What every game developer should know about

# Mathematical Optimization (Part 2)

Intelligent Computational Media
Aalto University
Prof. Perttu Hämäläinen
perttu.hamalainen@aalto.fi

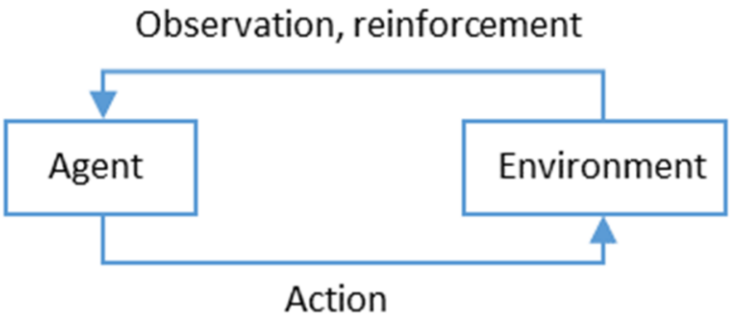
## From CMA-ES to Deep Reinforcement Learning

- Recap: CMA-ES
- Optimizing actions conditional on observations
- Optimizing action sequences

Learning goal: understand how the simple but powerful sampling approach of CMA-ES can be extended to Deep RL

I'm going to explain reinforcement learning and policy optimization visually, highlighting the similarities to CMA-ES and building on what we learned in the previous part of this lecture. This is new this year, and I put the talk together just yesterday. It's perhaps not the standard pedagogical approach, but let's see how it goes.

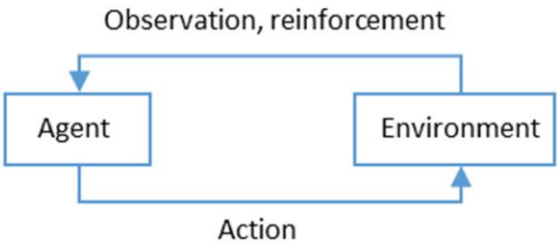
## (Deep) Reinforcement learning (RL)

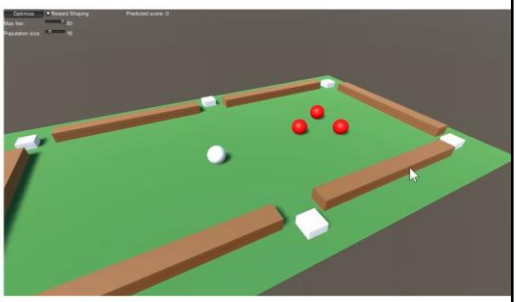



This is the basic very generic framework we will discuss. How can one learn the optimal actions for an agent that receives observations and some reinforcement from some environment? The reinforcement can be either a reward to maximize or a cost to minimize.

Deep RL is a buzzword, but one should note that RL research goes back decades, and the deep just means that one uses a deep neural network somewhere in the algorithm.

# Policy optimization



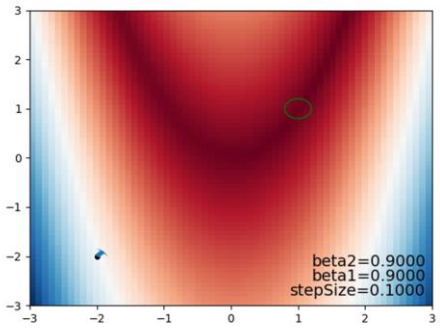



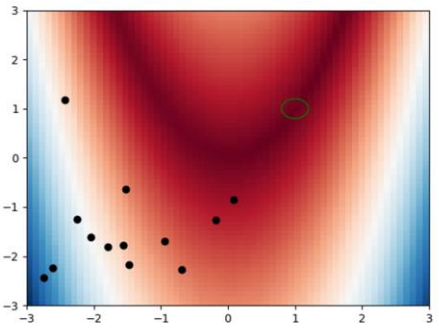
Policy optimization is another common term, basically a synonym to RL. It means that instead of optimizing a single action like a billiards shot, one optimizes a so-called policy function that maps observed world state to actions. In case of billiards, the state is defined by the positions of the balls.

Note that in the general case, one talks about observations, but in the context of this talk, we will simplify by assuming a fully observed world. Thus, we will henceforth replace ”observation” with ”state”.

# Recap from last lecture

- Gradient-based optimization (Adam)
- Population-based optimization (CMA-ES)

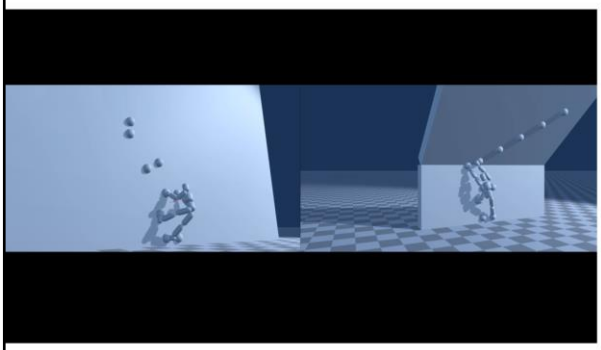


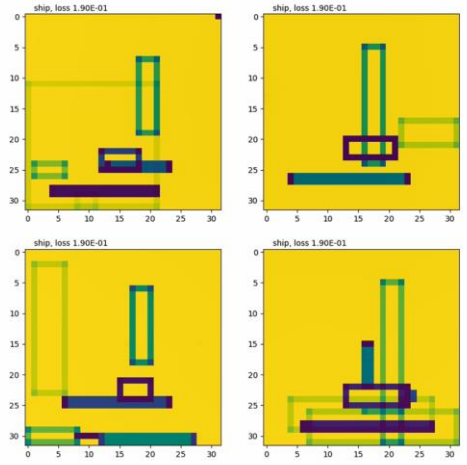



Left: Adam, the gradient-based optimization method that is both powerful and simple to implement, and scales up to billions of parameters. This makes it a good first choice if gradients are available.

Right: CMA-ES, the leading sampling-based or population-based method. This lecture builds on CMA-ES, so let's review it a bit more. This is how it proceeds on this 2d problem.

# CMA-ES applications






CMA-ES can also handle complex problems like controlling a humanoid climber, or optimizing abstract ”paintings” that fool a neural network image classifier.

## Algorithm (CMA-ES)

1. Sample $\mathbf{x}_1 \ldots \mathbf{x}_N$
2. Evaluate $f(\mathbf{x}_i)$ for all $i$=1...N
3. Prune the worst half
4. Adapt the sampling distribution (mean, variance) based on the remaining samples

Repeat the above until sampling budget exhausted

Here is the basic CMA-ES algorithm.

## Notation for optimizing actions (this lecture)

1. Sample actions $\mathbf{a}_1 ... \mathbf{a}_N$
2. Evaluate reward function $r(\mathbf{a}_i)$ for all $i$=1...N
3. Prune the worst half
4. Adapt the sampling distribution (mean, variance) based on the remaining samples

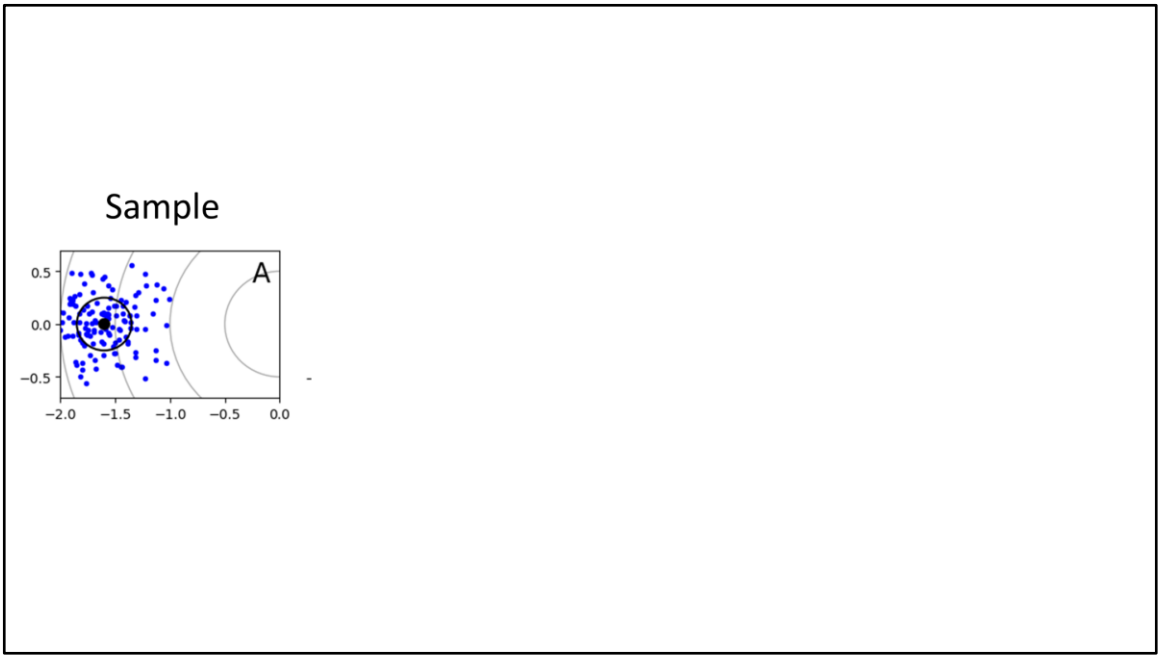
In this talk, the focus is on optimizing actions given some reward function. Thus, we replace **x** with **a** and $f(\mathbf{x})$ with $r(\mathbf{a})$

The reward function is maximized. Equivalently, one can also use a minimized cost function.

