Autonomous and Adaptive Systems

Policy Gradient Methods

Mirco Musolesi

mircomusolesi@acm.org

Policy Gradient Methods

- In the previous lectures we discussed methods that were based on the calculation of action values. We refer to them as as *action-value methods*.
- We learned the values of the actions and then select actions based on the estimated action values. The learning can happen with the optimisation of the policy at the same time (e.g., control problem), but we need the values of the actions in the first place.
- We now consider a different type of methods that learn instead a parametrised policy that can select actions *without consulting a value function*.
- A value function can be used to learn the policy parameter, but it is not required for action selection.

Policy Gradient Methods

- We use the notation $\theta \in \mathbb{R}^{d'}$ for the policy's parameter vector.

- We indicate the probability that action a is taken at time t given that the environment is in state s a time t with parameters θ as follows:

$$\pi(a \mid s, \theta) = Pr\{A_t = a \mid S_t = s, \theta_t = \theta\}$$

- If a method uses a learned value function as well, the value function's weight vector is denoted with $\mathbf{w} \in \mathbb{R}^d$.

Policy Gradient Methods

- We consider methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\theta)$ with respect to the policy parameter.

- The goal of these methods is to maximise performance, so their updates approximate gradient ascent in J as follows:

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t)$$

where $\nabla \hat{J}(\theta_t) \in \mathbb{R}^d$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to the argument θ_t.

- We refer to all the methods that follow this schema as *policy gradient methods*.

Policy Approximation

- In policy gradient methods, the policy can be parameterised in several possible way.

- The only constraint is that $\pi(a \mid s, \theta)$ is differentiable, i.e., as long as $\nabla \pi(a \mid s, \theta)$ with respect to θ exists and it is finite for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$ with $\theta \in \mathbb{R}^{d'}$.

- We need to ensure exploration and, therefore, one goal is to make sure that the policy will never become deterministic, i.e. that $\pi(a \mid s, \theta) = (0,1)$ for all s, a, θ.

Policy Parametrisation and Parametrised Action Preferences

- A standard way for parametrisation of policies is to derive first parametrised numerical preferences $h(a \mid s, \theta) \in \mathbb{R}$ for each state-action pair.

- The actions with the highest preferences in each state are given the highest probabilities of being selected.

- A typical mapping between the preferences $h(a \mid s, \theta)$ and the probabilities $\pi(a \mid s, \theta)$ is obtained through the use of an exponential softmax distribution:

$$\pi(a \mid s, \theta) \doteq \frac{e^{h(a|s,\theta)}}{\sum_b e^{h(b|s,\theta)}}$$

- We call this type of policy parametrisation *softmax in action preferences*.

