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

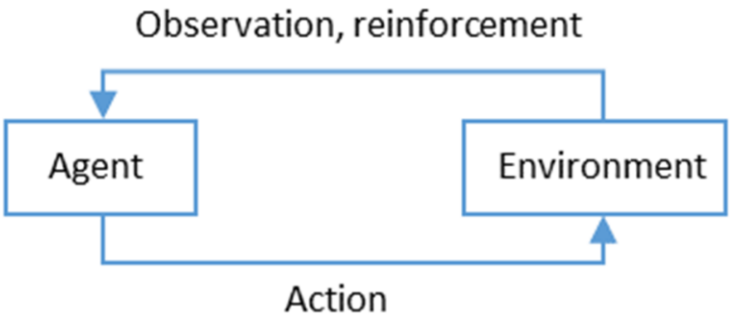
- Recap: CMA-ES
- Optimizing actions conditional on observed state
- Optimizing action sequences
- Common problems and tricks of the trade

Learning goals:

- Understand how the simple but powerful sampling approach of CMA-ES can be extended to Deep RL
- Terminology and visual & mathematical intuitions that help in understanding RL articles and research papers

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.

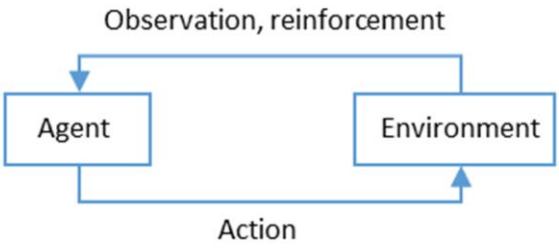
(Deep) Reinforcement learning (RL)

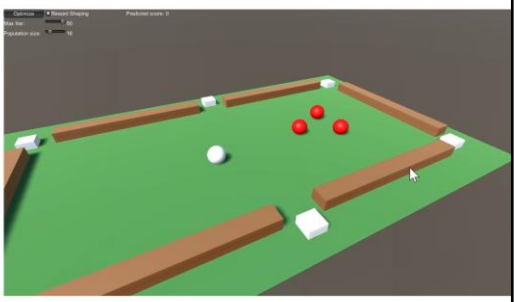


This is the basic very generic framework we will discuss. How can one learn the optimal actions for an agent that receives observations and some reinforcement from some environment? The reinforcement can be either a reward to maximize or a cost to minimize.

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



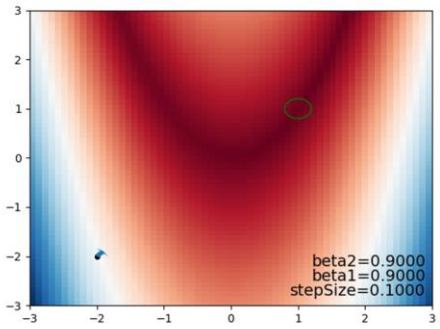


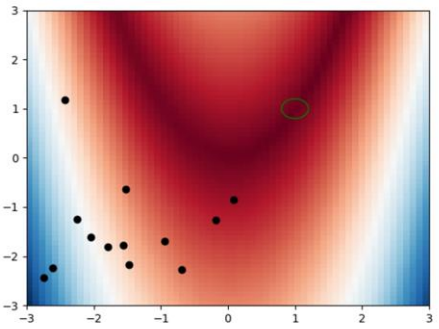
Policy optimization is another common term, basically a synonym to RL. It means that instead of optimizing a single action like a billiards shot, one optimizes a so-called policy function that maps observed world state to actions. In case of billiards, the state is defined by the positions of the balls.

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)

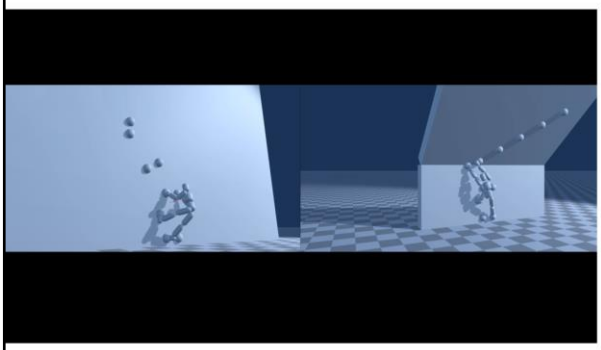


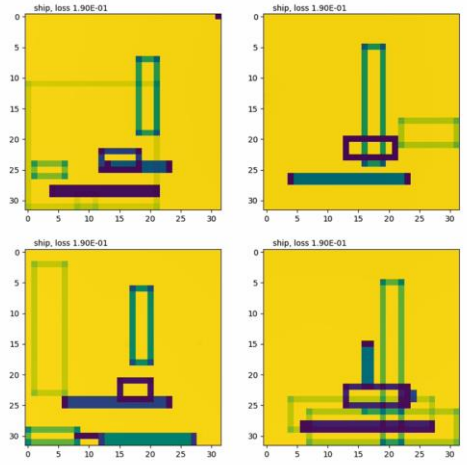


Left: Adam, the gradient-based optimization method that is both powerful and simple to implement, and scales up to billions of parameters. This makes it a good first choice if gradients are available.

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.

CMA-ES algorithm (one iteration)

1. Sample $\mathbf{x}_1 \ldots \mathbf{x}_N$
2. Evaluate $f(\mathbf{x}_i)$ for all i=1...N
3. Sort and compute sample weights w_i
4. Fit the sampling distribution (mean, variance) to the weighted samples.

Goal of the distribution fitting: Make sampling $\mathbf{x}_i$ with good $f(\mathbf{x}_i)$ more probable.

Here is the basic CMA-ES iteration. This is typically repeated until one gets good enough results or runs out of computing budget.

Notation for optimizing actions (this lecture)

1. Sample $\mathbf{a}_1 ... \mathbf{a}_N$
2. Evaluate $r(\mathbf{a}_i)$ for all i=1...N
3. Sort and compute sample weights w_i
4. Fit the sampling distribution (mean, variance) to the weighted samples.

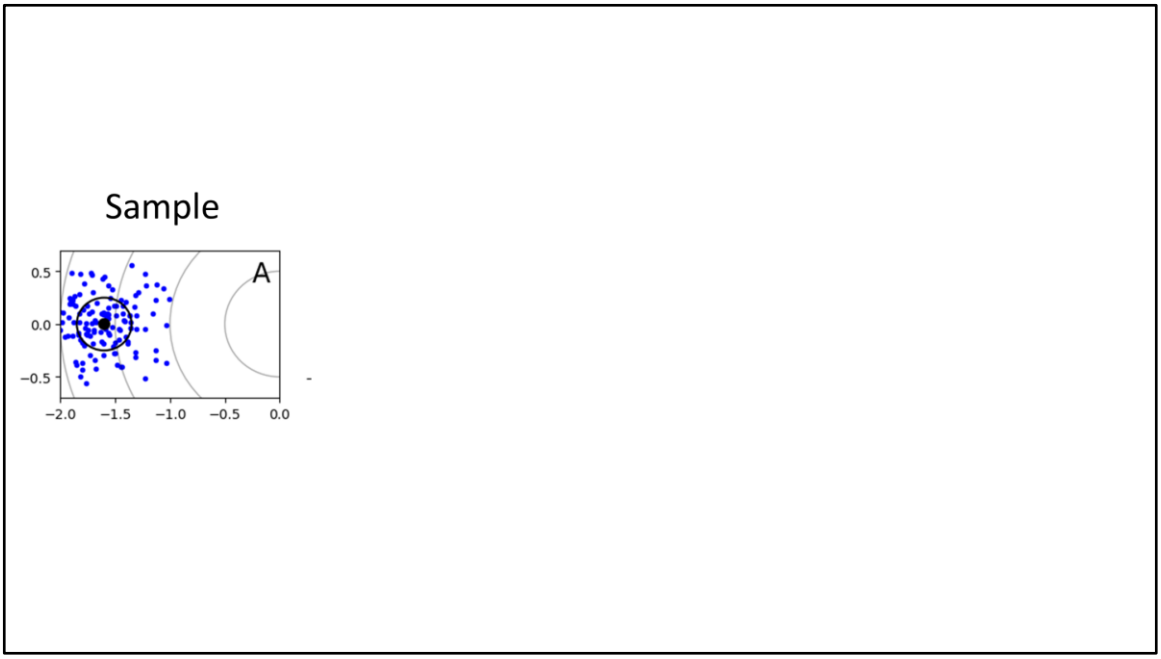
In this talk, the focus is on optimizing actions given some reward function. Thus, we replace **x** with **a** and $f(\mathbf{x})$ with $r(\mathbf{a})$

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

Notation for optimizing actions (this lecture)