## Notation for optimizing actions (this lecture)

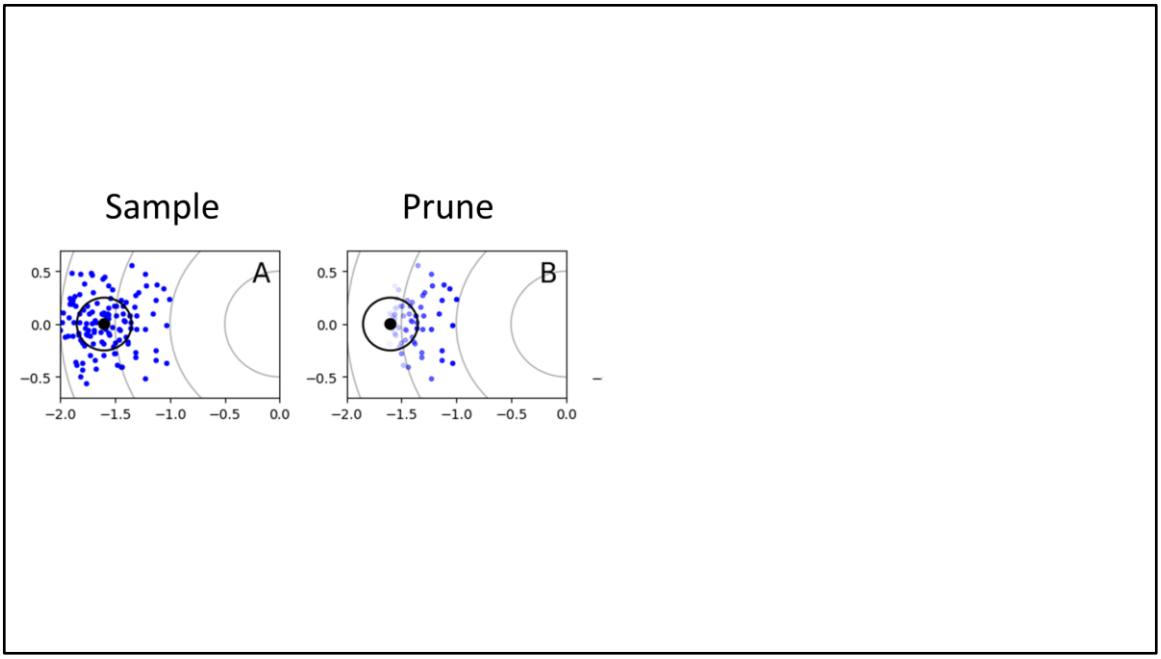
1. Sample actions $\mathbf{a}_1 \ldots \mathbf{a}_N$
2. Evaluate reward function $r(\mathbf{a}_i)$ for all $i$=1...N
3. Prune the worst half
4. Adapt the sampling distribution (mean, variance) based on the remaining samples

The steps 1-3 should be clear. But how does the adaptation of step 4 work?

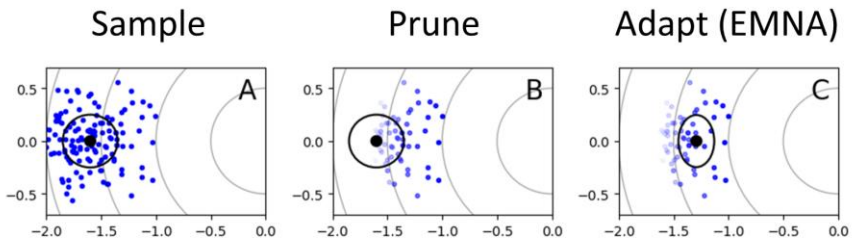



Black dot: mean. Black ellipse: one standard deviation of the 2D Gaussian distribution. Blue dots denot the samples drawn from the Gaussian.

The gray curves are the objective function isocontours, i.e., the optimum is on the right.




After generating and evaluating samples, CMA-ES prunes them. The opacity of the blue samples illustrates their reward function values.

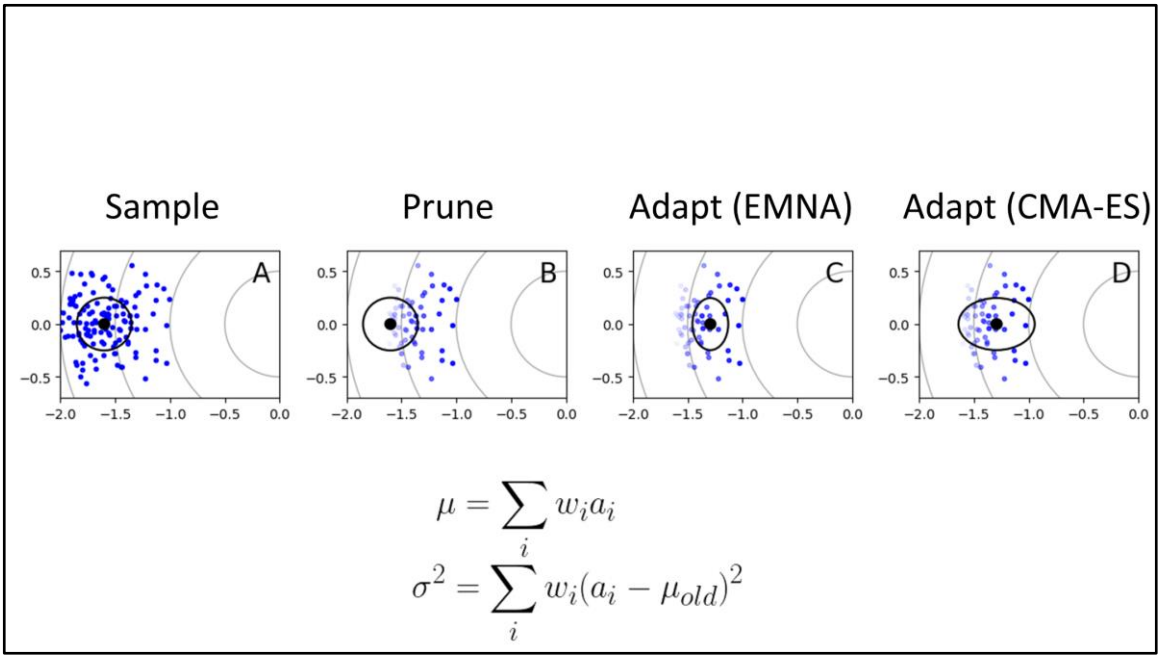



$$\mu = \sum_i w_i a_i$$

$$\sigma^2 = \sum_i w_i (a_i - \mu)^2$$

After pruning, the sampling distribution is fitted to the remaining samples. Basically, the mean μ and variance $\sigma^2$ are computed as weighted sums, the weights *w* computed based on the objective function values such that they sum to 1.

However, if one does it directly like illustrated, it actually corresponds to the so-called EMNA (Estimation of Multivariate Normal) algorithm, which has the problem that it may increase exploration in irrelevant directions – here, along the vertical axis – and make slow progress.



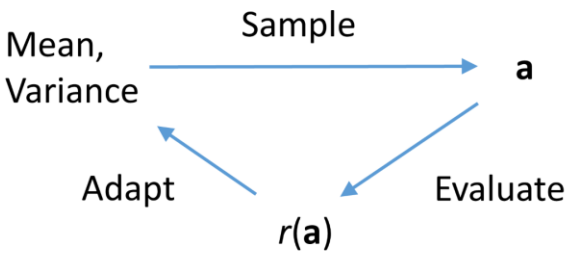

CMA-ES has some extra tricks that elongate the distribution in the progress direction, which makes it more probable that the next iteration will produce good samples. You can think of this as a form of momentum.

CMA-ES first updates the variance and only then updates the mean, or equivalently, uses the old mean in updating the variance. This elongates the distribution in the correct direction, as illustrated here. You can thi

Additionally, CMA-ES has the so-called evolution path heuristic, which further amplifies this effect.

NOTE: The equations depict the simple case of a diagonal covariance matrix, i.e., having just one $\sigma^2$ for each optimized variable. With a full covariance matrix, the update is slightly more complex. Readers interested in the details are pointed to Hansen's tutorial: https://arxiv.org/pdf/1604.00772.pdf

## CMA-ES



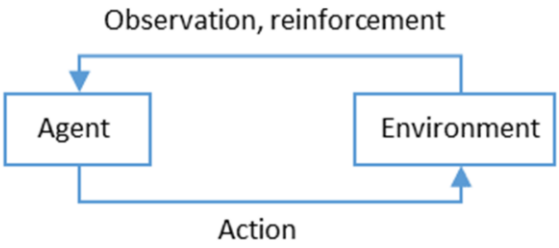

Visually, we have this kind of a feedback loop

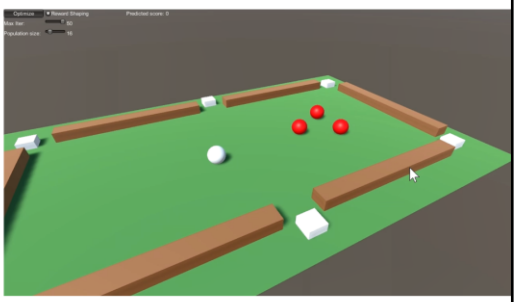
## From CMA-ES to Deep Reinforcement Learning

- Recap: CMA-ES
- **Optimizing actions conditional on observations**
- Optimizing action sequences

Now, let's see what happens when we try to extend this approach to learning to act optimally in a variety of situations

# Real life: Countless variations of the same problems

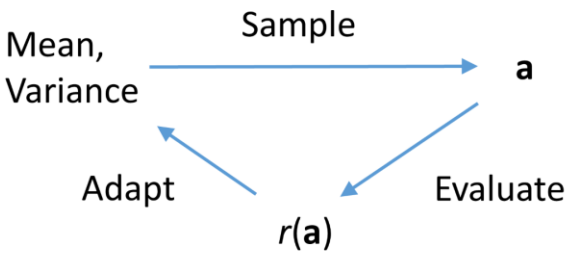


In reality, we often face a large variety of problems, like in case of billiards, where we basically always optimize shot speed and direction, but the ball configurations provide infinite variants of the same problem.