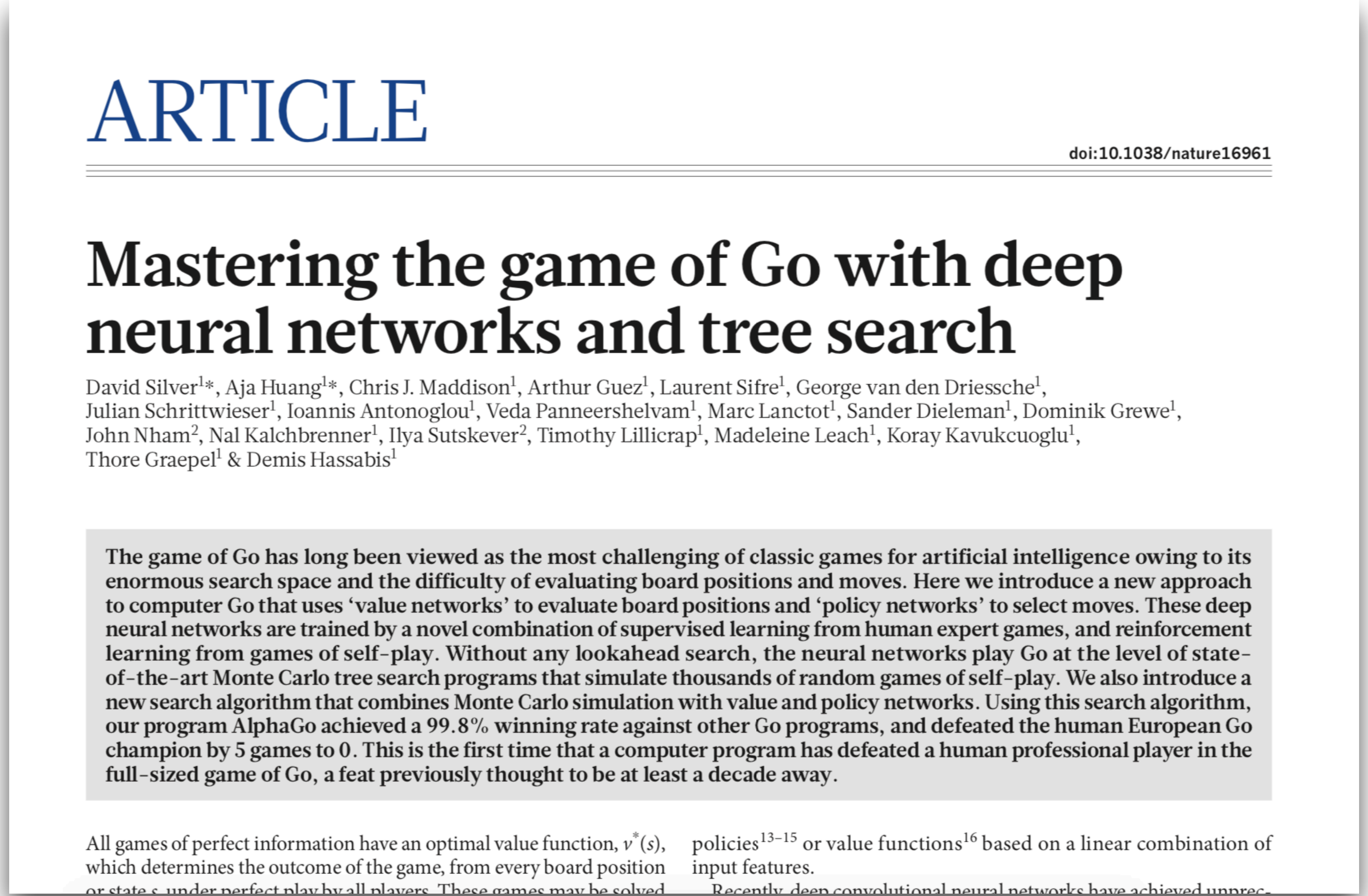
- Note that we are not deriving the value functions and then apply a policy (let's say ϵ-greedy). We are deriving directly the probability distributions of the actions given the states, i.e., the policy itself.

Policy Parametrisation and Parametrised Action Preferences

- The action preferences themselves $h(a \mid s, \theta)$ can be parametrised arbitrarily.

- For example, they might be computed by a deep artificial neural network (ANN), where θ is the vector of all the connection weights.

- This is used for example in the AlphaGo system.



ARTICLE

doi:10.1038/nature16961

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

All games of perfect information have an optimal value function, $v^*(s)$, which determines the outcome of the game, from every board position or state s, under perfect play by all players. These games may be solved

policies[13–15] or value functions[16] based on a linear combination of input features.

Recently, deep convolutional neural networks have achieved unprec-

Advantages of Using Policy Approximation according to Softmax in Action Preferences

- One advantage of parameterising policies according to the softmax in action preferences is that the approximate policy can approach a deterministic policy.
- In fact, with ϵ-greedy selection over action values, there is always a probability ϵ of selecting a random action.
- One possibility is to use softmax distribution on the action values, but this will not allow to reach a deterministic policy.
    - Action values will always differ and, therefore, there will be always a non-null probability of selecting a different action.
    - Action preferences are different since they do not approach specific values: instead they are driven to produce the optimal stochastic policy.
    - If the optimal policy is deterministic, the preferences of the optimal actions will be driven infinitely higher than all suboptimal actions (the output of the softmax will be then very close to 1, i.e., close to determinism).

Advantages of Using Policy Approximation according to Softmax in Action Preferences

- The second advantage of parameterising policies according to the softmax in action preferences is that it enables the selection of actions with arbitrary probabilities.

- In some situations the best approximate policy might be stochastic, especially in games of imperfect information.

- Action-value methods do not have a natural way of finding stochastic optimal policies; instead, policy approximation methods can.