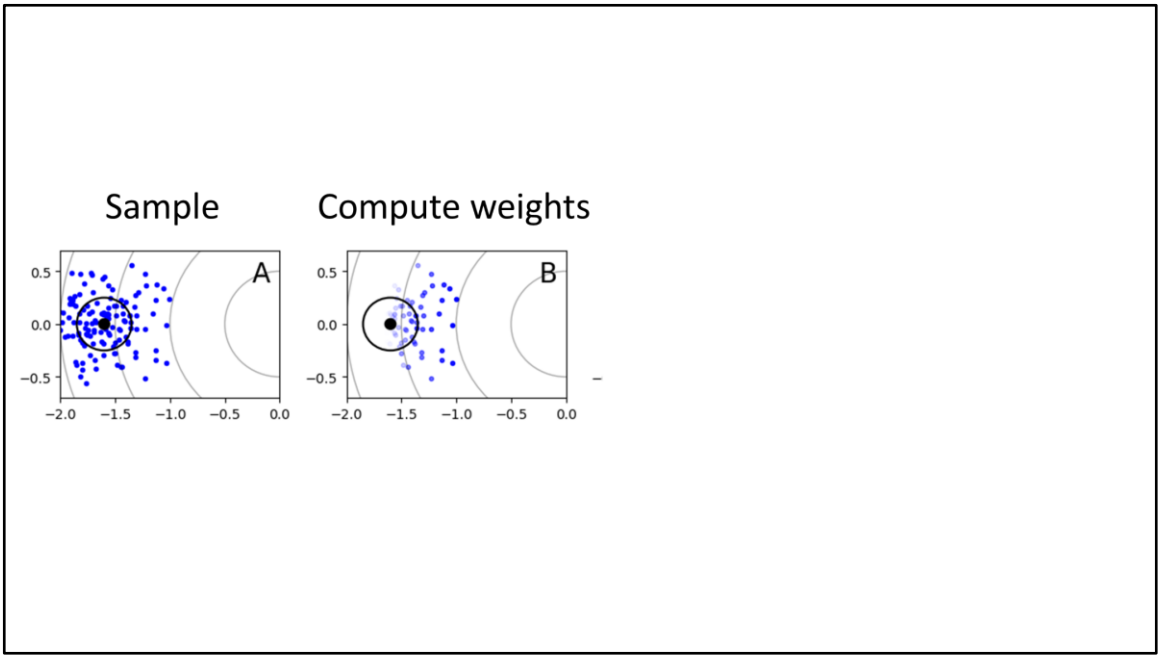
1. Sample $\mathbf{a}_1 \ldots \mathbf{a}_N$
2. Evaluate $r(\mathbf{a}_i)$ for all i=1...N
3. Sort and compute sample weights w_i
4. Fit the sampling distribution (mean, variance) to the weighted remaining samples. Make sampling $\mathbf{a}_i$ with good $r(\mathbf{a}_i)$ more probable.

The steps 1-2 should be clear. Let's now take a closer look at how the weight computing and distribution fitting works.



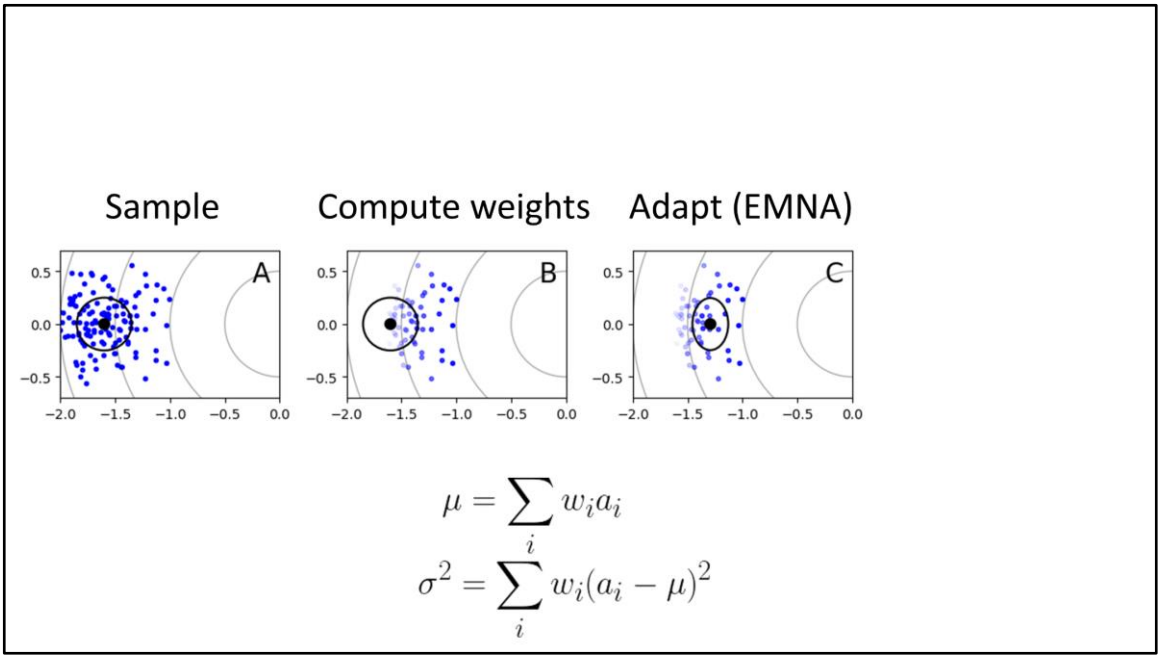
Black dot: mean. Black ellipse: one standard deviation of the 2D Gaussian distribution. Blue dots denote the samples drawn from the Gaussian.

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



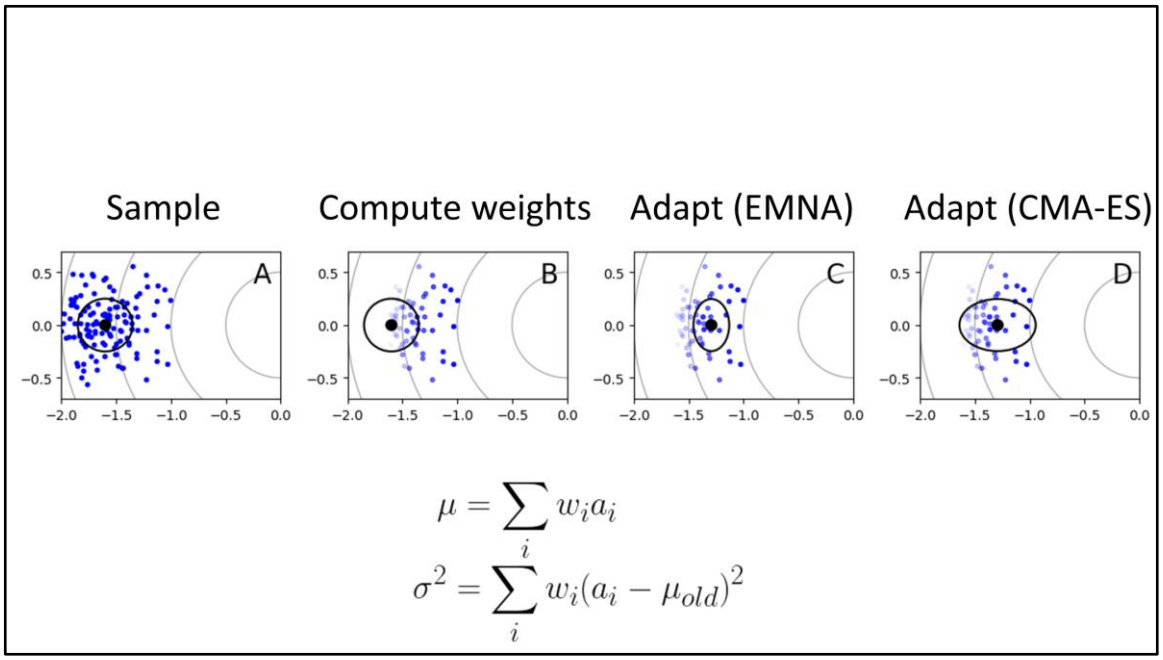
After generating and evaluating samples, we compute the $f(\mathbf{x})$ or $r(\mathbf{a})$ values, and then convert them to weights. Here, the opacity of the samples illustrates the weights.

By default, CMA-ES prunes the the worst half of the samples, i.e., sets their weight to zero. The remaining weights are based on sample sorting such that the samples with best objective function values get highest weights.



After pruning, the sampling distribution is fitted to the remaining samples. Basically, the mean μ and variance σ^2 are computed as weighted sums using the weights *w.* In math and statistics terminology, this corresponds to weighted maximum likelihood estimation of the Gaussian distribution parameters, i.e., setting the distribution parameters such that they maximize the probability of the data.

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.



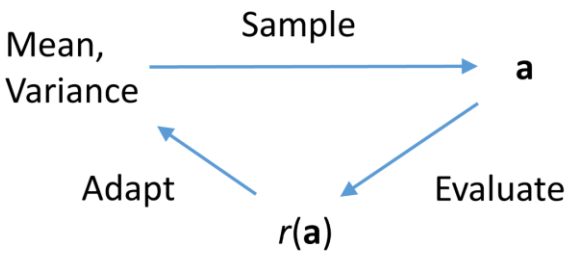
CMA-ES has some modifications that elongate the distribution in the progress direction, which makes it more probable that the next iteration will produce good samples. You can think of this as a form of momentum.

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 think of this as a form of momentum.