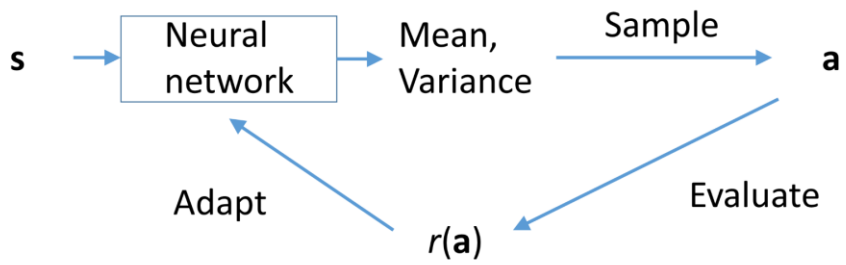
Instead of optimizing each problem instance from scratch, can one learn to act optimally based on the observed problem parameters, or in RL lingo, the observed world state?

## CMA-ES

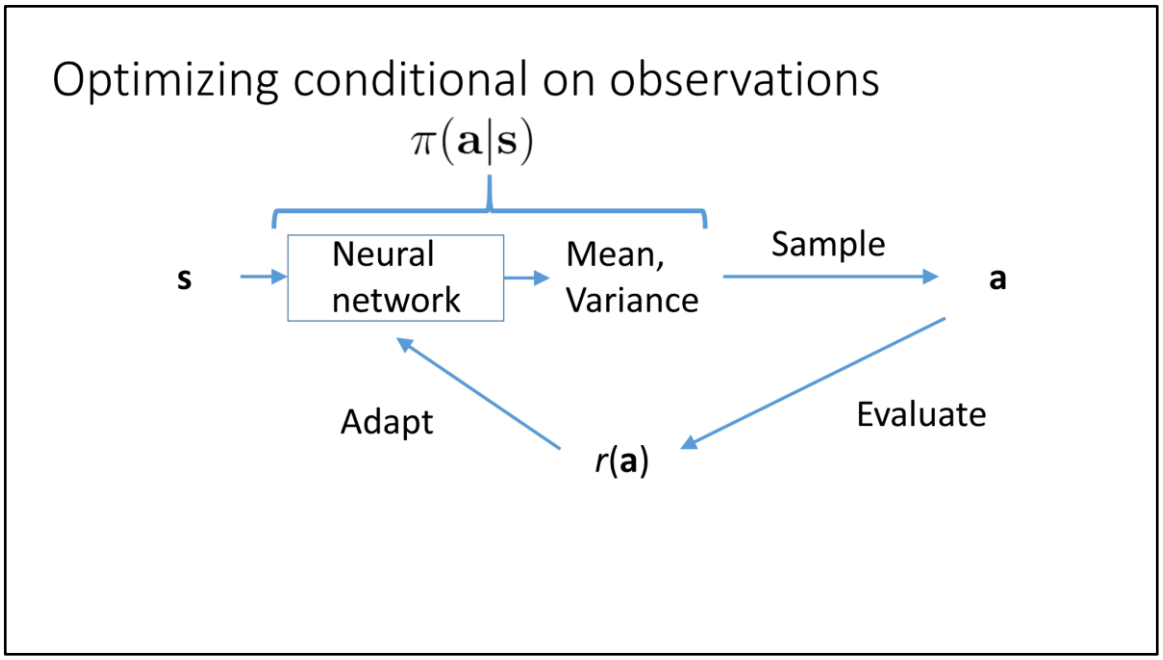



Let's incorporate observations into this graph

## Optimizing conditional on state

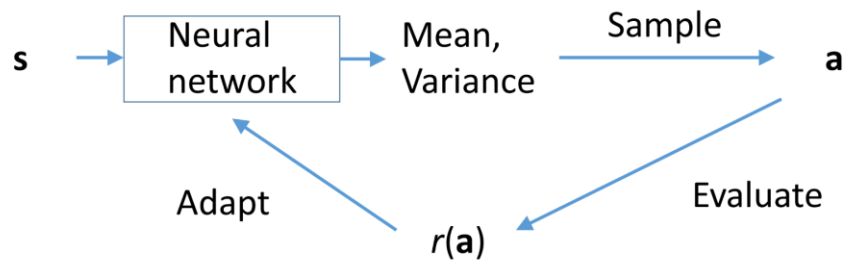



The standard approach is that the sampling distribution depends on the state **s**, which a so-called policy network maps to the mean and variance.




The distribution of actions given the observations or world state is also called the policy distribution, usually denoted π(a | s) in RL papers.

## Optimizing conditional on observations




The mean and variance are algorithm state variables.
A neural network can be used to store and interpolate the variables for each observation.
Instead of assigning to a variable, one trains the network with input-output pairs.

In effect, this is a process for solving an infinite number of action optimization problems at the same time, one for each observed problem configuration.

## Algorithm

1. Sample actions $\mathbf{a}_1 ... \mathbf{a}_N$
2. Evaluate reward function $r(\mathbf{a}_i)$ for all $i$=1...N
3. Prune the worst half
4. Adapt the sampling distribution (mean, variance) based on the remaining samples

Now, let's look at what needs to be changed if one wants to change CMA-ES to incorporate the observations

## Algorithm

1. Sample states $\mathbf{s}_1 ... \mathbf{s}_N$ and actions $\mathbf{a}_1 ... \mathbf{a}_N$
2. Evaluate $r(\mathbf{a}_i, \mathbf{s}_i)$ for all $i$=1...N
3. Prune the worst half
4. Train the policy network with the remaining samples

Basically, one needs to sample both observations and actions, train the policy network instead of simply recomputing the mean and variance

## Algorithm

1. Sample states $\mathbf{s}_1 \ldots \mathbf{s}_N$ and actions $\mathbf{a}_1 \ldots \mathbf{a}_N$
2. Evaluate $r(\mathbf{a}_i, \mathbf{s}_i)$ for all $i$=1...N
3. Prune the worst half — How?
4. Train the policy network with the remaining samples

The problem becomes this part. If we simply remove the worst half of all actions, we may discard good actions that just happened to have a bad f(x,o) value because of the observation.

For example, consider a simulated character that tries to learn to walk and f(x,o) is a reward function. The character will encounter both fallen and upright states, and all upright states will typically have better f(x,o), but we still want to learn from the best actions of the fallen states. Thus, we can't simply discard the worst K% of the data

## Algorithm

1. Sample states $\mathbf{s}_1 \ldots \mathbf{s}_N$ and actions $\mathbf{a}_1 \ldots \mathbf{a}_N$
2. Evaluate $r(\mathbf{a}_i, \mathbf{s}_i)$ for all $i$=1...N
3. Compute advantages $A(\mathbf{a}_i, \mathbf{s}_i) = r(\mathbf{a}_i, \mathbf{s}_i) - V(\mathbf{s}_i)$
4. Train the policy network with advantage-weighted samples

One solution is to replace the pruning with computing so-called advantages.

## Algorithm

1. Sample observed states $\mathbf{s}_1 ... \mathbf{s}_N$ and actions $\mathbf{a}_1 ... \mathbf{a}_N$
2. Evaluate $r(\mathbf{a}_i, \mathbf{s}_i)$ for all $i$=1...N
3. Compute advantages $A(\mathbf{a}_i, \mathbf{s}_i) = r(\mathbf{a}_i, \mathbf{s}_i) - V(\mathbf{s}_i)$
4. Train the policy network with advantage-weighted samples

$V(\mathbf{s}_i)$ is the so-called value function, denoting the average $r(\mathbf{a},\mathbf{s})$ for a given $\mathbf{s}$.

The advantage is positive if the action gives better than average results.

In practice, $V(\mathbf{o})$ can be estimated by another neural network trained using all actions and observations.

## Training with advantage-weighted samples

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

To understand the training loss, let's first consider updating the policy mean. Intuitively, one might try this loss function.

**NOTE:** I'm going to use some slides to explain the loss function, but if you only want to apply some existing algorithm, you don't need to care about this. I'm only explaining it to help those who might want to read some RL algorithms. Once I understood these intuitions myself, I had much easier time reading the papers.

## Training with advantage-weighted samples

The sampling mean output by the policy network.

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

## Training with advantage-weighted samples

Neural network parameters that we optimize

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

What we optimize are the policy network parameters, usually denoted by θ

## Training with advantage-weighted samples

Squared distance between the policy and actions

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

The basic loss is just the plain old mean squared error, using the sampled actions as the training targets

## Training with advantage-weighted samples

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

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

Of course, simply sampling from the policy and then training the policy with the samples doesn't make sense as such.

However, this changes when we introduce the advantages as weights. The larger the weight, the more the training will focus on making the policy network output match the action. Conversely, if the advantage is negative, we are saying that the policy mean should be as far from it as possible.

On the other hand, actions with zero advantages make no difference, and the training works as if those were pruned.

## Training with advantage-weighted samples

Average over all samples in minibatch

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

Finally, as with most minibatch training, we take the average over all samples in the minibatch

## Training with advantage-weighted samples

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

So, to recap, minimizing this loss makes the policy mean attracted to positive-advantage actions and gravitate away from negative-advantage actions. This sounds like what one would want.

