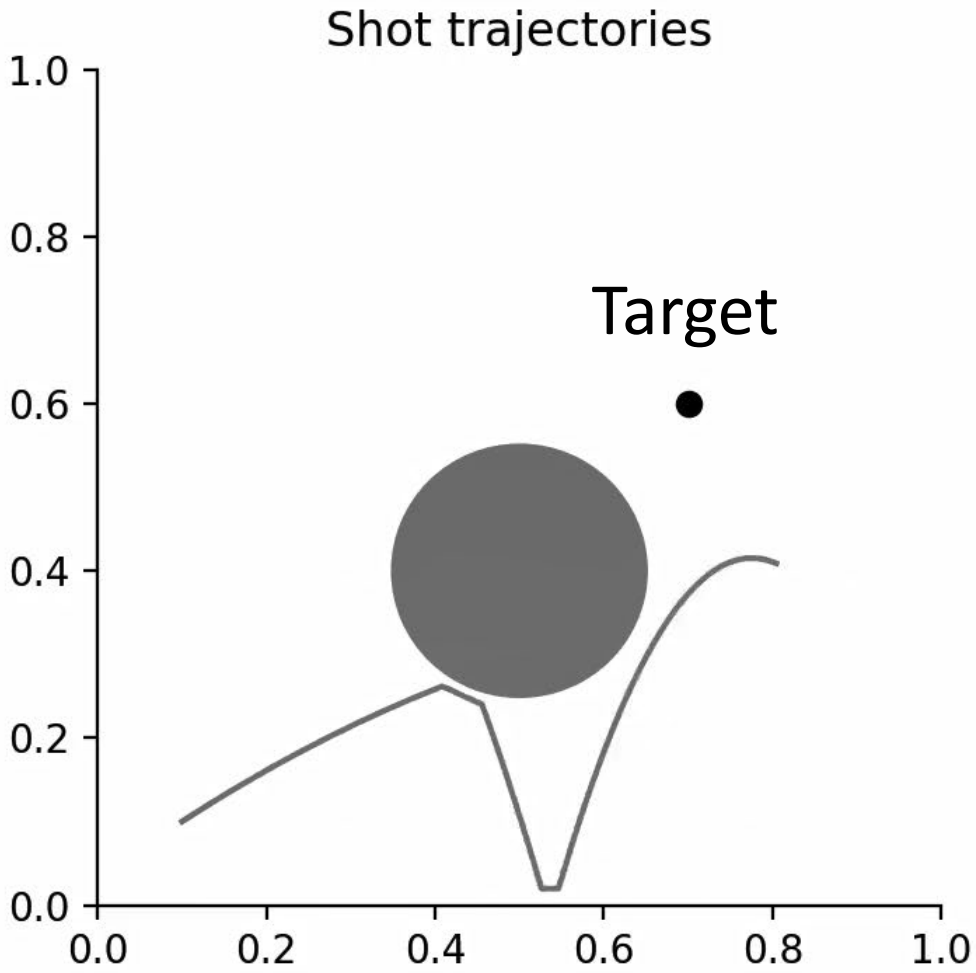
Non-convex,
unimodal

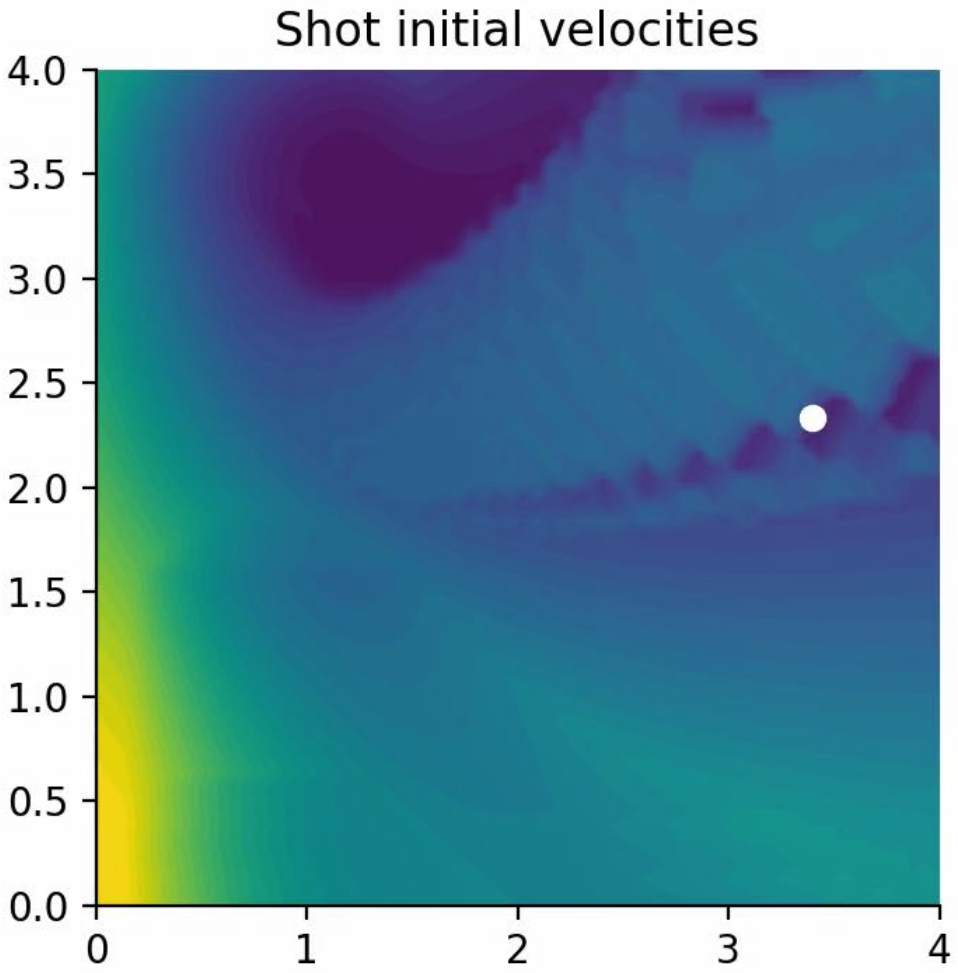


Non-convex,
multimodal



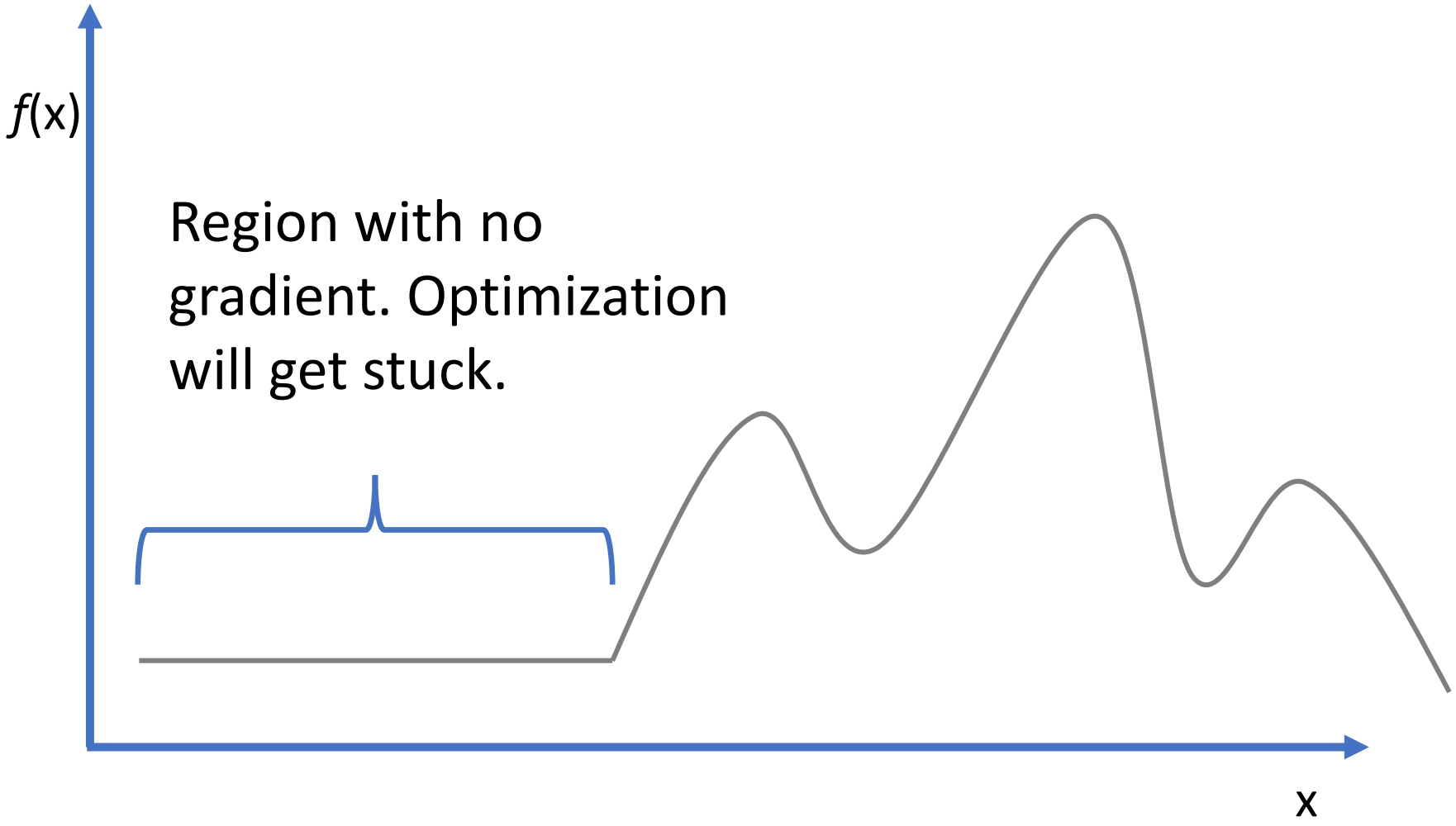
Optimize ball throw initial velocity to hit the target





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

One might even have $f(\mathbf{x})$ with no gradient



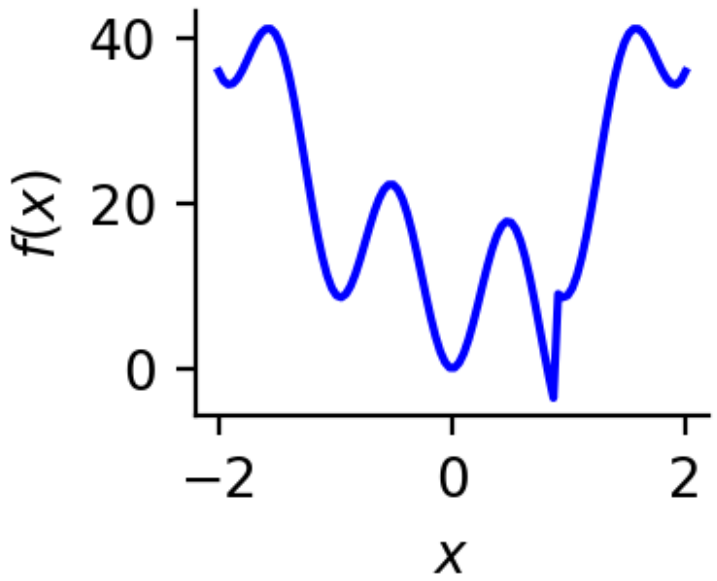
Sampling-based optimization

How to handle multimodality and zero gradient regions?

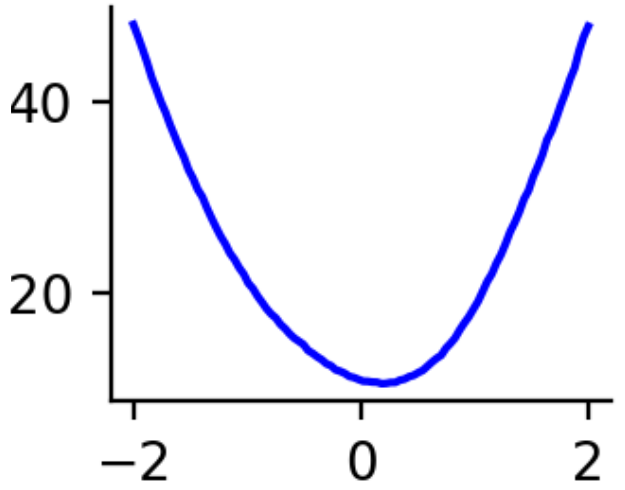
Key intuition: A difficult objective function can be smoothed (blurred) to help optimization

- Common: Gaussian-blurring of the objective.
- Assumption: The blurred optimum is good/close enough

Original $f(x)$

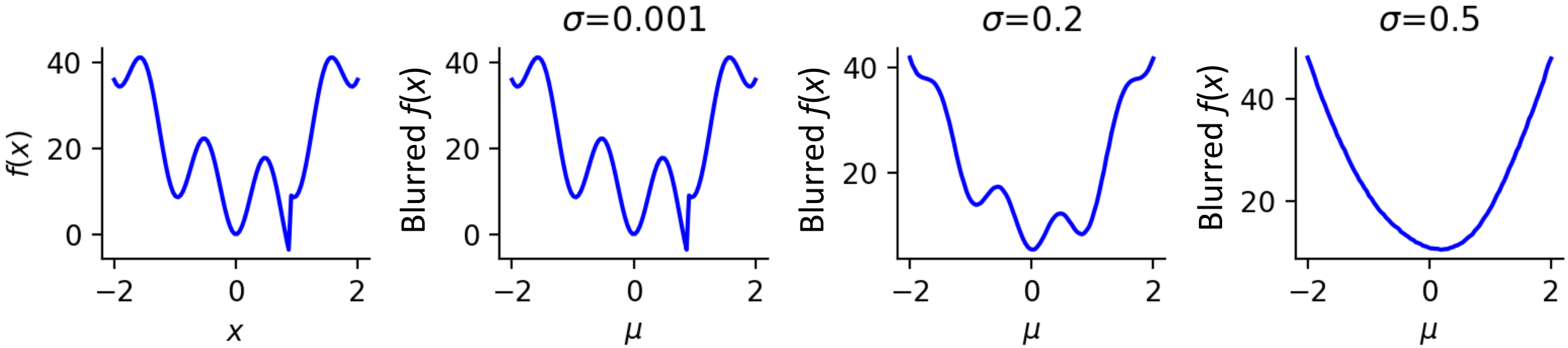


Blurred $f(x)$



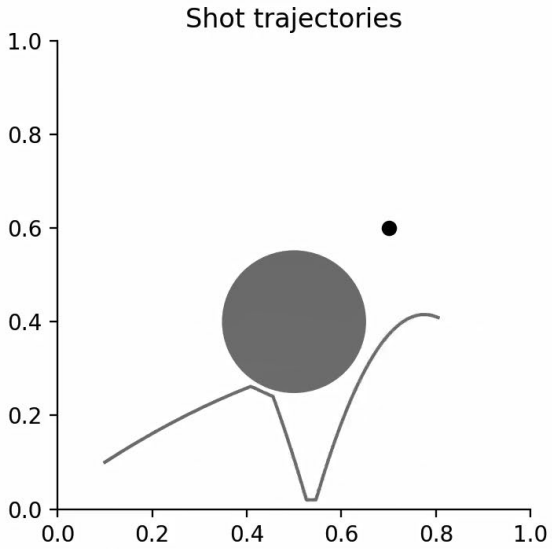
Sampling allows approximate Gaussian blurring in high-dimensional optimization

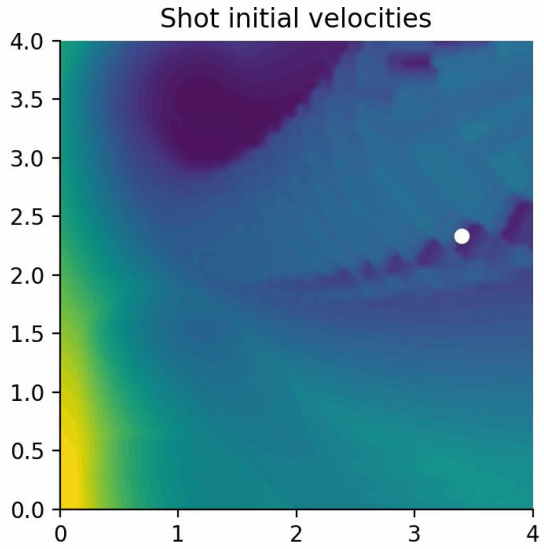
- Gaussian-blurred $f(\mathbf{x})$ ≈ average of $f(\mathbf{x}')$ over $\mathbf{x}'$ sampled from a Gaussian (Normal) distribution with mean $\boldsymbol{\mu}=\mathbf{x}$
- The distribution's standard deviation $\boldsymbol{\sigma}$ is the Gaussian blurring amount



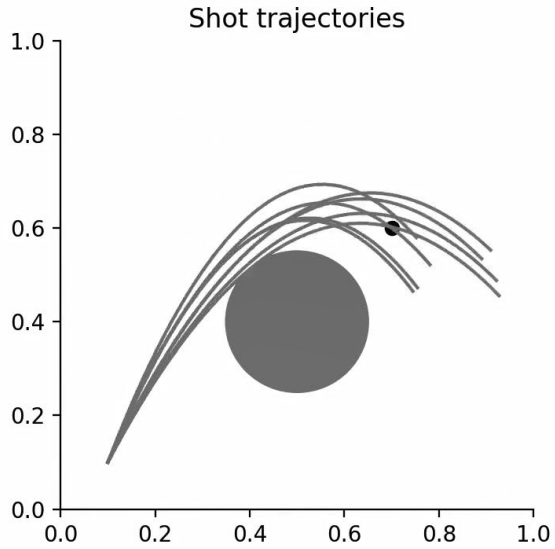
Example

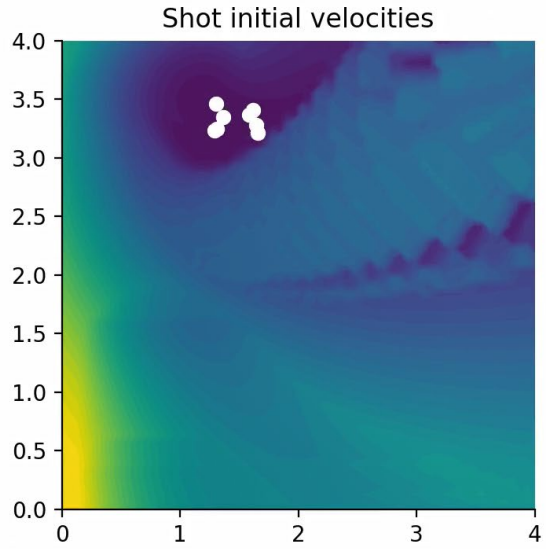
Adam without sampling





Adam with sampling

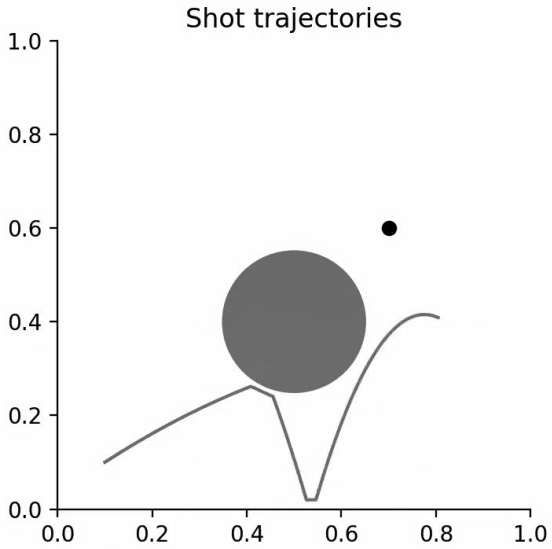


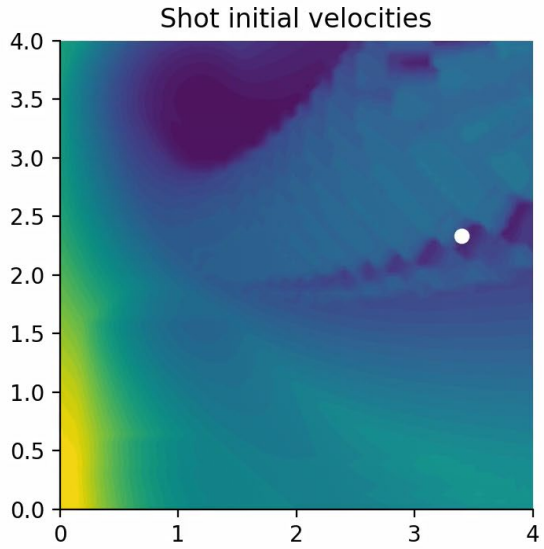


With sampling: Instead of updating **x**, update **μ**, $\boldsymbol{\sigma}$ based on gradient computed from samples. Can still use momentum etc.

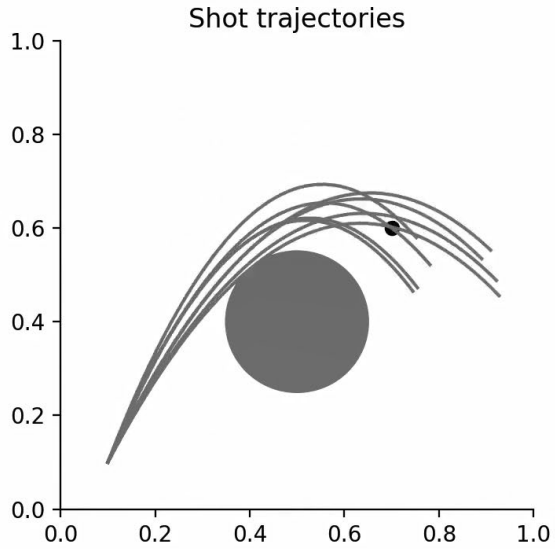
Example

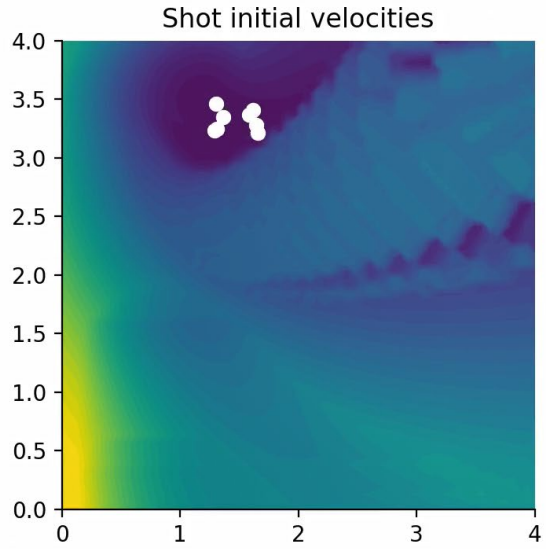
Adam without sampling





Adam with sampling





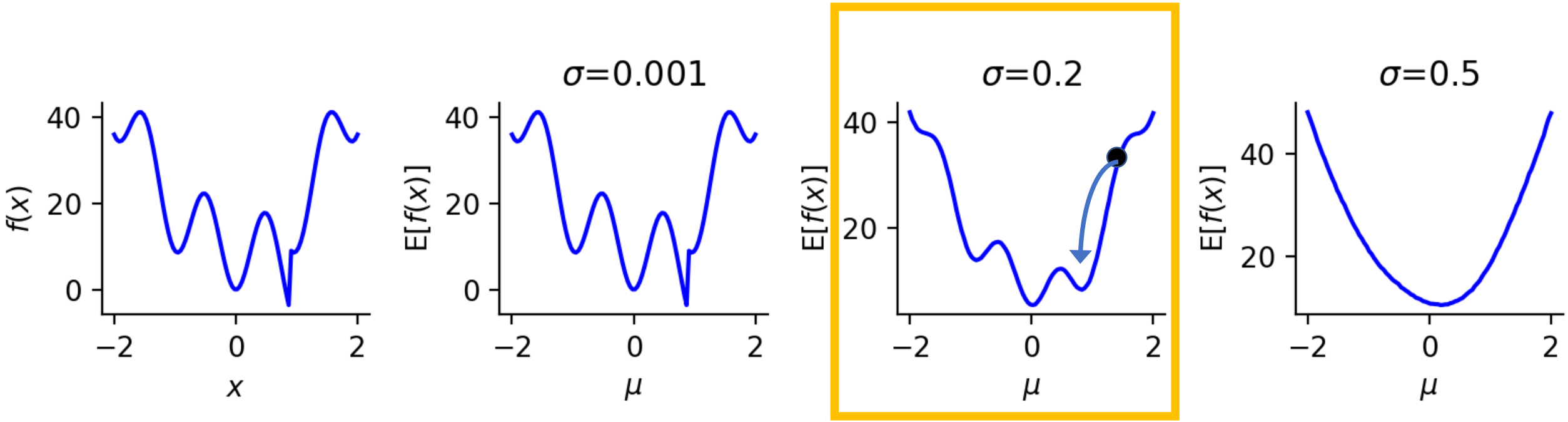
The mean μ gives an estimate of the optimum. This becomes more accurate as the optimization progresses.

Perception Engines: Sampling-based optimization of brush strokes to fool a neural net image classifier

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

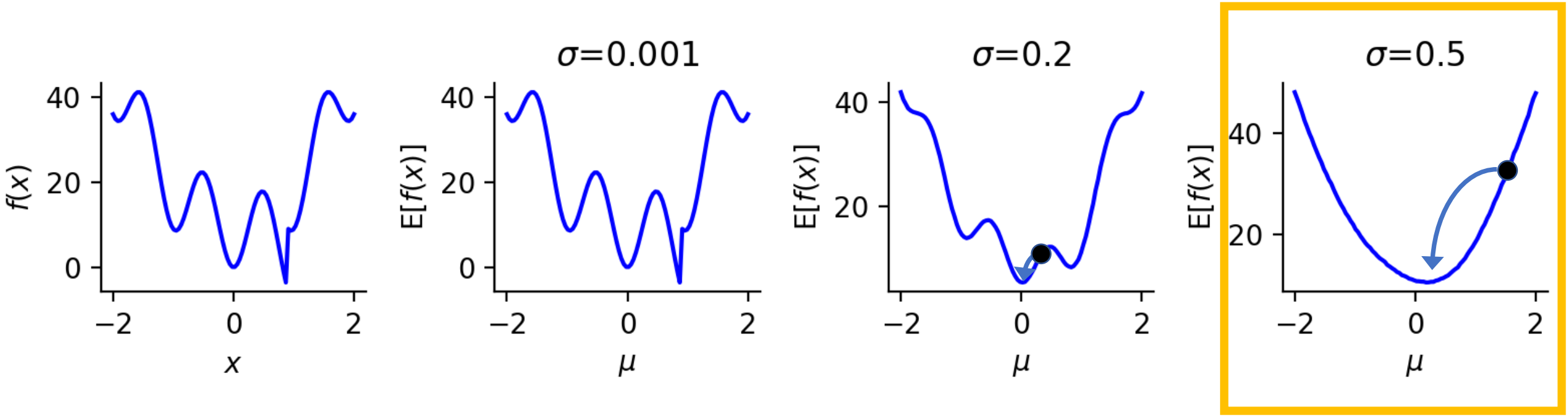
https://medium.com/@tom_25234/synthetic-abstractions-8f0e8f69f390

Key intuition: Low initial σ increases the probability of getting trapped in a bad local optimum



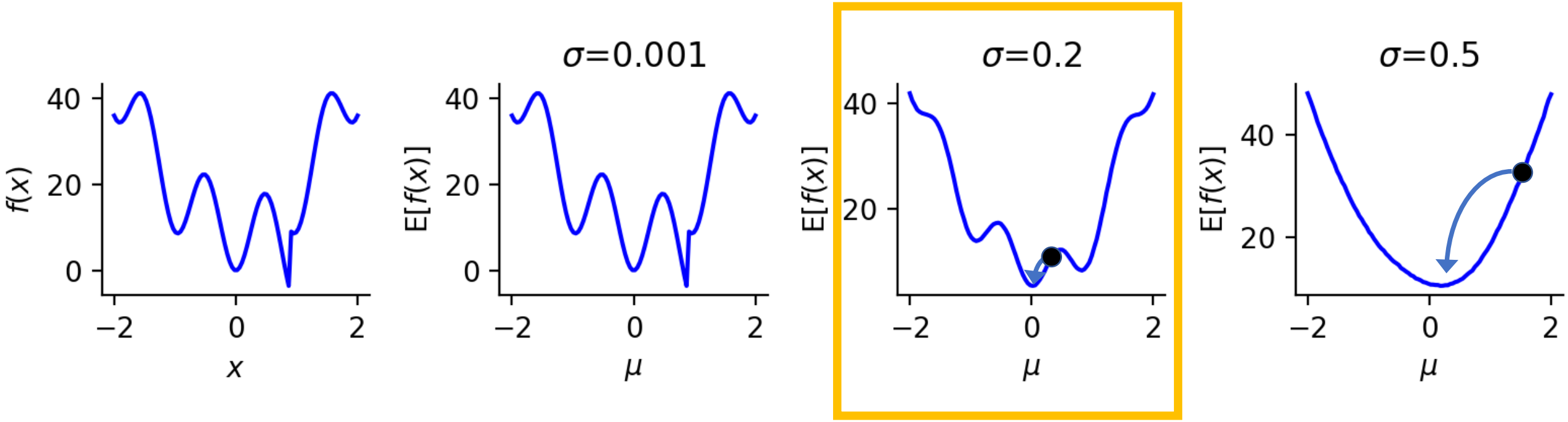
Better to start with high σ and gradually decrease it