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 σ^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



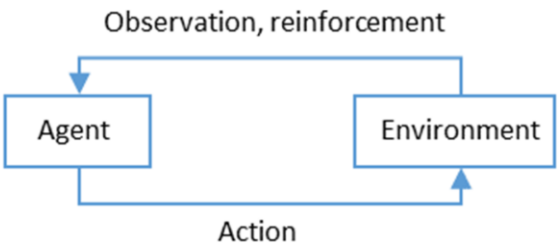
Visually, we have this kind of a feedback loop

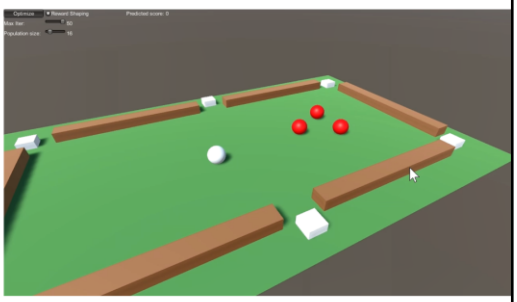
From CMA-ES to Deep Reinforcement Learning

- Recap: CMA-ES
- **Optimizing actions conditional on observed state**
- Optimizing action sequences
- Common problems and tricks of the trade

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

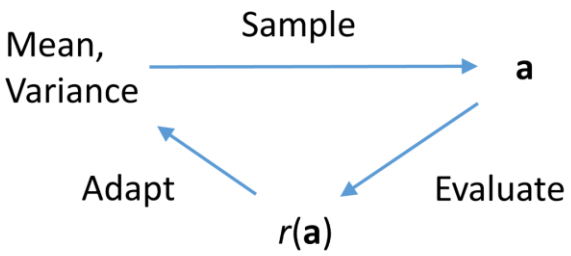




In reality, we often face a large variety of problems, like in case of billiards, where we basically always optimize shot speed and direction, but the ball configurations provide infinite variants of the same problem.

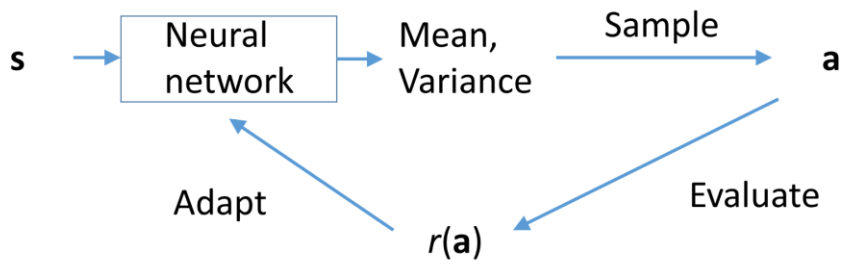
Instead of optimizing each problem instance from scratch (which is slow), can one learn to act optimally based on the problem parameters, or in RL lingo, the observed world state?

CMA-ES

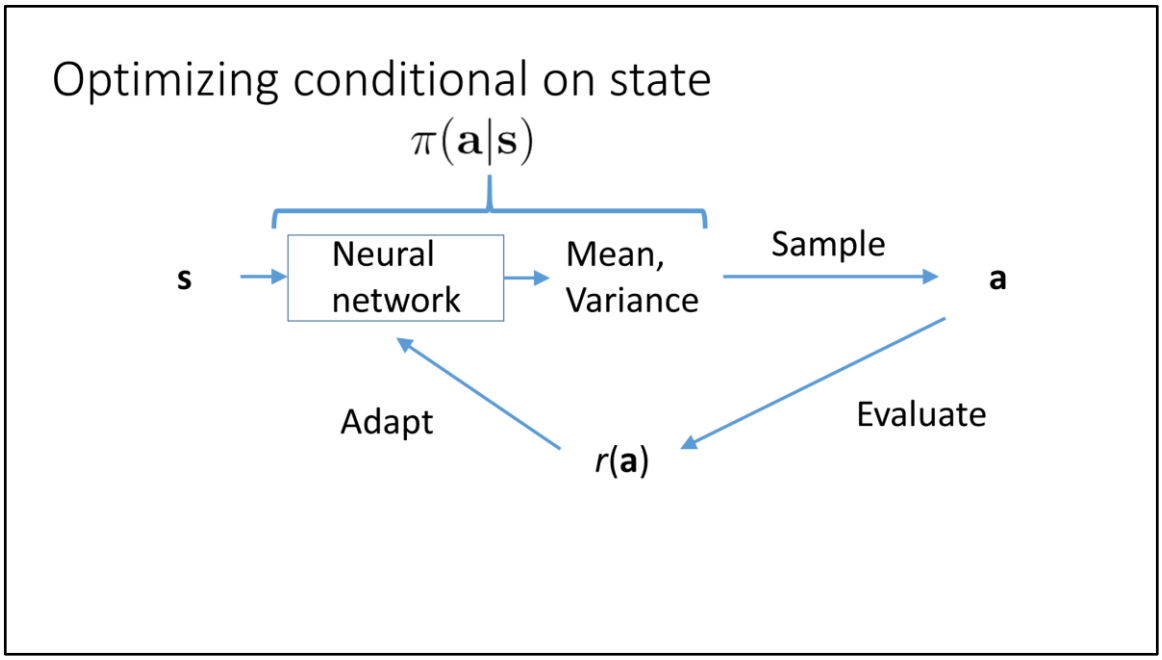


Let's incorporate observations into this graph

Optimizing conditional on state

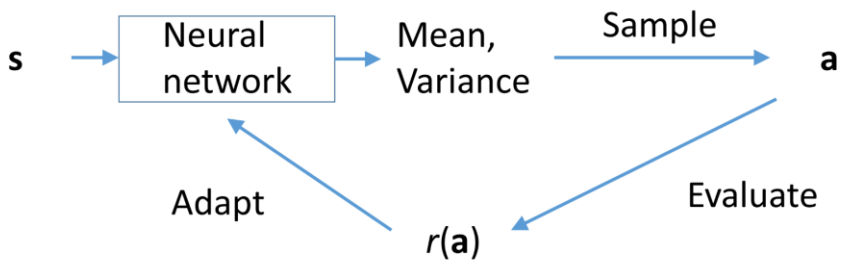


The standard approach is that the sampling distribution depends on the state **s**, which a so-called policy network maps to the mean and variance.



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 state



The mean and variance are algorithm state variables.
When optimizing for multiple problem configurations **s**, the algorithm parameters become functions of **s**, encoded as neural network parameters.

Instead of assigning to a variable, one updates the function by training the network.

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 $\mathbf{a}_1 ... \mathbf{a}_N$
2. Evaluate $r(\mathbf{a}_i)$ for all i=1...N
3. Sort and compute sample weights w_i
4. Fit the sampling distribution (mean, variance) to the weighted samples. Make sampling $\mathbf{a}_i$ with good $r(\mathbf{a}_i)$ more probable.

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 $\mathbf{s}_1 \ldots \mathbf{s}_N$ and actions $\mathbf{a}_1 \ldots \mathbf{a}_N$
2. Evaluate $r(\mathbf{a}_i, \mathbf{s}_i)$ for all i=1...N
3. Sort and compute sample weights w_i
4. Fit the sampling distribution (mean, variance) to the weighted samples. Make sampling $\mathbf{a}_i$ with good $r(\mathbf{a}_i, \mathbf{s}_i)$ more probable.

Basically, one needs to sample both states 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. Sort and compute sample weights w_i
4. Fit the sampling distribution (mean, variance) to the weighted samples. Make sampling $\mathbf{a}_i$ with good $r(\mathbf{a}_i, \mathbf{s}_i)$ more probable.

How?

The problem becomes this part. If we simply sort the actions and prune the worst half like in basic CMA-ES, we may discard good actions that just happened to have a bad $r(\mathbf{a},\mathbf{s})$ value because of the state.

For example, consider a simulated character that tries to learn to walk and gets rewards for staying upright and moving forward. The character will encounter both fallen and upright states, and all upright states will typically have better $r(\mathbf{a},\mathbf{s})$, 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.

To make the sorting work, one should separately sort the actions for each state. However, this is not possible as with a high-dimensional state space like that of a simulated human body, we typically only sample each state once, although we may sample states that are close to each other.

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 $A(\mathbf{a}_i, \mathbf{s}_i) = r(\mathbf{a}_i, \mathbf{s}_i) - V(\mathbf{s}_i)$

One solution is to replace the weight computing, sorting and pruning of CMA-ES with computing so-called advantage function values.

NOTE: In practice, if you want to just use a Deep RL algorithm that has source code available, you don't need to handle the advantage computing or think about the network training yourself.

In the following slides, I will explain these in more detail, also involving some math. This is optional content, mainly provided 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.

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 $A(\mathbf{a}_i, \mathbf{s}_i) = r(\mathbf{a}_i, \mathbf{s}_i) - V(\mathbf{s}_i)$

$V(\mathbf{s}_i)$ is the so-called value function, denoting the average $r(\mathbf{a},\mathbf{s})$ for a given $\mathbf{s}$ (for a single optimized action, more about action sequences later)

The advantages can be used as sample weights such that for each state, averagely good actions have zero weight and better than average actions have higher than zero weights.

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

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. Fit the sampling distribution (mean, variance) to the advantage-weighted samples. Make sampling $\mathbf{a}_i$ with high $A(\mathbf{a}_i, \mathbf{s}_i)$ more probable.

In summary, using the advantages provides a (rough) analogue to CMA-ES which can be implemented without the sorting and pruning operations.

Now, let's see how one goes about fitting the policy distribution to the advantage-weighted samples.

Training loss

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

This is a common policy network training loss function in RL papers. Minimizing the loss can be interpreted as advantage-weighted maximum likelihood estimation of the policy network parameters θ; in other words, minimizing the loss will fit the policy to the advantage-weighted samples, similar to the

It is also the so-called advantage-based policy gradient loss, i.e., the gradient of the loss with respect to the policy network parameters θ gives the direction of improvement when one tries to maximize the rewards. The mathematical proof is involved and we will skip it, and focus on some basic intuitions.

Training loss

Average over all samples in minibatch

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

The first thing to note is that like with most neural network training, the loss is an average over a minibatch

Training loss

Minimizing the loss = maximizing the average

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

Training loss

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

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

Training loss

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

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

Training loss

The advantage values,
treated as constants
(computed before training)

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

Training loss

Maximize this = maximize the probability of actions with positive advantages

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

The log is monotonously increasing. Thus, when we maximize it, we maximize the π(a | s).