## Connection to gradient-based optimization?

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

$$||\mathbf{a} - \mu_\theta(\mathbf{s})||^2 \propto -\log(e^{-\frac{1}{2}||\mathbf{a} - \mu_\theta(\mathbf{s})||^2}) = -\log \pi_\theta(\mathbf{a}|\mathbf{s})$$

Now, one may wonder if there's any connection to gradient-based optimization, and how to update the variance. For this, it's useful to consider that the mean-squared error is actually proportional to the negated logarithm of the Gaussian policy probability density, if we assume that the variance for all variables is 1.

## Advantage-based policy gradient

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

Substituting this to the loss function results in a common formula for so-called advantage-based policy gradient RL. The gradient of this loss can be shown to correspond to a sampling-based estimate of the true gradient of the RL objective function. However, the proof is rather involved, and we won't go into details.

Minimizing this can also be thought as advantage-weighted maximum likelihood estimation of the policy parameters. Basically, minimizing the loss increases the probability of positive-advantage actions and decreases the probability of negative-advantage actions.

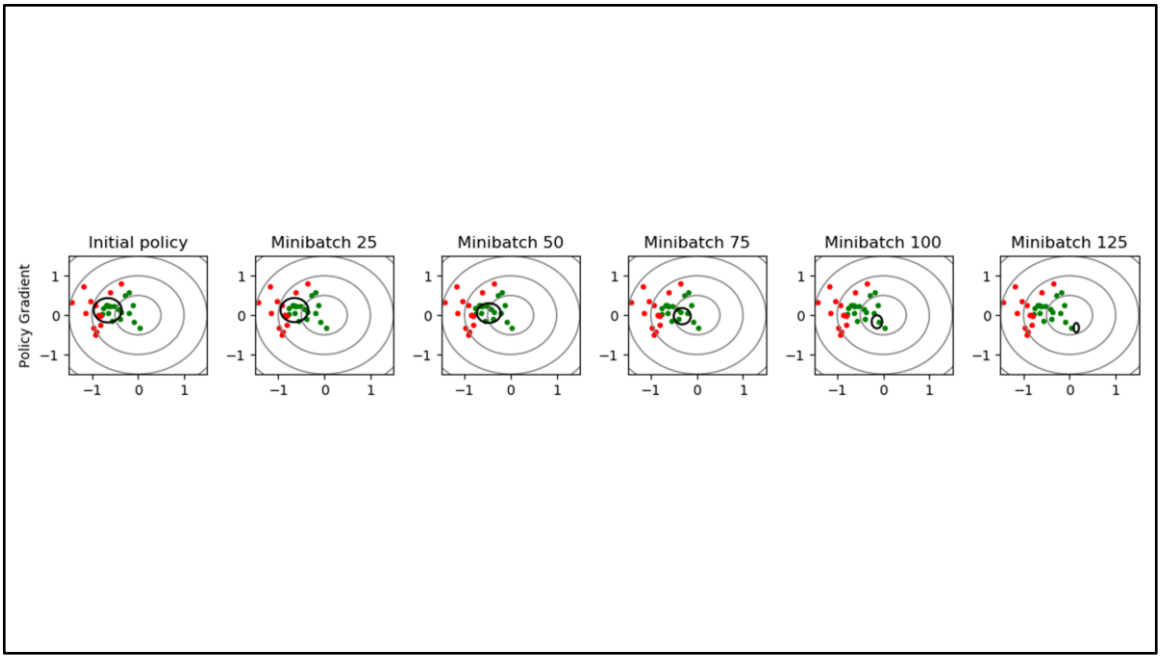
## Updating both mean and variance

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

The policy gradient formula also allows us to handle arbitrary policy distributions with arbitrary variances. For example, this is the loss for a Gaussian policy with diagonal covariance, which is a common choice in continuous control tasks like controlling physically simulated animation characters.

The diagonal elements of the covariance matrix are denoted by *c*. The *j* denote variable indices and *i* indexes over minibatch.

In summary, depending on the policy distribution, the policy learning loss function can become complex. Fortunately, you don't usually have to implement the loss yourself, as there are already many existing open source implementations of RL algorithms.

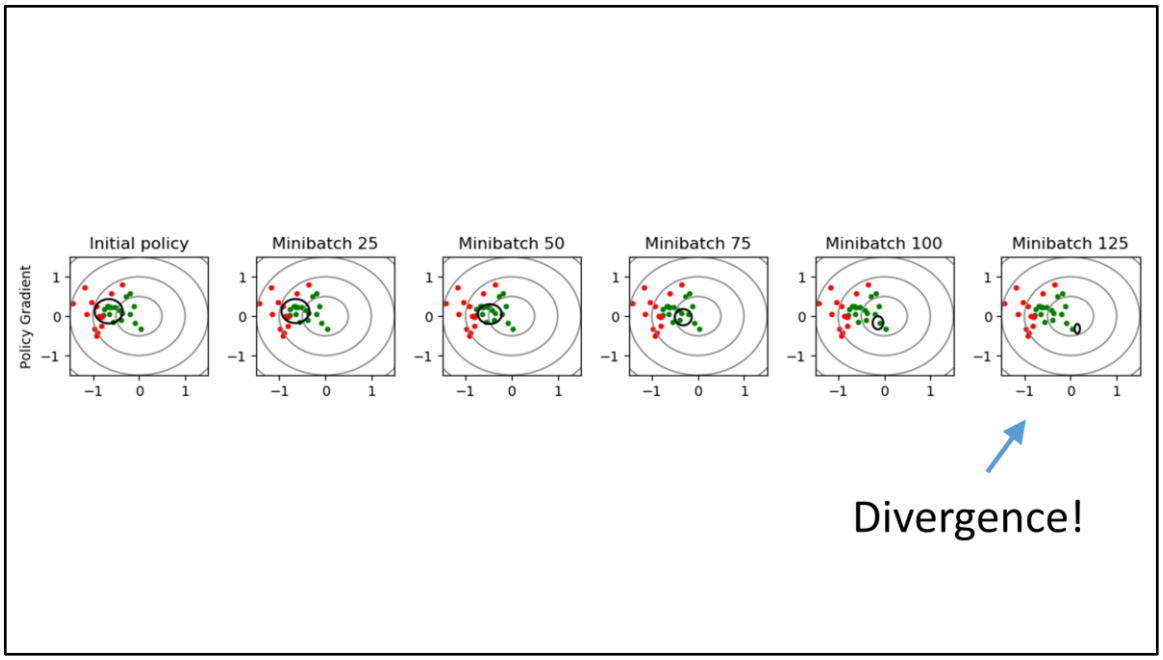



Let's see how this works when we train with the loss function for some minibatch Adam steps, using the samples collected in a single iteration of the algorithm. The policy gravitates towards the positive-advantage actions shown in green, and away from the negative-advantage actions shown in red.

Red: negative-advantage actions
Green: positive-advantage actions

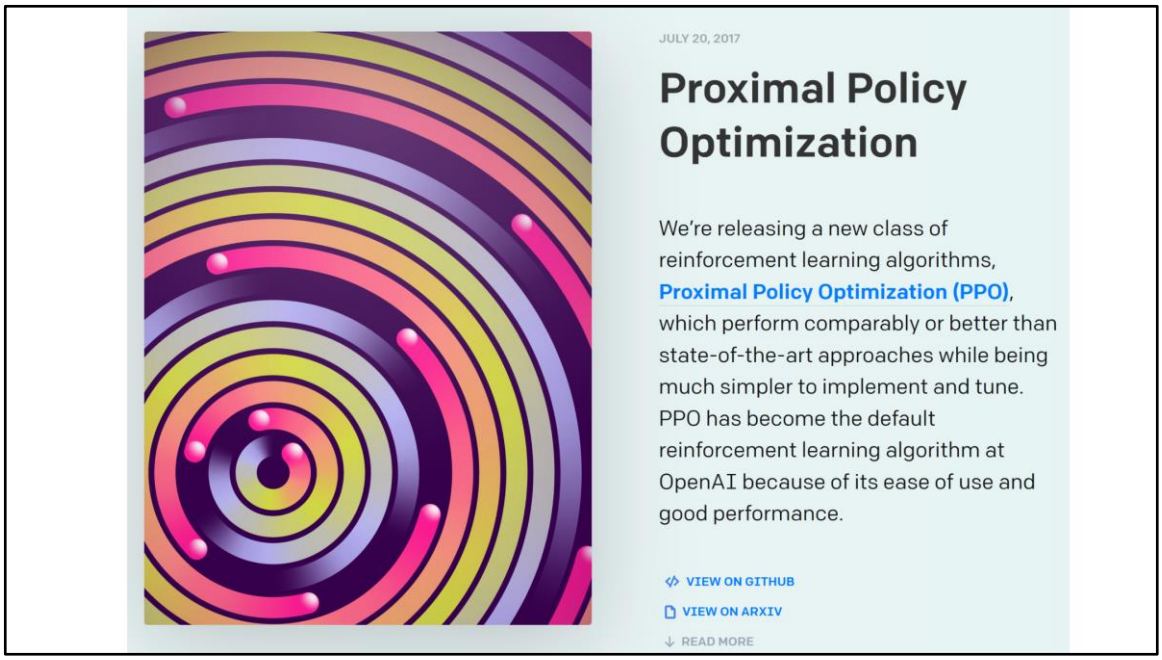
Black ellipse: one standard devation of the policy Gaussian

Here, we visualize only a single state.