- It is also worth noting that in some cases, policy approximation might be easier than value approximation.

Advantages of Using Policy Approximation according to Softmax in Action Preferences

- Finally, there is an important “theoretical” advantage. With continuous policy parameterisation, the action probabilities change smoothly as a function of the learned parameter.

- Indeed, in ϵ-greedy selection the action probabilities might change dramatically for a small change in the estimated action values, if that change results in a different action having the maximal value.

Policy Gradient Theorem

- We consider episodic learning and we define the performance measure as the value at the start of the episode.

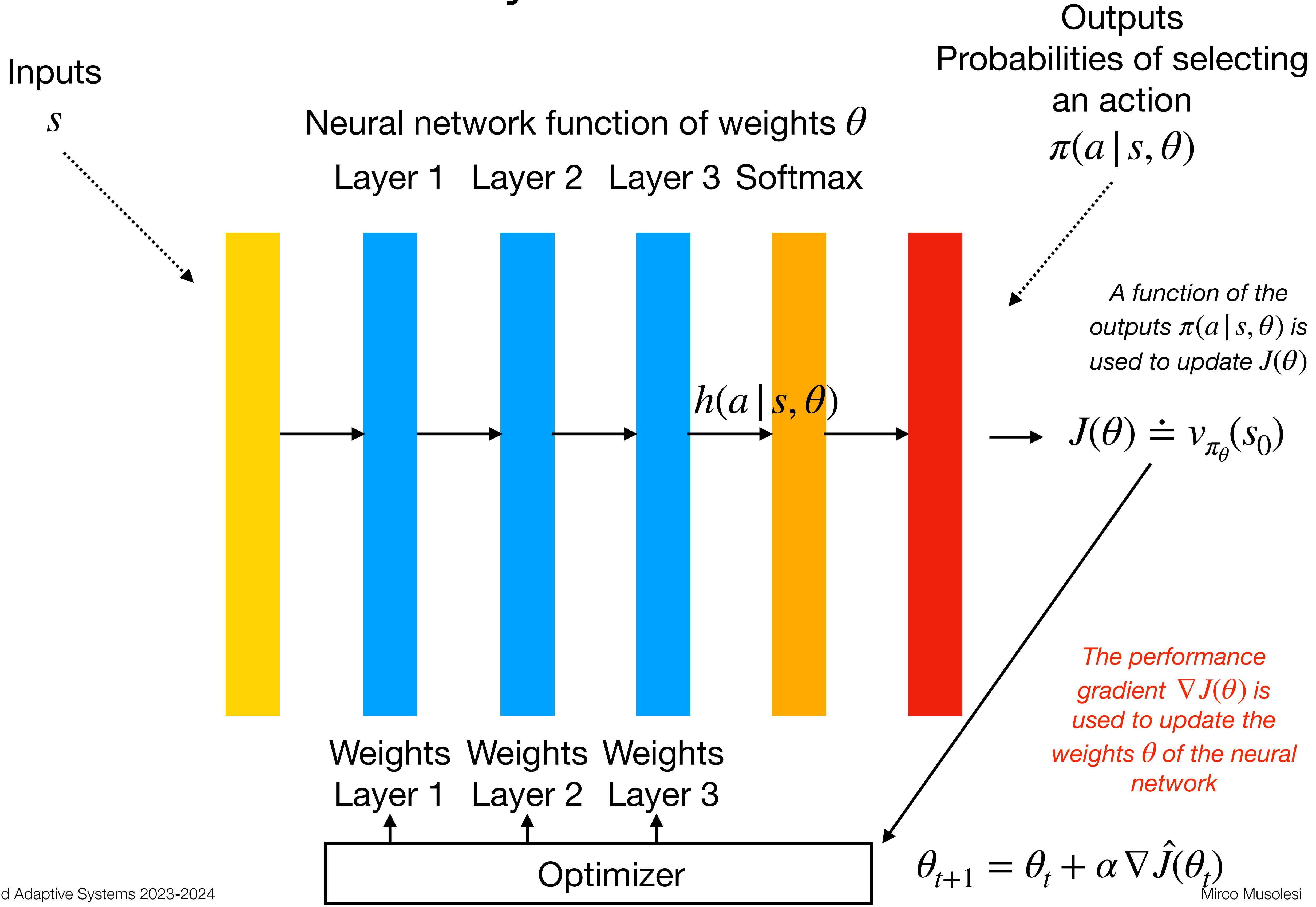
- We can simplify the notation by assuming that each episode starts in a non-random state s_0.