Start with σ=0.5

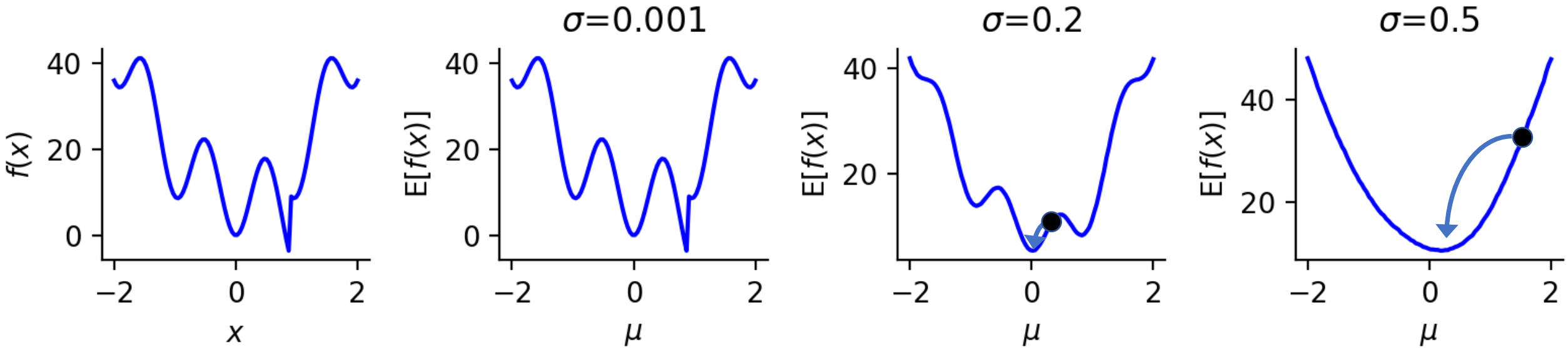


Better to start with high σ and gradually decrease it

Continue with σ=0.2



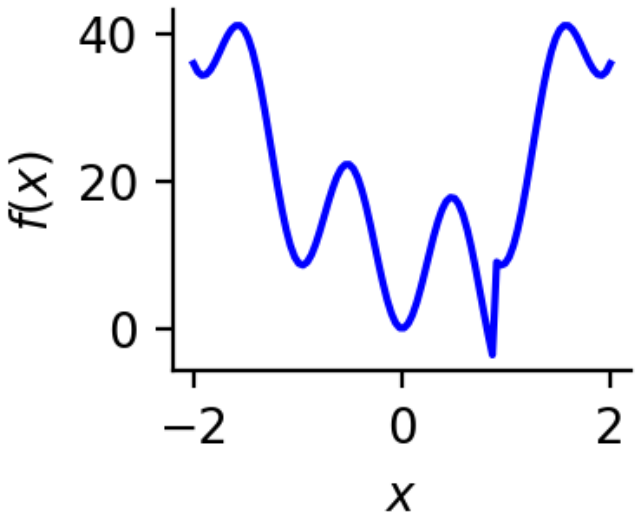
Better to start with high σ and gradually decrease it

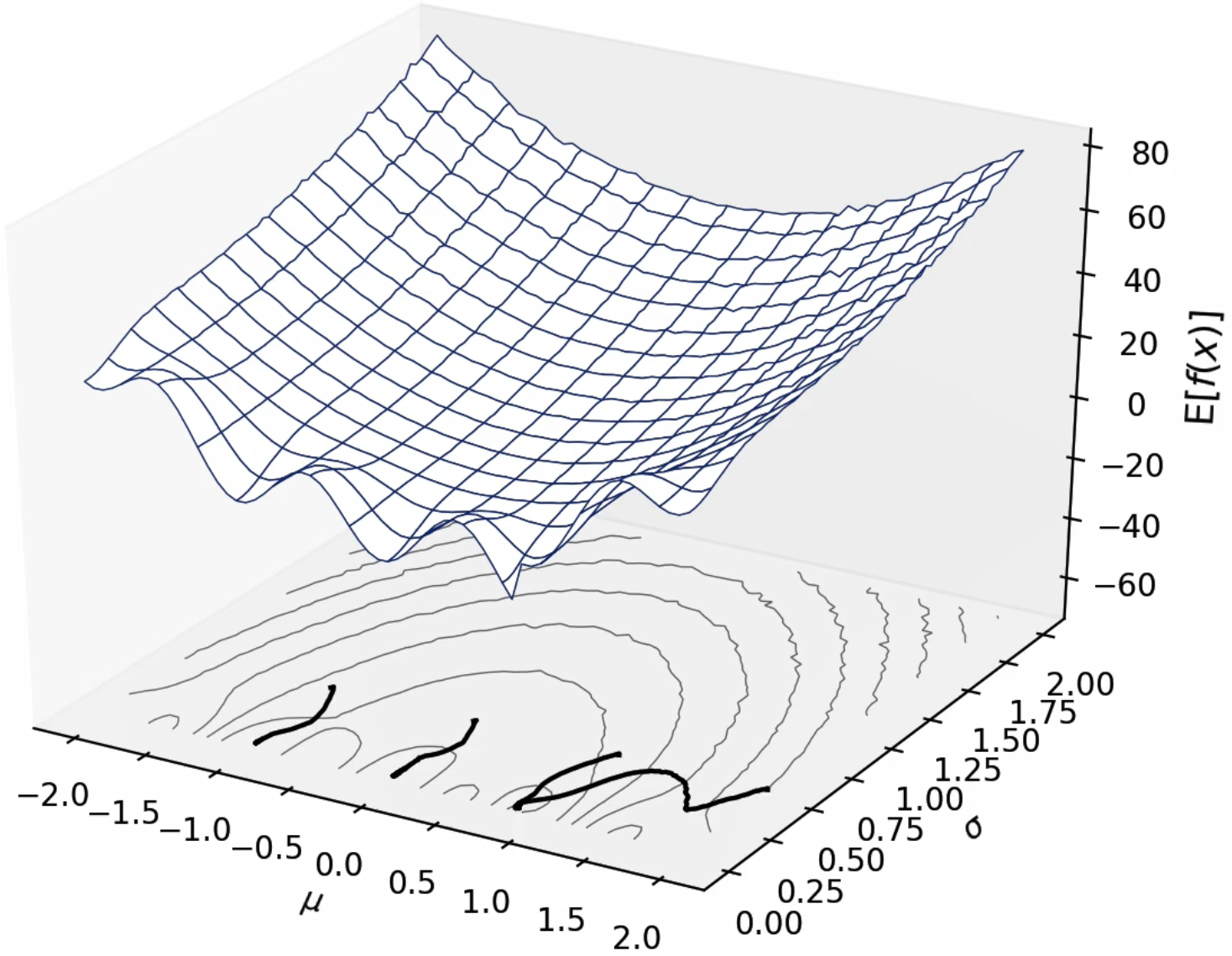


Will usually find the global optimum or a reasonably good local one.

Animation: Optimizing with different initial μ, σ.
With large σ, all runs converge to same optimum.

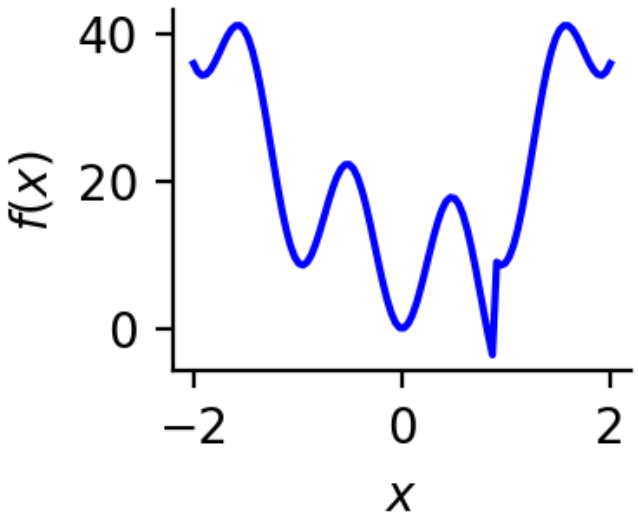
Original f(x)

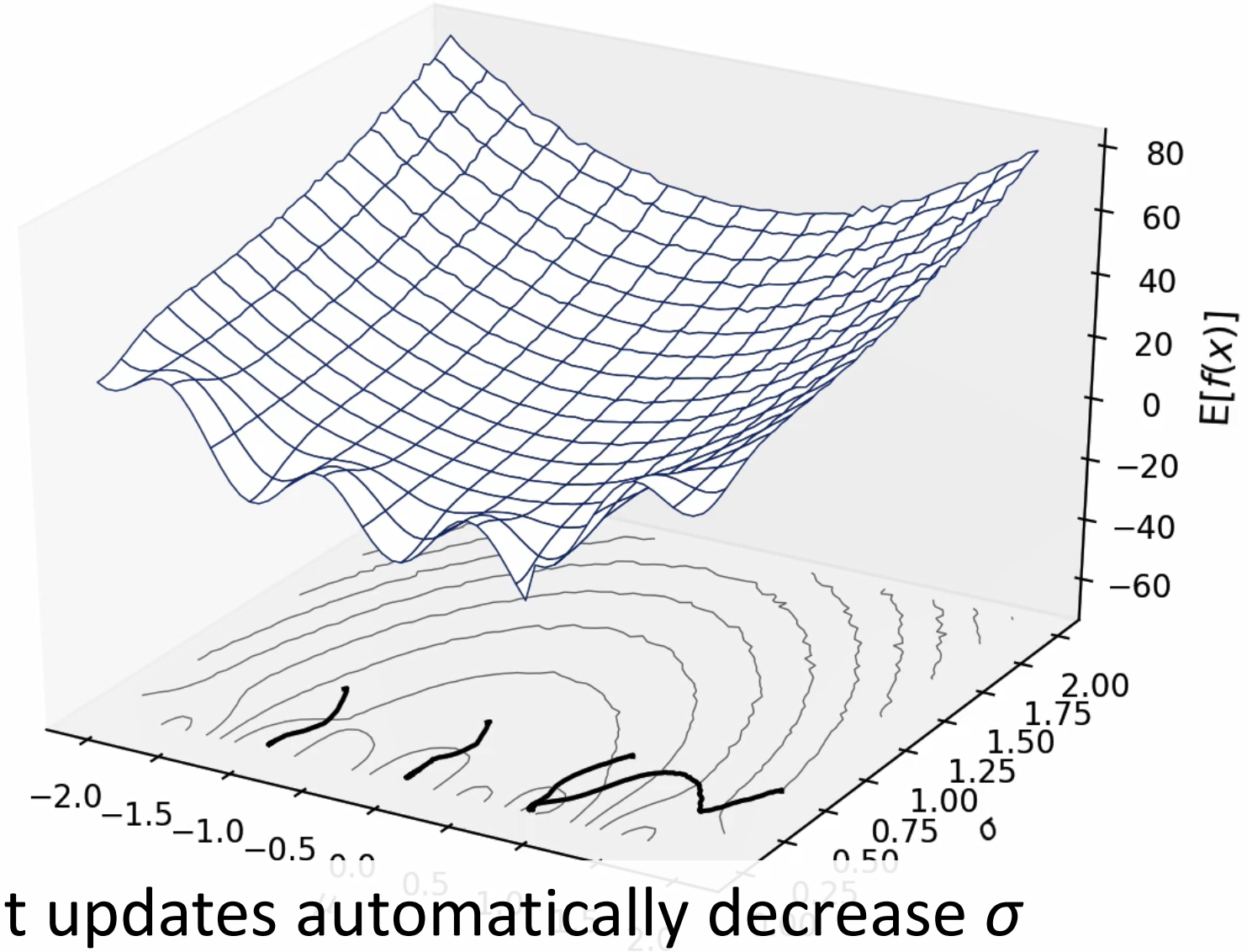




Animation: Optimizing with different initial μ, σ.
With large σ, all runs converge to same optimum.

Original f(x)

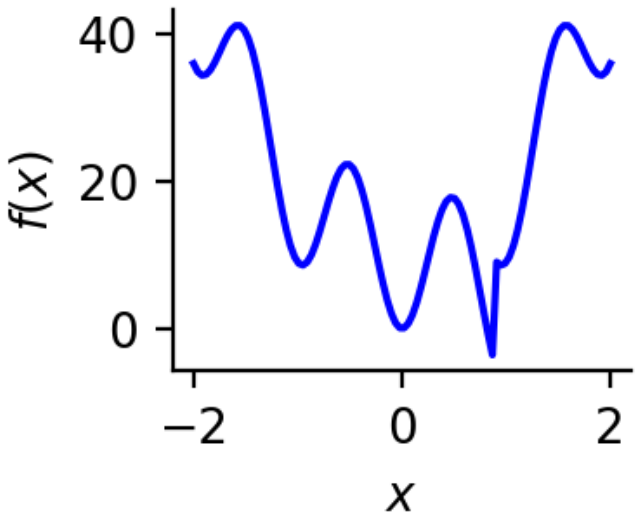


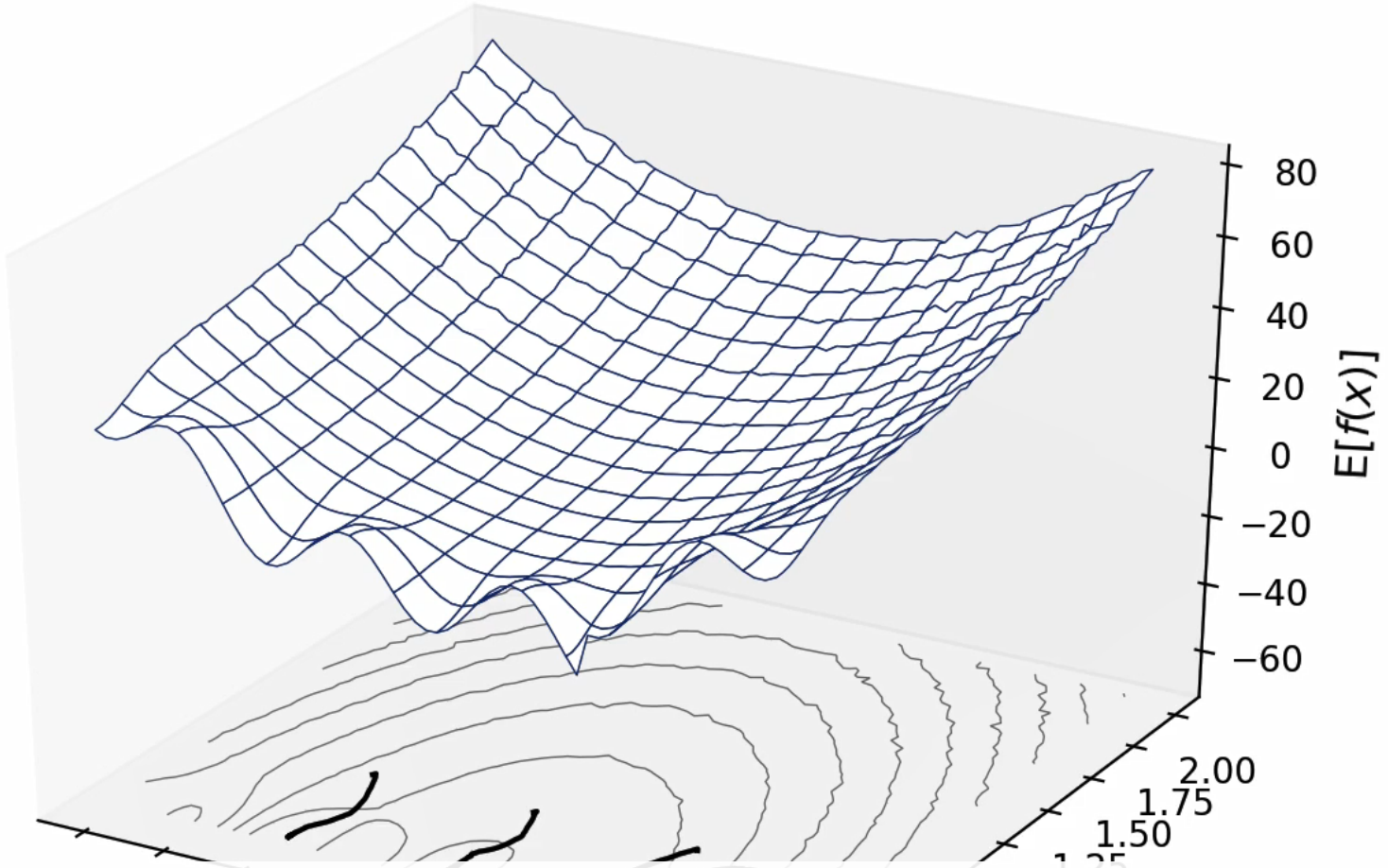


Here, the gradient updates automatically decrease σ

Animation: Optimizing with different initial μ, σ.
With large σ, all runs converge to same optimum.

Original f(x)





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

Algorithm

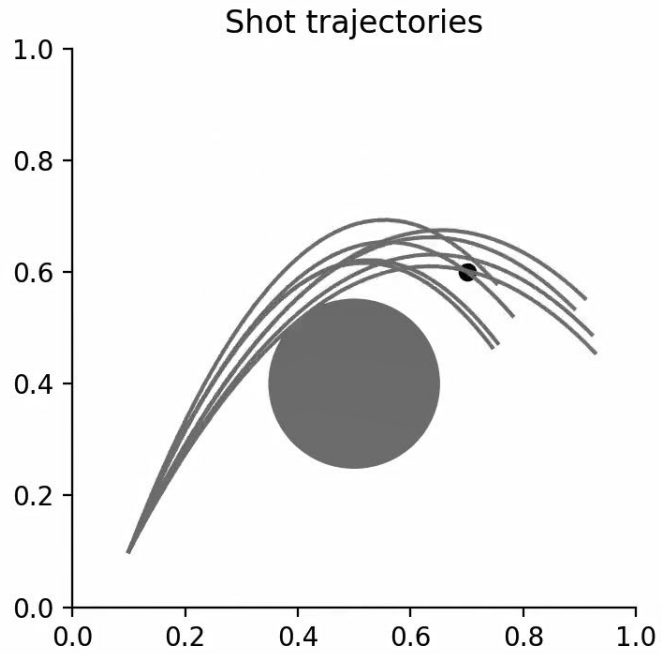
One iteration (repeat until results are good enough):

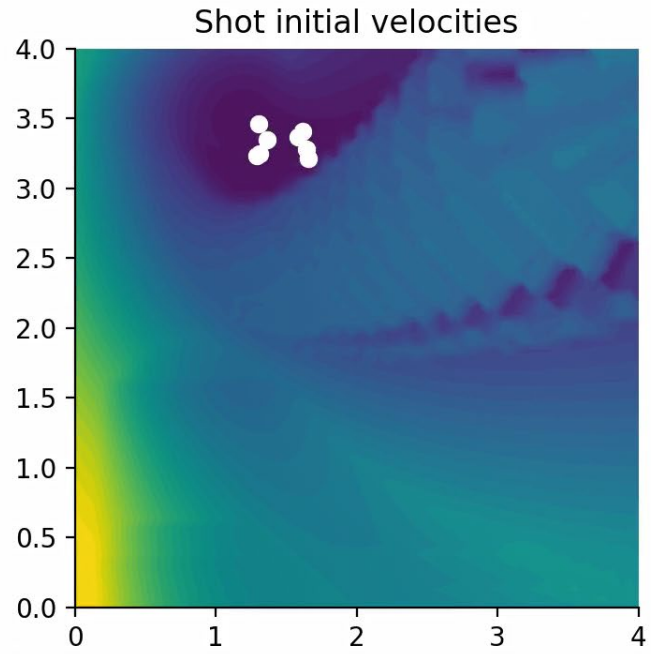
1. Sample multiple $\mathbf{x}$ from the exploration distribution $p(\mathbf{x})$
2. Evaluate $f(\mathbf{x})$ for all sampled $\mathbf{x}$
3. Follow the gradient: Nudge the parameters of the exploration distribution (e.g., $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$) so that it becomes more probable to sample again the $\mathbf{x}$ with high $f(\mathbf{x})$

Algorithm in code

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

Sample the $\mathbf{x} \sim p(\mathbf{x})$
Evaluate the $f(\mathbf{x})$
Use gradient to update $p(\mathbf{x})$





x_2

x_1

How to estimate the gradient from samples?


WARNING
Mathematics
Ahead

Gradient estimator for non-differentiable $f(\mathbf{x})$

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

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

Here, θ denotes the parameters of $p(\mathbf{x})$, e.g., $\theta = [\mu, \sigma]$

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

Gradient estimator for non-differentiable $f(\mathbf{x})$

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 estimator for non-differentiable $f(\mathbf{x})$

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 estimator for non-differentiable $f(\mathbf{x})$

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 $p_\theta(\mathbf{x})$

This approximation becomes more accurate with high N. Thus, if optimization does not work, one should first try increasing N.

Gradient estimator for non-differentiable $f(\mathbf{x})$

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 estimator for non-differentiable $f(\mathbf{x})$

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 estimator for non-differentiable $f(\mathbf{x})$

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 $f(\mathbf{x})$

Sounds very sensitive to the choice of $f(\mathbf{x})$. What if all f($\mathbf{x}$) are negative?

Gradient estimator for non-differentiable $f(\mathbf{x})$

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

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

$$A(\mathbf{x}_i) = f(\mathbf{x}_i) - \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[f(\mathbf{x})] \approx f(\mathbf{x}_i) - \frac{1}{N}\sum_{j=1}^{N} f(\mathbf{x}_j)$$

A common improvement: Increase the probability of $\mathbf{x}_i$ with positive advantage function values $A(\mathbf{x}_i)$, i.e., those $\mathbf{x}_i$ that work better than average.

Instead of directly computing the gradients, most ML code minimizes the corresponding loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{x}_i) \log p_\theta(\mathbf{x}_i)$$

$A(\mathbf{x}_i)$ is treated as a constant, gradient not propagated through it

Gradient estimator for differentiable $f(\mathbf{x})$

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 estimator for differentiable $f(\mathbf{x})$

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 of $f(\mathbf{x})$ at all sampled $\mathbf{x}_i$ and average over the samples

Gradient estimator for differentiable $f(\mathbf{x})$

Reparameterization trick:

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

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

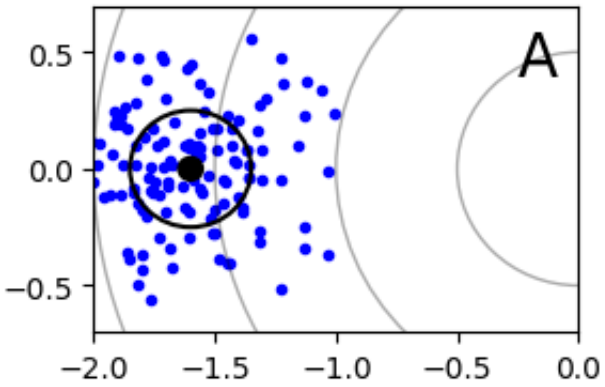

WA NG
M ematics
Ahead

Improving optimization efficiency

- Taking a single gradient step for each batch of samples can be expensive
- Cross-Entropy Method (CEM): Fit the exploration distribution to the best samples
  - This is equal to 1) setting the $f(\mathbf{x})$ or $A(\mathbf{x})$ values to 1 for the best samples and 0 for others, and 2) **taking multiple gradient steps** with the same samples
- Covariance Matrix Adaptation Evolution Strategy (CMA-ES): Similar to CEM, but implementing a form of momentum.
- Conceptually, CEM and CMA-ES are similar to sampling-based gradient optimization: They smooth $f(\mathbf{x})$ to be able to handle local optima, noise, and regions with no gradient.

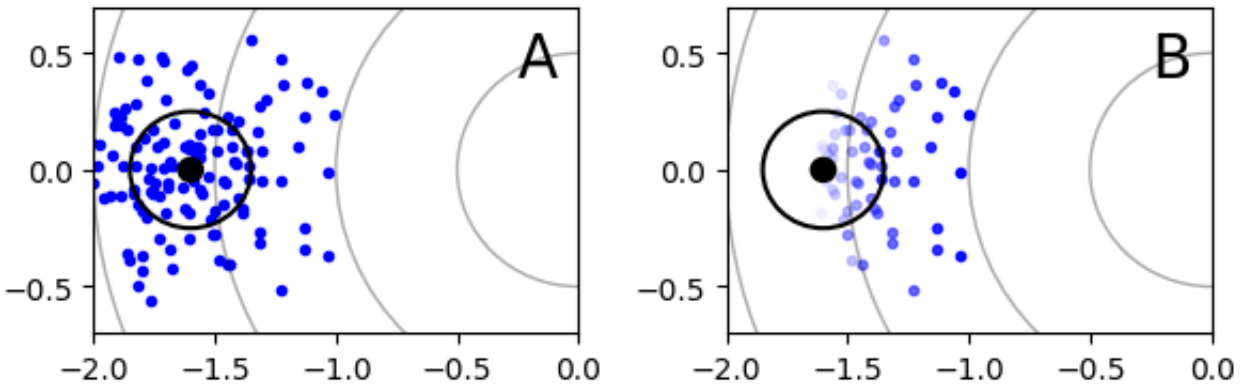
CMA-ES and CEM

Sample some **x**



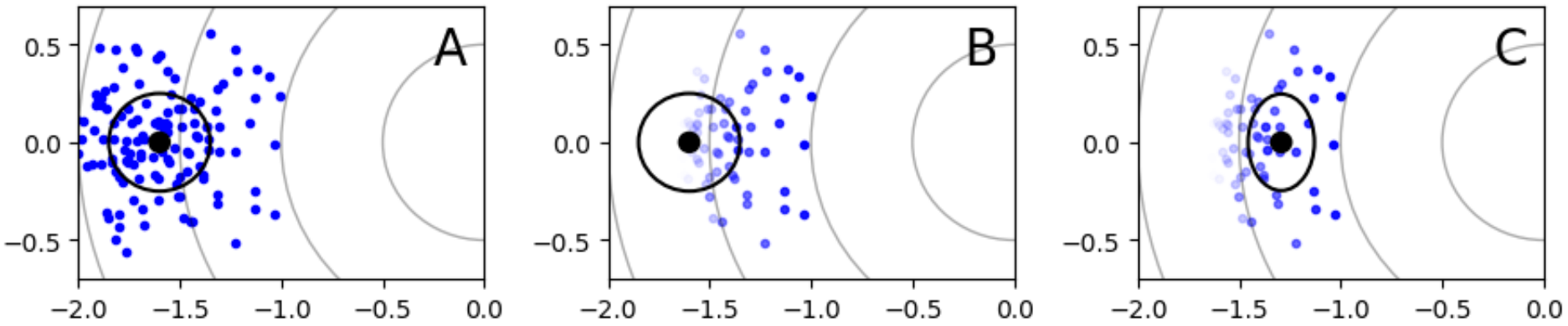
CMA-ES and CEM

Only keep the best **x**



CMA-ES and CEM

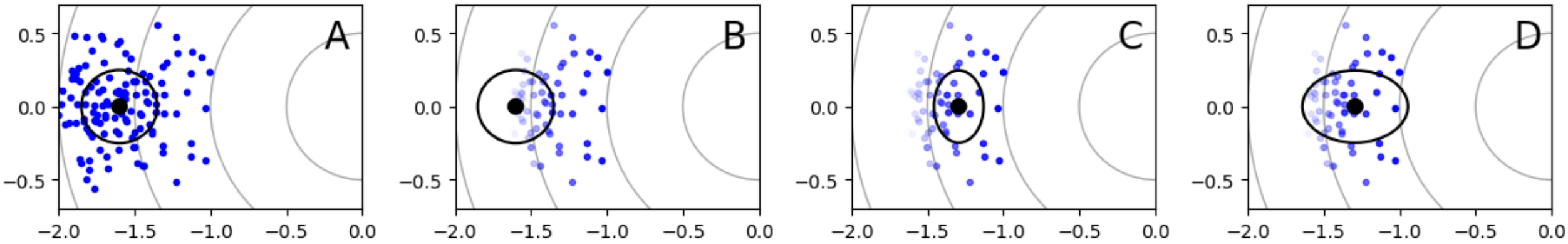
CEM: fit $p(\mathbf{x})$
to the best $\mathbf{x}$



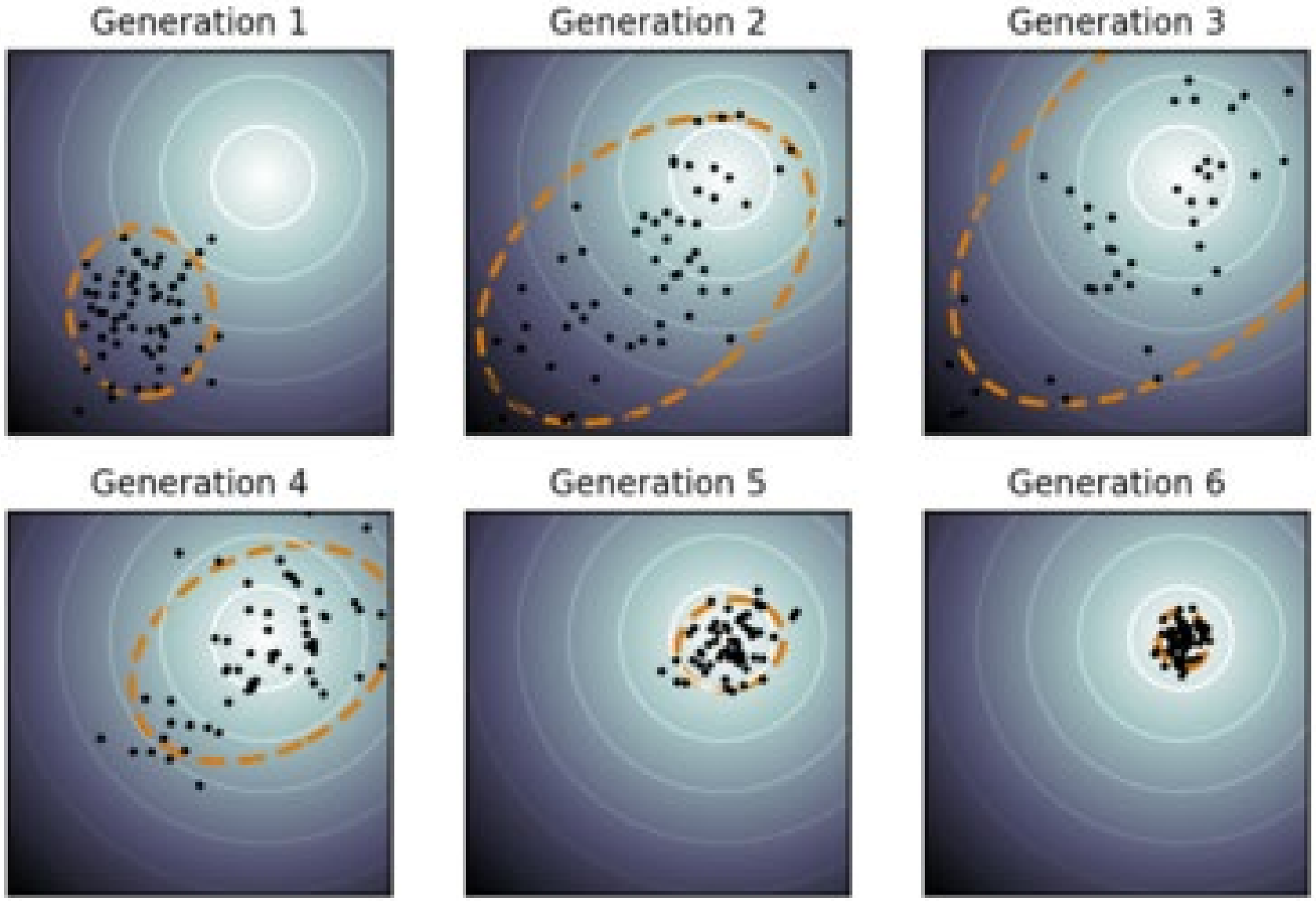
CMA-ES and CEM

CEM: fit $p(\mathbf{x})$ to the best $\mathbf{x}$

CMA-ES: $p(\mathbf{x})$ elongated in the progress direction



CMA-ES over multiple iterations



The CMA Evolution Strategy: A Tutorial

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

Contents

https://arxiv.org/pdf/1604.00772

A Tutorial on the Cross-Entropy Method

PIETER-TJERK DE BOER * ptdeboer@cs.utwente.nl
Department of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands