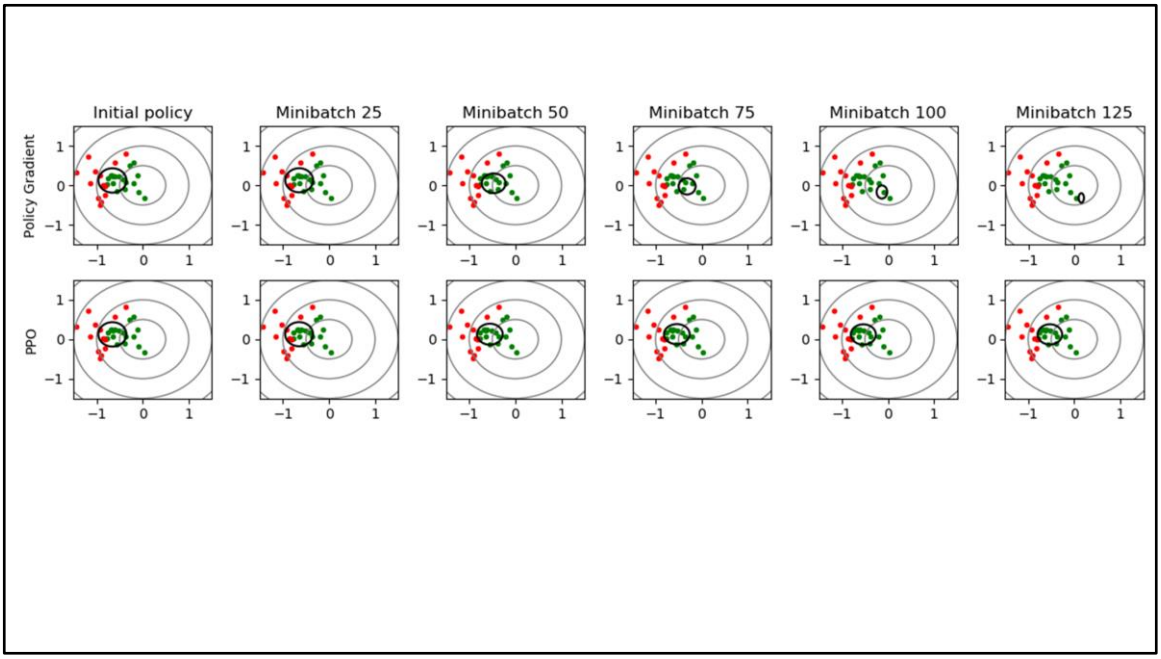

However, it is unstable if one trains for multiple minibatch gradient steps. Fine-tuning the step-size and minibatch count can be hard.

Strictly speaking, the policy gradient loss is only valid for a single gradient step, after which one should sample and evaluate new actions. However, this is very computing-intensive and one would like to get as much out of the samples of a single iteration as possible.



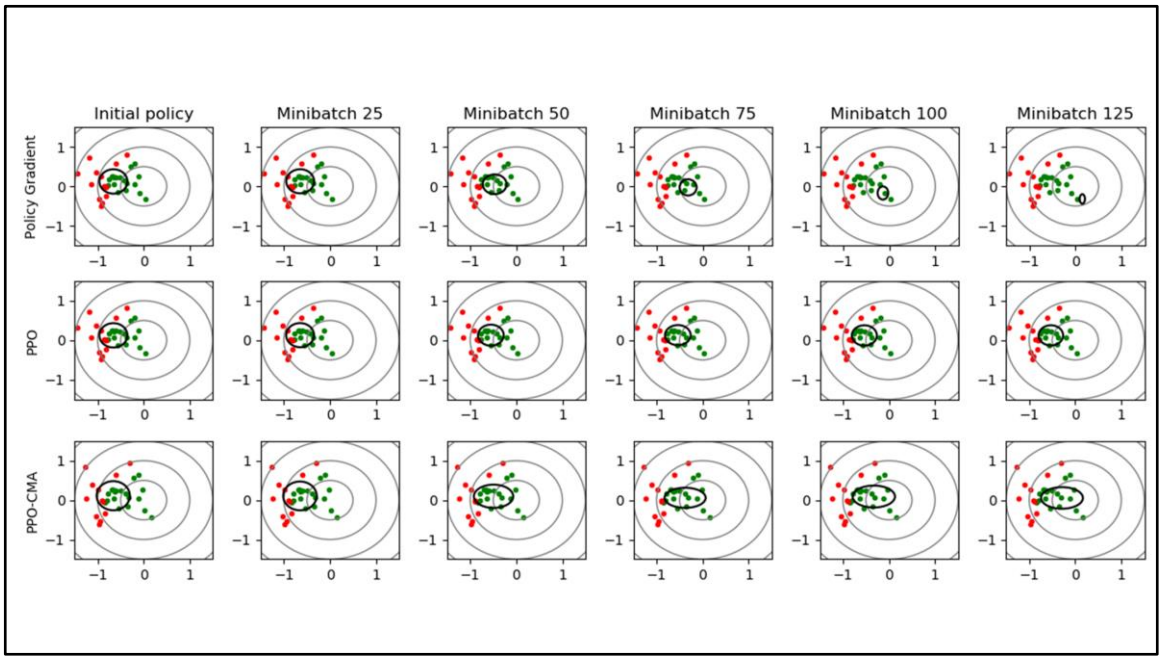

In 2017, the stability problem was addressed by Proximal Policy Optimization (PPO), which quickly became immensely popular. It is used in OpenAI's Dota AI and it's also the default in Unity's Machine Learning Agents. PPO is a simplified version of an earlier method called Trust-Region Policy Optimization

https://blog.openai.com/openai-baselines-ppo/




PPO removes the instability by using the so-called clipped surrogate loss function. This limits how big changes are allowed to the policy in each iteration. In other words, the policy is forced to stay in the proximity of the old policy and sampled actions, hence the name.

For each minibatch, the policy first makes some progress towards the optimum, but then stops updating.




PPO is fairly robust, but can converge slowly. We have recently published an algorithm called PPO-CMA, which employs techniques inspired by CMA-ES to elongate the policy Gaussian in the progress direction, accelerating progress.

The policy is updated to approximate the positive-advantage actions and the negative-advantage actions are converted to positive ones through a mirroring procedure.

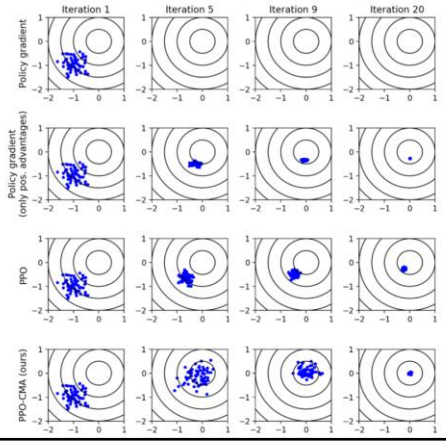
PPO-CMA can still be considered a proximal policy optimization method, though, as the policy will not diverge outside the sampled actions.

## PPO-CMA: Proximal Policy Optimization with Covariance Matrix Adaptation

Perttu Hämäläinen [1] Amin Babadi [1] Xiaoxiao Ma [1] Jaakko Lehtinen [1 2]

**Abstract**

Proximal Policy Optimization (PPO) is a highly popular model-free reinforcement learning (RL) approach. However, we observe that in a continuous action space, PPO can prematurely shrink the exploration variance, which leads to slow progress and may make the algorithm prone to getting stuck in local optima. Drawing inspiration from CMA-ES, a black-box evolutionary optimisation method designed for robustness in similar situations, we propose PPO-CMA, a proximal policy optimization approach that adaptively expands and contracts the exploration variance. With only minor algorithmic changes to PPO, our algorithm considerably improves performance in Roboschool continuous control benchmarks.




**1. Introduction**

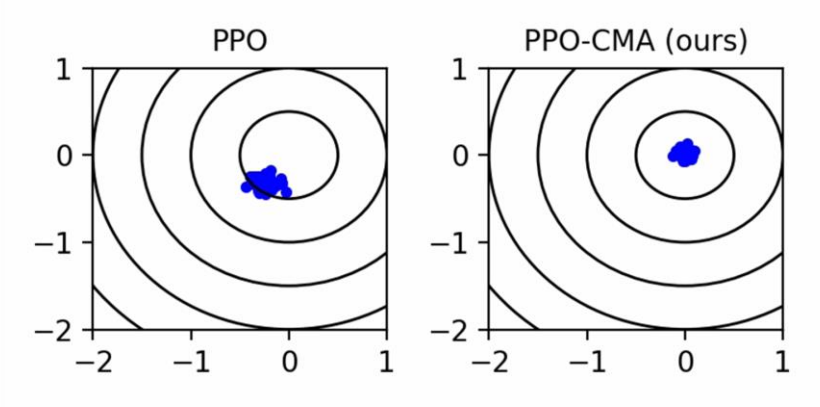
https://arxiv.org/pdf/1810.02541.pdf

https://github.com/ppocma/ppocma

The figure illustrates that over multiple iterations, the basic policy gradient is prone to divergence. Using PPO or policy gradient with only positive advantages helps with the divergence, but can lead to prematurely decaying exploration variance and slow final convergence.

PPO-CMA avoids the premature decay and implements CMA-ES style momentum that allows faster convergence.

## PPO vs. PPO-CMA over multiple iterations

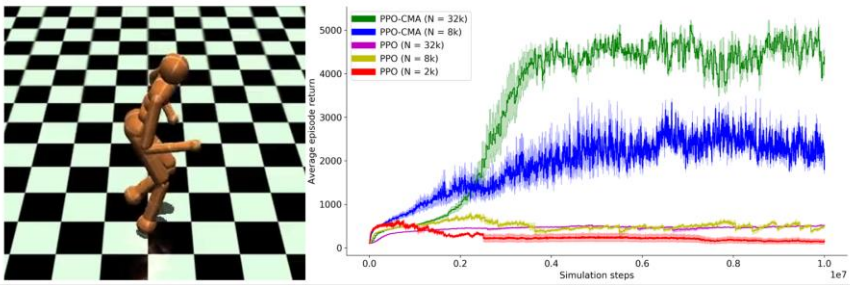



Here's the key result as an animation.