- Formally:

$$J(\theta) \doteq v_{\pi_\theta}(s_0)$$

where v_{π_θ} is the true value function for π_θ, the policy determined by θ.

Policy Networks



Inputs

s

Neural network function of weights θ

Layer 1 Layer 2 Layer 3 Softmax

Outputs

Probabilities of selecting an action

$\pi(a \mid s, \theta)$

A function of the outputs $\pi(a \mid s, \theta)$ is used to update $J(\theta)$

$h(a \mid s, \theta)$

$$J(\theta) \doteq v_{\pi_\theta}(s_0)$$

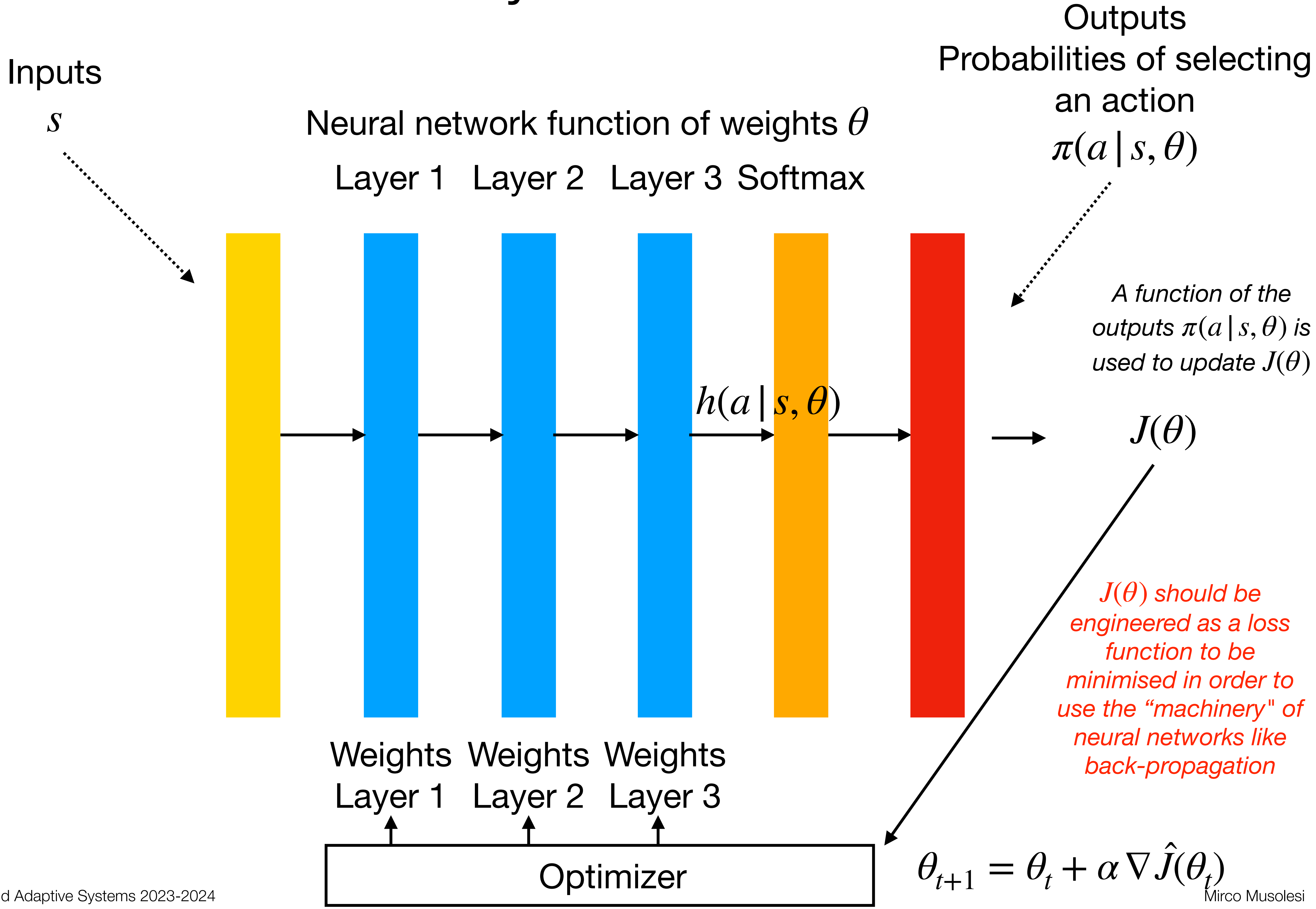
The performance gradient $\nabla J(\theta)$ is used to update the weights θ of the neural network