DIRK P. KROESE kroese@maths.uq.edu.au
Department of Mathematics, The University of Queensland, Brisbane 4072, Australia

SHIE MANNOR shie@ece.mcgill.ca
Department of Electrical and Computer Engineering, McGill University, 3480 University Street, Montreal, Quebec, Canada

REUVEN Y. RUBINSTEIN ierrr01@ie.technion.ac.il
Department of Industrial Engineering, Technion, Israel Institute of Technology, Haifa 32000, Israel

Received January 2003; Revised November 2003; Accepted January 2004

Abstract. The cross-entropy (CE) method is a new generic approach to combinatorial and multi-extremal optimization and rare event simulation. The purpose of this tutorial is to give a gentle introduction to the CE method. We present the CE methodology, the basic algorithm and its modifications, and discuss applications in combinatorial optimization and machine learning.

https://people.smp.uq.edu.au/DirkKroese/ps/aortut.pdf

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

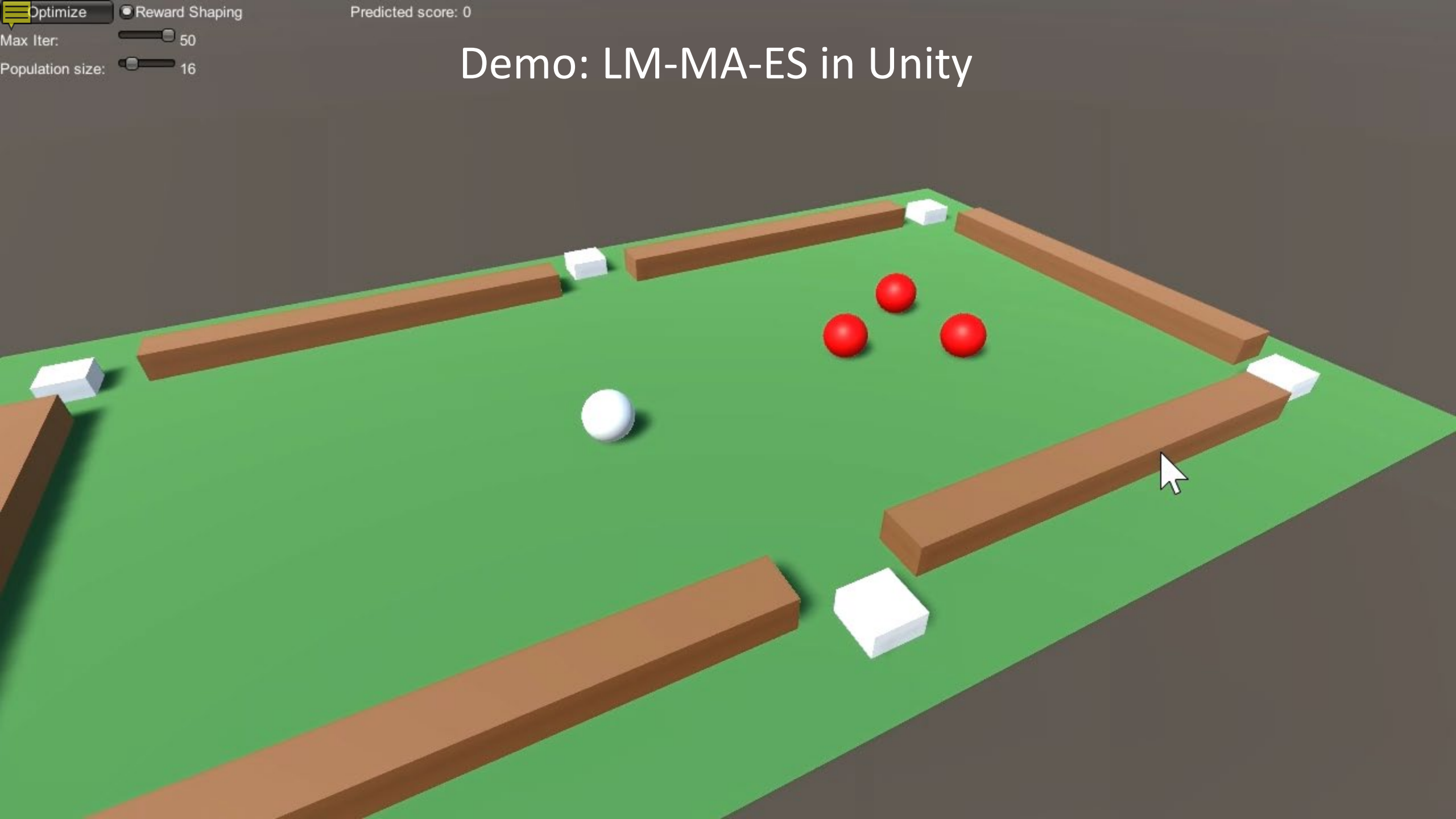
Hans-Georg Beyer
Research Center Process and Product Engineering
Vorarlberg University of Applied Sciences, Dornbirn, Austria
hans-georg.beyer@fhv.at

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)

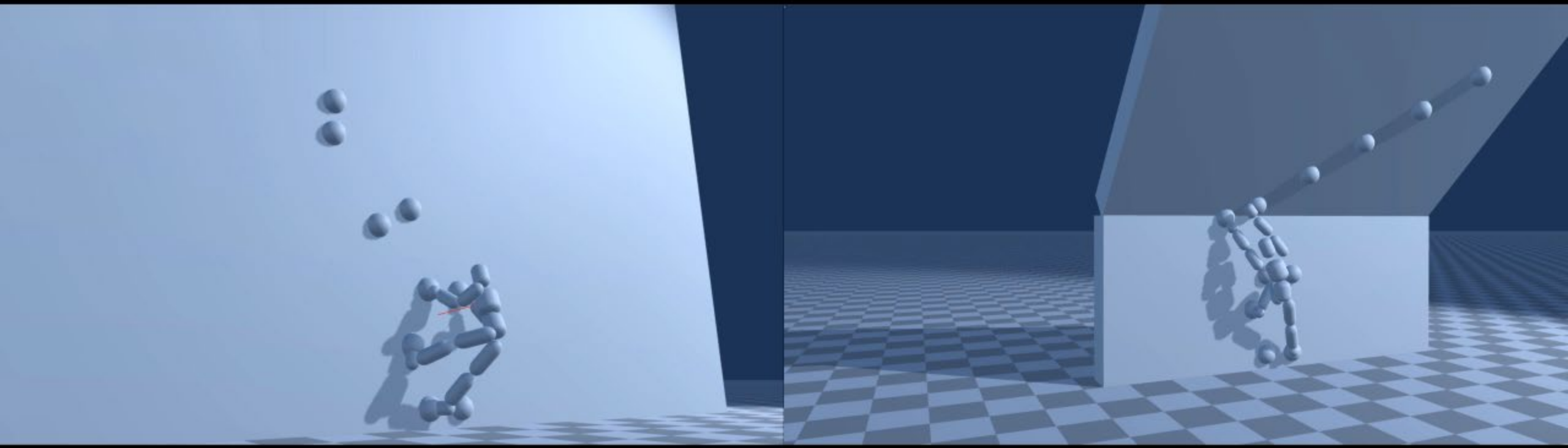
- Limited memory variant of CMA-ES, scales to millions of variables, O(N log(N))
- LM-MA-ES works even in generating adversarial images, something that had previously only been done with gradient-based optimization
- Unity C# implementation provided in the course repository



Demo: LM-MA-ES in Unity

In the Aalto Drawing game, we used LM-MA-ES in Unity to align user drawings with model images for similarity comparison

CMA-ES optimization of climbing movements (Naderi, Rajamäki & Hämäläinen 2017)




Interactive emergent movement. No training data.

Discrete/combinatorial optimization

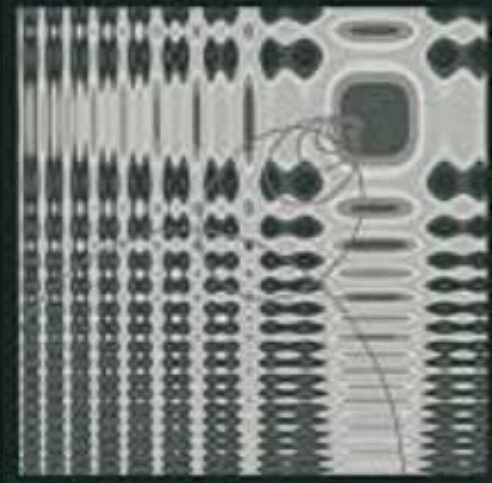
Discrete/combinatorial optimization

- 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.
- Fortunately, sampling-based gradient estimation still works. Mathematically, the optimized expectation is continuous even if x is discrete.
- For multiple variables (combinatorial optimization), one can often assume independence as $p(x,y)=p(x)p(y)$ and sample x,y from their own discrete distributions
- Popular in practice: CEM.
- CMA-ES is not applicable to the discrete case.

The Cross-Entropy Method

A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning

Reuven Y. Rubinstein
Dirk P. Kroese



Information Science & Statistics

Combinatorial Optimization of Graphical User Interface Designs

This article surveys combinatorial optimization as a flexible and powerful tool for computational generation and adaptation of graphical user interfaces (GUIs).

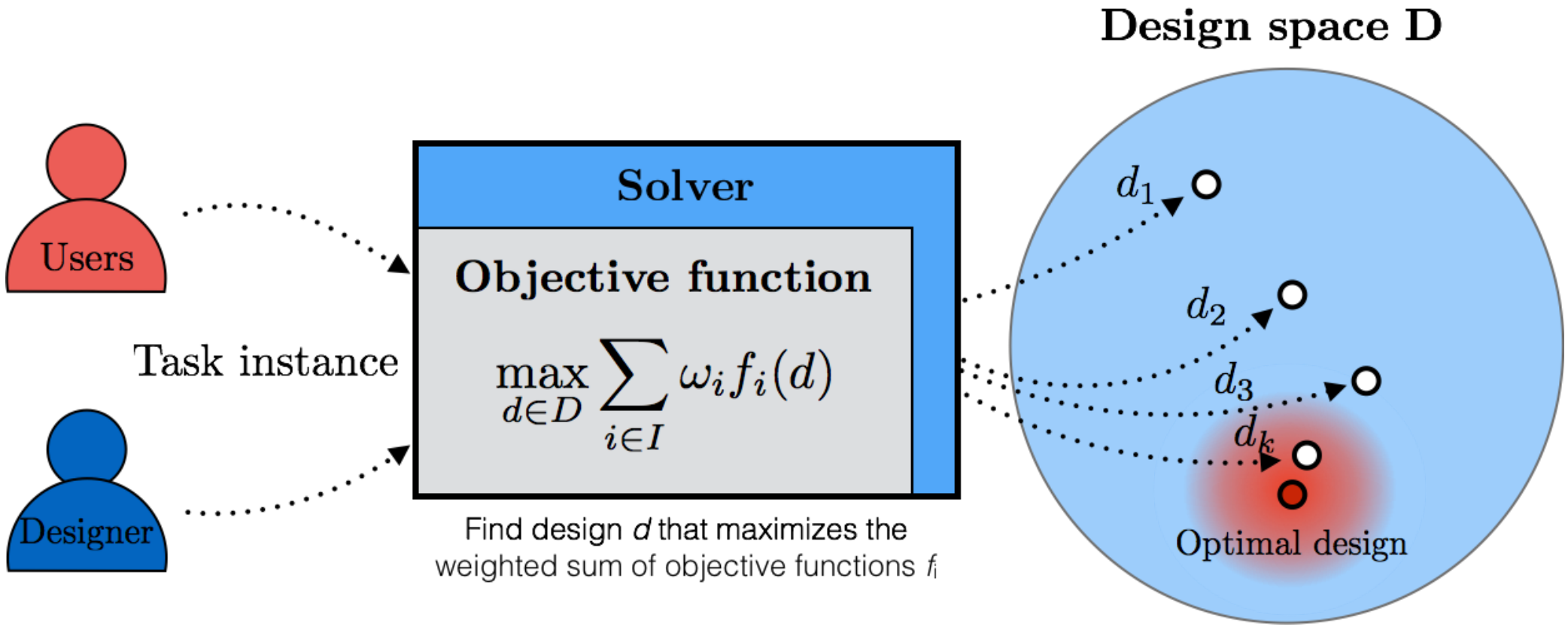
By Antti Oulasvirta, Niraj Ramesh Dayama, Morteza Shiripour, Maximilian John, and Andreas Karrenbauer

ABSTRACT | The graphical user interface (GUI) has become the prime means for interacting with computing systems. It leverages human perceptual and motor capabilities for elementary tasks such as command exploration and invocation, information search, and multitasking. For designing a GUI, numerous interconnected decisions must be made such that the outcome strikes a balance between human factors and technical objectives. Normally, design choices are specified manually and coded within the software by professional designers and developers. This article surveys combinatorial optimization as a flexible and powerful tool for computational generation and adaptation of GUIs. As recently as 15 years ago, applications were limited to keyboards and widget layouts. The obstacle has been the mathematical definition of design tasks, on the one hand, and the lack of objective functions that capture essential aspects of human behavior, on the other. This article presents definitions of layout design problems as integer programming tasks, a coherent formalism that permits identification of problem types, analysis of their complexity, and exploitation of known algorithmic solutions. It then surveys advances in formulating evaluative functions for common design-goal foci such as user performance and experience. The convergence of these two advances has expanded the range of solvable problems. Approaches to practical deployment are outlined with a wide spectrum of applications. This article concludes by discussing the position of this application area within optimization and human–computer interaction research and outlines challenges for future work.

KEYWORDS | Combinatorial optimization; computational design; graphical user interfaces (GUIs); human–computer interaction (HCI); integer programming; interactive optimization; meta-heuristic optimization.

I. INTRODUCTION

This article surveys combinatorial optimization approaches for graphical user interface (GUI) design. GUIs have become the prime user interface type for interacting with computing systems. They leverage our perceptual and



Oulasvirta's awesome Jupyter notebook on combinatorial optimization for UI design: https://urly.fi/1W9O

Heuristic for choosing an optimization method:

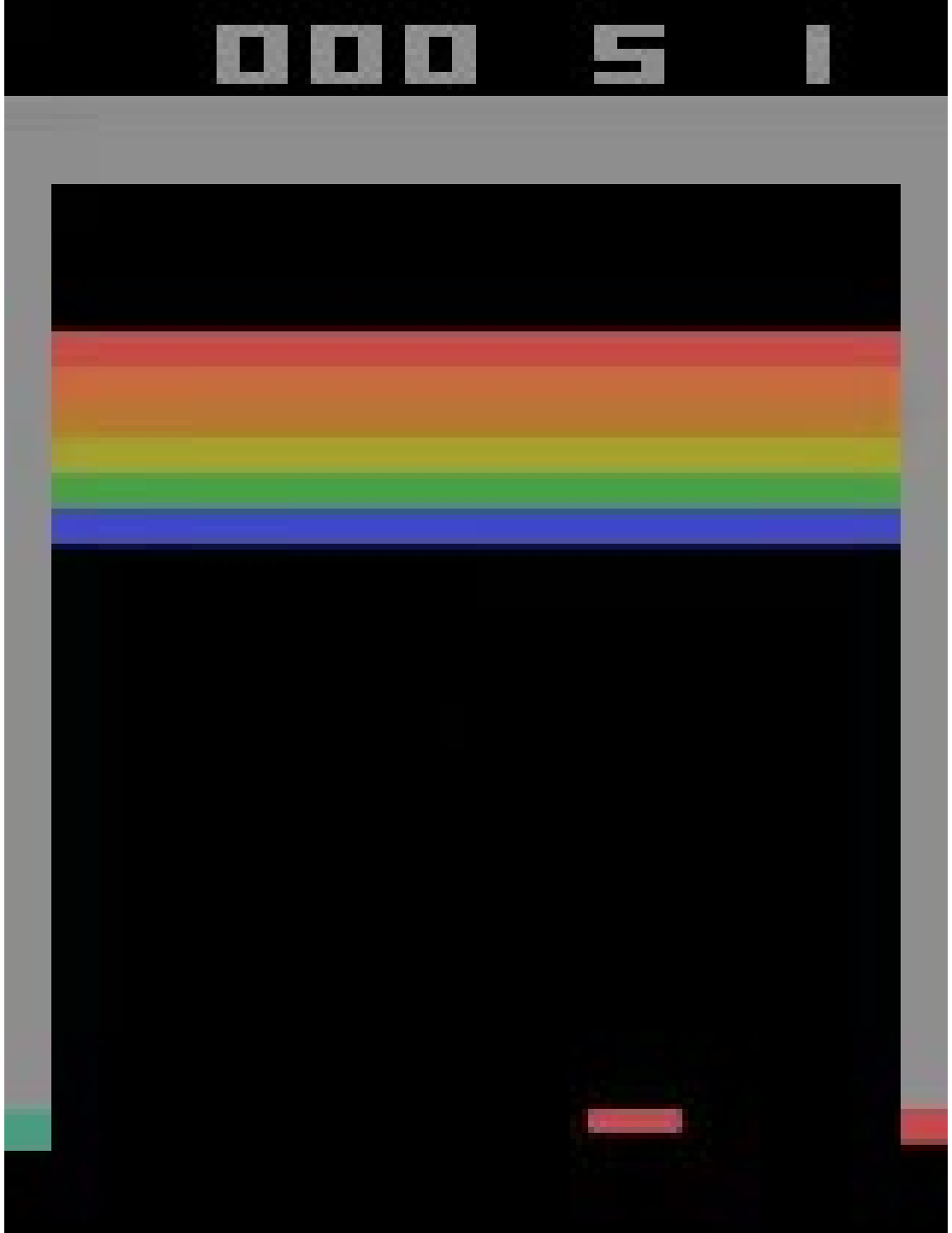
```
if differentiable f(x) with no difficult local optima:
    Use Adam
elif x is real-valued:
    if x is very high-dimensional (100+ variables):
        Use LM-MA-ES
    else:
        Use CMA-ES
else:
    Use CEM or Adam with sampling-based gradient
```

Break

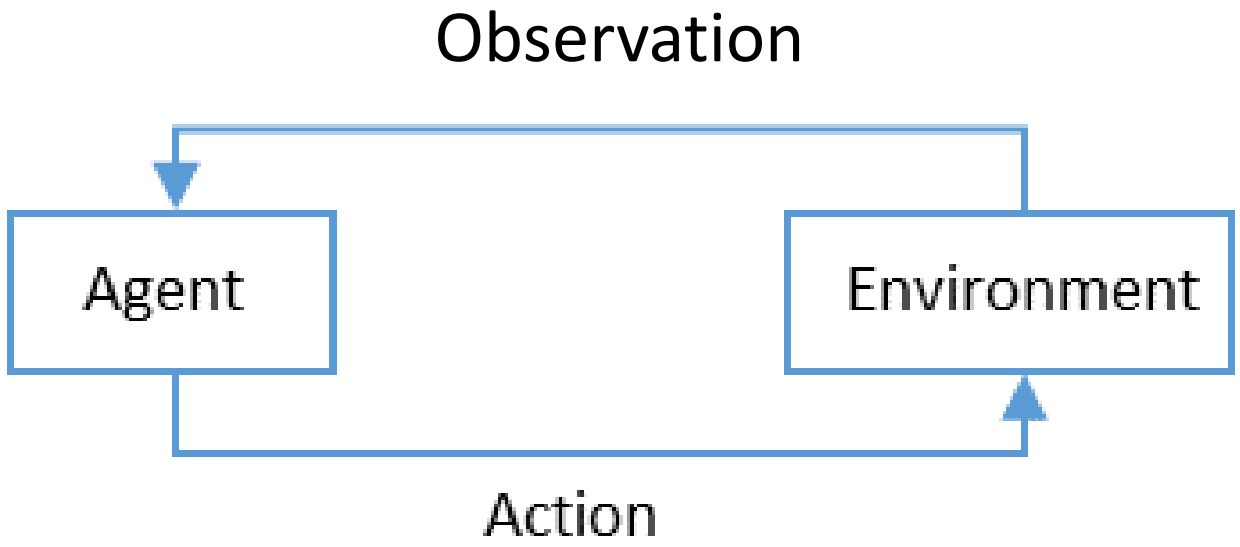
Deep Reinforcement learning (DRL)

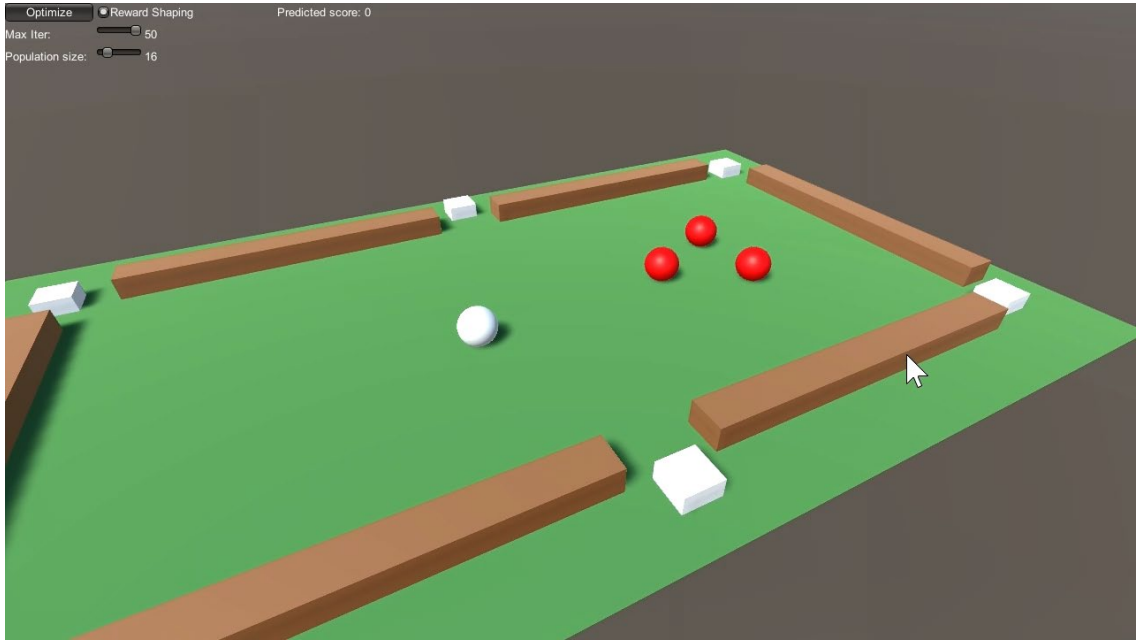
Mnih et al. 2015:

Human-level control of Atari games using Deep Reinforcement Learning

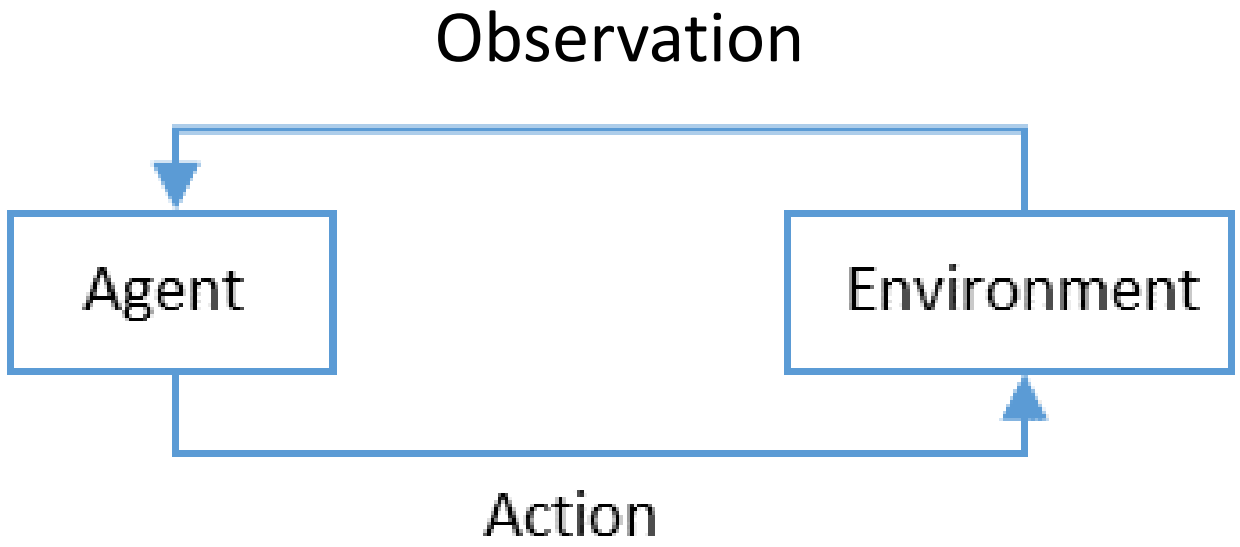


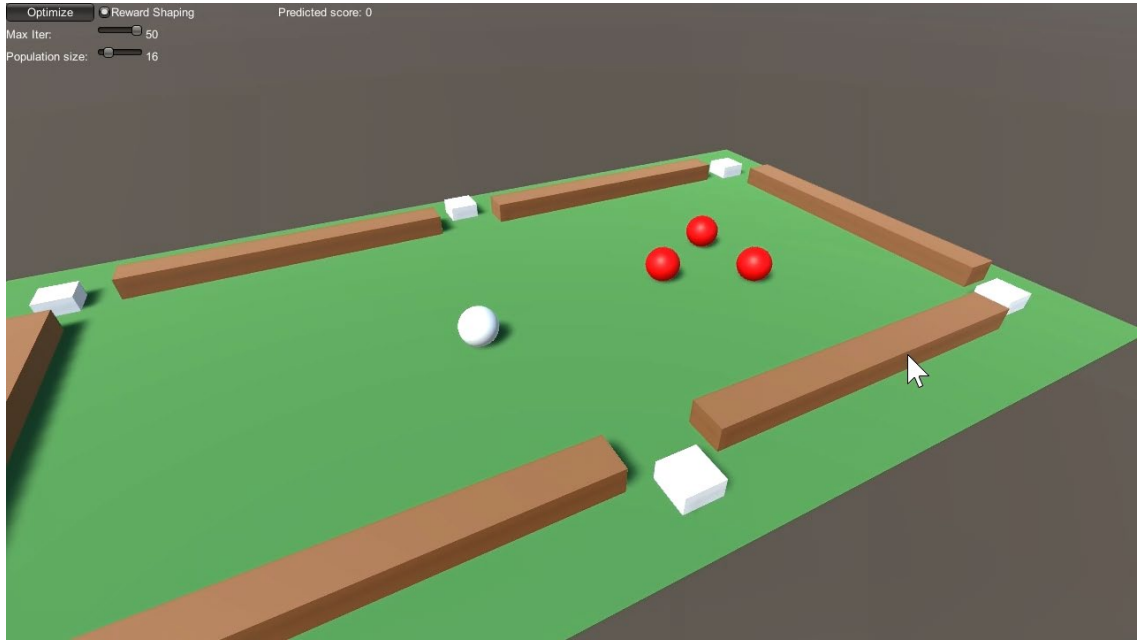
Real life: Countless variations of the same problems





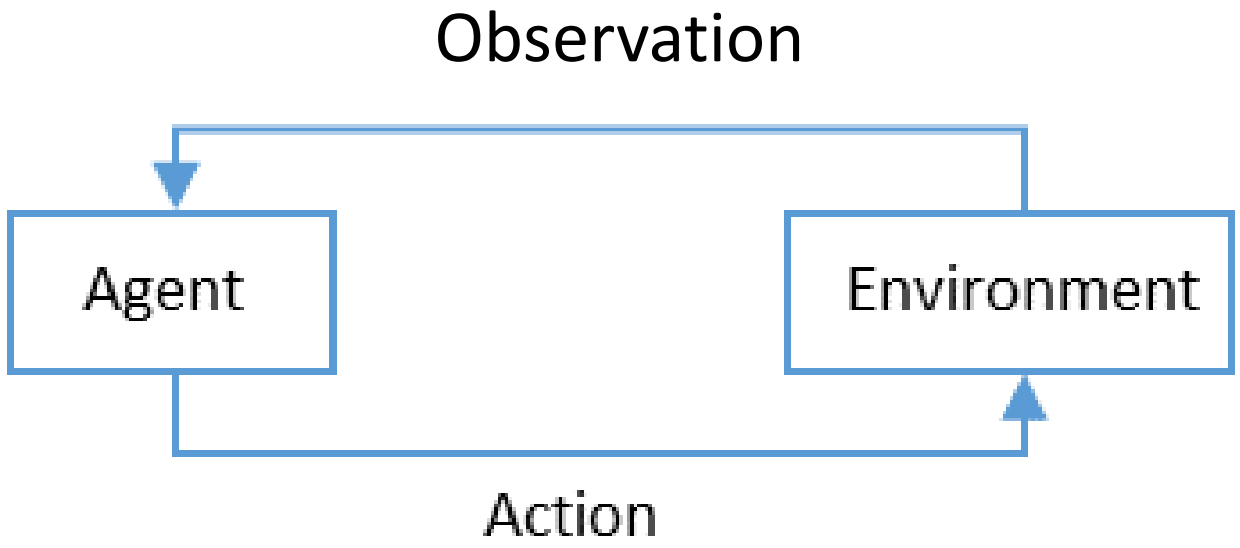
Real life: Countless variations of the same problems