The animation illustrates how PPO-CMA implements the CMA-ES –style momentum, converging much faster than PPO, which makes steady but slow progress.

## From CMA-ES to Deep Reinforcement Learning

- Recap: CMA-ES
- Optimizing actions conditional on observations
- **Optimizing action sequences**




Besides the simple didactic visualizations, PPO-CMA also works on complex cases like humanoid locomotion. Let's now look at what needs to be changed when one optimizes a long sequence of actions like muscle activations, where each action may have delayed rewards.

For example, if one gives the humanoid a reward of moving in a specific direction, the short-term easy solution is to simply fall in that direction – it requires very little effort. However, it obviously does not work for planning horizons above a second or two, as continuing the movement will be more effortful.

## Optimizing action sequences, a.k.a. episodic Reinforcement Learning

Until iteration simulation budget exhausted:
  Sample initial state, resulting in initial observation **s**
  Until terminal state encountered or time limit:
    Sample action **a** according to policy
    Execute action (simulate world model)
    Observe new state **s**' and reward *r*

*One episode*

Update the policy based on the collected experience tuples [**s**,**a**,**s**',*r*]

So far, we've only looked at optimizing a single action conditional on the observed state, e.g., a billiards shot conditional on the ball configuration.
In most cases, we are interested in action sequences, and the reward from an action might be delayed.

In this case, one typically collects experience using *episodes*, i.e., only the initial state of an episode is sampled, after which each new state results from executing the actions. The single action case can be thought as a special case of this, where each episode has only one action.

Note that here, we limit the discussion to so-called *on-policy* methods, where the actions are sampled from the current policy and only the data from one iteration is used for the update and then discarded. There are also *off-policy* methods that maintain the experience from multiple iterations in an experience replay buffer, and do not necessarily sample the actions from the policy.

## Optimizing action sequences, a.k.a. episodic Reinforcement Learning

Until iteration simulation budget exhausted:
- Sample initial state, resulting in initial observation **s**
- Until terminal state encountered or time limit:
  - Sample action **a** according to policy
  - Execute action (simulate world model)
  - Observe new state **s**' and reward *r*

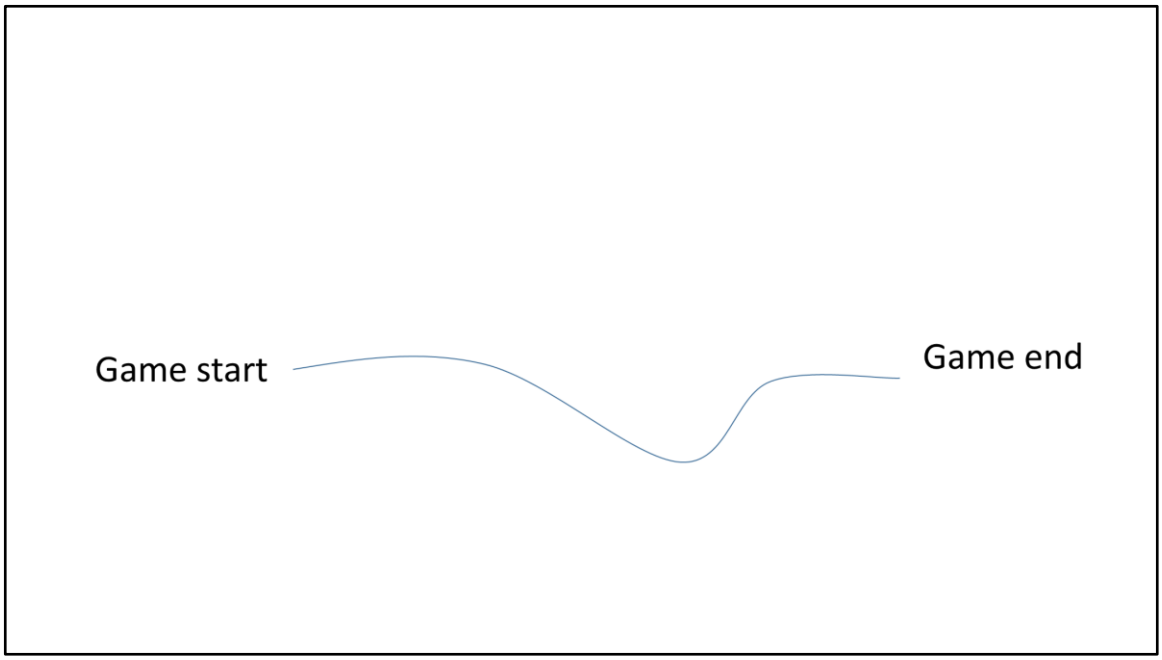
*One episode*

Update the policy based on the collected experience tuples [**s**,**a**,**s**',*r*]

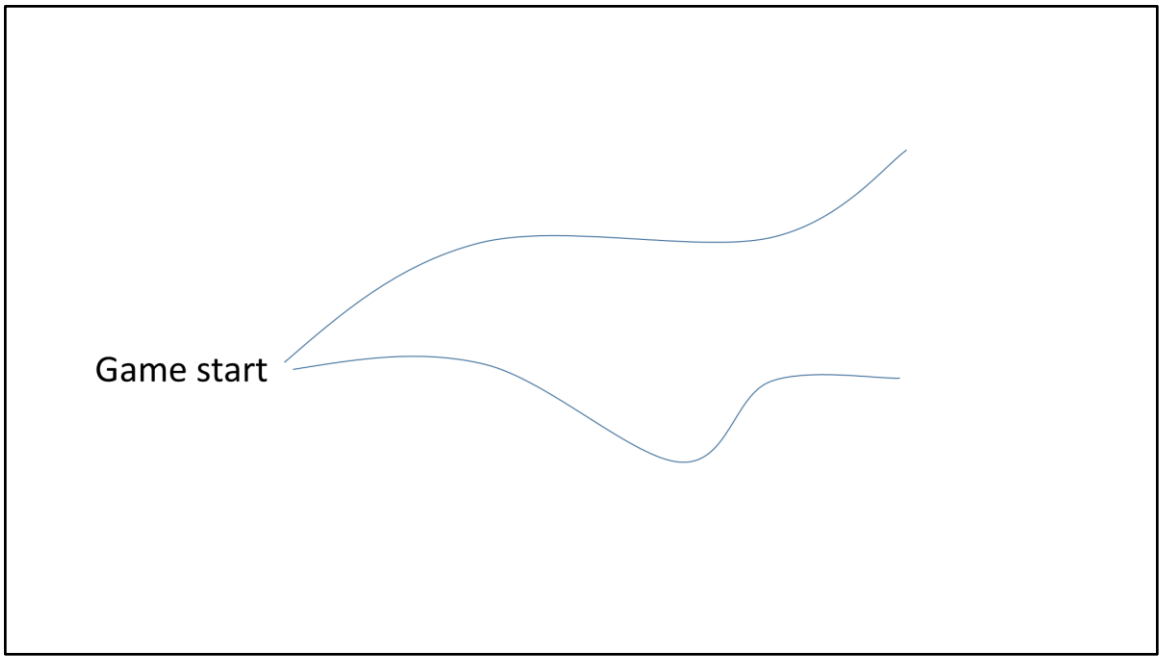
Objective: instead of *r*(**a**,**s**), maximize $\mathbb{E}\left[\sum_t \gamma^t r(\mathbf{a}_t, \mathbf{s}_t)\right]$, where *t* is time

We want to maximize the expected cumulative reward. In the formula, *t* denotes time and $\gamma$ is the reward discount factor, typically 0.99 or some other value close to 1. This means that the weight of the rewards decays exponentially with temporal distance, and rewards far in the future have less effect. $\gamma$ adjusts the balance between short-term and long-term gains.

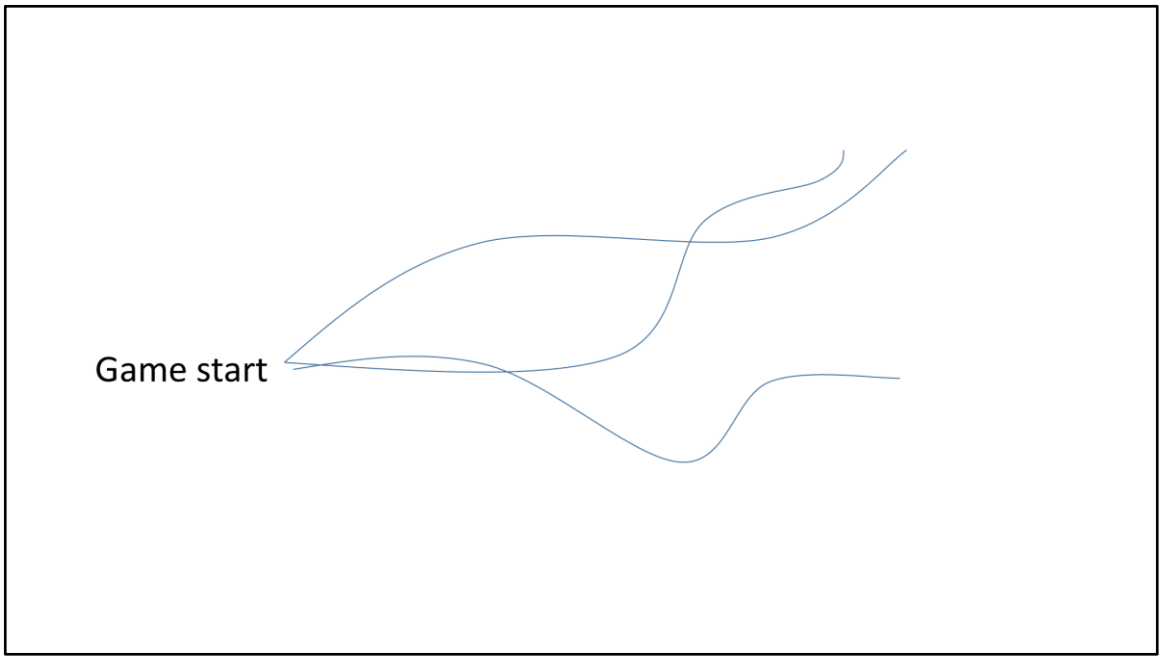
Again, the single action case discussed earlier is equal to this, with each episode having only one action. If *t* starts from zero, $\gamma^t$=1 for the first action.