Of course, if the advantages are negative, the optimization turns to minimization.

Actions with zero advantages make no difference, and the training works as if those were pruned.

In summary, the loss function leads to maximizing the probability of positive-advantage actions and minimizing the probability of negative-advantage actions.

For more, see https://arxiv.org/pdf/1810.02541.pdf

Gaussian policy with identity covariance matrix

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

Now, let's see an example of writing out the log of the policy distribution.

Here, we assume a Gaussian policy with 1 as the variance for each action variable, i.e., using an identity matrix as the covariance.

Training with advantage-weighted samples

The sampling mean output by the policy network.

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

Because of the choice of variance, we only have a term for the the mean.

Training with advantage-weighted samples

Neural network parameters that we optimize

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

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

Training with advantage-weighted samples

Squared distance between the policy and actions

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

In this case, the loss becomes 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

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

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.

Updating both mean and variance

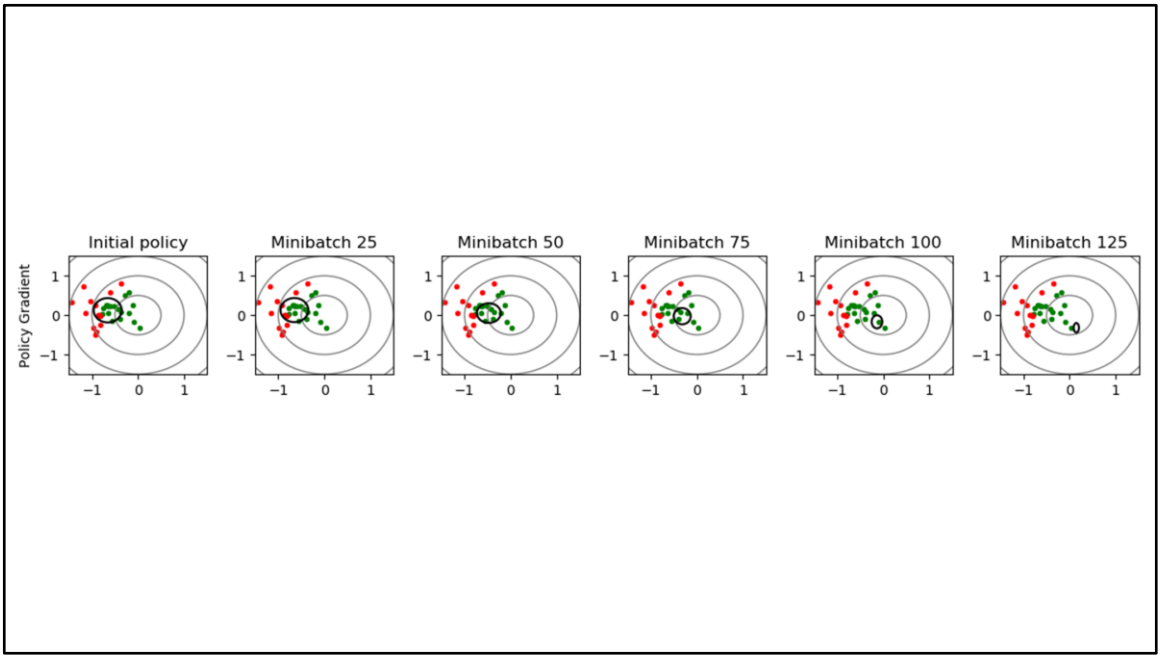
$$\mathcal{L} = \frac{1}{M}\sum_{i=1}^{M} A^{\pi}(\mathbf{s}_i, \mathbf{a}_i) \sum_j \left[\frac{(a_{i,j} - \mu_{j;\theta}(\mathbf{s}_i))^2}{c_{j;\theta}(\mathbf{s}_i)} + 0.5 \log c_{j;\theta}(\mathbf{s}_i) \right]$$

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.

Math part ends here

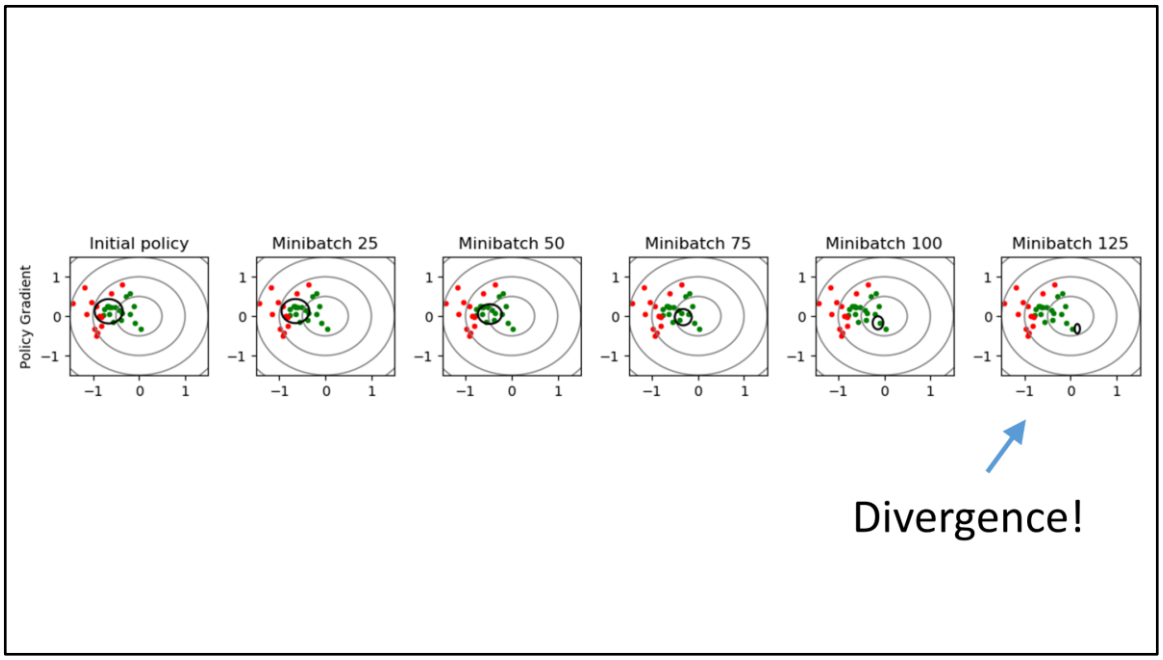


Let's see how this works when we train with the loss function for some minibatch Adam steps, using the samples collected in a single iteration of the algorithm. The policy gravitates towards the positive-advantage actions shown in green, and away from the negative-advantage actions shown in red.

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

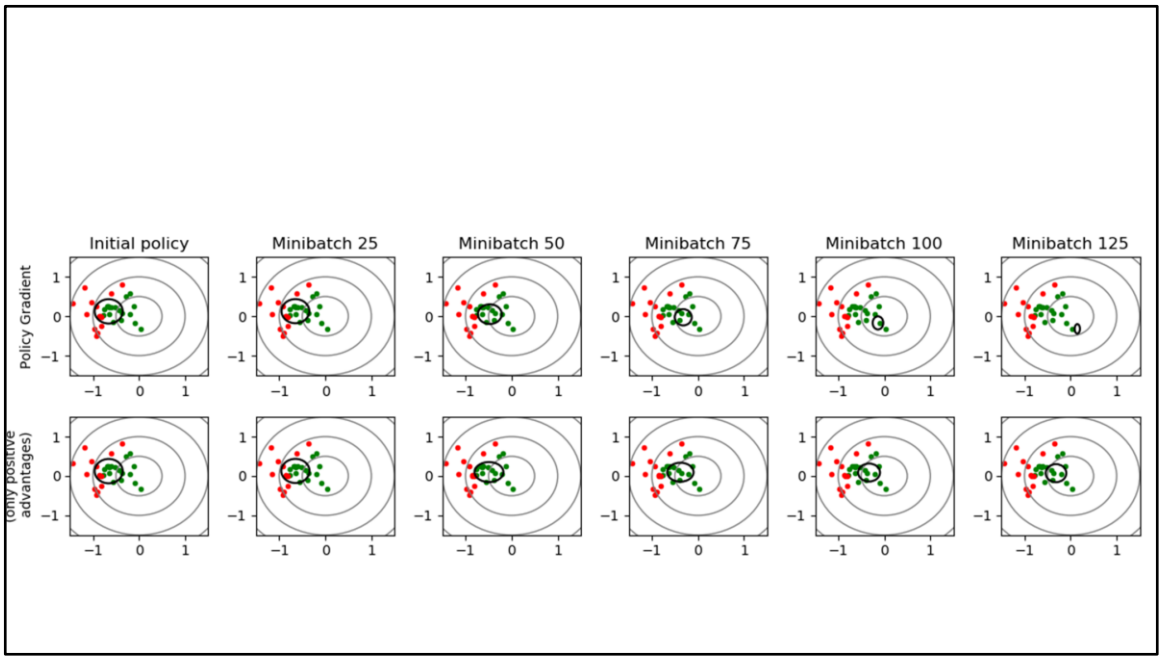
Black ellipse: one standard devation of the policy Gaussian

Here, we visualize only the actions of a single state in a hypothetical simple example.