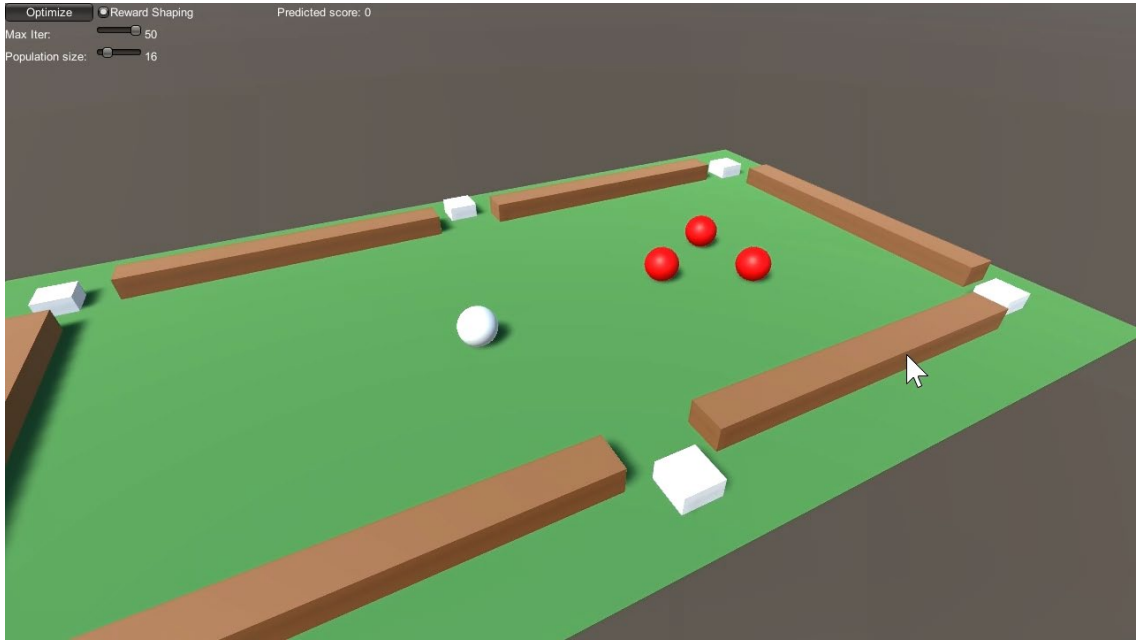




Observation: configuration of balls, i.e., problem definition for optimizing action

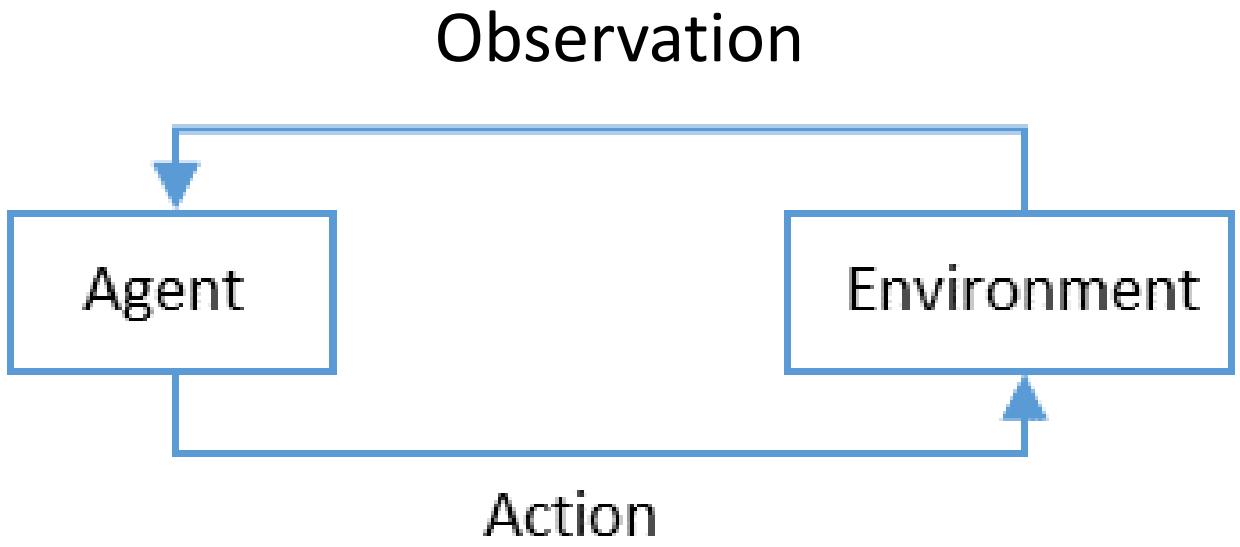
Real life: Countless variations of the same problems

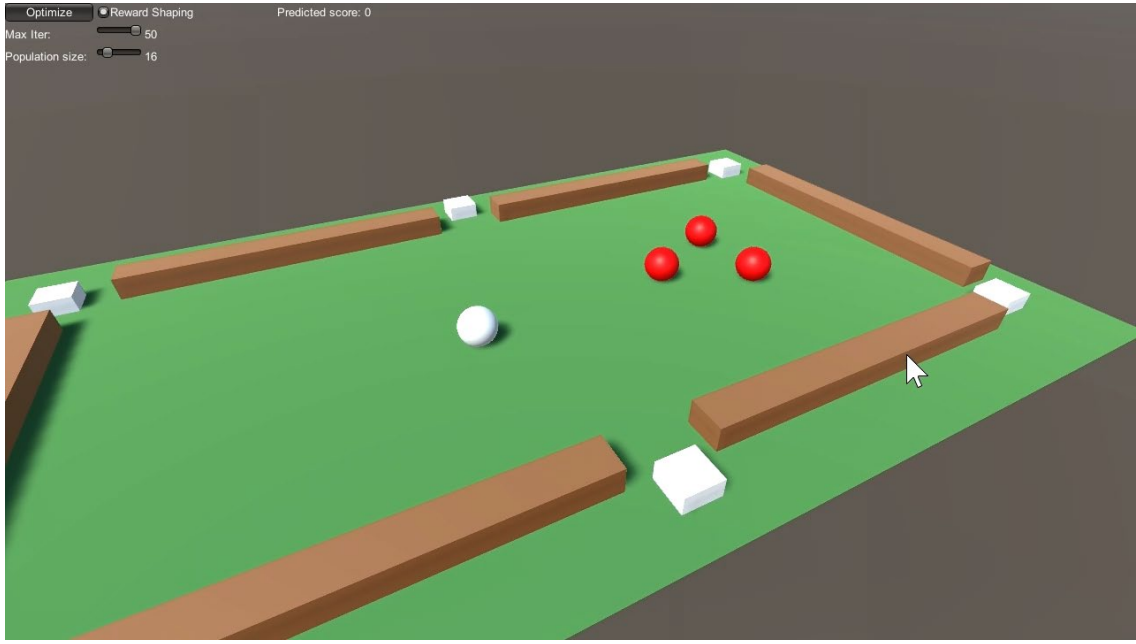




DRL goal: Instead of optimizing a single action **a**, optimize the parameters **θ** of a policy function $\mathbf{a}=\pi_{\boldsymbol{\theta}}(\mathbf{o})$ that gives the best action for any observation **o**.

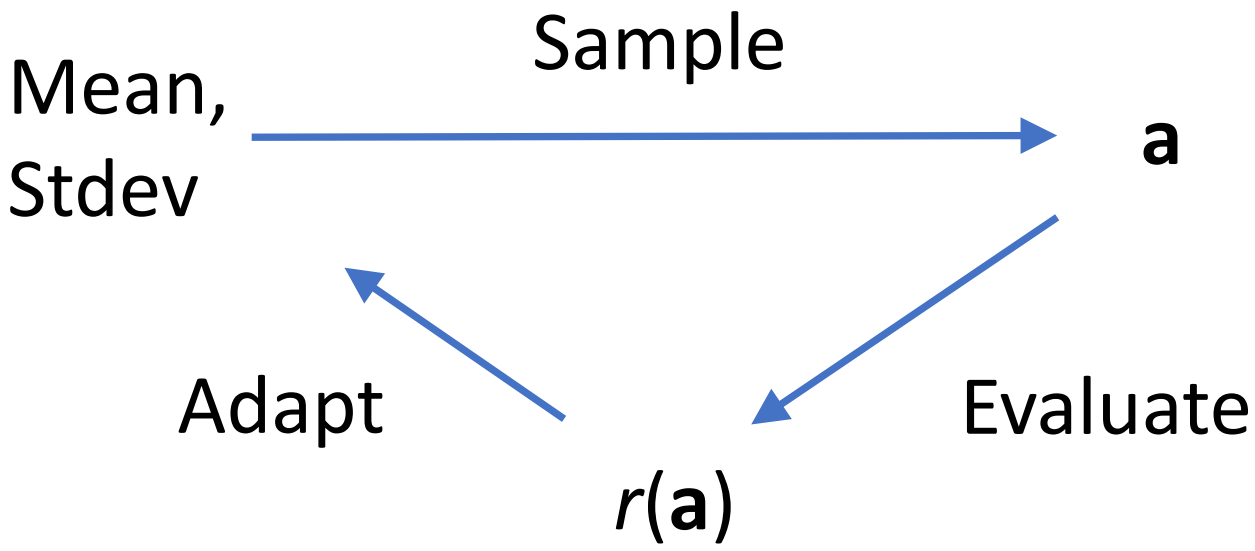
Real life: Countless variations of the same problems



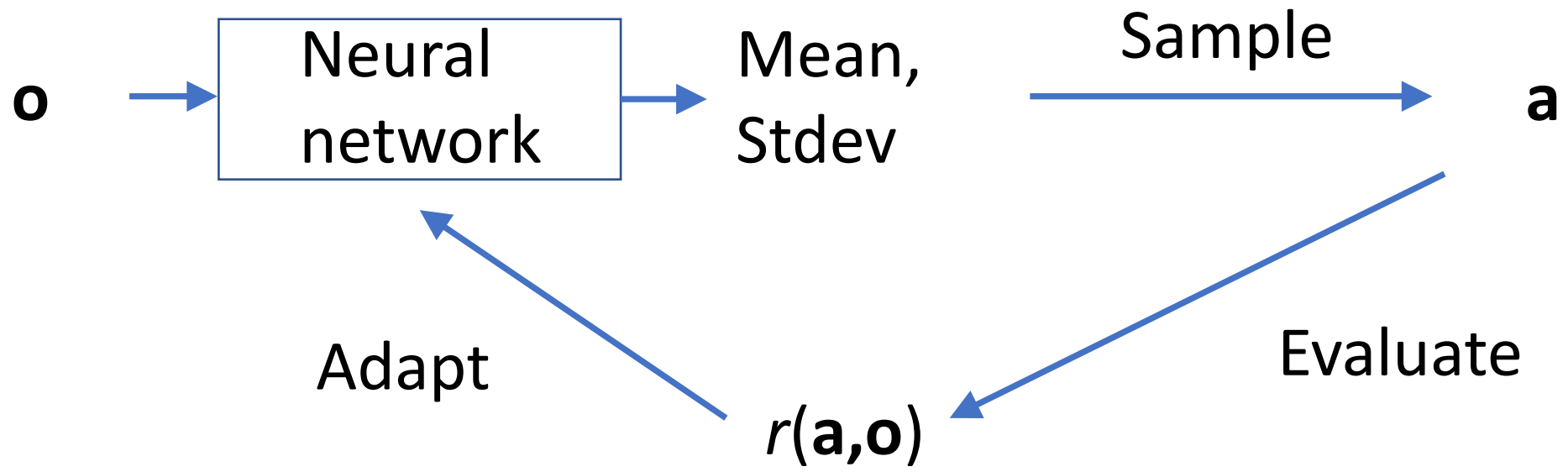


Essentially: Explore/sample different actions in different situations, memorize the best actions, and interpolate/extrapolate to be able to act in new situations.

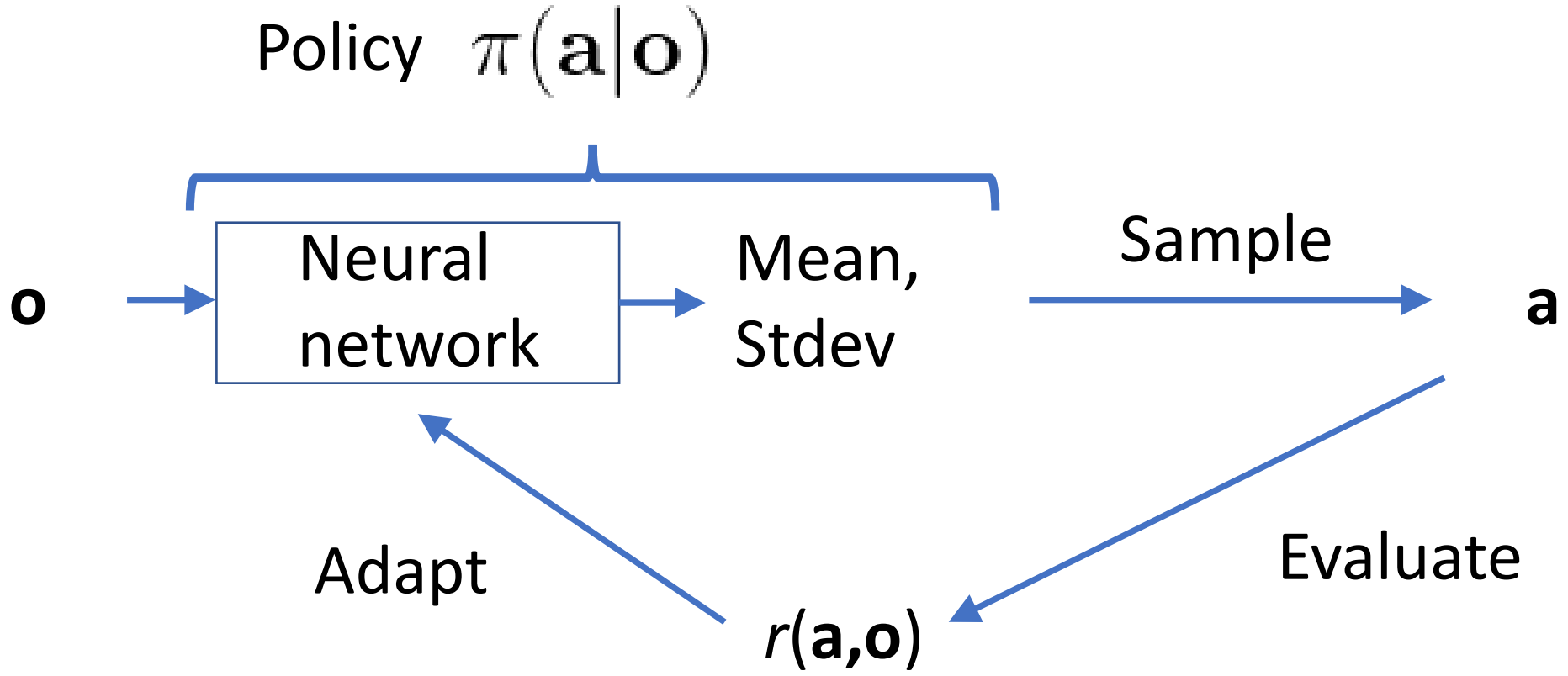
Optimizing actions independent of state (e.g., CMA-ES)



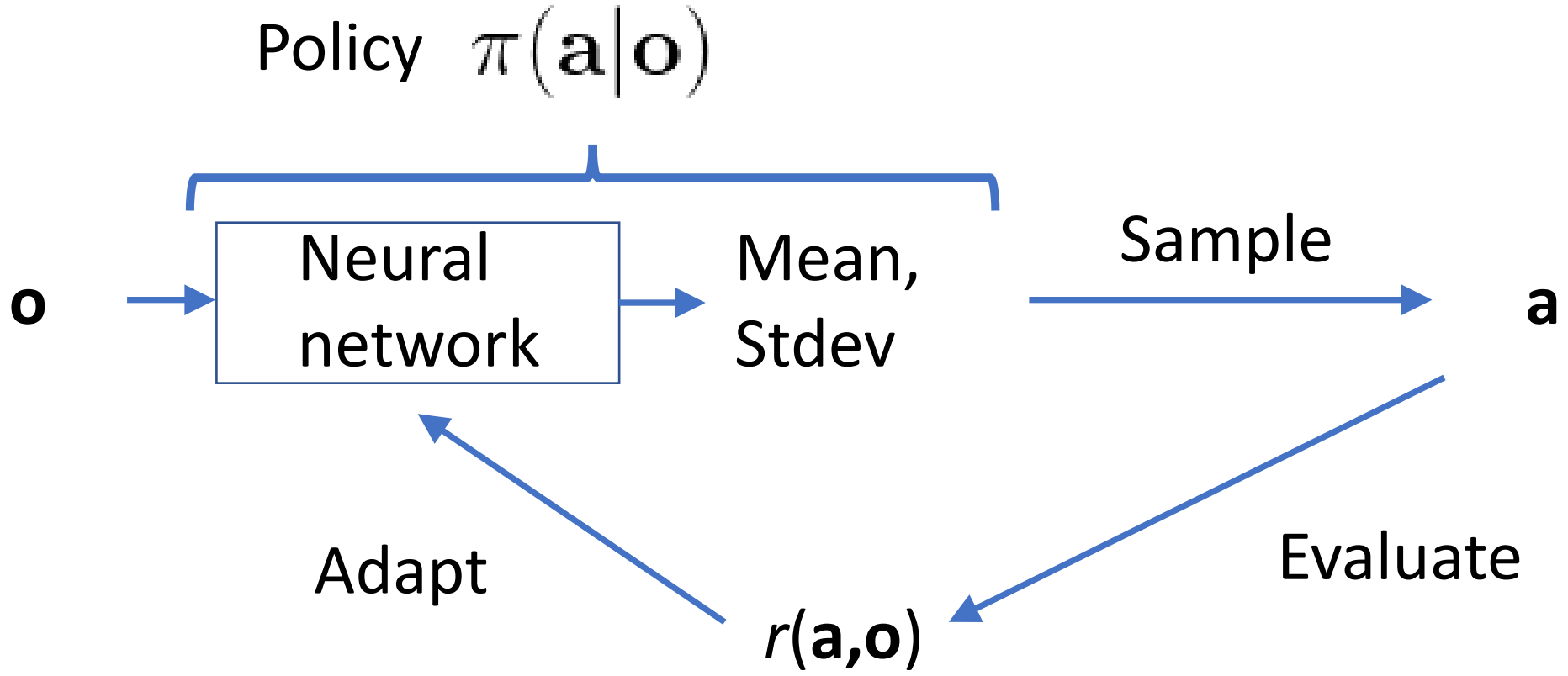
DRL: The action distribution depends on the observation (e.g., configuration of billiards balls)



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)



Reinforcement Learning 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

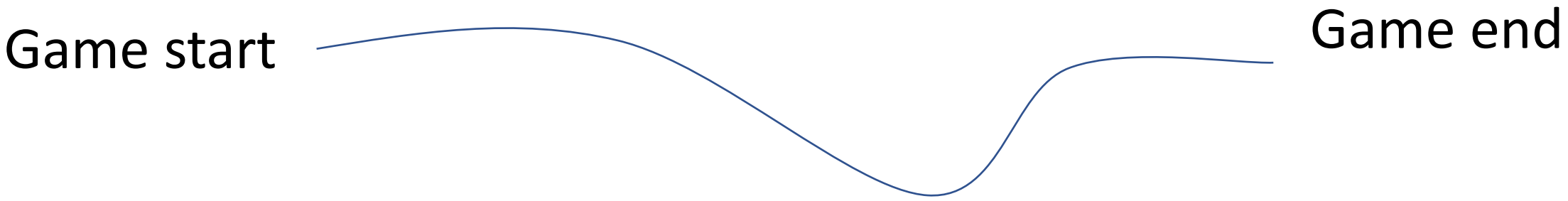
Execute action (simulate world model)

Observe new **o**’ and reward *r*

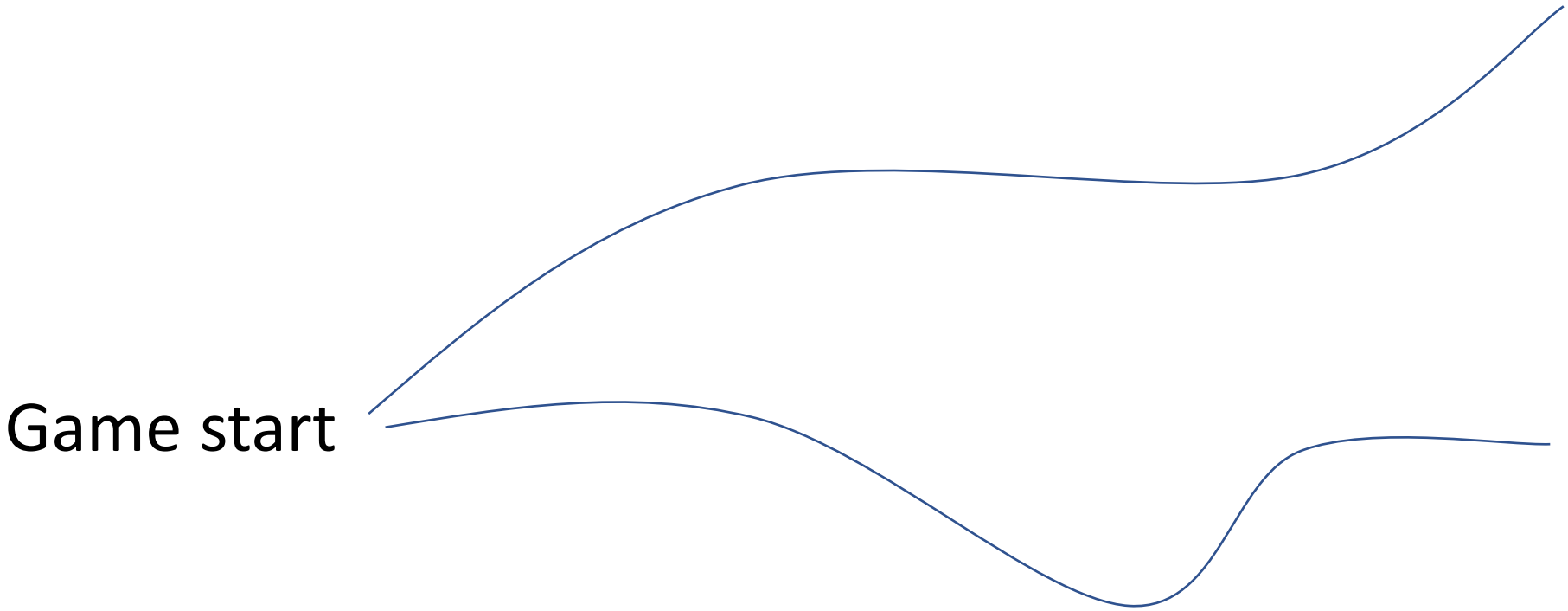
One episode

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

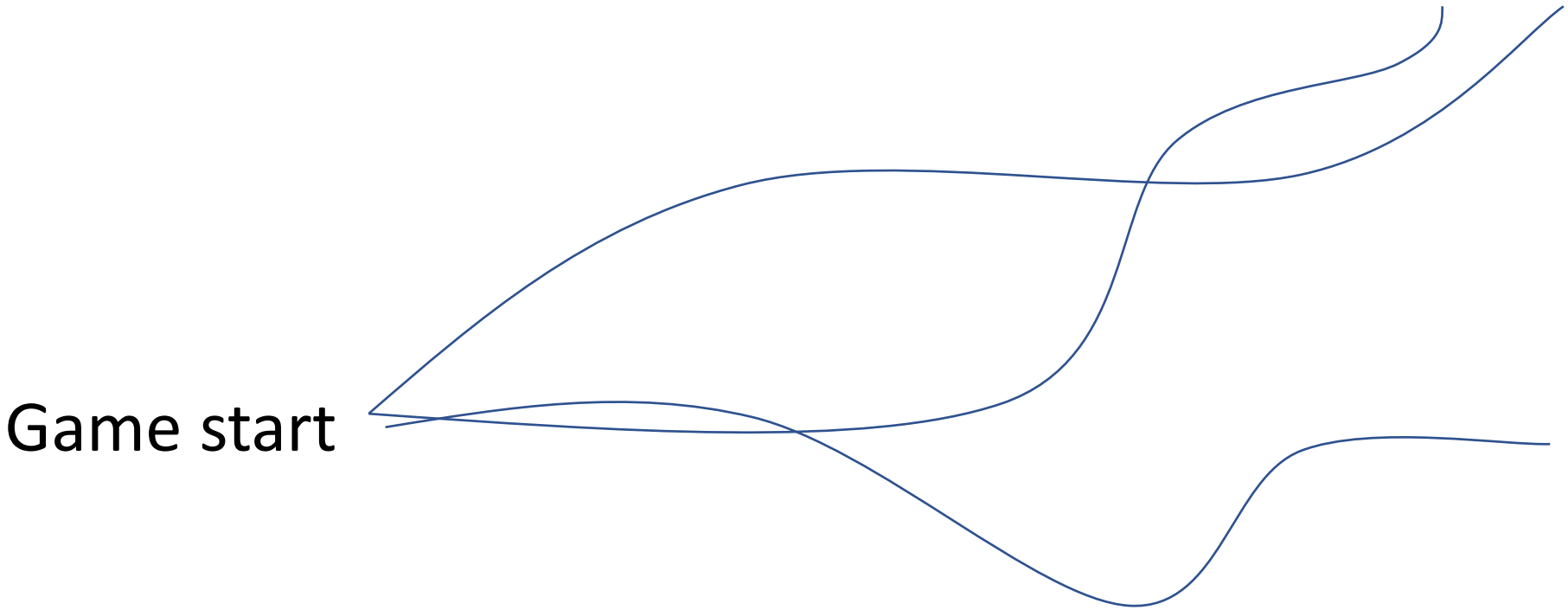
Each episode explores a trajectory in the state space



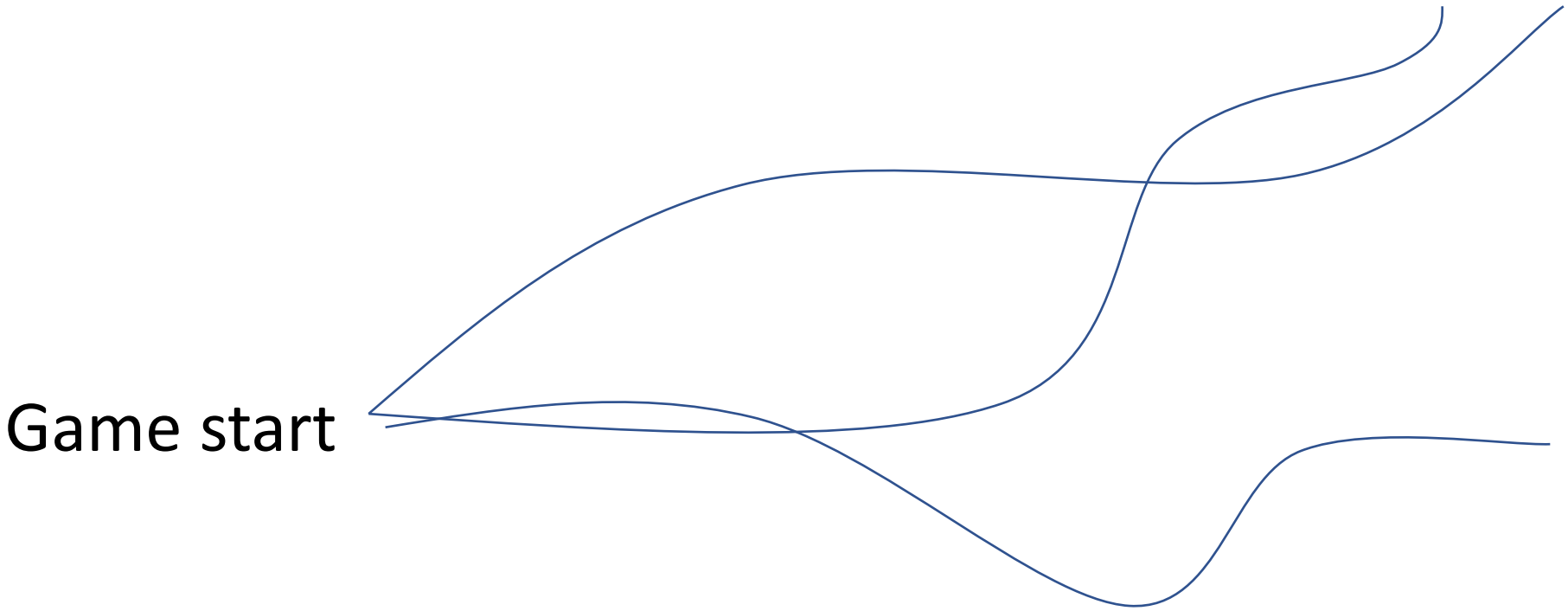
Each episode explores a trajectory in the state space



Each episode explores a trajectory in the state space



Each episode explores a trajectory in the state space

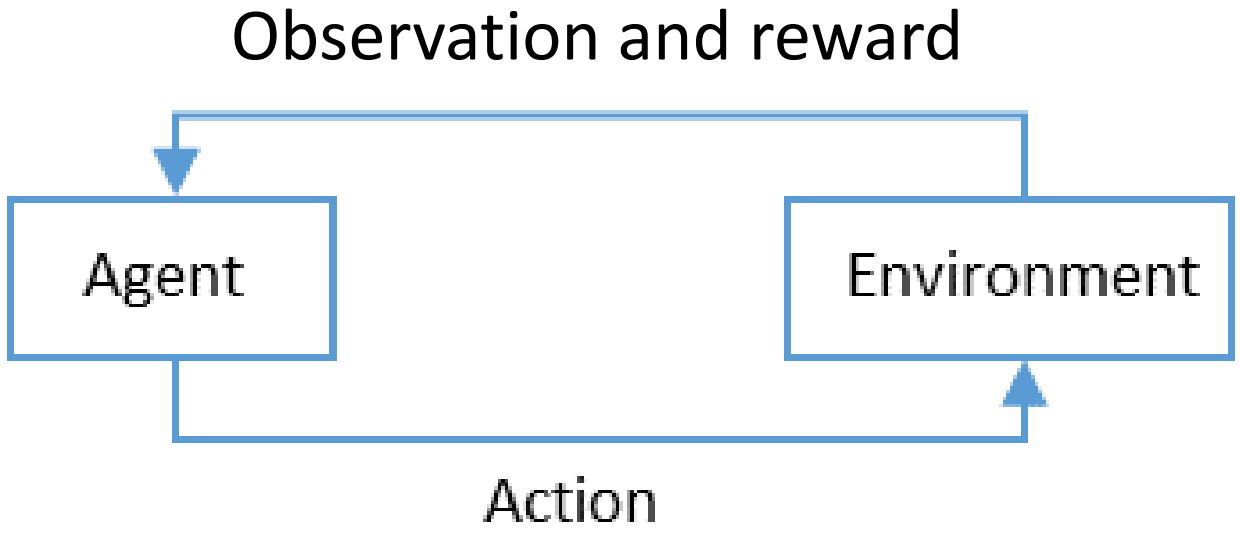


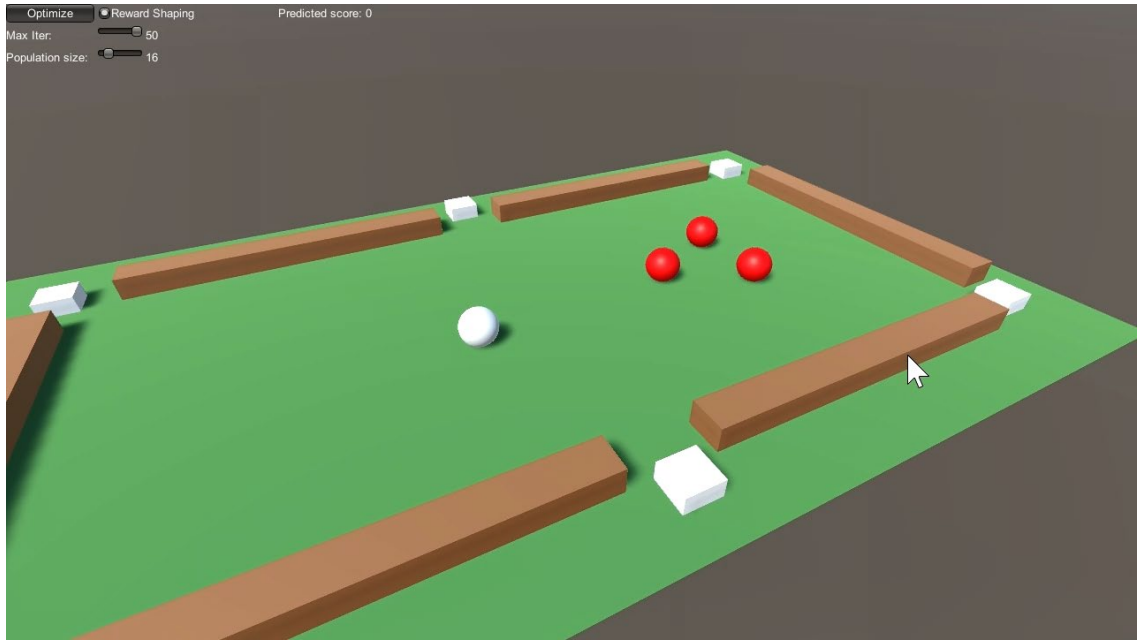
When enough episodes collected, use all the [**o**,**a**,**o**’,*r*] to compute the gradient

Applying RL in practice: Formulate Your Learning Problem as a 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 a discount factor 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**.
- DRL is a tool for discovering a policy that is optimal for a specific MDP => one adjusts AI behavior through changing the details of the **s**,**a**,r,γ

How does the discount factor γ work?