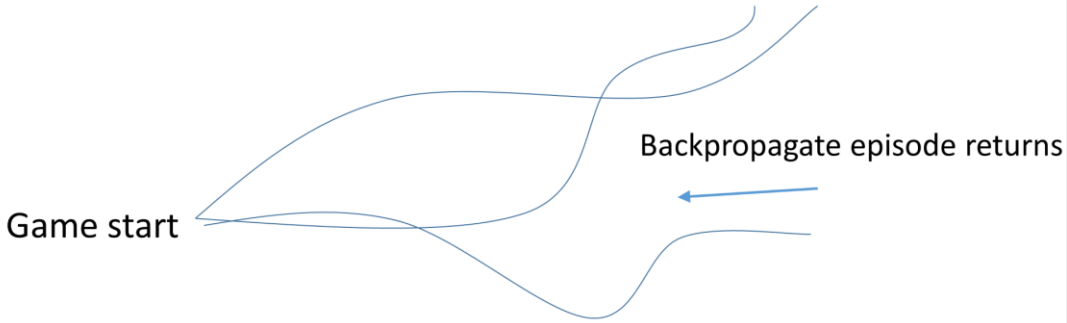



The episodes can be depicted as trajectories in the game state space

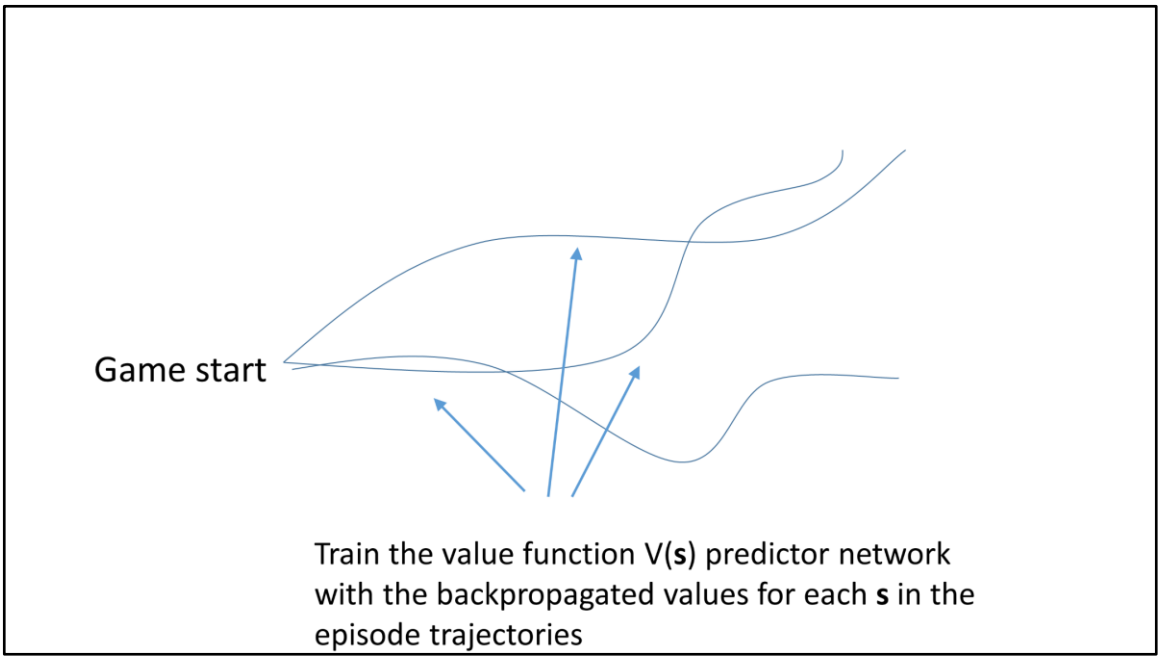



At each iteration, one executes one or more episodes, in this case plays a game one or more times

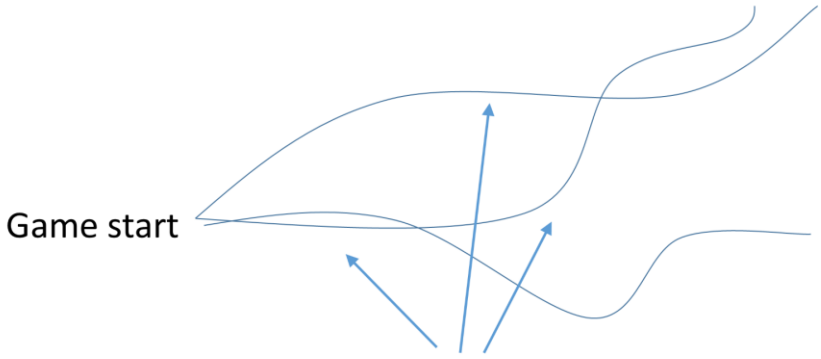

Game start




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




V(**s**) is the expectation of the returns R(**a**,**s**)=r(**a**,**s**) + $\gamma$ R(**a**,**s**'), assuming that actions are sampled from the policy




Compute advantages as A(**a**,**s**)=Q(**a**,**s**)-V(**s**),
Q(**a**,**s**)=r(**a**,**s**)+ $\gamma$ V(**s**') (Bellman backup)

Here, V(**s**) is approximated by the value function predictor network. V(**s**) is the expected value for state **s**, assuming that one samples the actions from the policy. Q(**a**,**s**) equals the expected value of taking action **a** in state **s** and then continuing on the policy. Thus, A(**a**,**s**)=Q(**a**,**s**)-V(**s**) is the benefit, that the single action **a** brings in state **s** over simply using the policy.

This is the basic formulation, which can however be unstable due to inaccuracy (bias) of the value function predictor network.

Both PPO and PPO-CMA use Generalized Advantage Estimation, which is a procedure that allows adjusting a tradeoff between bias for variance.

https://arxiv.org/abs/1506.02438

## Optimizing action sequences, a.k.a. episodic Reinforcement Learning

Until iteration simulation budget exhausted:
- Sample initial state, resulting in initial observation **s**
- Until terminal state encountered or time limit:
  - Sample action **a** according to policy
  - Execute action (simulate world model)
  - Observe new state **s'** and reward *r*

*One episode*

Bottom line: **This is very easy to implement in many applications!**

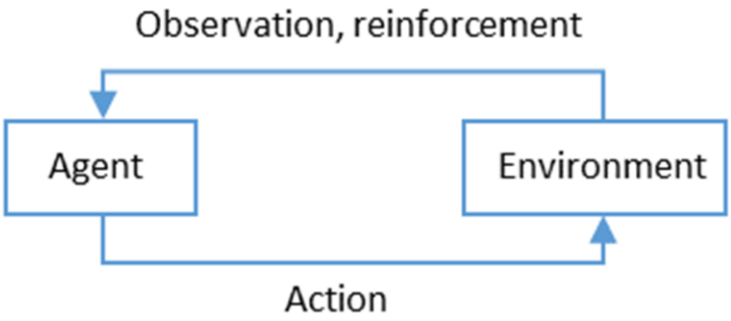
Sampling the initial state = start game level.

Terminal state = agent dies or completes the level.

Generalized Advantage Estimation is a procedure for estimating how much a single action contributes to the full optimization objective.