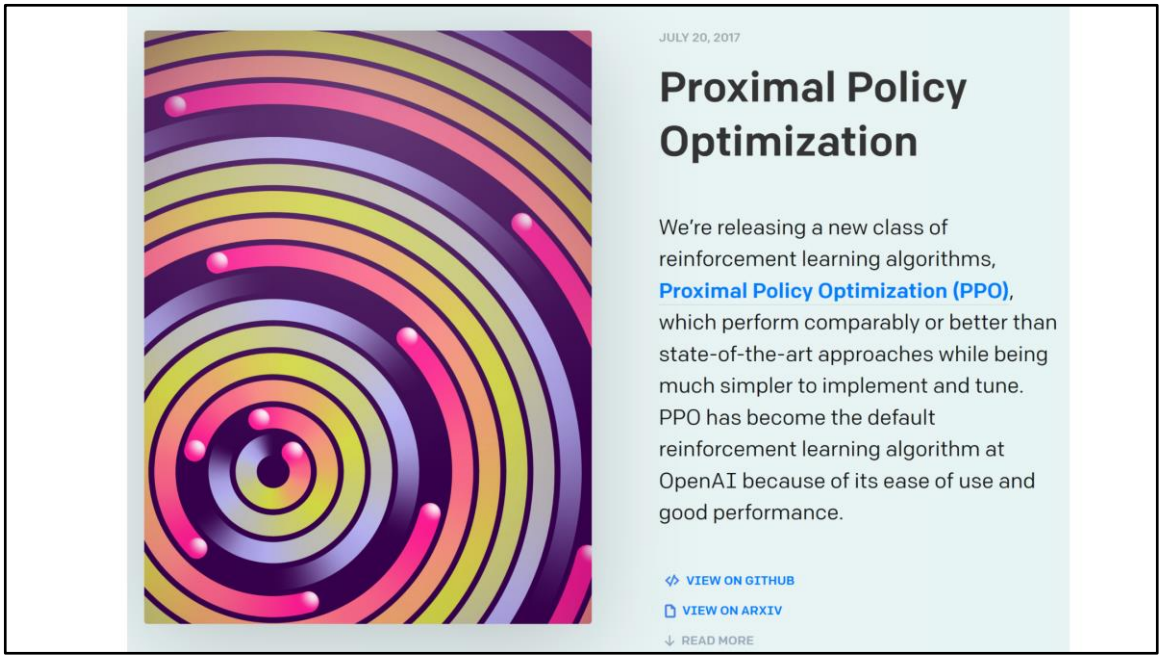
However, it is unstable if one trains for multiple minibatch gradient steps. Fine-tuning the step-size and minibatch count can be hard.

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.



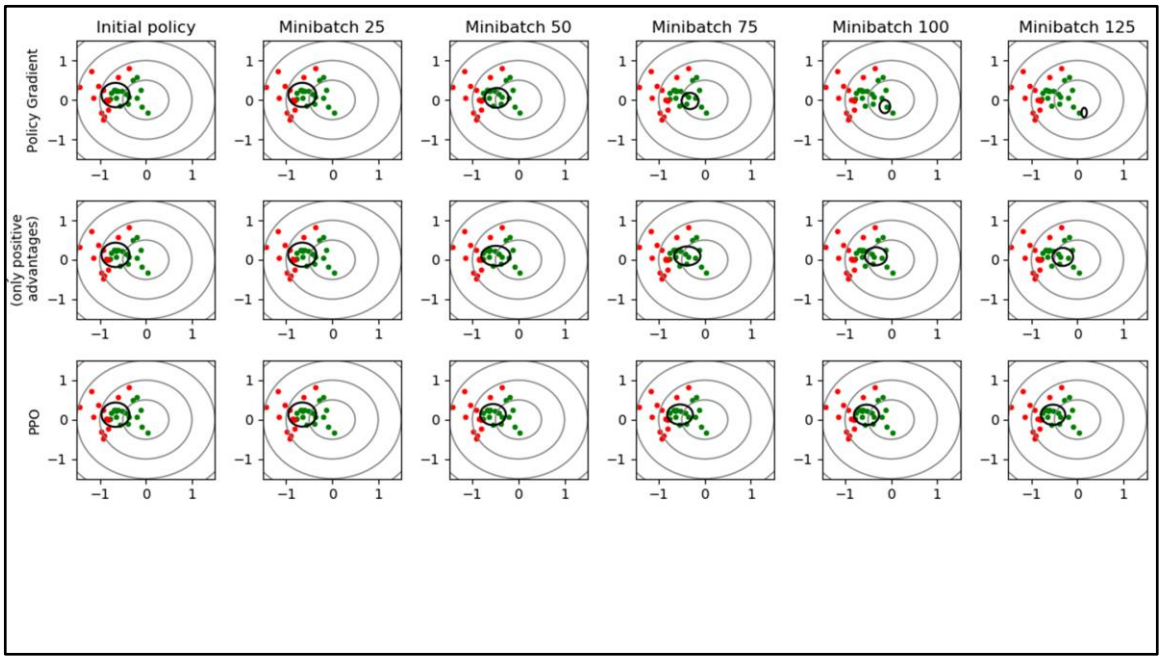
The problem is caused by the actions with negative advantages, which keep pushing the policy further away with each minibatch. From the weighted policy fitting perspective, using negative weights does not really make sense, and training can be made stable by simply discarding the negative advantage actions, or equivalently, setting their weight to 0.

However, this discards information; it should be possible to use negative-advantage actions instead of just discarding them.



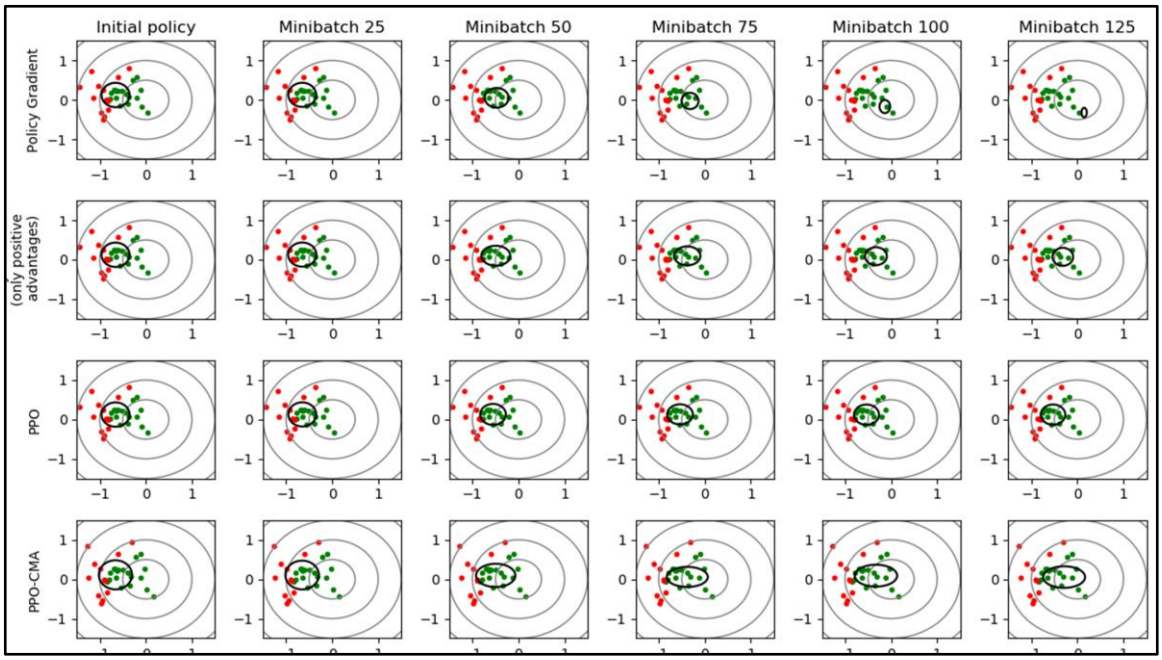
In 2017, the stability problem was addressed by Proximal Policy Optimization (PPO), which quickly became immensely popular. It is used in OpenAI's Dota AI and it's also the default in Unity's Machine Learning Agents. PPO is a simplified version of an earlier method called Trust-Region Policy Optimization

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



PPO does not discard the negative-advantage actions, but instead removes the instability by using a 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 extended PPO with 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.

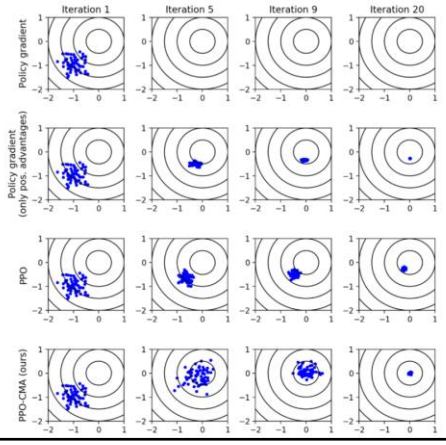
PPO-CMA can still be considered a proximal policy optimization method, though, as the policy will not diverge outside the sampled actions.