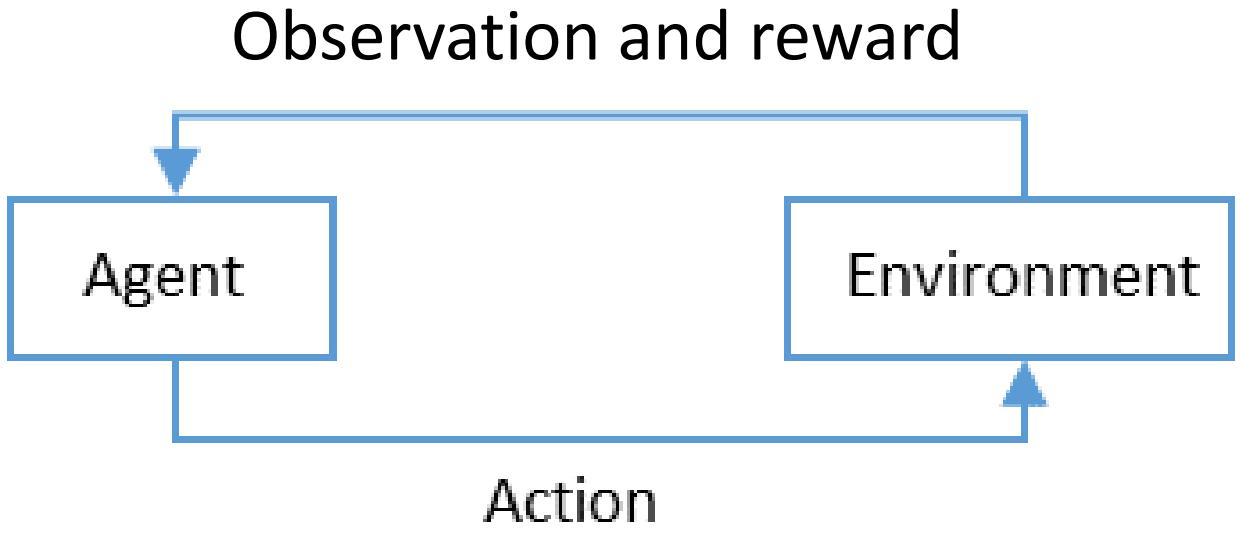


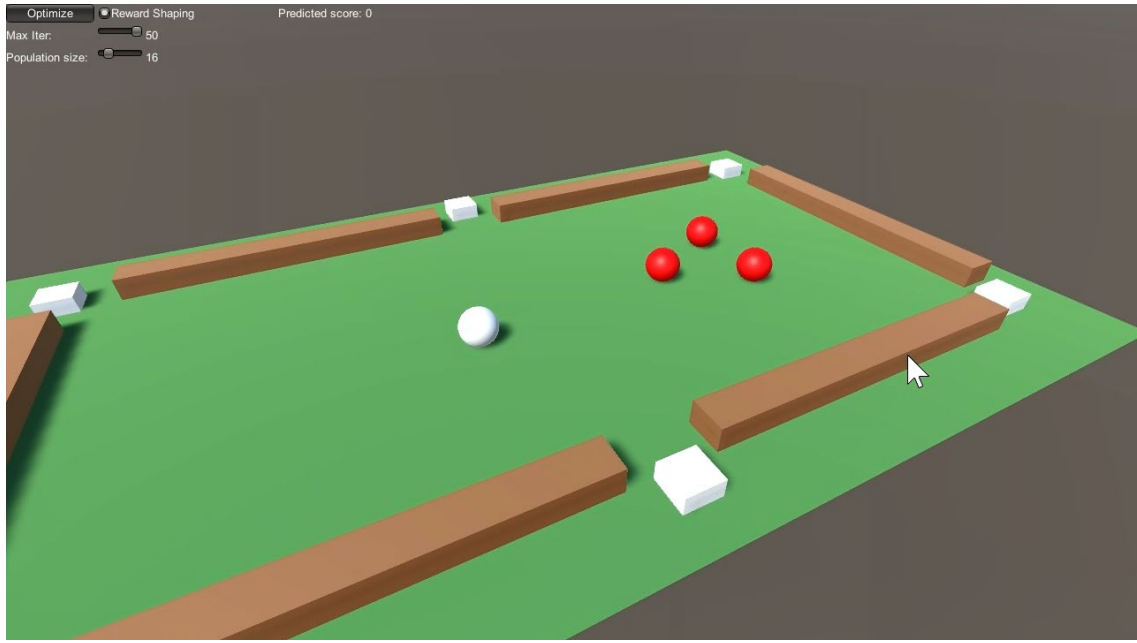


DRL optimization objective: Maximize the expected sum of (future) rewards:

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

How does the discount factor γ work?



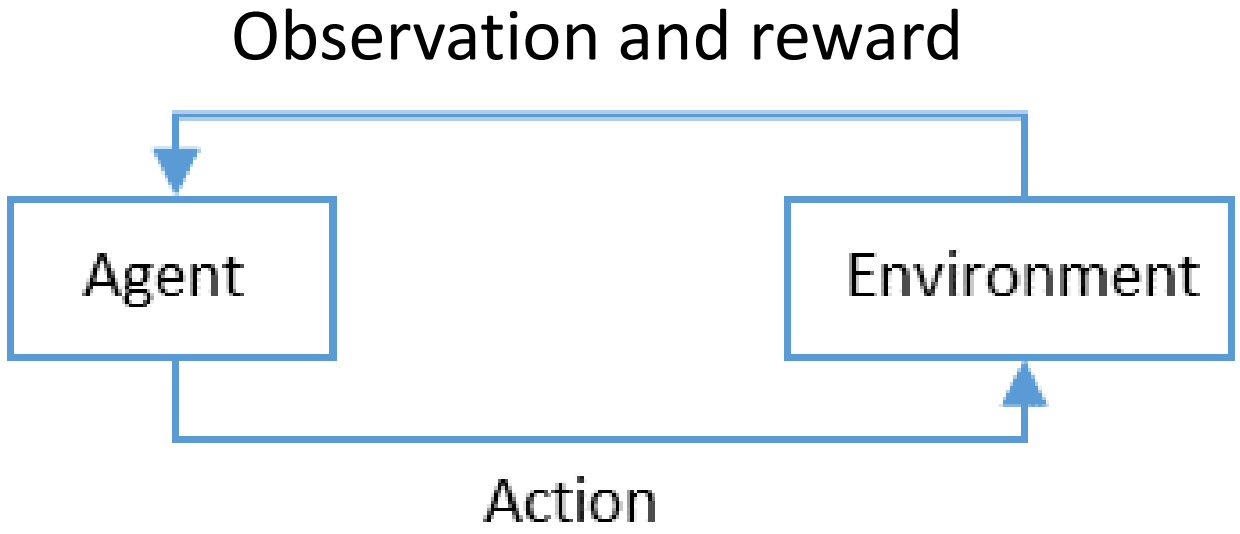


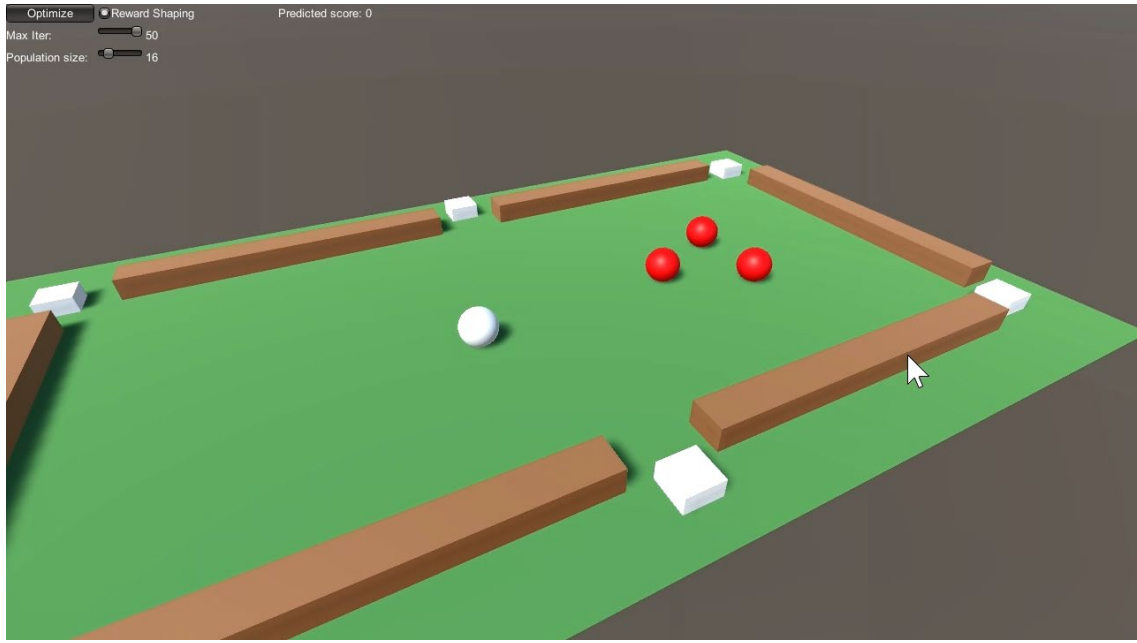
DRL optimization objective: Maximize the expected sum of (future) rewards:

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

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

How does the discount factor γ work?



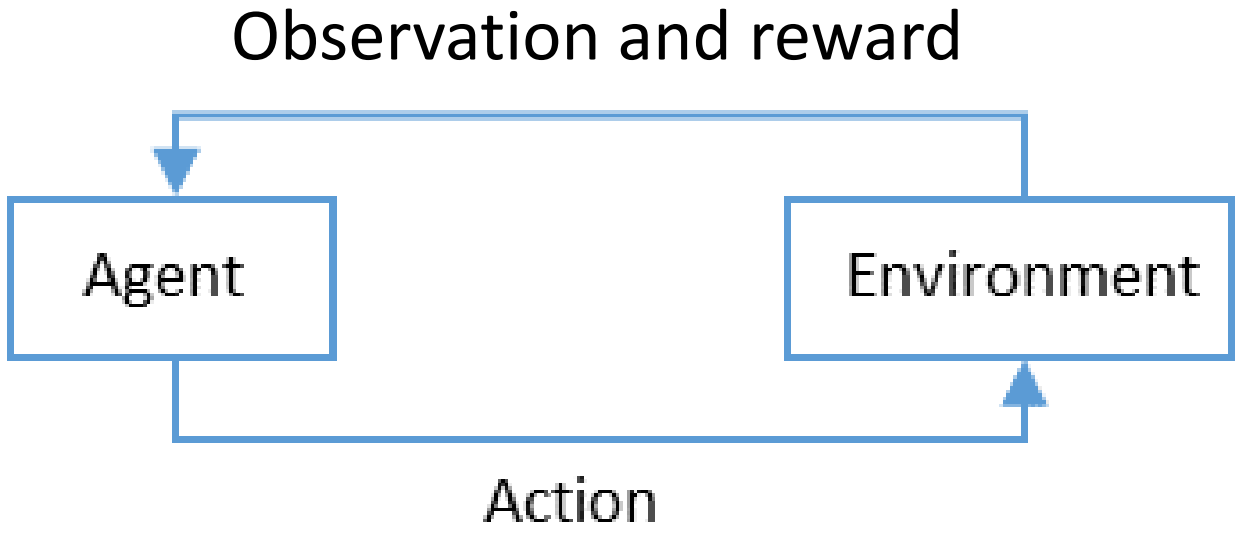


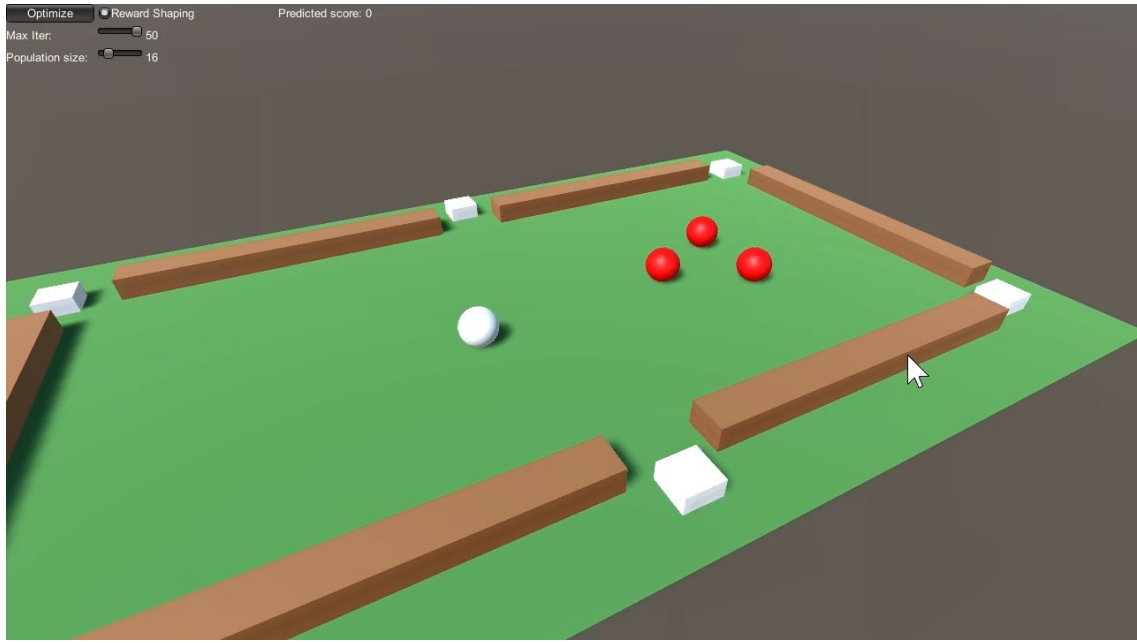
DRL optimization objective: Maximize the expected sum of (future) rewards:

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

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

How does the discount factor γ work?



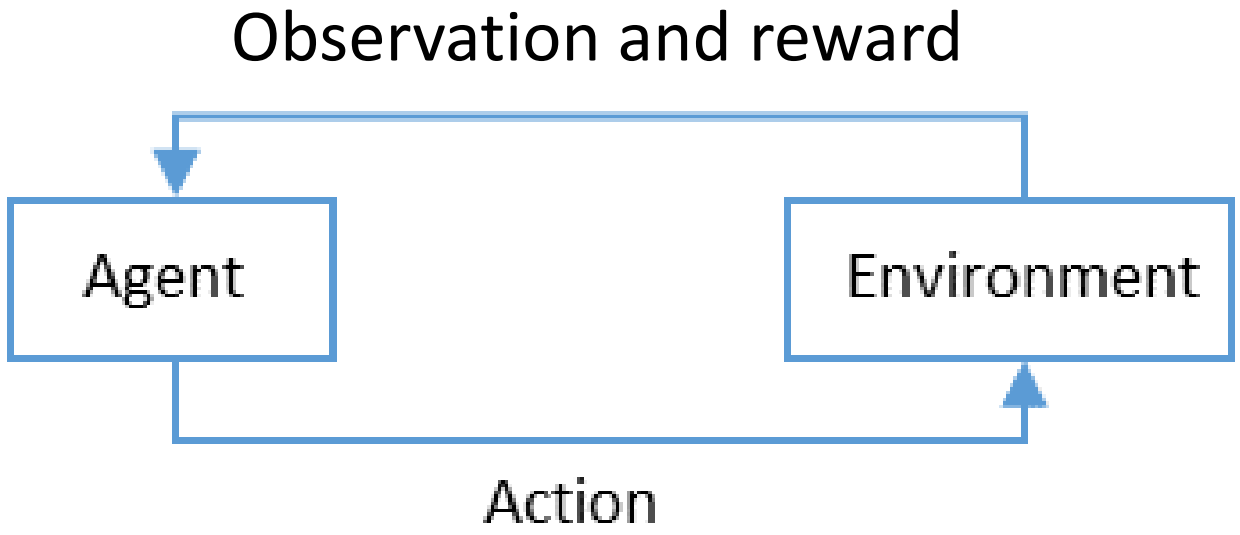


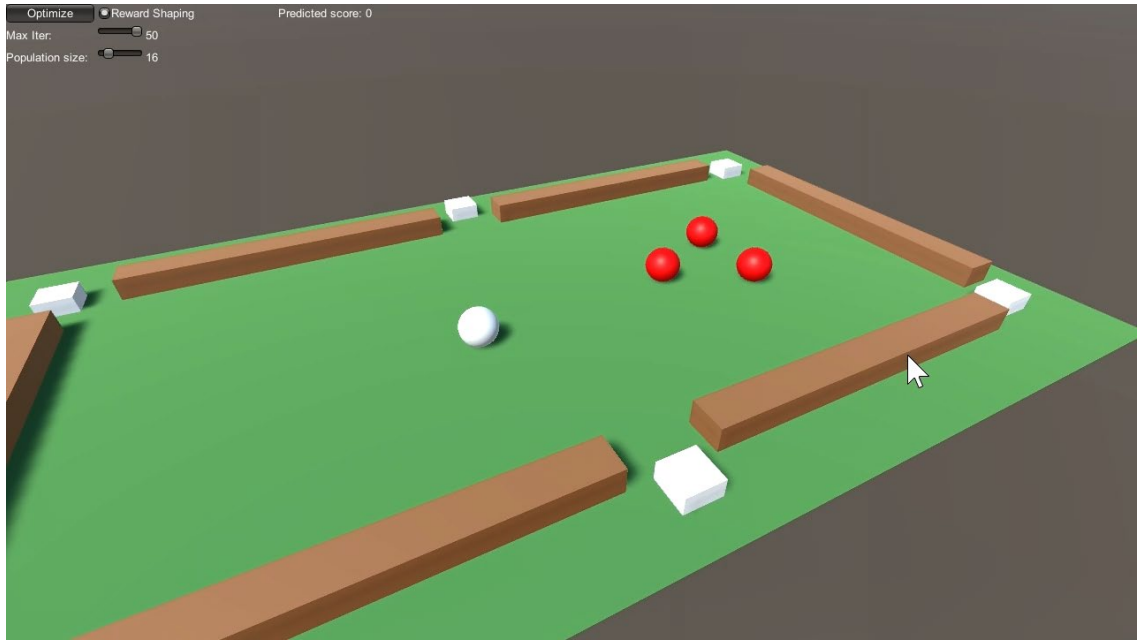
DRL optimization objective: Maximize the expected sum of (future) rewards:

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

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

How does the discount factor γ work?



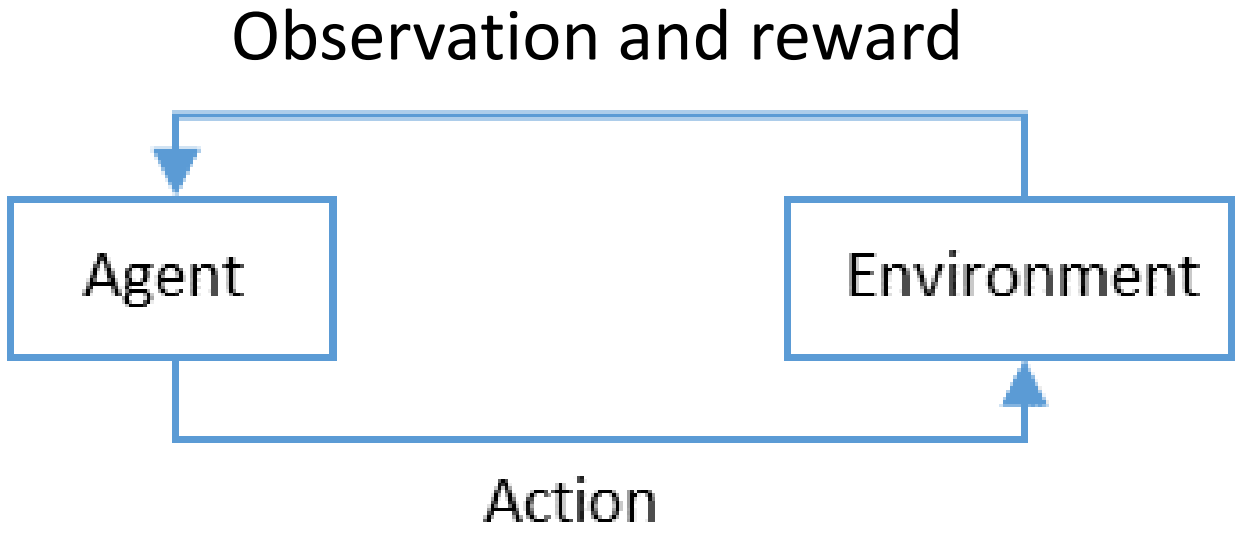


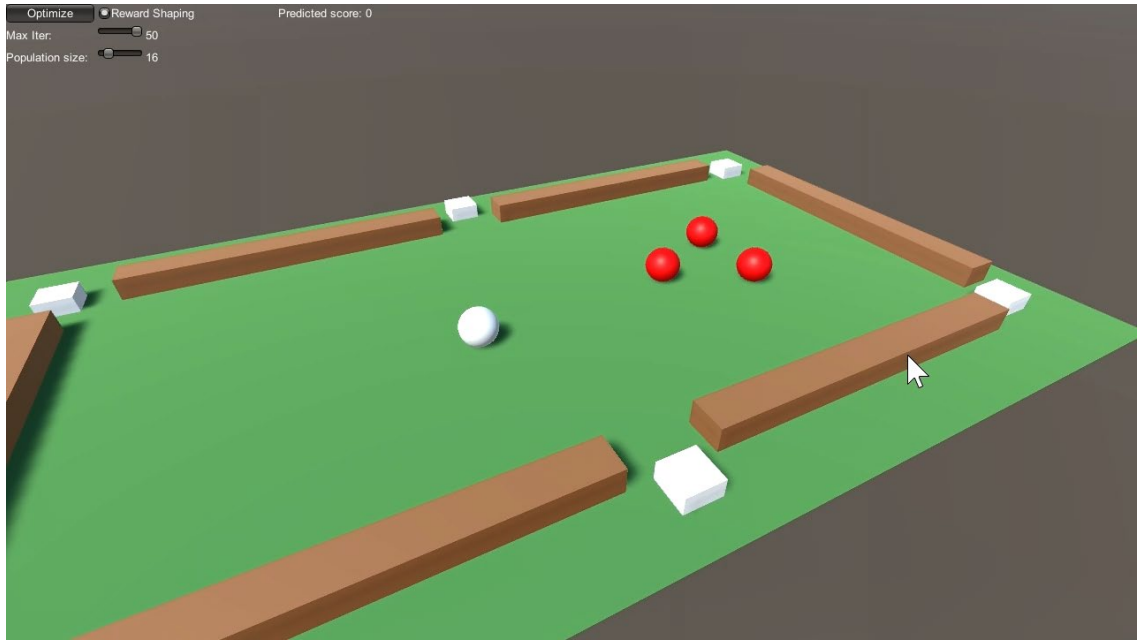
DRL optimization objective: Maximize the expected sum of (future) rewards:

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

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

How does the discount factor γ work?



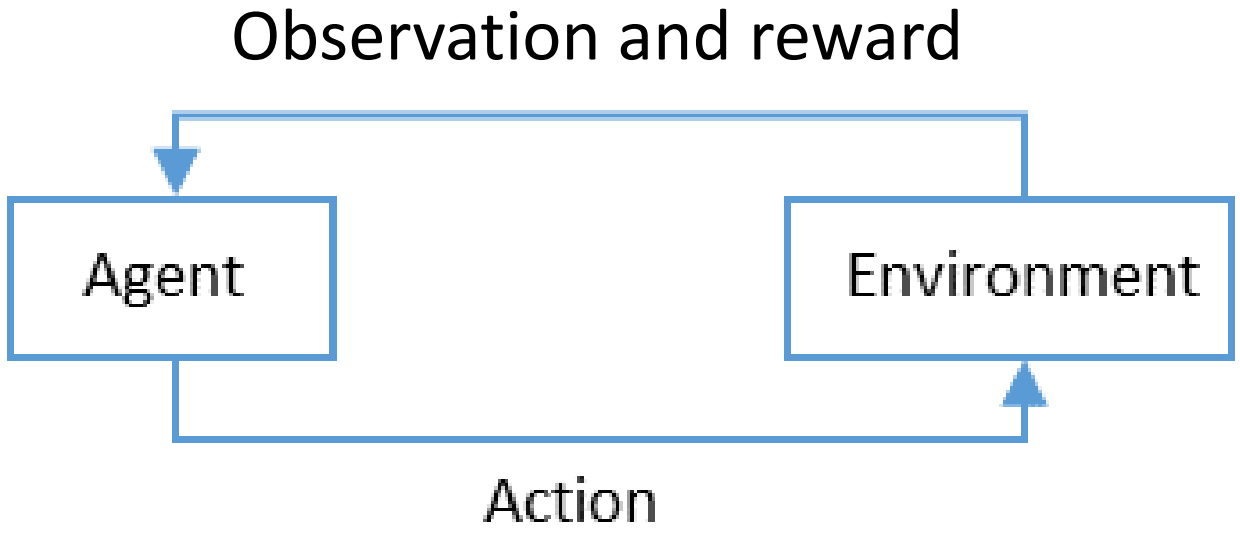


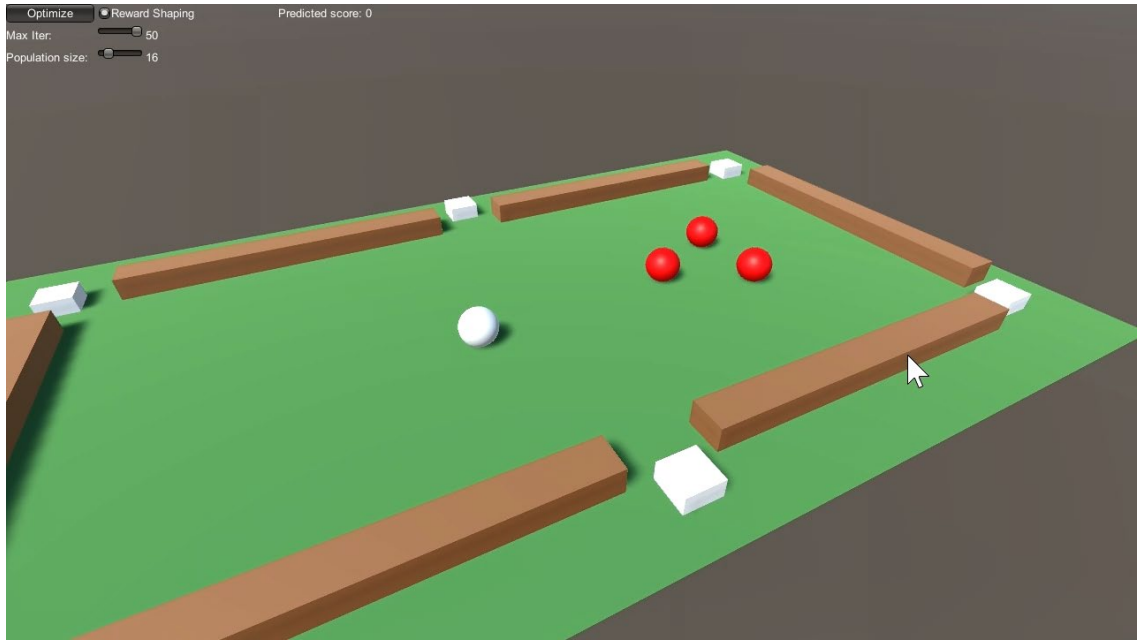
DRL optimization objective: Maximize the expected sum of (future) rewards:

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

A time-dependent multiplier for how much reward at time *t* contributes to the objective.

How does the discount factor γ work?