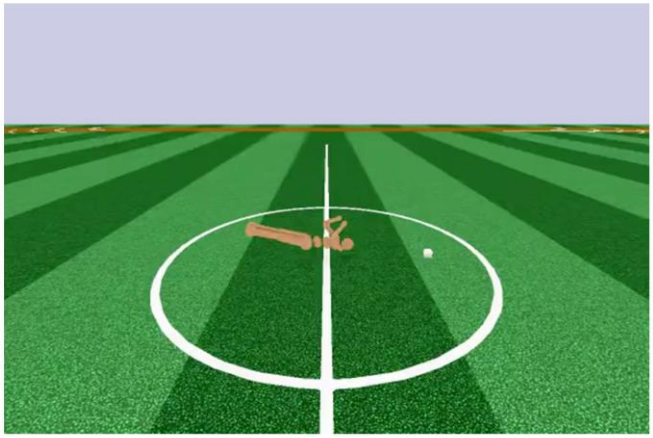
https://arxiv.org/abs/1506.02438

## Reinforcement learning



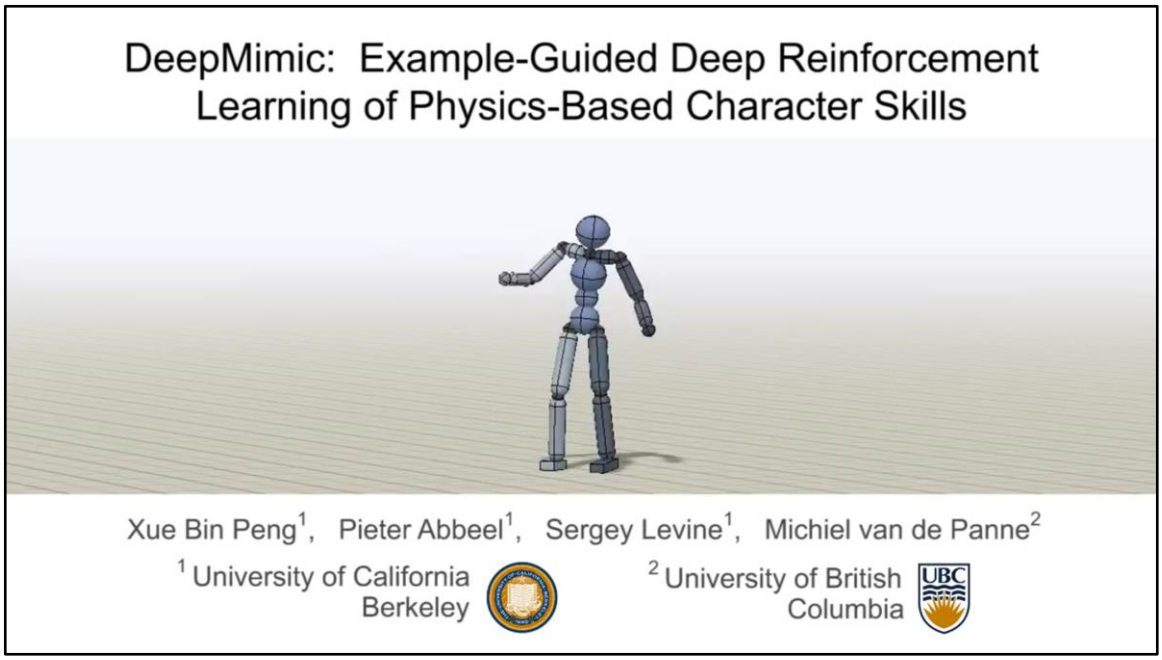

Basically, all you need to do is create an environment that takes in actions and outputs rewards and state observations

## Reinforcement learning

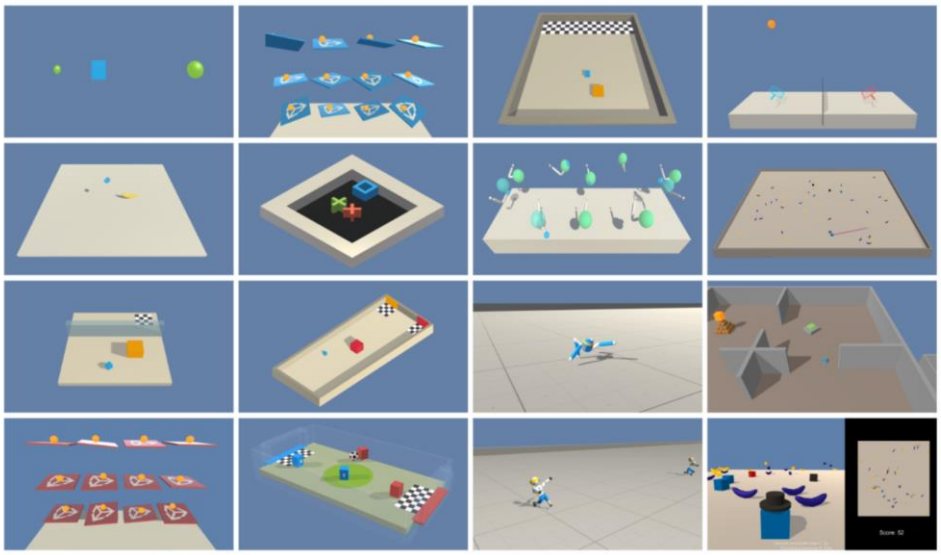
Environments commonly used by researchers include retro Atari games as well as humanoid locomotion tasks.

Now that you know how the agents basically sample their actions randomly, you can see this in the gameplay and movement. One active topic of research is how to produce aesthetically pleasing and natural movement style.




Here's one recent paper that solves the quality problem using motion capture data. I'll talk more about this later.

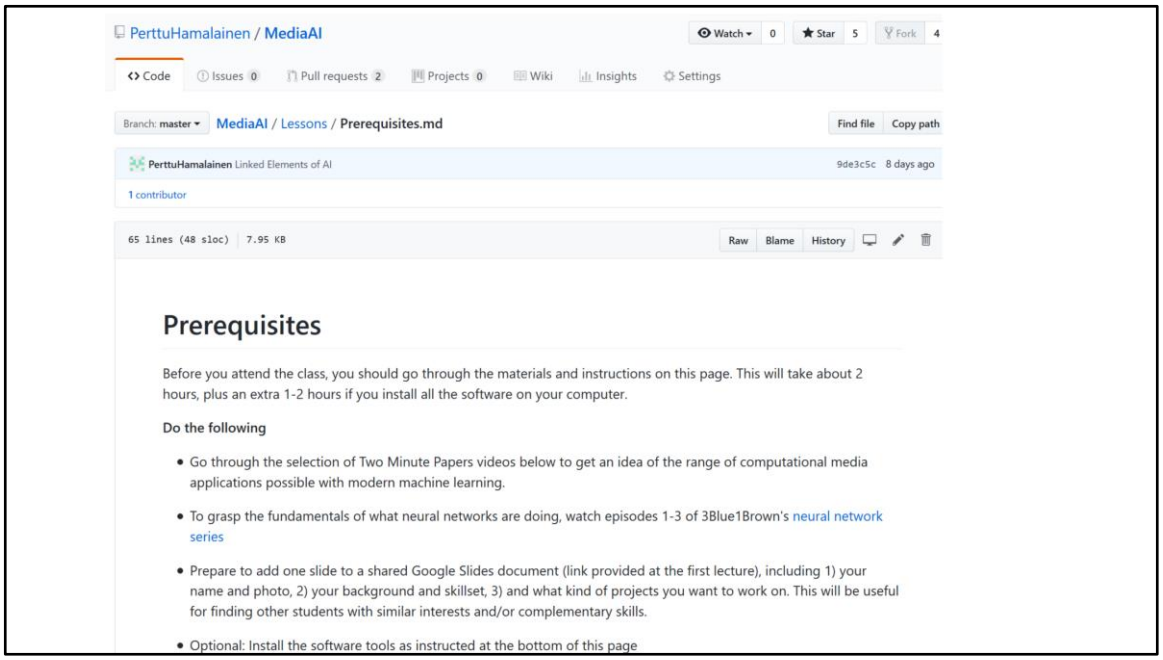
https://arxiv.org/pdf/1804.02717.pdf



Unity ML-Agents Toolkit (Beta)

Unity ML Agents framework implements a number of agents and environments, and it's probably the easiest way to start learning how to use reinforcement learning.

https://github.com/Unity-Technologies/ml-agents




Instructions for installing Unity ML are provided at the bottom of the course Prerequisites page.


After installing, you should be able to run the scenes, e.g., this ball balancing demo.

Important to be able to run episodes in parallel for efficiency, same as the trajectories in the billiards case I showed you yesterday.

https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/

# In Python: OpenAI Gym

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

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

Most researchers use the OpenAI Gym interface and standard environments. Unity ML also provides a Gym wrapper, using which you can try any RL algorithm implemented in Python (provided that there are no Tensorflow etc. version conflicts)

## Some methods to try

- PPO (Schulman et al. 2017): Default in Unity Machine Learning Agents, very popular.
- PPO-CMA (Hämäläinen et al. 2018)
- Soft Actor Critic (Haarnoja et al. 2018)
- Relative Entropy Regularized Policy Iteration (Abdolmaleki et al. 2018)

PPO: Quite robust, but not the fastest and with continuous actions (float-valued actions), the exploration variance can shrink prematurely.

PPO-CMA: Fixes PPO's variance adaptation problem, but only applicable for continous actions and Gaussian policy. These work, e.g., in character animation but not with retro Atari games, where the set of actions is discrete, and it's better to have the policy network output a discrete probability distribution (which one can do using softmax output similar to image classification).

SAC: Many consider this the new state of the art. It's an *off-policy method*, which means that it does not discard the exprience from previous iterations but instead utilizes all experience in the policy updates.

RERP: A new off-policy method from Google Deepmind that uses a somewhat similar, but probabilistically motivated "fit policy to best actions" approach as PPO-CMA.

## Summary: Algorithms

- Reinforcement learning – a.k.a. policy optimization – optimizes the parameters of a policy to maximize expected future-discounted rewards. The $\gamma$ parameter adjusts the discounting.
- Can be thought of as multiple sampling-based optimizations in parallel, one for each possible state. Same principles apply:
  - Randomly try different actions, increase the probability of good actions, decrease the probability of bad actions
  - Use momentum to accelerate (PPO-CMA)
  - Reward function design and action parameterization matter: the random sampling should find at least some good actions for any learning to happen.
  - Reward shaping often needed, similar to the CMA-ES billiards example
- Algorithm state variables (e.g., sampling mean & variance) become as functions of agent state, encoded as neural network parameters

## Summary: Applications

- Widely applicable: animation, robotics, service optimization (facebook feed)
- Common frameworks: Unity ML Agents, OpenAI Gym
- What you need to implement: **environment & reward function**. Unity ML includes the popular PPO algorithm. You don't need to care about the advantage estimation etc.
- Exercise: check out Unity ML Agents, think of applications

The rumor has it facebook uses engagement such as likes and clicks as the reward.