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

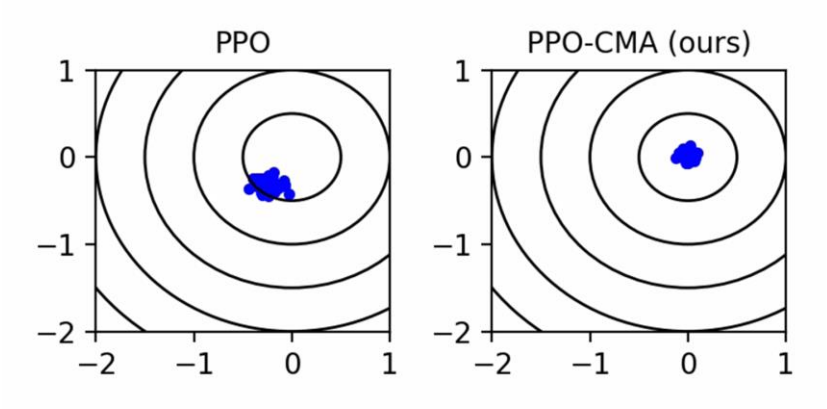
https://arxiv.org/pdf/1810.02541.pdf

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



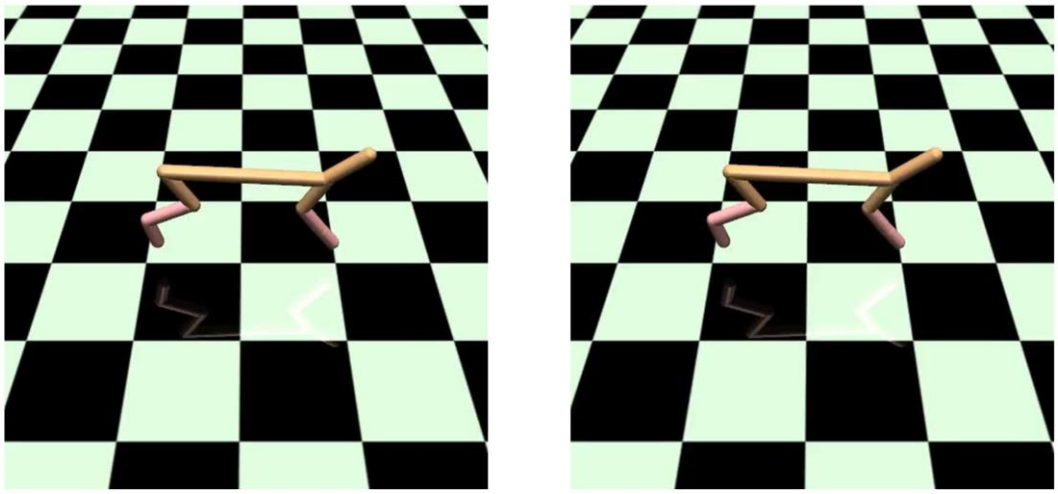
Here's the key result as an animation.

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

What does all this mean?

- RL = try out random actions, then try variations of the actions that yield good results. Can be very similar to CMA-ES, but with neural networks added.
- To make progress, the random sampling must yield at least some good actions!
- Thus, one needs:
  - A well-designed reward function
  - Lots of samples in difficult problems
- There's no guarantee of success, and like other optimization methods, RL can get stuck in a local optimum

Getting stuck to a local optimum

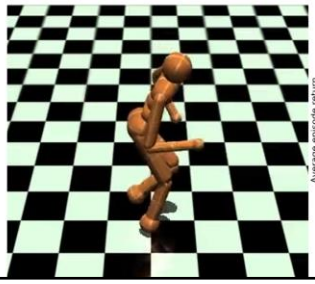


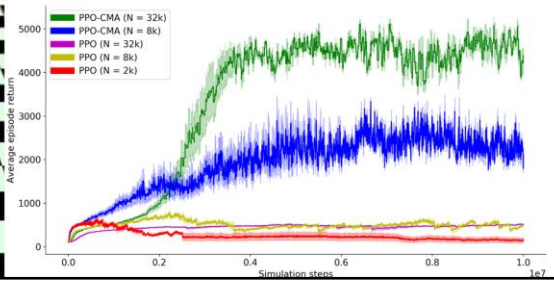
Source: https://www.alexirpan.com/2018/02/14/rl-hard.html

Solutions: redesign the reward, increase iteration simulation budget, or try a different and supposedly better algorithm

From CMA-ES to Deep Reinforcement Learning

- Recap: CMA-ES
- Optimizing actions conditional on observed state
- **Optimizing action sequences**
- Common problems and tricks of the trade





So far, our algorithm and visualizations have focused on the simple case of optimizing the reward of a single action in each state.

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 **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 **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*

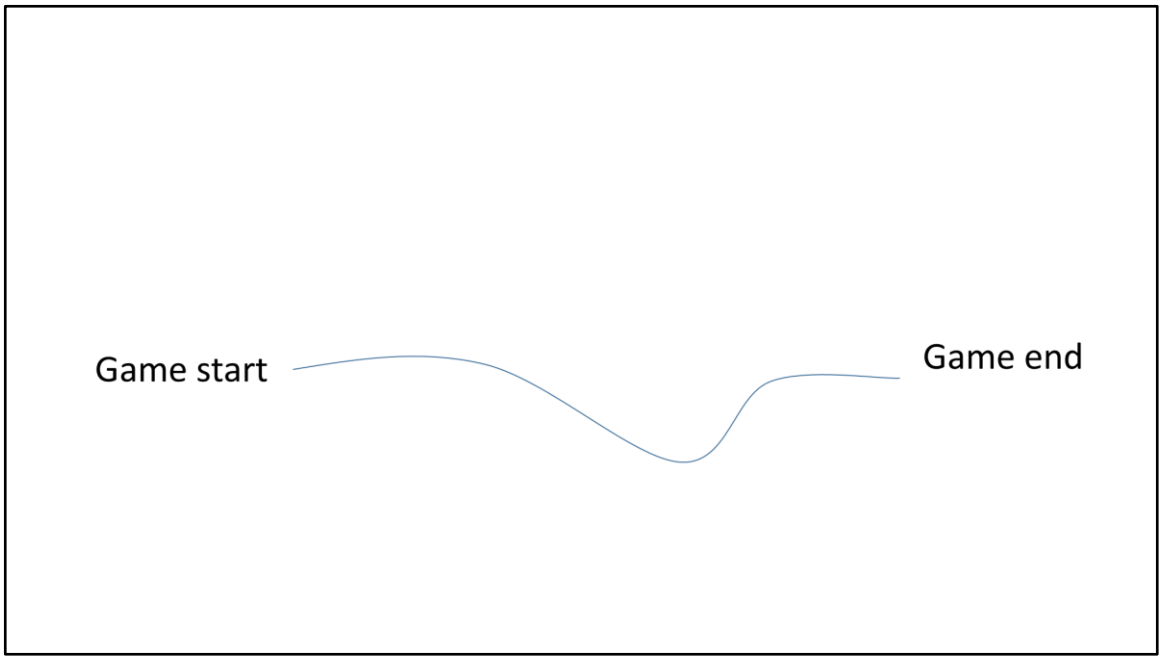
One episode

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

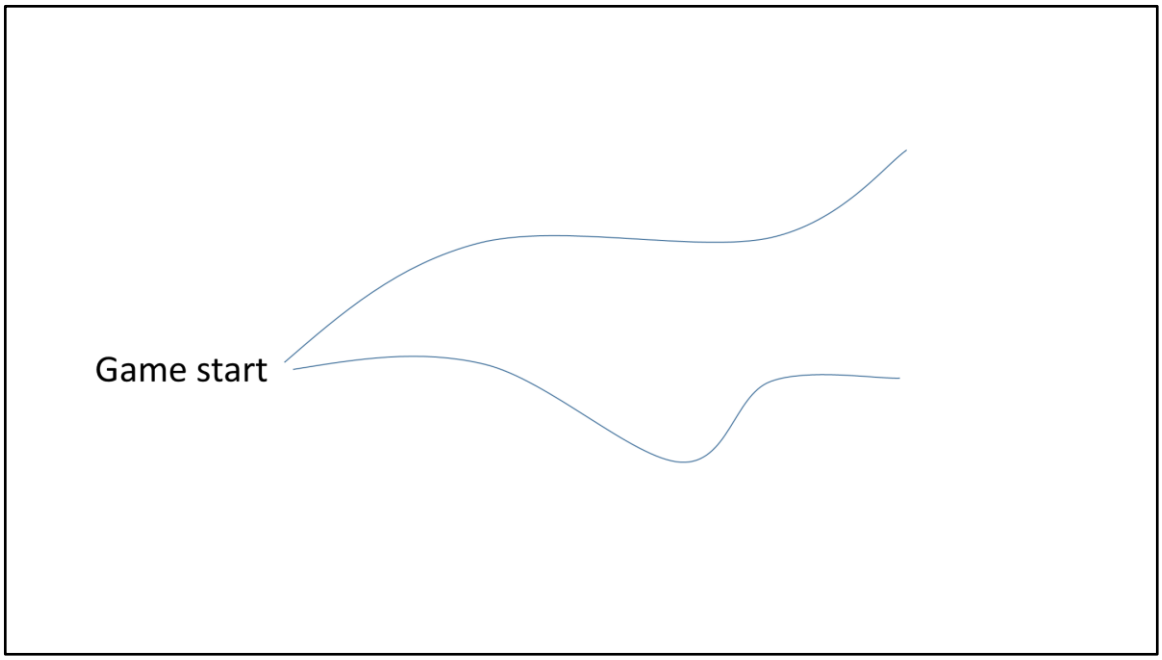
Objective: instead of *r*(**a**,**s**), maximize $\mathbb{E}\left[\sum_t \gamma^t r(\mathbf{a}_t, \mathbf{s}_t)\right]$, where *t* is time

We want to maximize the expected cumulative reward. In the formula, *t* denotes time and γ 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. γ adjusts the balance between short-term and long-term gains.