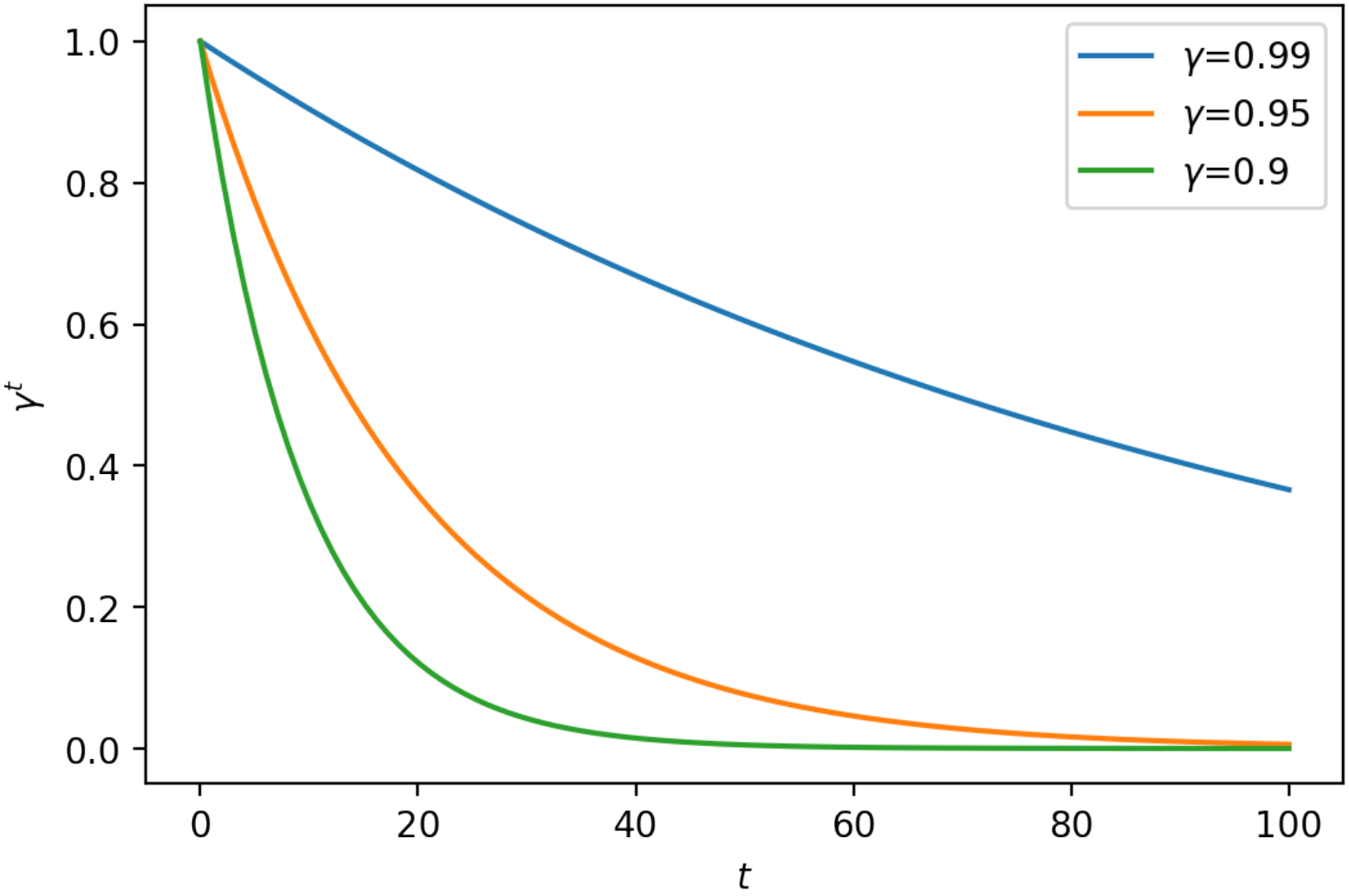




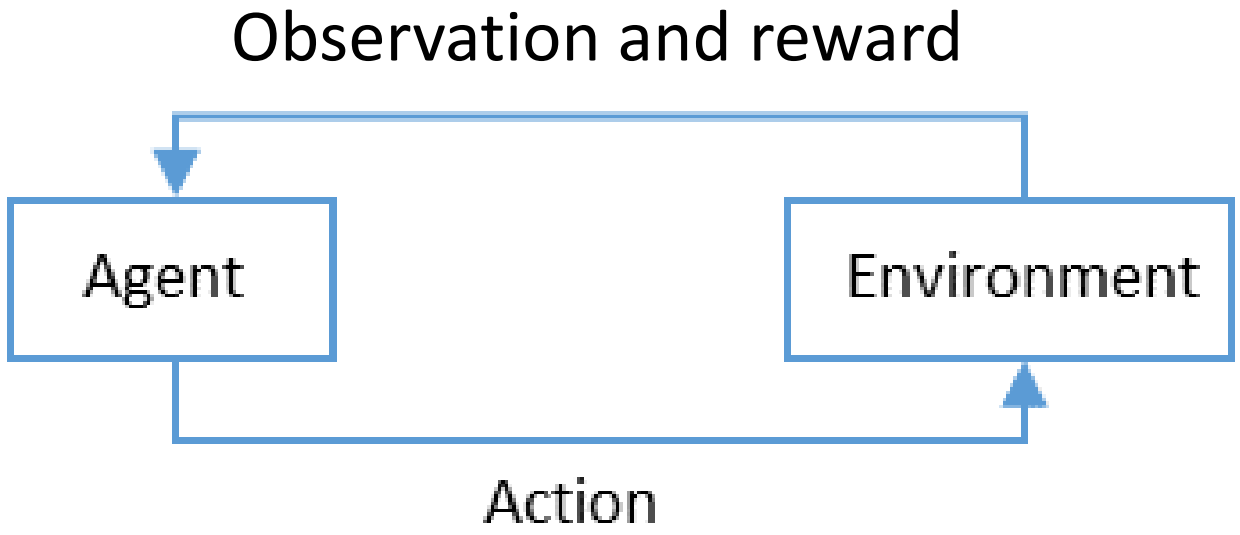
γ is in the range *0…1. For t=0,* γ^t =1. As t increases, γ^t decreases exponentially.

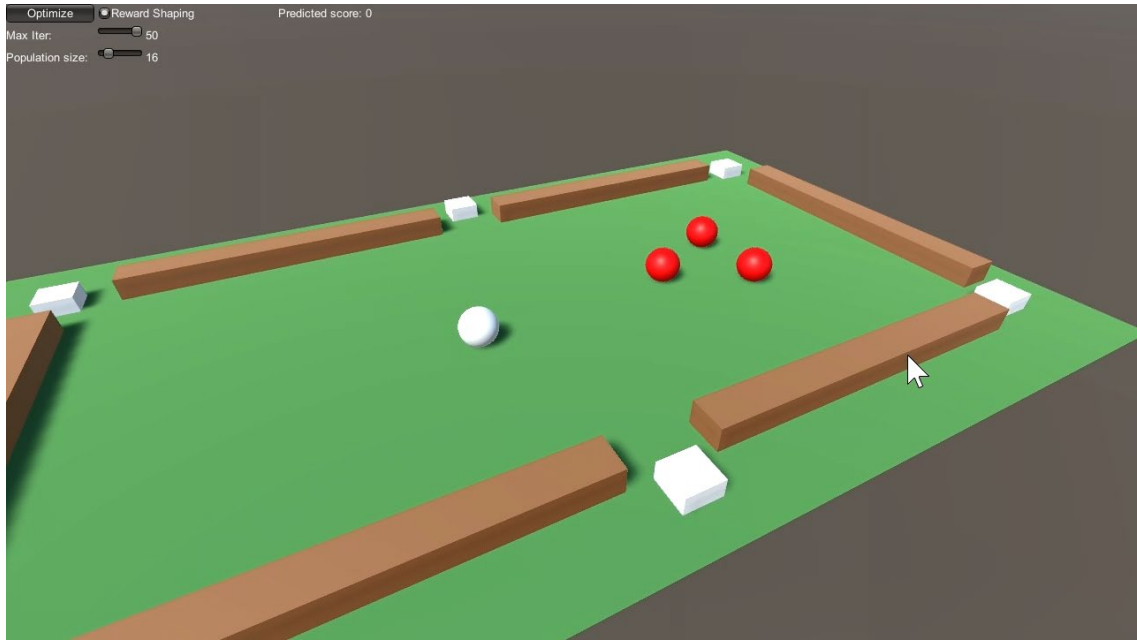
$$\mathbb{E}[\sum_t \gamma^t r(\mathbf{a}_t, \mathbf{o}_t)]$$

Small discount γ: Immediate rewards matter more



If γ=0, DRL only optimizes the immediate reward





In this case, γ^t =0, except at t=0. In the billiards example, DRL would optimize each individual shot without considering the next shots.

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


WARNING
Mathematics
Ahead

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i)\log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i)\log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

Key observation: Both losses are very similar.

(Here, we denote the optimized variables as **a** on the left, instead of **x**)

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i)\log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

More specifically, this DRL formulation is the advantage-based policy gradient algorithm, a.k.a. Advantage Actor Critic (A2C)

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i) \log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

Both: average over a batch of samples

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i)\log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

Both: minimizing the loss maximizes the average

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i)\log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

Left: the θ are the action sampling distribution parameters, e.g., mean and stdev
Right: the θ are the weights of a neural network that outputs the mean and stdev

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i) \log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

DRL: The probability of sampling action $\mathbf{a}_i$ is conditioned on the state $\mathbf{s}_i$ (or a state observation **o**)

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i) \log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

DRL: The advantage of action $\mathbf{a}_i$ depends on state $\mathbf{s}_i$

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i) \log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

For example, the efficiency of a specific billiards shot angle and force depends on the configuration of balls.

DRL as sampling-based gradient optimization

Sampling-based gradient optimization loss function

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} A(\mathbf{a}_i) \log p_\theta(\mathbf{a}_i)$$

DRL training loss function

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

Both: If the advantage for an action $\mathbf{a}_i$ is positive, reducing the loss increases the probability of sampling that action again.

Advantage computation

Sampling-based optimization

$$A(\mathbf{a}) = f(\mathbf{a}) - \mathbb{E}[f(\mathbf{a})]$$

DRL

$$A^{\pi}(\mathbf{a}, \mathbf{s}) = Q^{\pi}(\mathbf{a}, \mathbf{s}) - V^{\pi}(\mathbf{s})$$

The main differences between DRL and generic sampling-based optimization are in the specifics of the advantage computation.

Advantage computation

Sampling-based optimization

$$A(\mathbf{a}) = f(\mathbf{a}) - \mathbb{E}[f(\mathbf{a})]$$

DRL

$$A^{\pi}(\mathbf{a}, \mathbf{s}) = Q^{\pi}(\mathbf{a}, \mathbf{s}) - V^{\pi}(\mathbf{s})$$

Although the formulation differs, they are conceptually the same:
The advantage corresponds to how much better **a** is than average

Advantage computation

Sampling-based optimization

$$A(\mathbf{a}) = f(\mathbf{a}) - \mathbb{E}[f(\mathbf{a})]$$

DRL

$$A^{\pi}(\mathbf{a}, \mathbf{s}) = Q^{\pi}(\mathbf{a}, \mathbf{s}) - V^{\pi}(\mathbf{s})$$

Value function: How good the current policy π is for state **s**, on average.

Advantage computation

Sampling-based optimization

$$A(\mathbf{a}) = f(\mathbf{a}) - \mathbb{E}[f(\mathbf{a})]$$

DRL

$$A^{\pi}(\mathbf{a}, \mathbf{s}) = Q^{\pi}(\mathbf{a}, \mathbf{s}) - V^{\pi}(\mathbf{s})$$

$$V^{\pi}(\mathbf{s}) = \mathbb{E}_{\mathbf{s}_0=\mathbf{s}, \mathbf{a}_t \sim \pi(\mathbf{a}_t|\mathbf{s}_t)}[\sum_t \gamma^t r(\mathbf{a}_t, \mathbf{s}_t)]$$

More specifically, $V^{\pi}(\mathbf{s})$ equals the expected sum of future rewards if starting from state **s** and sampling actions from the current policy π.

Advantage computation

Sampling-based optimization

$$A(\mathbf{a}) = f(\mathbf{a}) - \mathbb{E}[f(\mathbf{a})]$$

DRL

$$A^{\pi}(\mathbf{a}, \mathbf{s}) = Q^{\pi}(\mathbf{a}, \mathbf{s}) - V^{\pi}(\mathbf{s})$$

$$Q^{\pi}(\mathbf{a}, \mathbf{s}) = r(\mathbf{a}, \mathbf{s}) + \gamma V^{\pi}(\mathbf{s}')$$

Action-Value function: Expected sum of future rewards if starting from state **s**, first taking action **a**, and then continuing on the current policy π from the next state **s'**

Advantage computation

Sampling-based optimization

$$A(\mathbf{a}) = f(\mathbf{a}) - \mathbb{E}[f(\mathbf{a})]$$

DRL

$$A^{\pi}(\mathbf{a}, \mathbf{s}) = Q^{\pi}(\mathbf{a}, \mathbf{s}) - V^{\pi}(\mathbf{s})$$

$$Q^{\pi}(\mathbf{a}, \mathbf{s}) = r(\mathbf{a}, \mathbf{s}) + \gamma V^{\pi}(\mathbf{s}')$$

DRL advantage $A^{\pi}(\mathbf{a},\mathbf{s})$ = how much improvement would we get if we always take action **a** in state **s**, but otherwise don't change the policy.

Evaluating V^{π} in practical implementations

Sampling-based optimization

$$A(\mathbf{a}) = f(\mathbf{a}) - \mathbb{E}[f(\mathbf{a})]$$

DRL

$$A^{\pi}(\mathbf{a}, \mathbf{s}) = Q^{\pi}(\mathbf{a}, \mathbf{s}) - V^{\pi}(\mathbf{s})$$

$$Q^{\pi}(\mathbf{a}, \mathbf{s}) = r(\mathbf{a}, \mathbf{s}) + \gamma V^{\pi}(\mathbf{s}')$$

A value predictor network (a *critic*) is trained to approximate $V^{\pi}(\mathbf{s})$. Simply plugging the predictions to the equations is unstable, however, and some extra tricks are needed. See, e.g., https://arxiv.org/abs/1506.02438

Advantage computation: The case of $\gamma = 0$

Sampling-based optimization

$$A(\mathbf{a}) = f(\mathbf{a}) - \mathbb{E}[f(\mathbf{a})]$$

DRL

$$A^{\pi}(\mathbf{a}, \mathbf{s}) = Q^{\pi}(\mathbf{a}, \mathbf{s}) - V^{\pi}(\mathbf{s})$$

$$Q^{\pi}(\mathbf{a}, \mathbf{s}) = r(\mathbf{a}, \mathbf{s}) + \gamma V^{\pi}(\mathbf{s}')$$

0

$$V^{\pi}(\mathbf{s}) = \mathbb{E}_{\mathbf{s}_0 = \mathbf{s}, \mathbf{a}_t \sim \pi(\mathbf{a}_t | \mathbf{s}_t)}[\sum_t \gamma^t r(\mathbf{a}_t, \mathbf{s}_t)]$$

0 for t>0

Advantage computation: The case of $\gamma = 0$

Sampling-based optimization

$$A(\mathbf{a}) = f(\mathbf{a}) - \mathbb{E}[f(\mathbf{a})]$$

DRL

$$A^{\pi}(\mathbf{a}, \mathbf{s}) = r(\mathbf{a}, \mathbf{s}) - \mathbb{E}[r(\mathbf{a}, \mathbf{s})]$$

DRL advantages simplify and become the same as in generic sampling-based optimization. The expectations can be approximated by averages over samples.

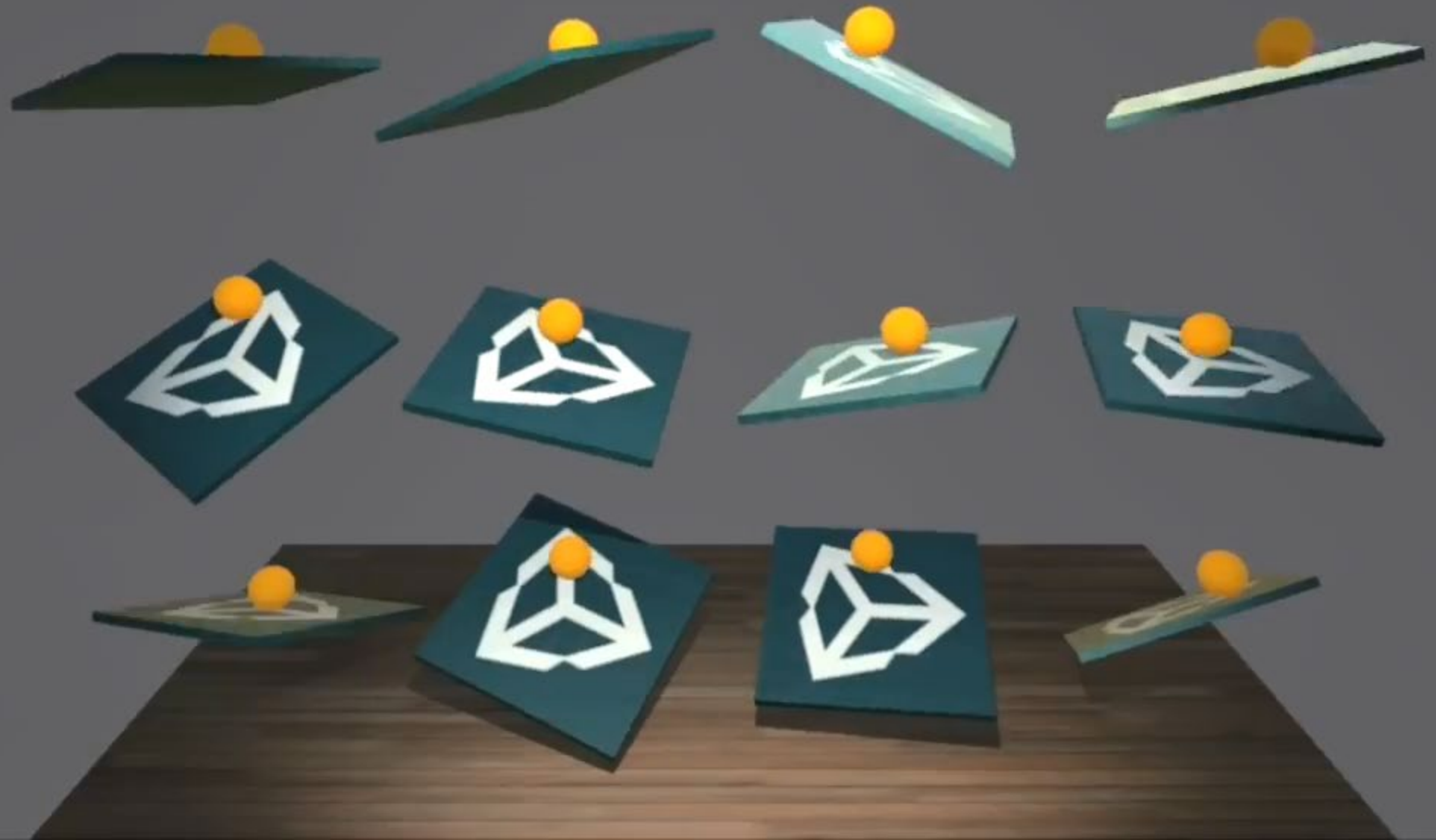
In DRL, the averaging requires that one samples multiple **a** for each **s**, which can be costly. On the other hand, no value predictor network is needed.

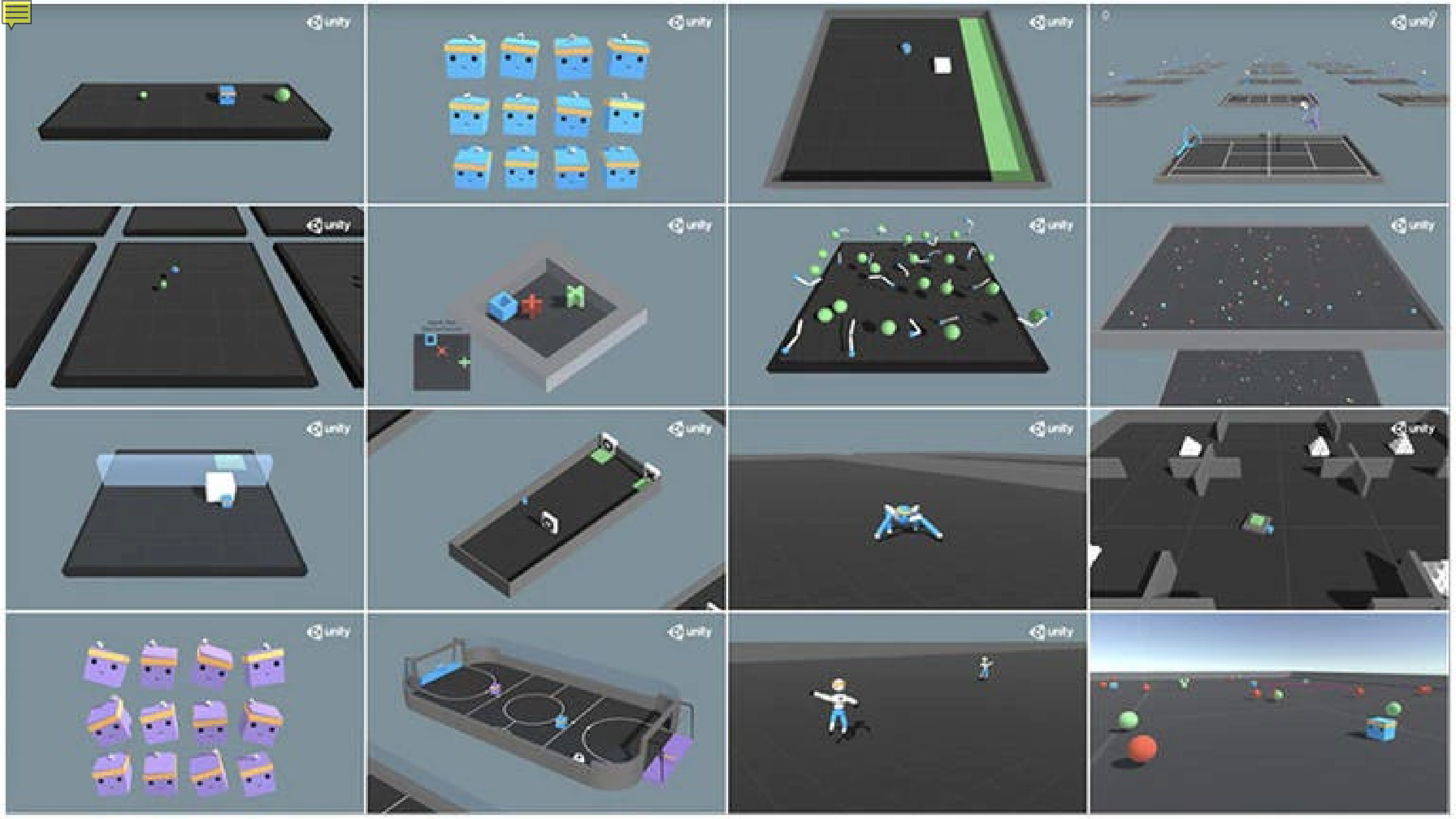

WA NG
M ematics
Ahead

Common DRL algorithms

- Proximal Policy Optimization (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 (SAC, Haarnoja et al. 2018): Originally only for continuous actions, but has been extended to discrete. Can be more efficient than PPO, but is mathematically much more complex and can be hard to finetune.
- Implementations are available in common DRL libraries:
  - https://github.com/DLR-RM/stable-baselines3 (Python)
  - https://github.com/fabiopardo/tonic (Python)
  - https://unity.com/products/machine-learning-agents (Unity 3D game engine)

Unity Machine Learning Agents, PPO





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

(Algorithm used: PPO)

SEPTEMBER 17, 2019 • 9 MINUTE READ

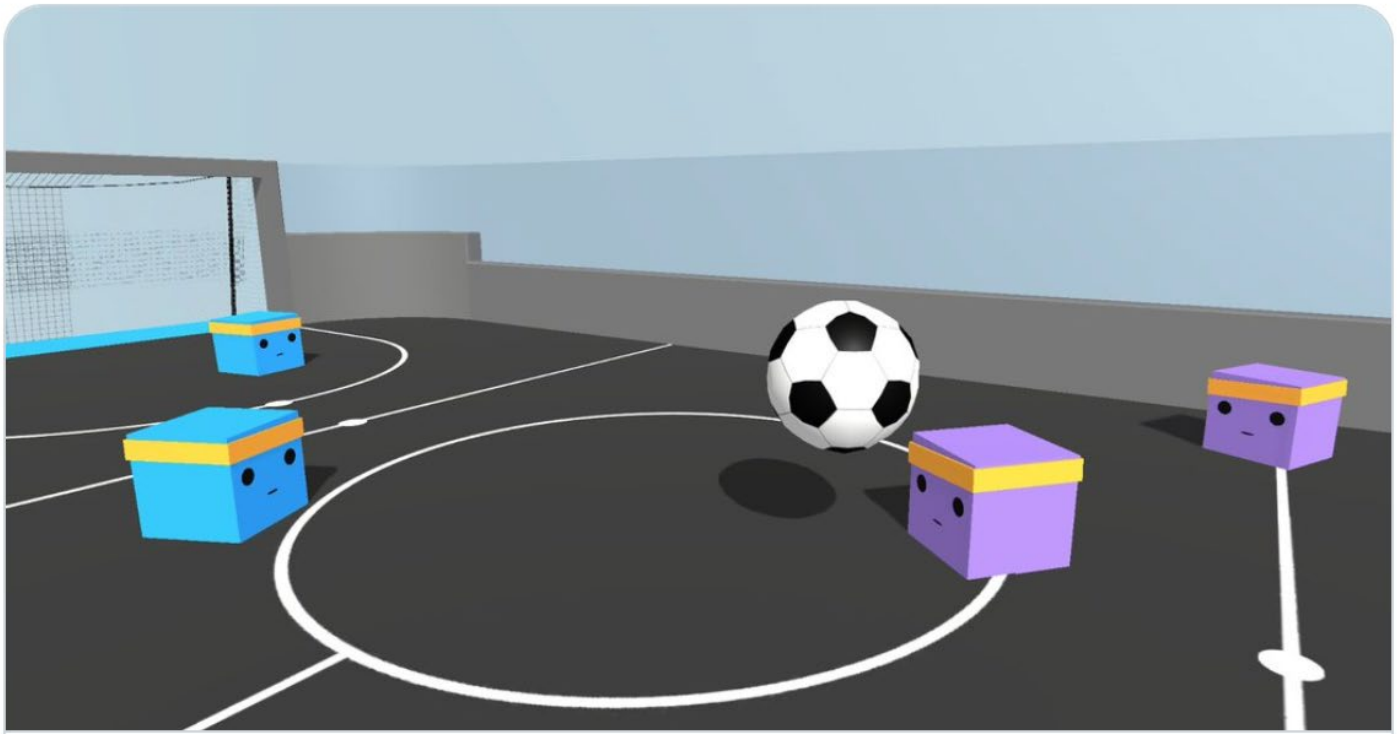
Emergent Tool Use from Multi-Agent Interaction

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...

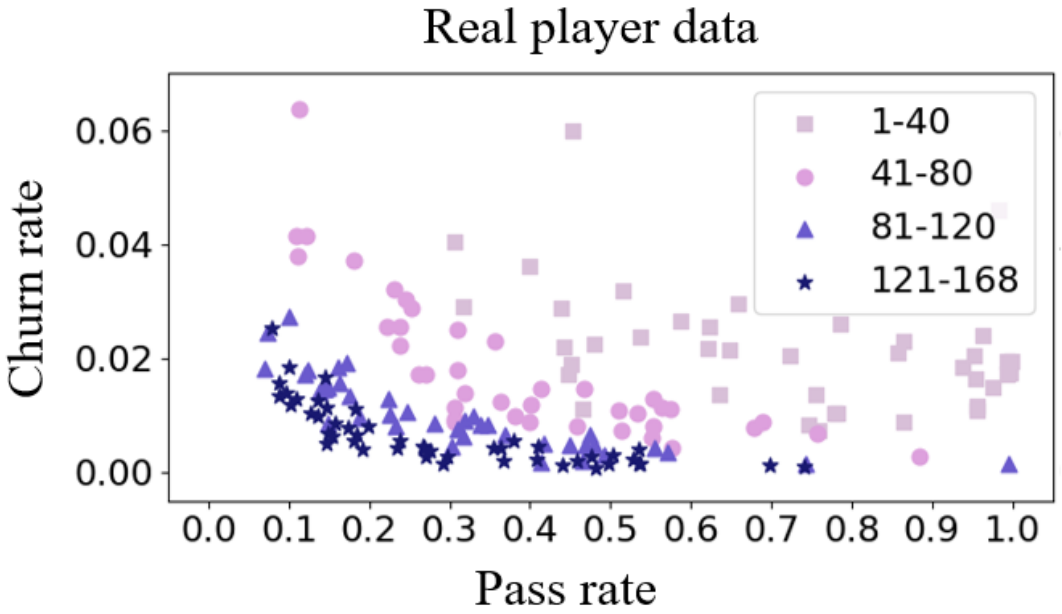
In the latest release of the ML-Agents Toolkit (v0.14), we have added a self-play feature that provides the capability to train competitive agents...

blogs.unity3d.com

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

Predicting player behavior and experience using DRL agents

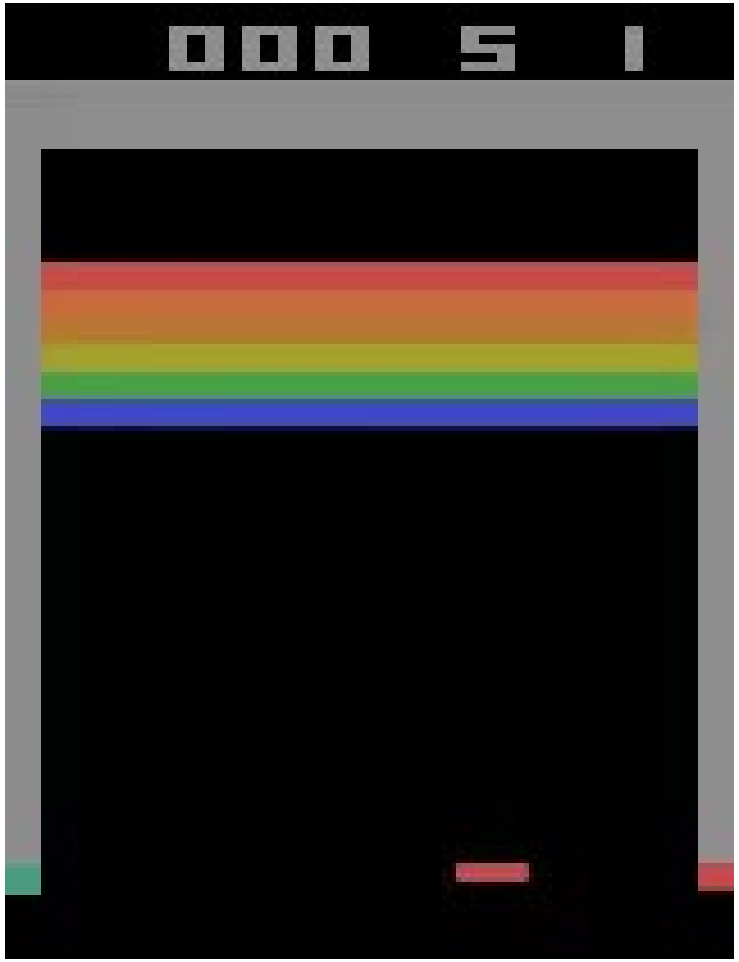
- An Aalto-Rovio collaboration: https://twitter.com/perttu_h/status/1302536834002628608?s=20
- We use DRL agents to predict pass rate (i.e., level difficulty, a crucial part of game experience) and churn rate (behavioral metric, how many players quit at each level)