Weights Layer 1 Weights Layer 2 Weights Layer 3

Optimizer

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t)$$

Policy Networks

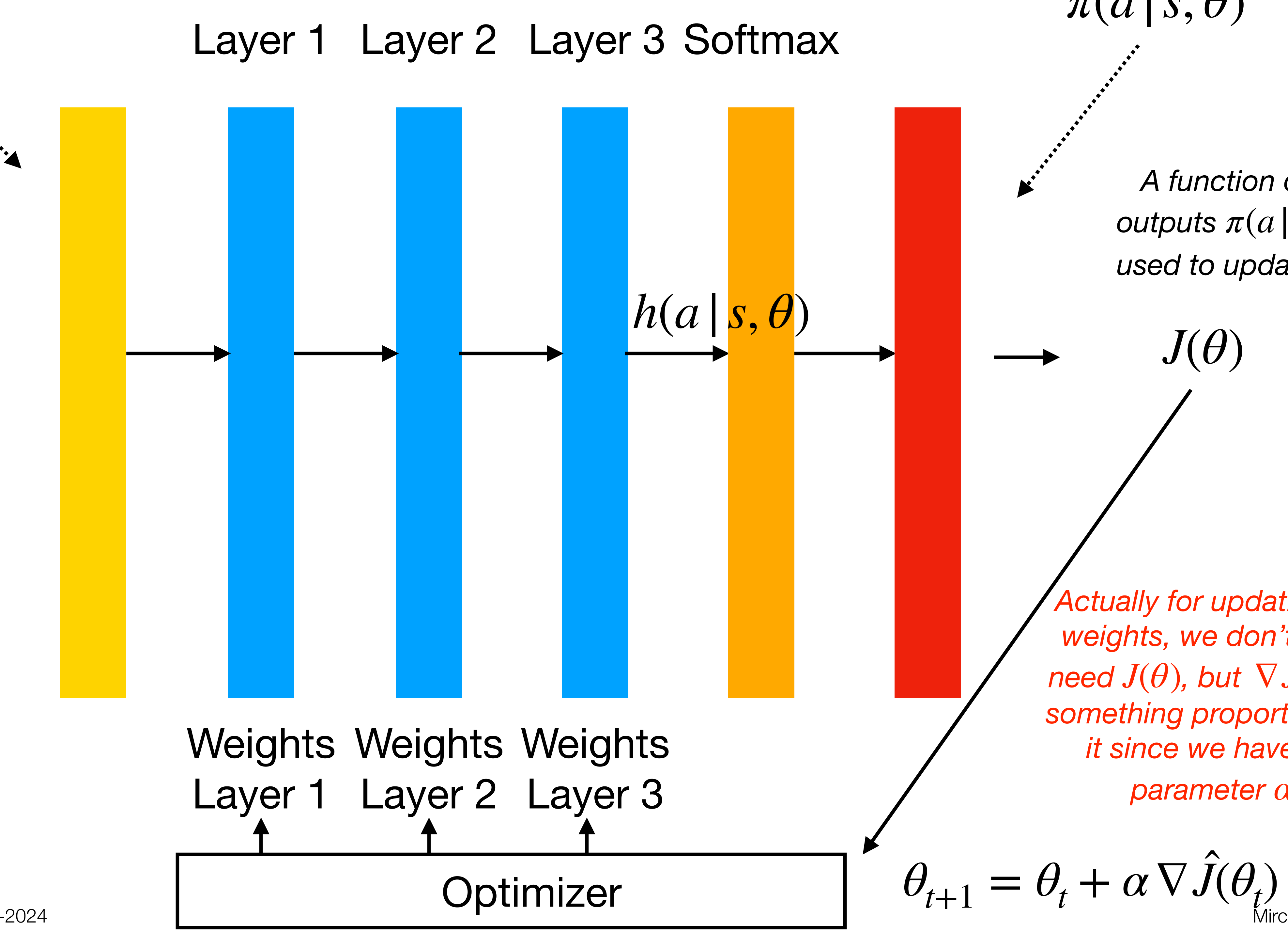


A function of the outputs $\pi(a \mid s, \theta)$ is used to update $J(\theta)$

$J(\theta)$ should be engineered as a loss function to be minimised in order to use the "machinery" of neural networks like back-propagation