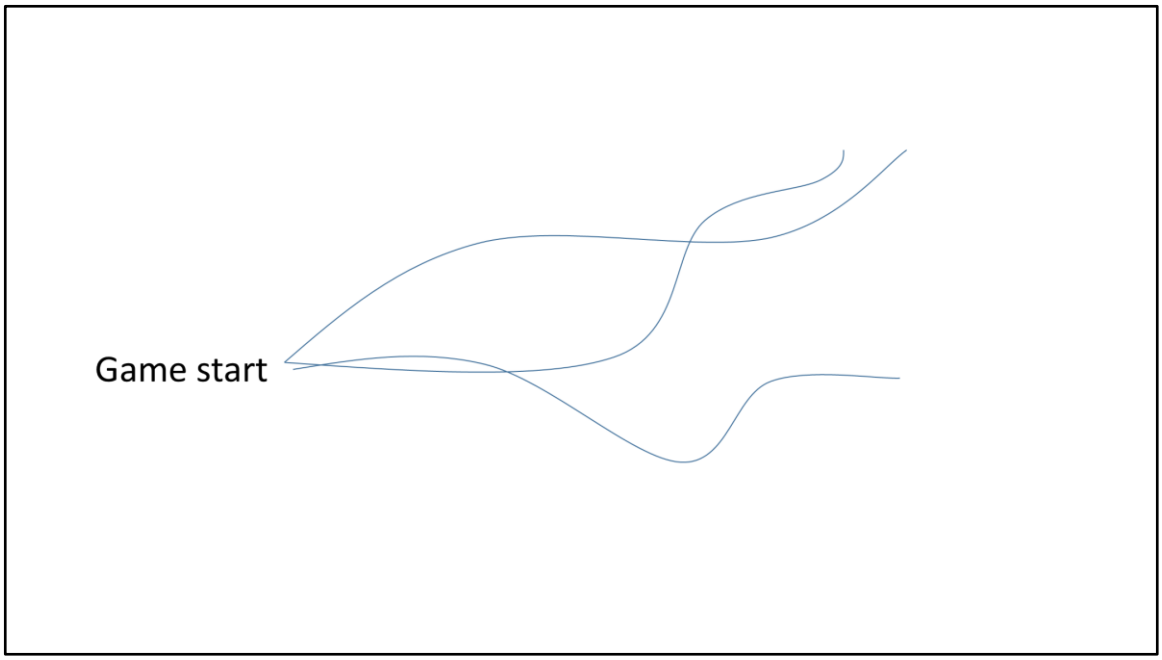
Again, the single action case discussed earlier is equal to this, with each episode having only one action. If *t* starts from zero, γ^t=1 for the first action.



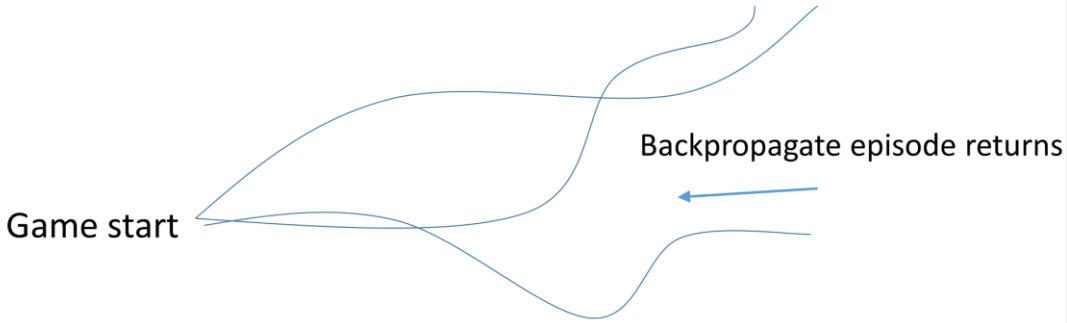
The episodes can be depicted as trajectories in the game state space



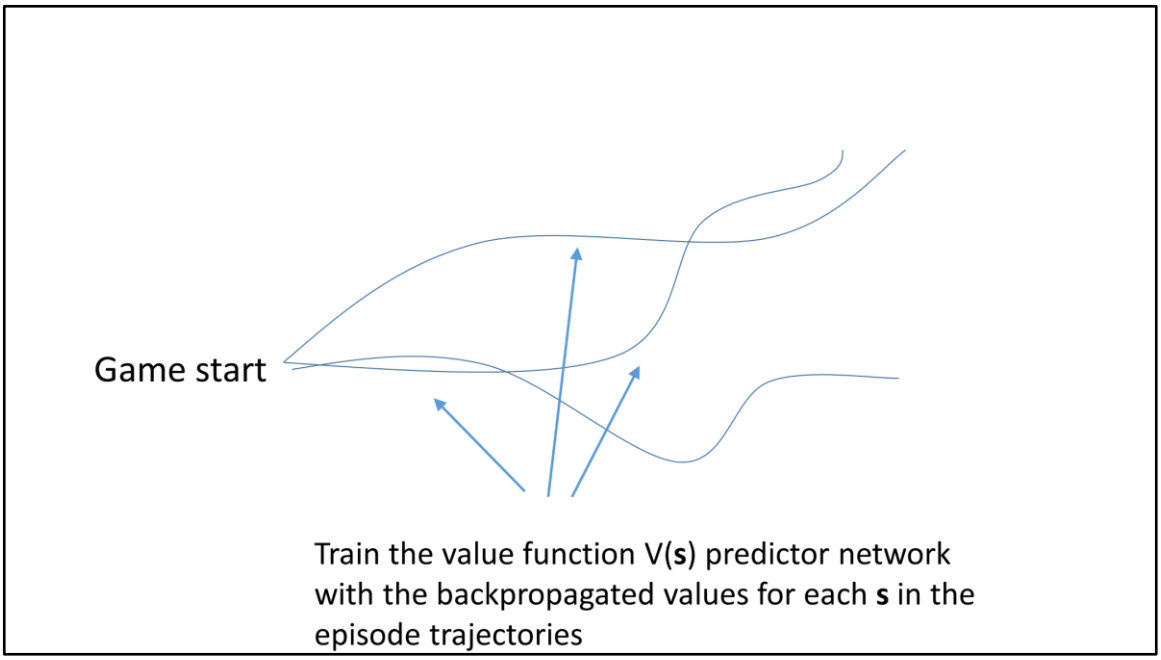
At each iteration, one executes one or more episodes, in this case plays a game one or more times


Game start

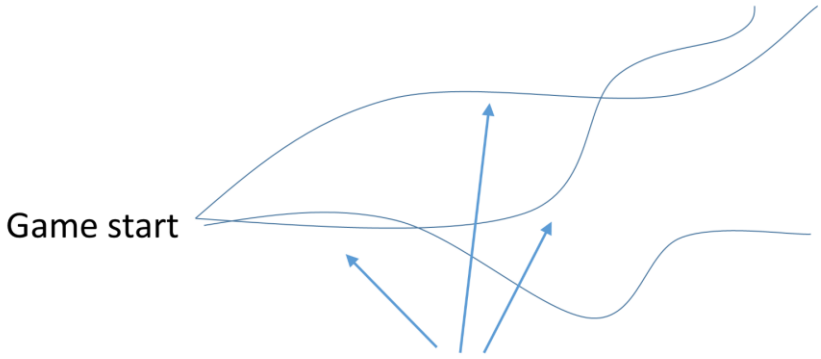
Again, I'm going to digress a bit into the technical details that are not necessary for applying the algorithms, but are useful for implementing and designing them.



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



V(**s**) is the expectation of the returns R(**a**,**s**)=r(**a**,**s**) + γ 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**)+ γ V(**s**') (Bellman recursion)

Here, V(**s**) is approximated by the value function predictor network. V(**s**) is the expected return for state **s**, assuming that one samples the actions from the policy. Q(**a**,**s**) equals the expected return 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

Training loss

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

After the advantages have been estimated, one can use the same policy gradient formula as in the single action case.

Back to the high-level part...

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*

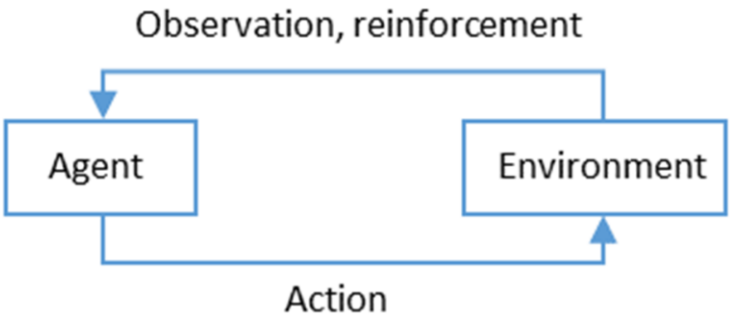
One episode

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

Sampling the initial state = start game level.

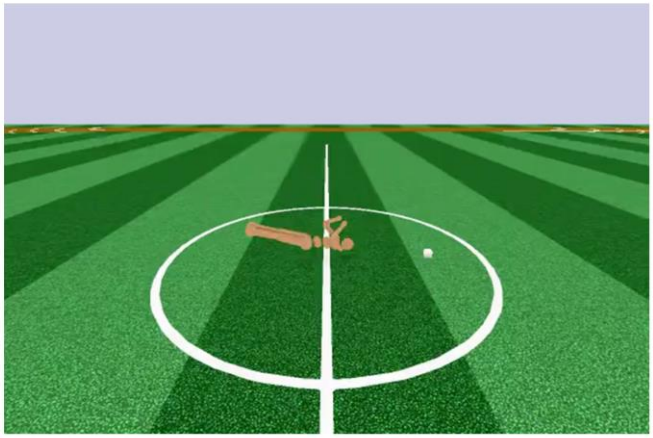
Terminal state = agent dies or completes the level.