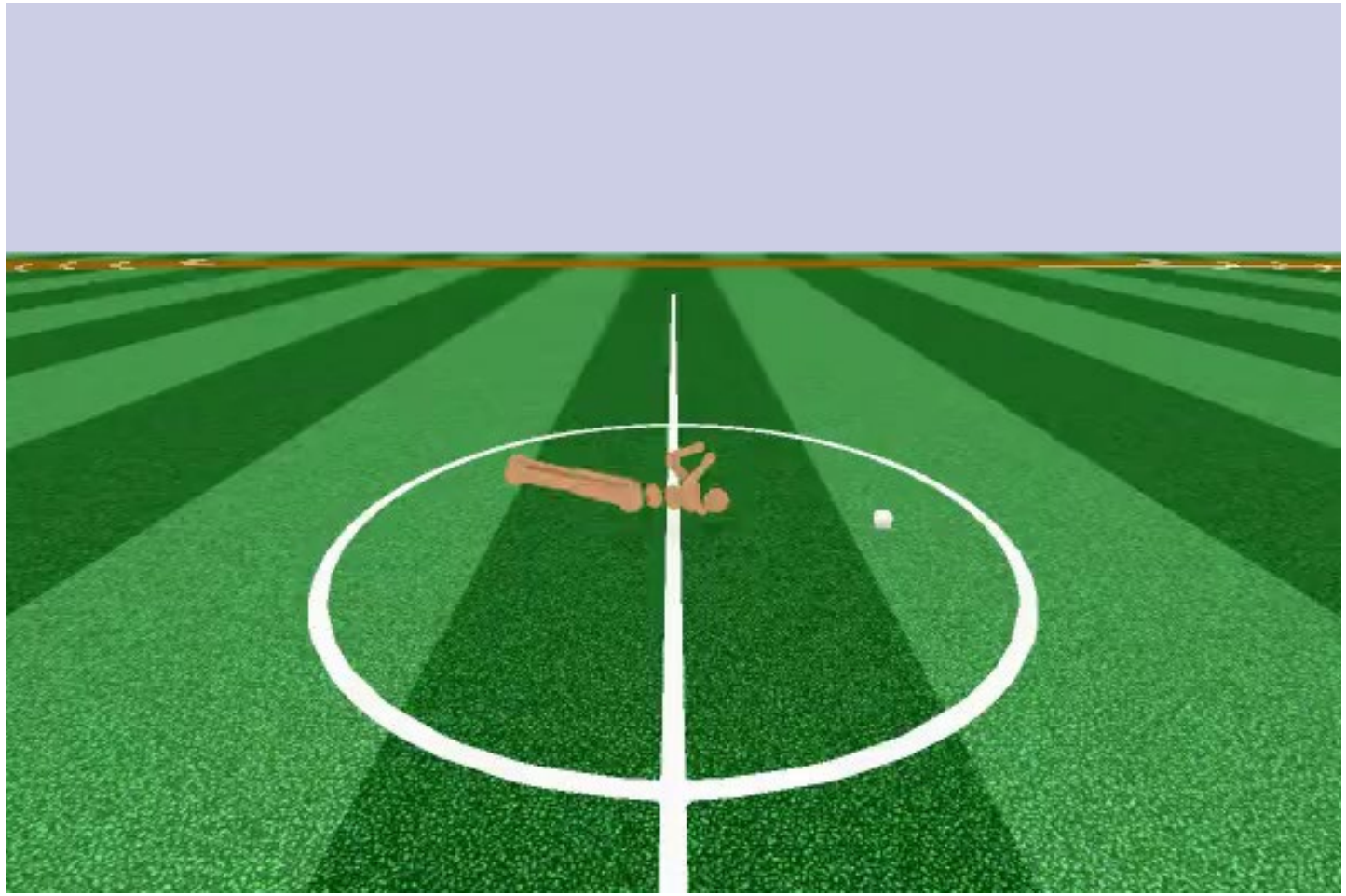


DRL outside game development: OpenAI Gym

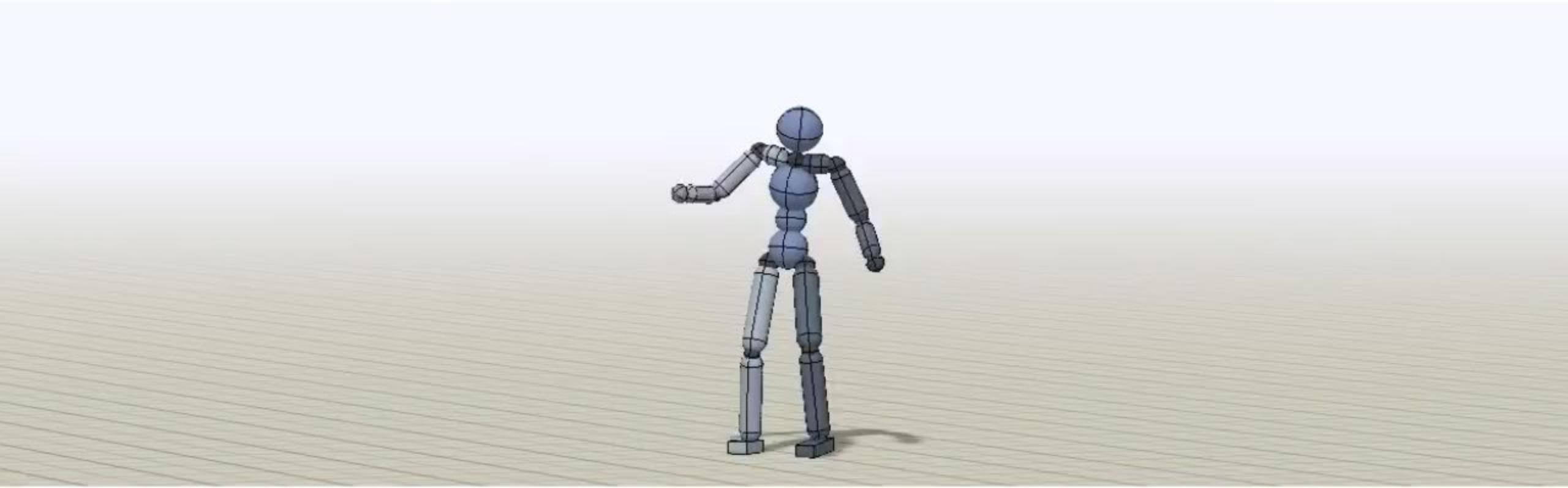
```
import gym
env = gym.make("CartPole-v1")
observation = env.reset()
for _ in range(1000):
  env.render()
  action = env.action_space.sample() # your agent here (this takes random actions)
  observation, reward, done, info = env.step(action)

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

DRL actions are inherently noisy





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



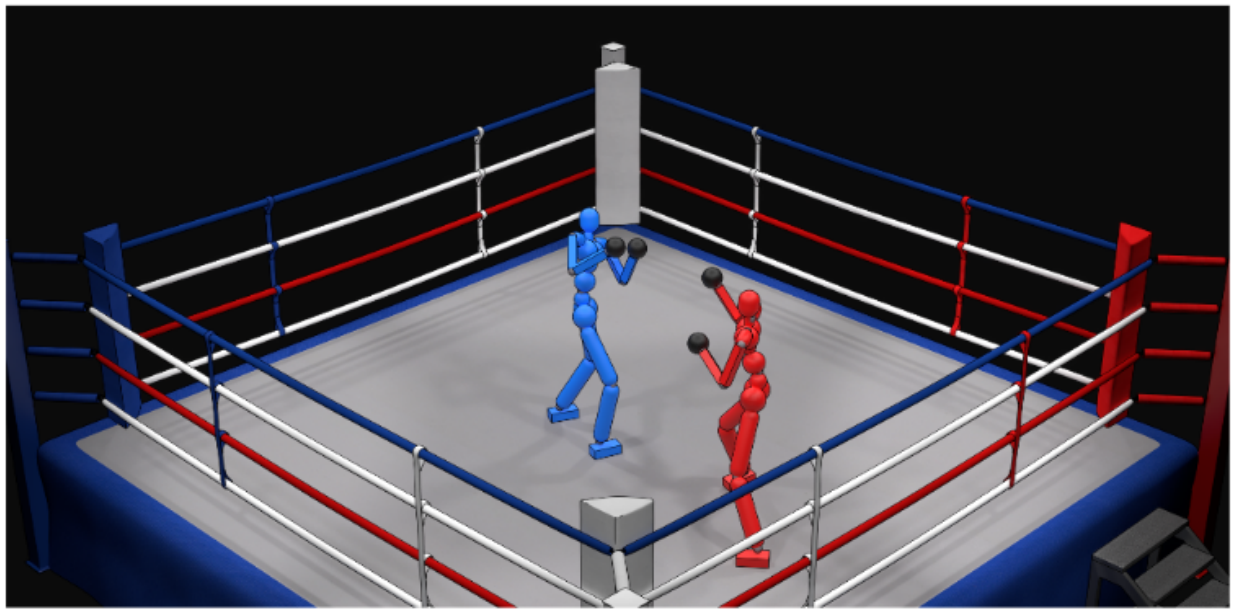
Xue Bin Peng[1], Pieter Abbeel[1], Sergey Levine[1], Michiel van de Panne[2]

[1] University of California Berkeley

[2] University of British Columbia

Control Strategies for Physically Simulated Characters Performing Two-player Competitive Sports

JUNGDAM WON, Facebook AI Research
DEEPAK GOPINATH, Facebook AI Research
JESSICA HODGINS, Facebook AI Research



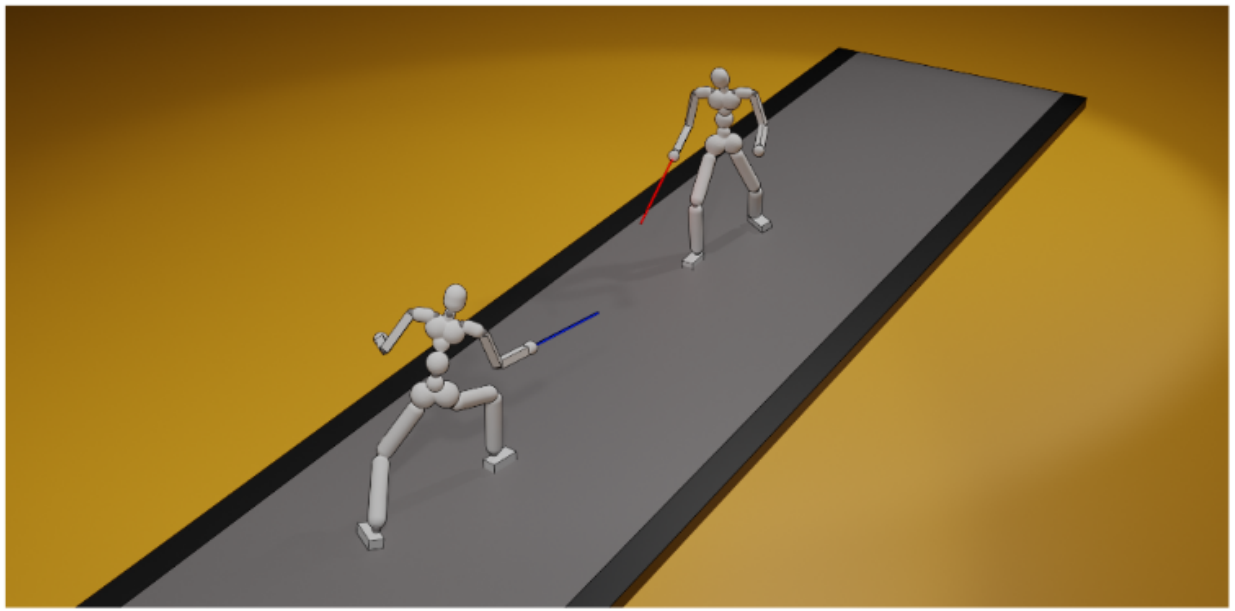


Fig. 1. Characters performing two-player competitive sports such as boxing (left) and fencing (right) using learned control strategies.

Current research topics in RL and policy learning

- Learning high-quality movement without reference data
- Model-based RL: Constructing predictive world dynamics models to allow imagination-based planning: https://github.com/danijar/dreamerv2
- (Self-)supervised learning as an alternative to RL: https://montreal.ubisoft.com/en/supertrack-motion-tracking-for-physically-simulated-characters-using-supervised-learning/, https://bair.berkeley.edu/blog/2020/10/13/supervised-rl/
- Offline RL: Learning policies based on arbitrary datasets, without constantly collecting new data https://arxiv.org/abs/2005.01643
- Utilizing the general world knowledge of large models like CLIP and GPT-3: https://wenlong.page/language-planner/, https://cliport.github.io/
- Accelerating learning through massively parallel simulation on GPU: https://github.com/google/brax, https://developer.nvidia.com/isaac-sim

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 the MDP or PO-MDP
  - Define how simulation state is mapped to a fixed length observation vector
  - Define how the episode initial state is sampled (in OpenAI gym, this happens when the DRL algorithm 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 the 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 teaching humans…)
- Test training with some DRL algorithm such as PPO or SAC

My agent does not learn or learns slowly. What to do?

- Make sure you exploit your CPU and GPU to the max. Check your CPU and GPU utilization during training. Are you using all CPU cores?
- In Unity, don’t train in the editor. Train using standalone builds.
- In Unity, speed up the game as much as possible during the training (Set Time.timeScale to the maximum value)
- Train with multiple threads or processes. In Unity, see: https://blog.unity.com/technology/training-your-agents-7-times-faster-with-ml-agents
- Check for other common problems listed in the following slides

Typical problems: Normalization

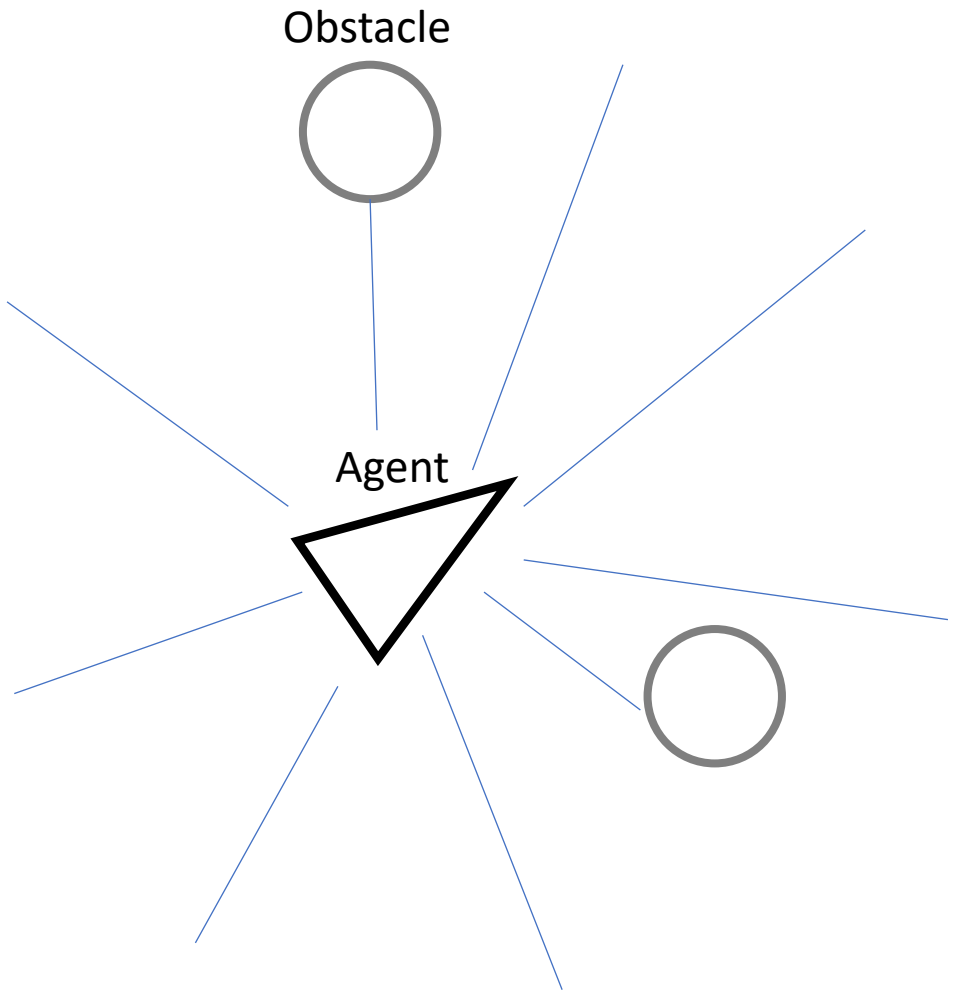
- Make sure that the reward function is in range 0…1 or close to it
- Make sure the observations are in range -1…1 or have mean 0 and standard deviation 1
- Reason: Neural network optimization is highly sensitive to initialization. The DRL networks are typically initialized assuming this kind of normalized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

Typical problems: Incomplete observations

- The next state/observation and the reward should be predictable based on only the current observation. Otherwise, the problem is not an MDP and learning will be slow or impossible
- For instance, if training a spaceship AI, imagine yourself in the cockpit with no windows and all you can see is the observation vector. Does it provide enough information for you to act optimally?

Typical problems: Incomplete observations

- Example: Raycast sensing in a game can be more efficient than training the agent simply based on the rendered game pixels
- Important: The observation vector should contain the hit distances for each ray, and possibly the velocities of the hit objects
- Simply observing which rays hit an obstacle is not enough for deciding how to avoid them



Typical problems: Early termination and negative rewards

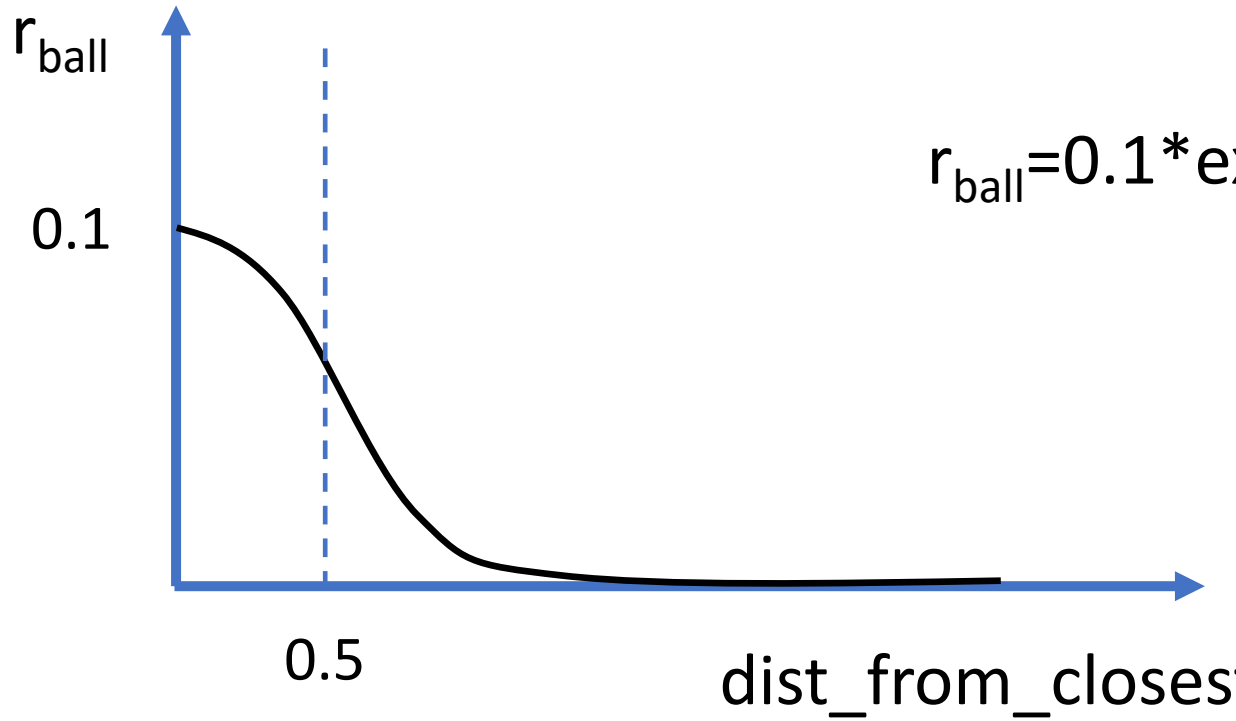
- Episode termination when the agent fails (e.g., walking agent falls down) can reduce gradient noise and speed up training, compared to running every episode until a timeout
- However, if the agent can receive negative rewards, it may optimize by failing as fast as possible to avoid the negative rewards
- 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 (makes it 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, use a smaller discount factor to reduce the noise in summing together the future rewards. Try reward shaping (next slide)
- 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)

Reward shaping in billiards

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



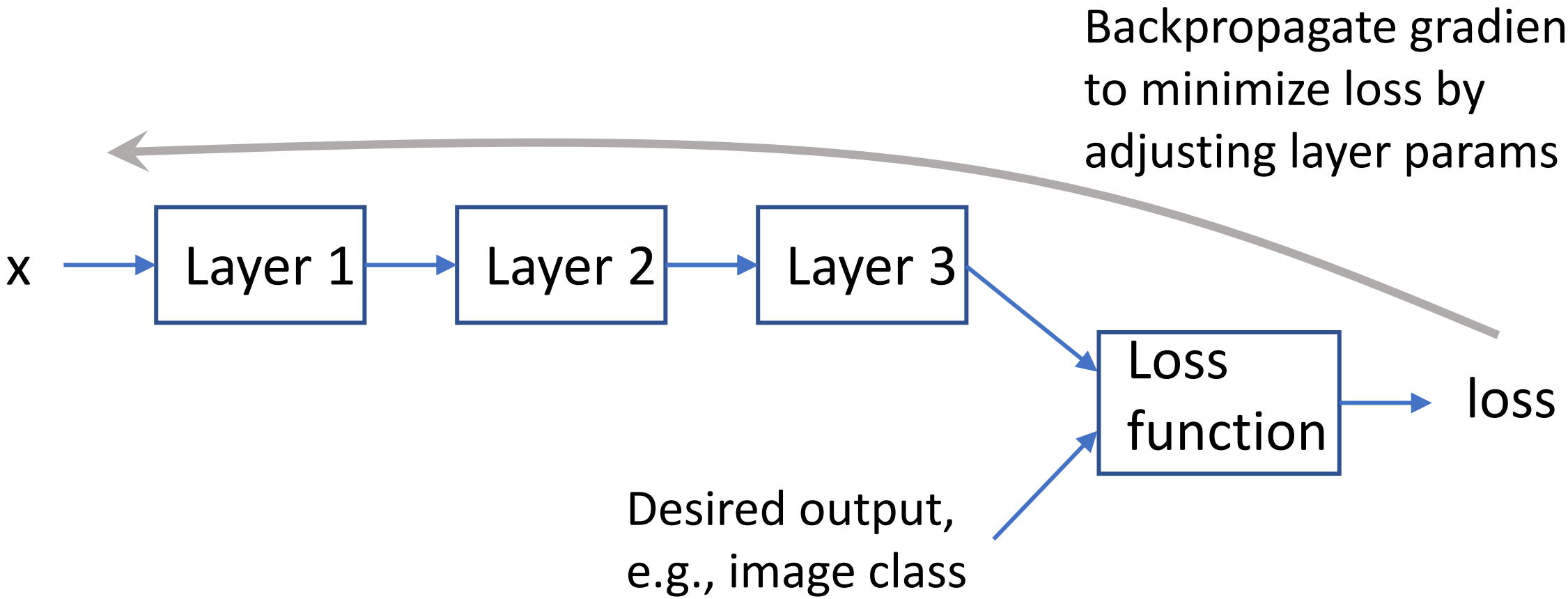
$$r_{ball}=0.1*\exp(-0.5*dist\_from\_closest^2/0.5^2)$$

Recap: Optimization theory

- Optimization: Hill climbing or descent on the optimization landscape defined by $f(\mathbf{x})$
- Gradient –based optimization: Follow the gradient (direction of steepest ascent/descend on the landscape), with momentum to avoid getting stuck at the bottom of valleys or top of ridges (e.g., Adam). Tensorflow and Pytorch allow computing gradients automatically.
- Sampling –based optimization: The $f(\mathbf{x})$ doesn't need to be directly differentiable to estimate gradients based on samples. A Gaussian search distribution = Gaussian smoothing of $f(\mathbf{x})$ to make optimization easier. Practical methods: CMA-ES, Adam with sampling-based gradient
- Discrete optimization: Sampling-based optimization still works.
- DRL: Optimizing actions conditional on observed state. What's needed: formulate your problem as a (PO-)MDP, i.e., define the actions, observations, rewards, and initial state distribution

Recap: Optimization applications

- Machine learning: Optimize the weights and biases of a neural network to minimize some error metric



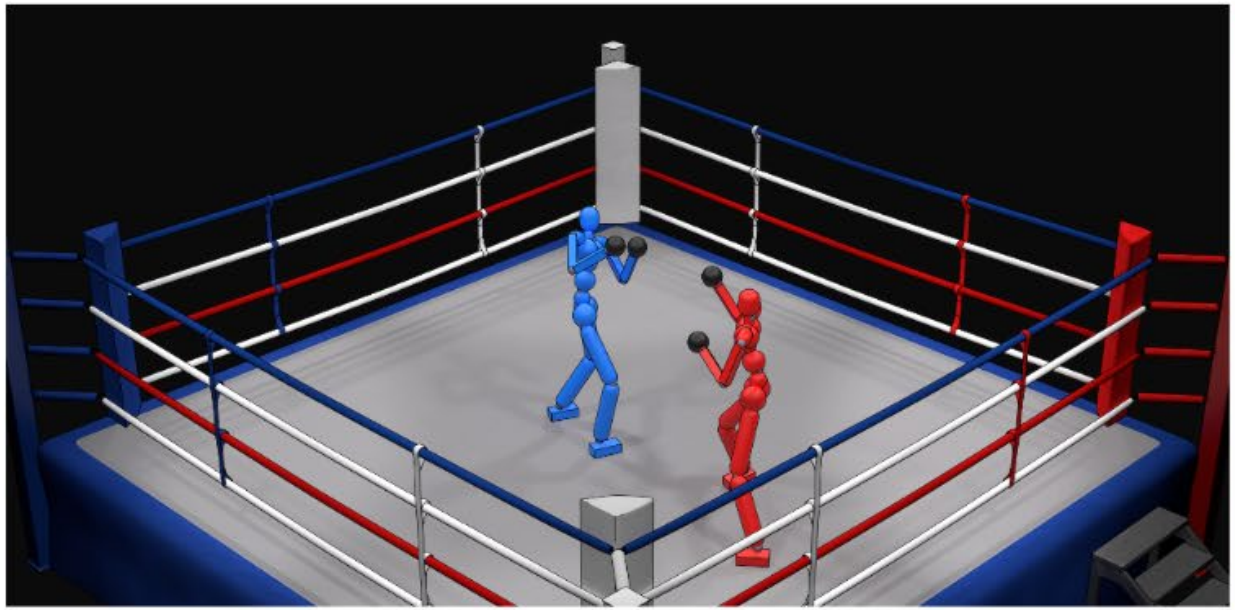
Recap: Optimization applications

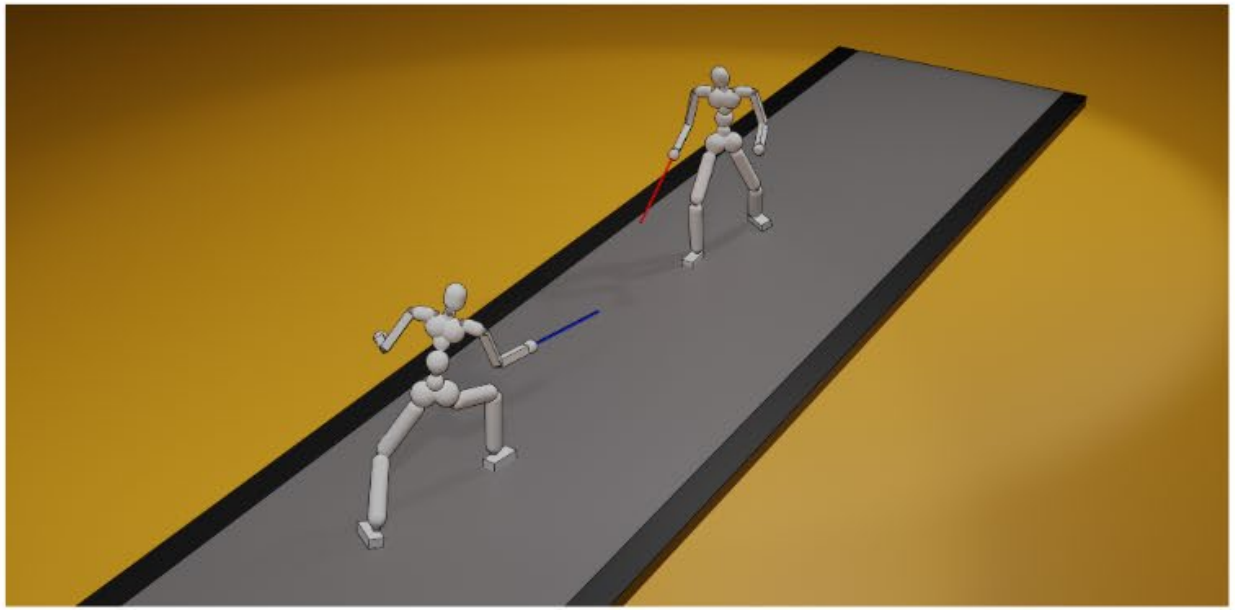
- Image synthesis: Optimize a drawing to fool a neural network thinking it's some object

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

Recap: Optimization applications

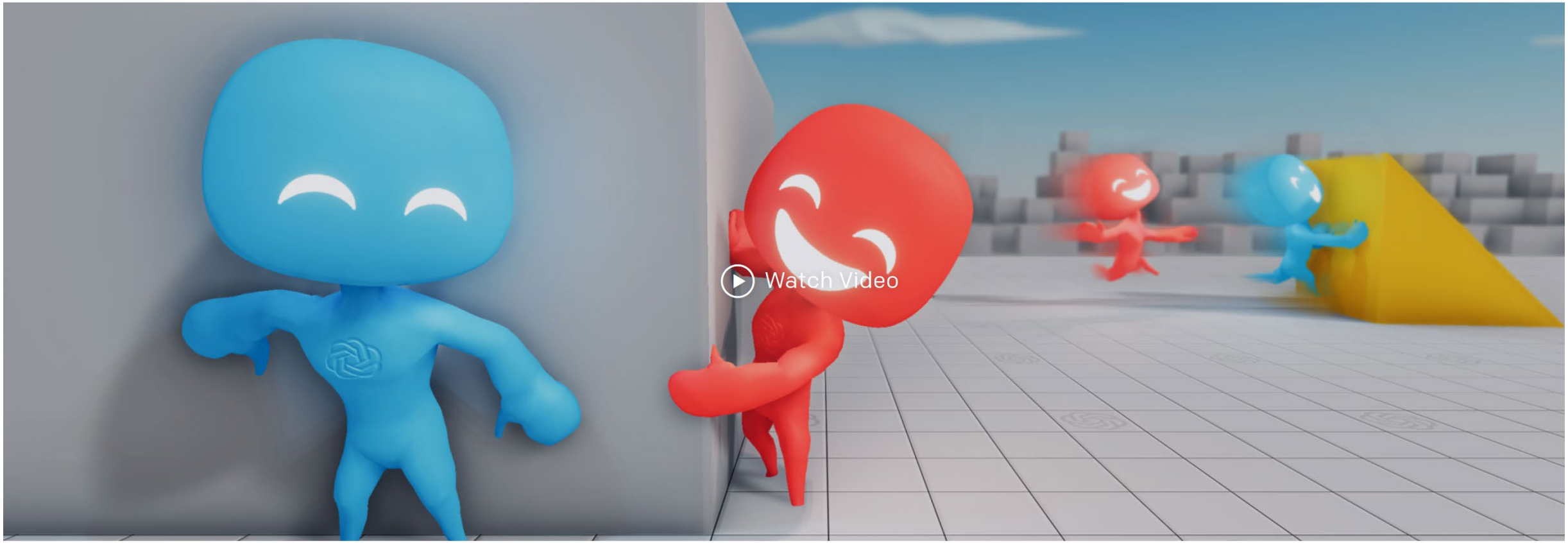
- Animation: Optimize a simulated humanoid's muscle activations to make it achieve a goal like walking or running





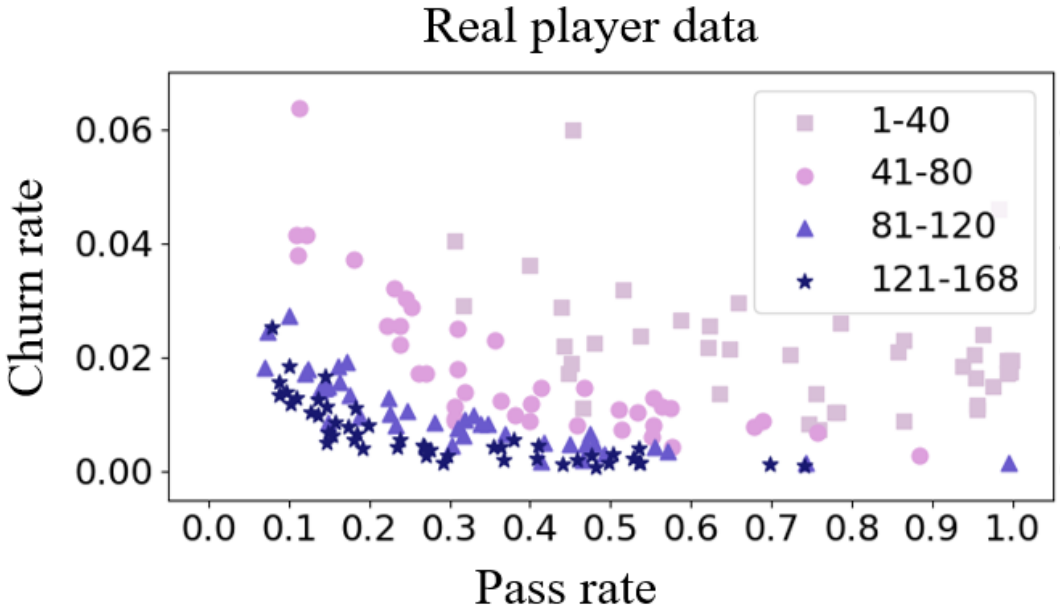
Recap: Optimization applications

- Game AI, artificial life: Optimize the actions of agents to make them achieve goals like performing billiards trick shot, winning a game of Go, or the hiders avoiding the seekers in a game of hide-and-seek.



Recap: Optimization applications

- Game content: Train AI agents for automatic playtesting. Optimize procedural generation parameters based on some model of player behavior or experience.



Fundamentally, all AI & ML is optimization

Optional old content follows, might be still useful to some

Forward search methods

Selected mode:Getting up

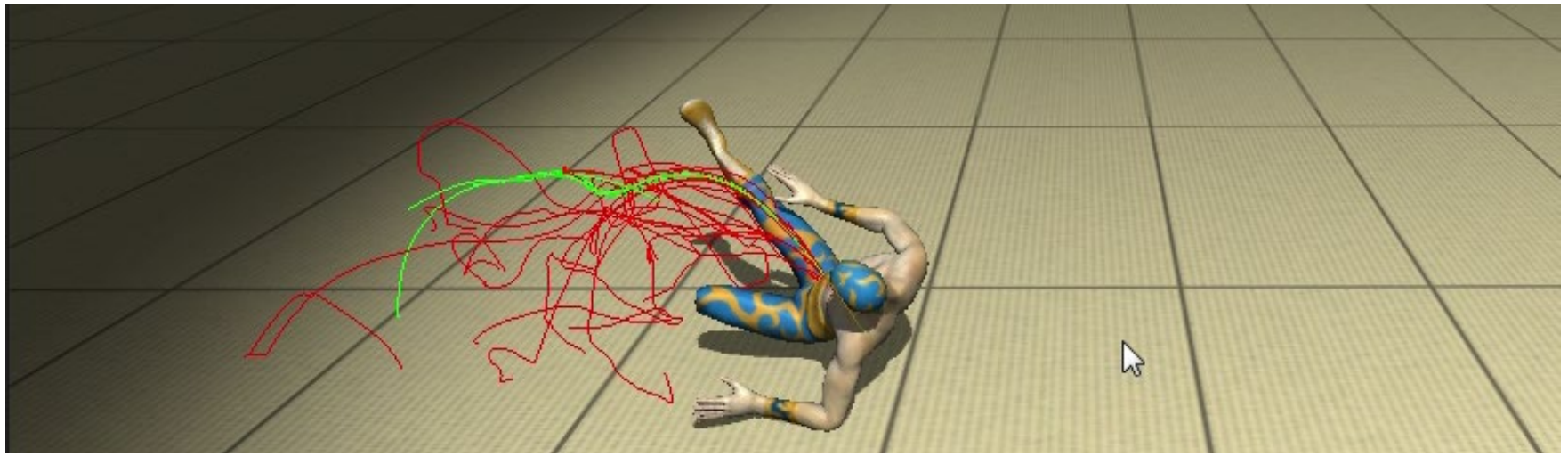
Online Motion Synthesis using Sequential Monte Carlo

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

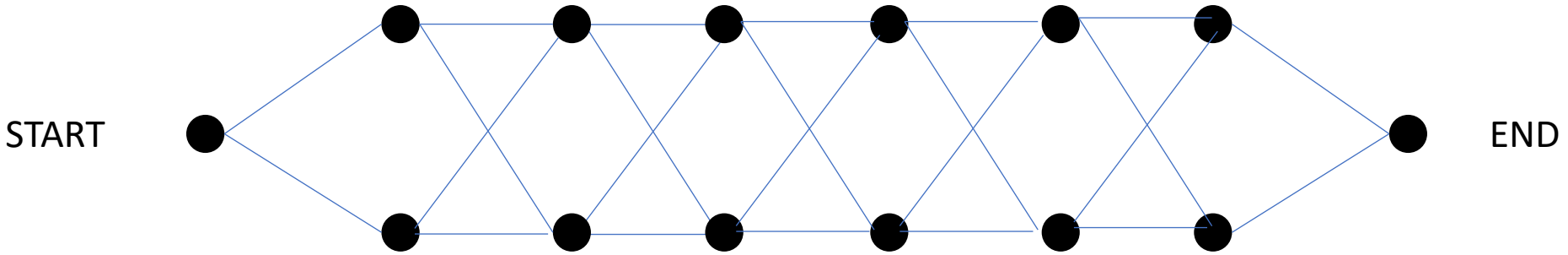
Deep RL:
+ Once trained, very computationally efficient
- Training can be very slow and may not converge

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

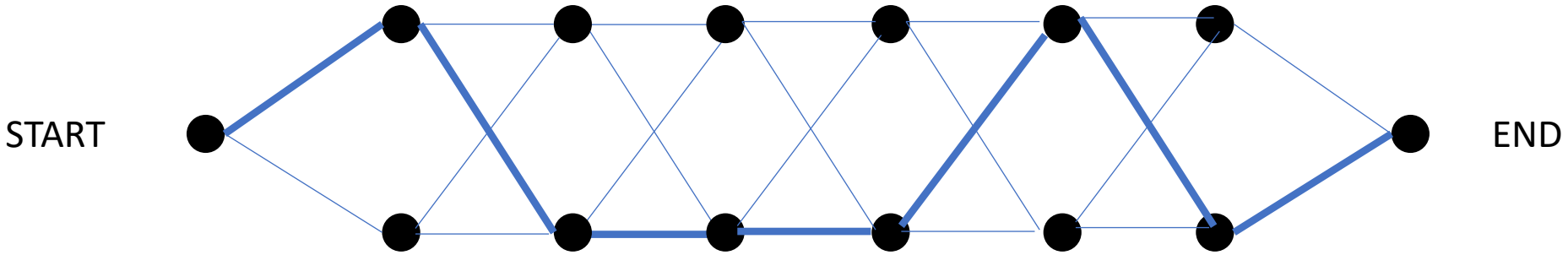
Combining search & neural networks can allow adjusting the tradeoffs.

Why is searching for action sequences hard?

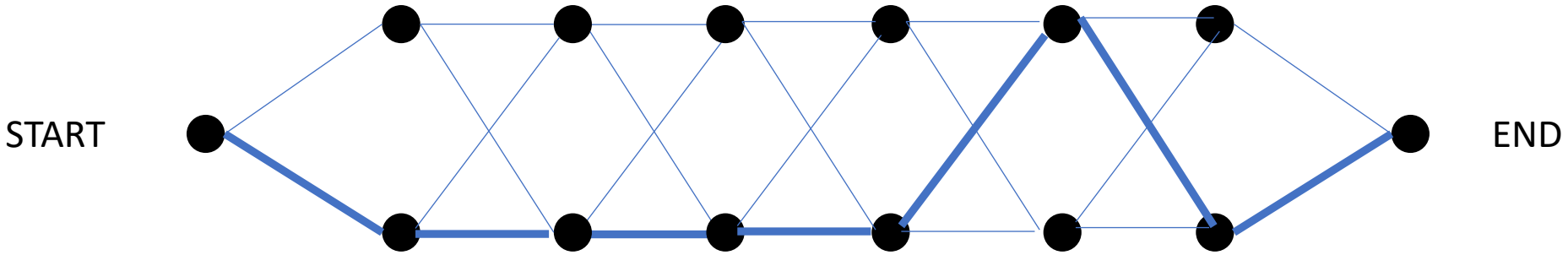
Curse of dimensionality



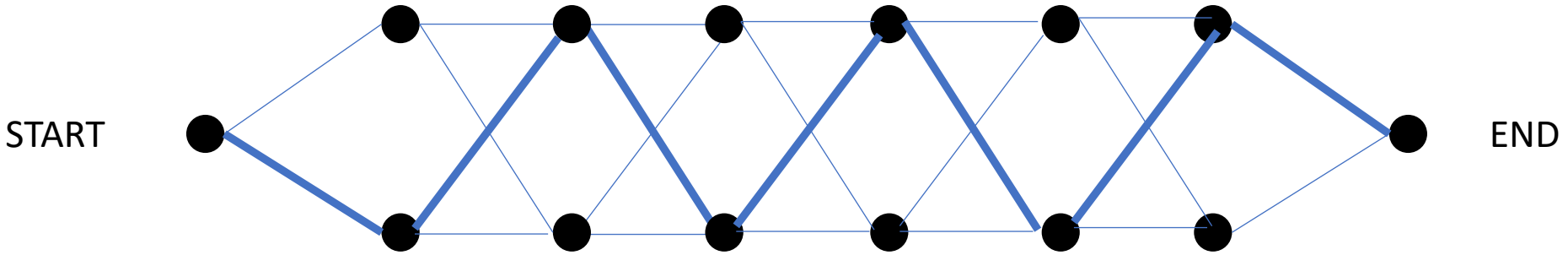
Curse of dimensionality



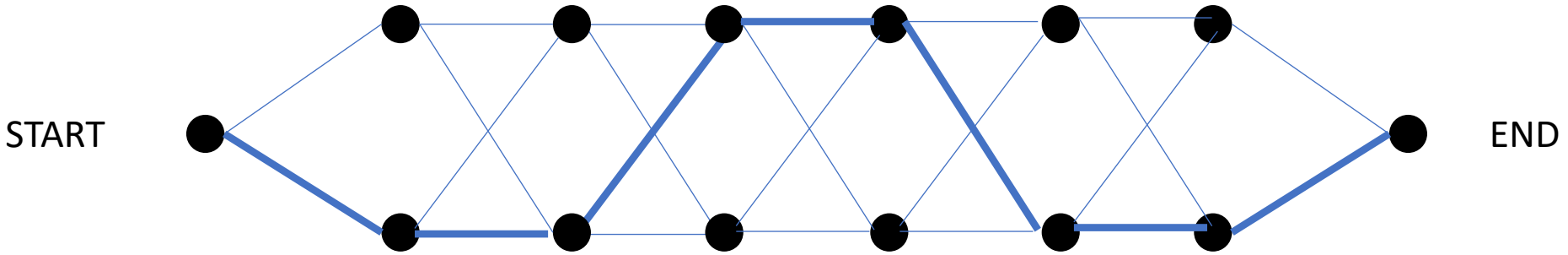
Curse of dimensionality



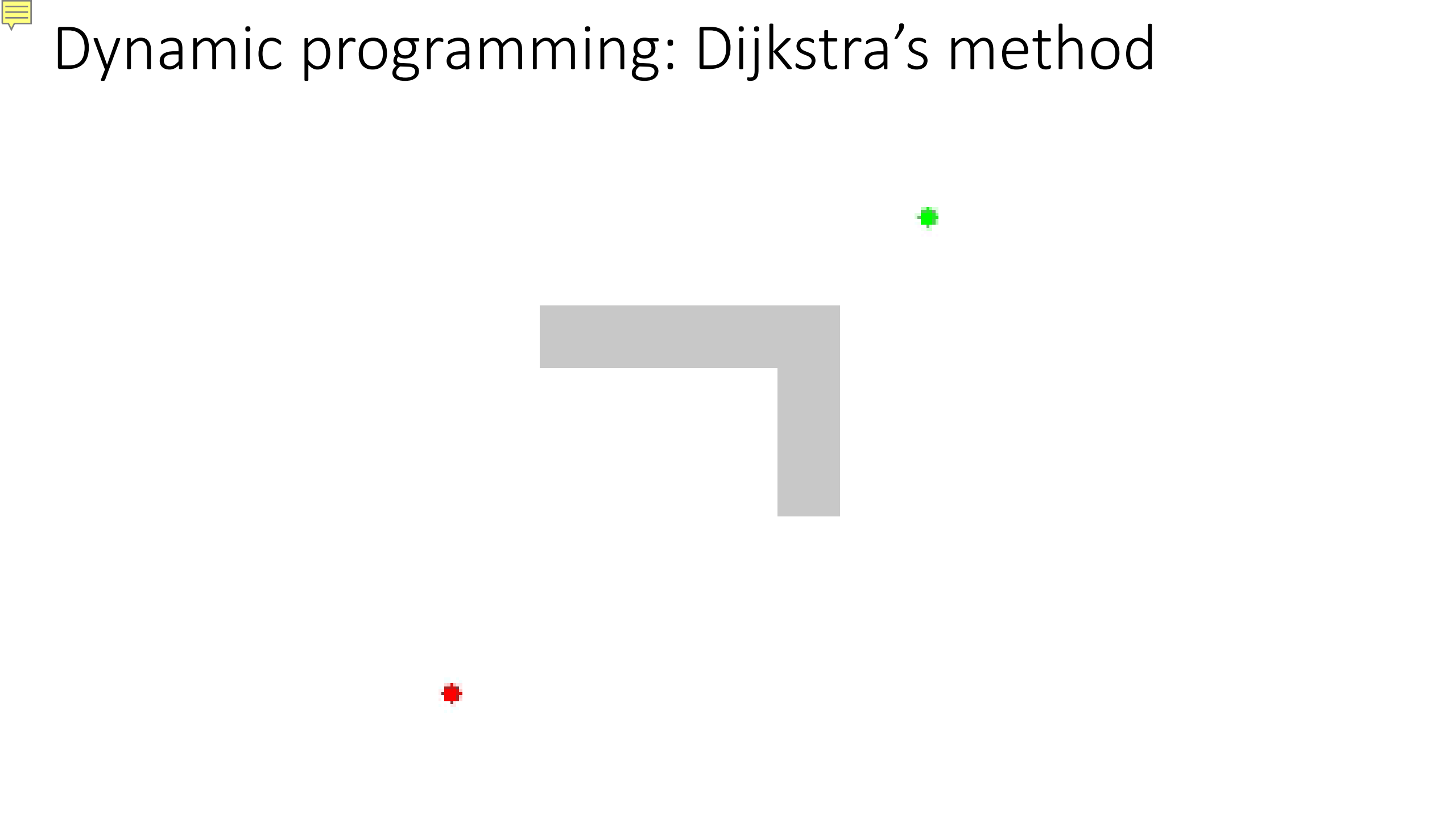
Curse of dimensionality



Curse of dimensionality



Dynamic programming: Dijkstra’s method

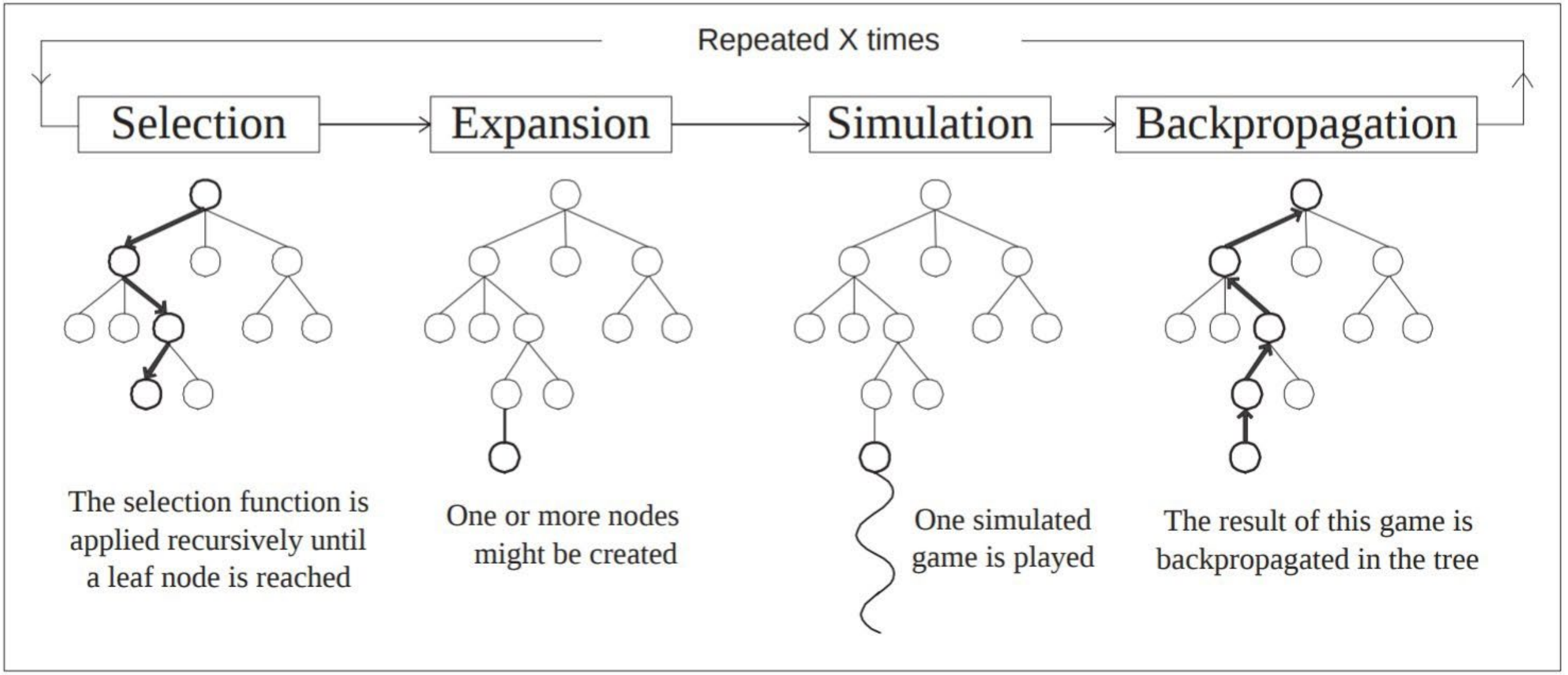


From Dijkstra to A*

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-

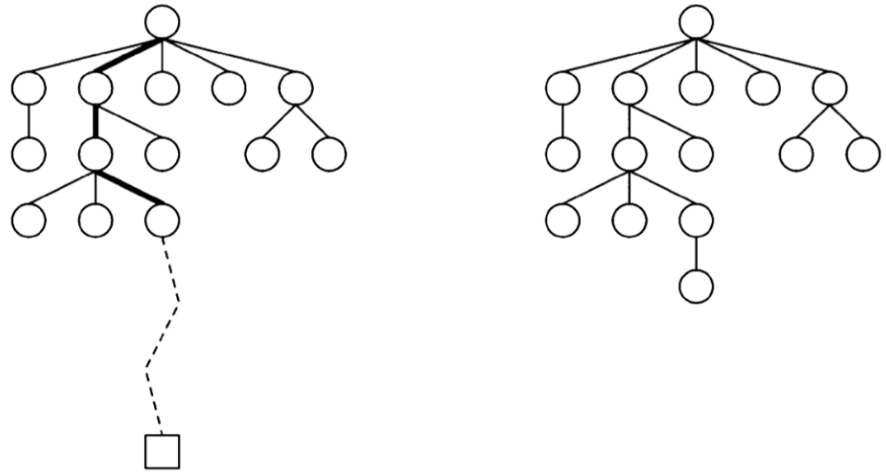


Fig. 1. The basic MCTS process [17].

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

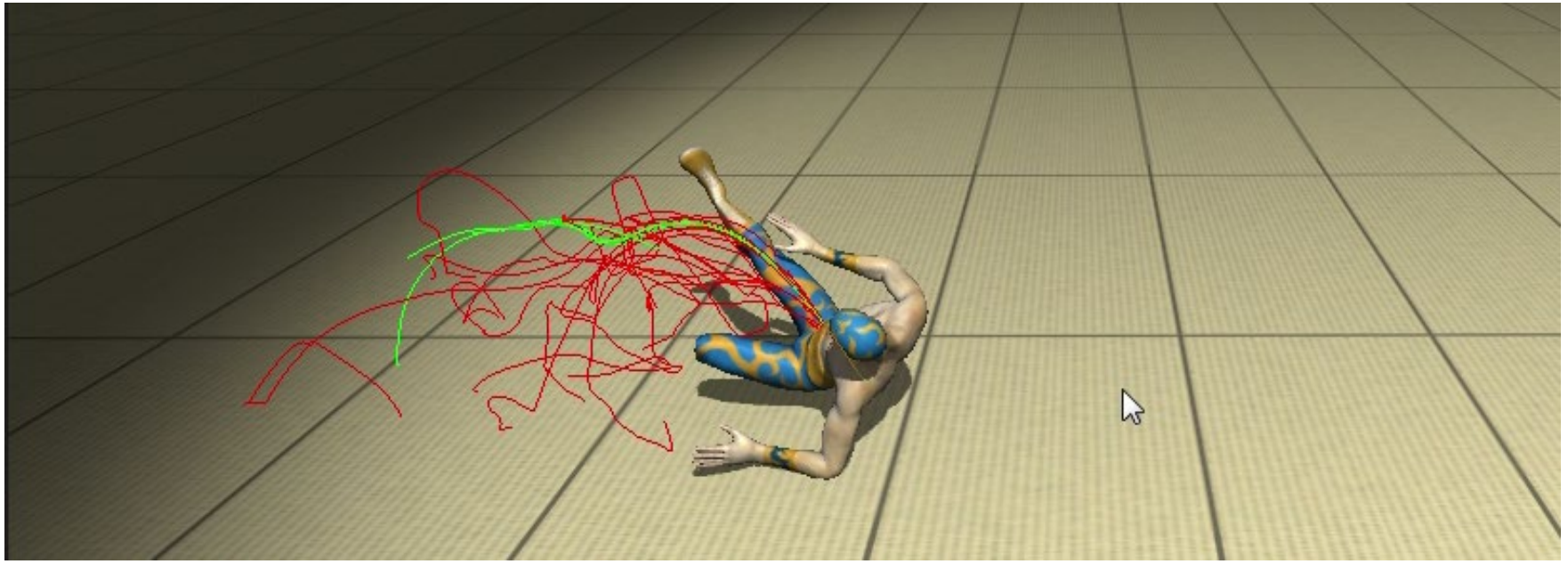
MCTS as a Deep Reinforcement Learning alternative

- 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.

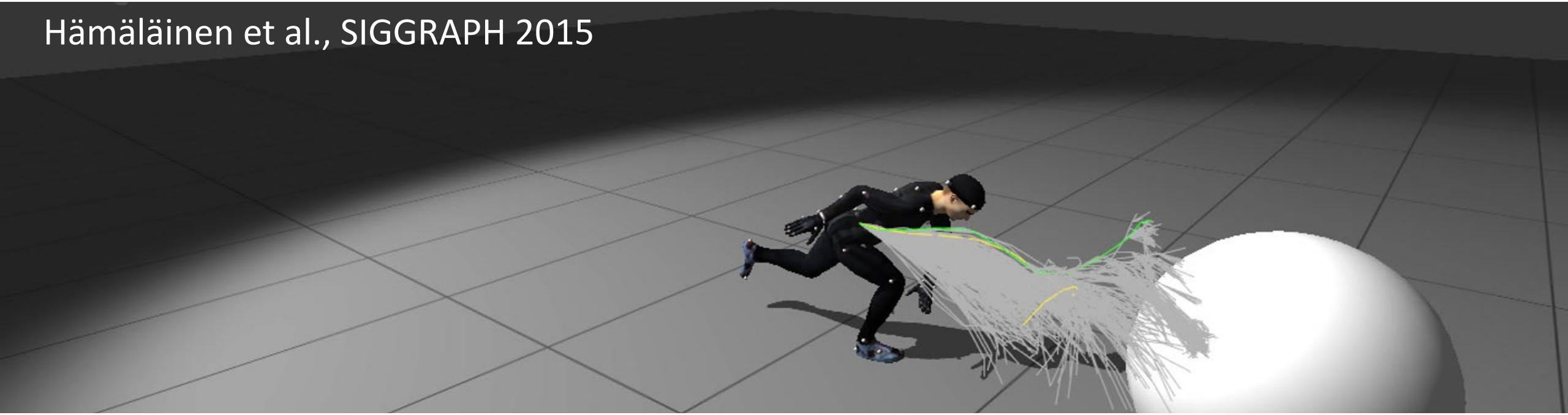
What about continuous actions?

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)

Optimizing action sequences: Summary

Deep RL:

+ Produces a policy network that can compute actions very efficiently
- Training can be very slow

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?

Recap: Optimization theory

- Optimization: Hill climbing/descent on the optimization landscape defined by $f(\mathbf{x})$
- Gradient –based optimization: Follow the gradient (direction of steepest ascent/descend on the landscape), with momentum to avoid getting stuck at the bottom of valleys or top of ridges (e.g., Adam). Tensorflow and Pytorch allow computing gradients of any computation written using them.
- Sampling –based optimization: The $f(\mathbf{x})$ doesn't need to be directly differentiable to estimate gradients based on samples. A Gaussian sampling distribution = Gaussian-blurred $f(\mathbf{x})$ that can smooth away local optima. Practical methods: CMA-ES, PPO, SAC.
- Discrete optimization: Sampling-based optimization still works.
- Action sequences: DRL, MCTS. What's needed: formulate your problem as a (PO-)MDP, i.e., define the actions, observations, rewards, and initial state distribution

Recap: Optimization applications

- Machine learning: Optimize the weights and biases of a neural network to minimize some error metric
- Image synthesis: Optimize a drawing to fool a neural network thinking it's some object
- Animation: Optimize a simulated humanoid's muscle activations to make it achieve a goal like walking or running
- Game AI, artificial life: Optimize the actions of agents to make them achieve goals like performing billiards trick shot, winning a game of Go, or the hiders avoiding the seekers in a game of hide-and-seek.
- Game content: Optimize procedural generation parameters based on some model of player behavior or experience
- Sometimes, the optimization or AI training can be made a game, challenging the player to design a learning environment

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

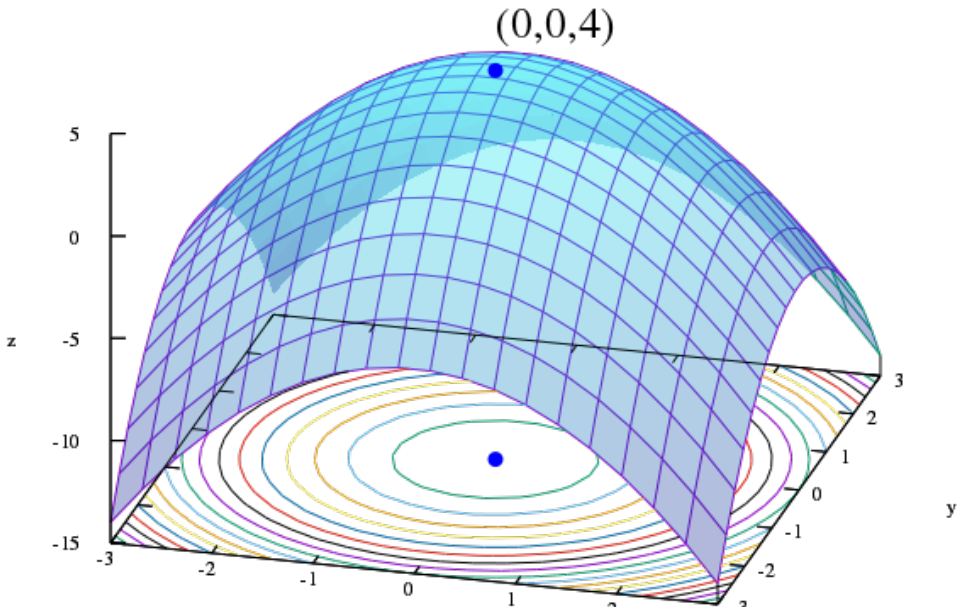
Papers on neuroevolution

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).

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)