Reinforcement learning



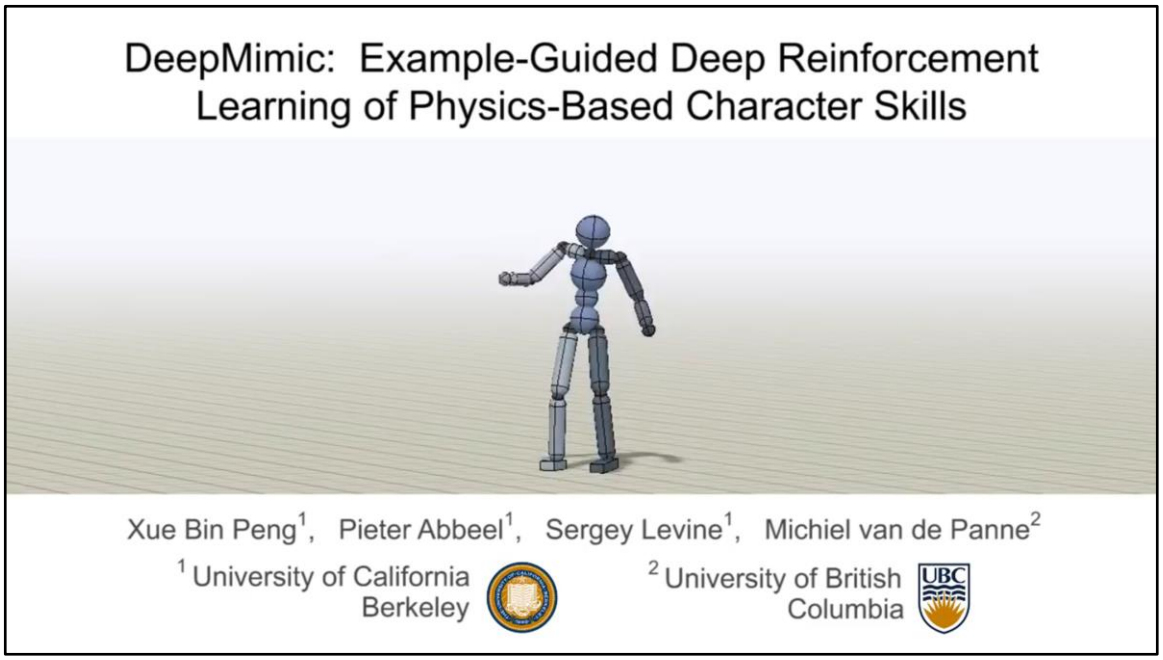
Basically, all you need to do is create an environment that takes in actions and outputs rewards and state observations

Reinforcement learning



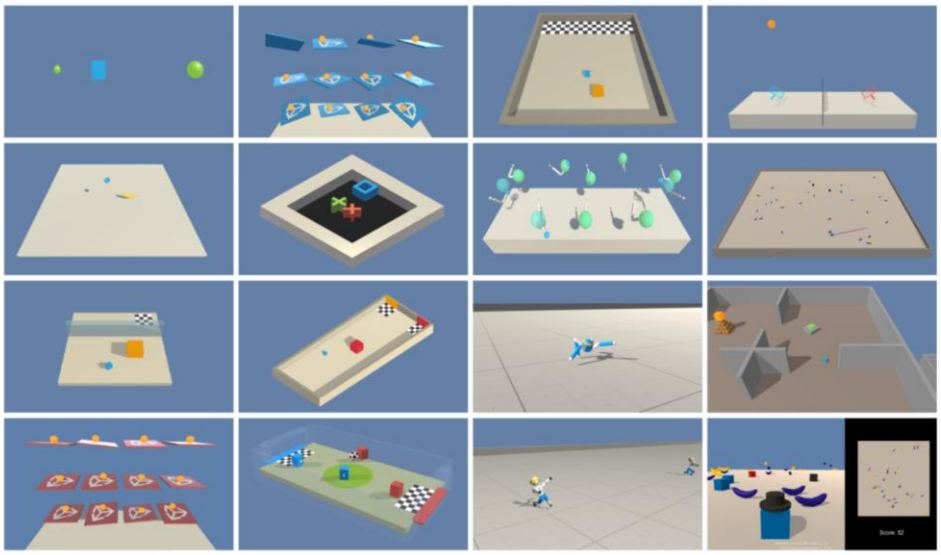
Environments commonly used by researchers include retro Atari games as well as humanoid locomotion tasks.

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. As the learning algorithm, this paper uses PPO.

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



Unity ML-Agents Toolkit (Beta)

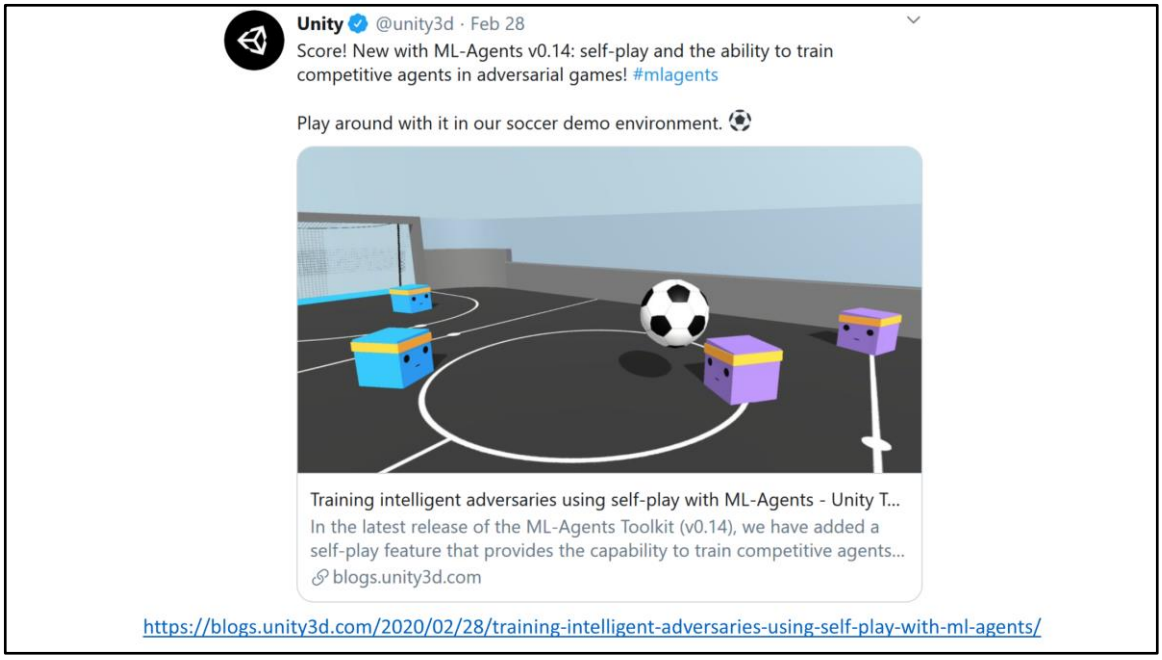
Unity ML Agents framework implements a number of agents and environments, and it's probably the easiest way to start learning how to use reinforcement learning.

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

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/



https://blogs.unity3d.com/2020/02/28/training-intelligent-adversaries-using-self-play-with-ml-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)
- Soft Actor Critic (Haarnoja et al. 2018)

Both these are available in Unity ML Agents.

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

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, from December 2018.

RERP: A new method from Google Deepmind (December 2018) that uses a somewhat similar, but mathematically more elaborate"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 γ 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
  - 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

- RL is 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.
- If PPO is very slow, try using the Gym wrapper of Unity ML and train using the Python code of some other algorithm.
- Exercise: check out Unity ML Agents, think of applications

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

Coffee break

From CMA-ES to Deep Reinforcement Learning

- Recap: CMA-ES
- Optimizing actions conditional on observed state
- Optimizing action sequences
- **Common problems and tricks of the trade**

Finally, I'll continue a bit on some practical knowledge that is hard to glean from the papers. Basically, these are issues that I have often discussed with students

Checklist for avoiding common pitfalls

1. Everything that your reward and next state depend on must be included in the observed state.
2. Ensure sufficiently large iteration simulation budget (PPO, PPO-CMA) or experience replay buffer (SAC, DQN). For example, humanoid movement needs 32k, sometimes 64k simulation steps per iteration with PPO and PPO-CMA
3. Try to make your episode initial state distribution cover the all the states that your agent needs to act in
4. Ensure that your rewards are in a reasonable range, e.g., -1...1
5. Consider terminating episodes at clear failures, but if you do, ensure that your rewards are never negative
6. Each iteration must result in discovering rewards that guide the learning (i.e., not just zero or random rewards)
7. Your initial policy distribution must be able to generate all actions of interest.
8. Use an effective action parameterization, e.g., joint angles instead of torques
9. Use an effective action frequency
10. Use correct network architecture for your state observations and actions, e.g., convolutional network if state is observed as images.
11. Use reference data if you can, e.g., like in DeepMimic
12. Don't forget: you get what you reward, often with some surprises.

Failing to comply with these requirements can all easily lead to the agent not learning anything.

If the next state or reward depend on variables not included in the state and action, you have a partially observed policy and need a recurrent policy network

TODO: one slide per checklist item, with explanations and examples