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t)$$

Policy Networks



Inputs

s

Neural network function of weights θ

Layer 1 Layer 2 Layer 3 Softmax

Outputs
Probabilities of selecting an action
$\pi(a \mid s, \theta)$

A function of the outputs $\pi(a \mid s, \theta)$ is used to update $J(\theta)$

Actually for updating the weights, we don't really need $J(\theta)$, but $\nabla J(\theta)$ (or something proportional to it since we have the parameter α)

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t)$$

Policy Gradient Theorem

- Key question: how can we estimate the performance gradient with respect to the policy parameter when the gradient depends on the unknown effect of policy changes on the state distribution?

- Fortunately, the policy gradient theorem provides an analytic expression for the gradient of performance with respect to the policy parameter that does not involve the derivative of the state distributions.

- In particular, the policy gradient theorem says that:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a \mid s, \theta)$$

where the gradients are column vectors of partial derivatives with respect to the components of θ and π denotes the policy corresponding to parameter vector θ. $\mu(s)$ is the on-policy distribution under π (i.e., the fraction of time spent in each state normalised to sum to 1).

- If you are interested, you can find the proof in Chapter 13 of Sutton and Barto's book.

REINFORCE

- REINFORCE is a Monte Carlo Policy Gradient *control* algorithm, i.e., the update leads to the optimal policy.
- It is based on stochastic gradient ascent as discussed above.
    - We need a way to obtain samples such that the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter.
    - The sample gradients need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α, which is otherwise arbitrary.
    - The policy gradient theorem gives an exact expression proportional to the gradient.
        - We need to find a way of sampling whose expectation equals or approximates this expression.

REINFORCE

- Notice that the right-hand side of the policy gradient theorem is a sum over states weighted by how often the states occur under the target policy π. If π is followed, the states will be encountered in these proportions.

- More formally:

$$\begin{aligned}\nabla J(\theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a \mid s, \theta) \\ &= \mathbb{E}_\pi[\sum_a q_\pi(S_t, a) \nabla \pi(a \mid S_t, \theta)]\end{aligned}$$

REINFORCE

- In theory we can stop here and instantiate the stochastic gradient algorithm:

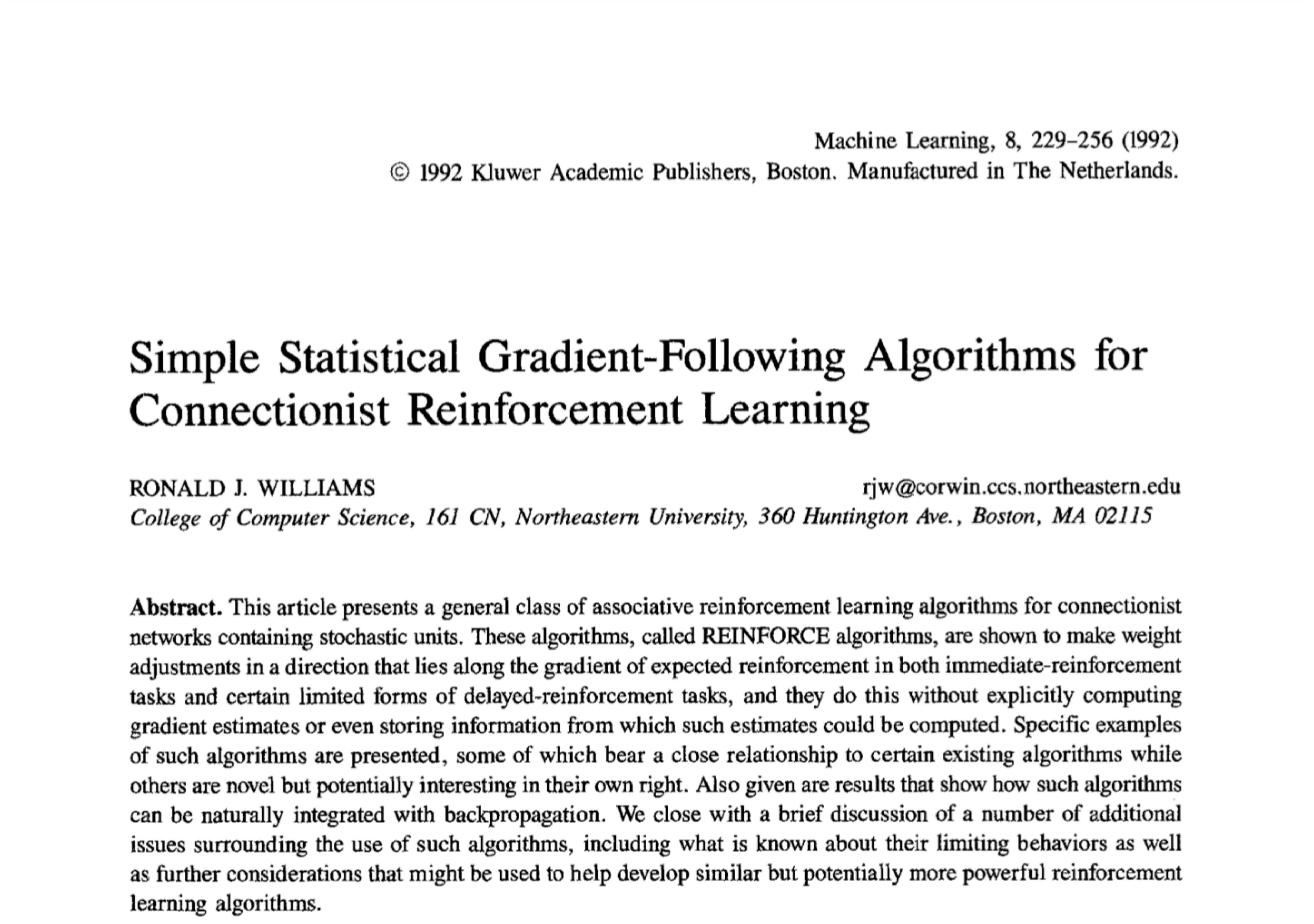
$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t)$$

as follows:

$$\theta_{t+1} \doteq \theta_t + \alpha \sum_a \hat{q}(S_t, a, \mathbf{w}) \nabla \pi(a \,|\, S_t, \theta)$$

where $\hat{q}$ is a learned approximation to q_π.

- This is called an all-action method because it involves the updates of all the actions.
- However, we are interested in an algorithm whose update at time t involves only A_t, the action taken at time t. This is called REINFORCE algorithm proposed by Williams in 1992.



Machine Learning, 8, 229–256 (1992)
© 1992 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

RONALD J. WILLIAMS rjw@corwin.ccs.northeastern.edu
College of Computer Science, 161 CN, Northeastern University, 360 Huntington Ave., Boston, MA 02115

Abstract. This article presents a general class of associative reinforcement learning algorithms for connectionist networks containing stochastic units. These algorithms, called REINFORCE algorithms, are shown to make weight adjustments in a direction that lies along the gradient of expected reinforcement in both immediate-reinforcement tasks and certain limited forms of delayed-reinforcement tasks, and they do this without explicitly computing gradient estimates or even storing information from which such estimates could be computed. Specific examples of such algorithms are presented, some of which bear a close relationship to certain existing algorithms while others are novel but potentially interesting in their own right. Also given are results that show how such algorithms can be naturally integrated with backpropagation. We close with a brief discussion of a number of additional issues surrounding the use of such algorithms, including what is known about their limiting behaviors as well as further considerations that might be used to help develop similar but potentially more powerful reinforcement learning algorithms.

REINFORCE

- We have the following derivation

$$\nabla J(\theta) \propto \mathbb{E}_\pi[\sum_a q_\pi(S_t, a)\nabla\pi(a|S_t,\theta)]$$

$$= \mathbb{E}_\pi[\sum_a \pi(a|S_t,\theta)q_\pi(S_t,a)\frac{\nabla\pi(a|S_t,\theta)}{\pi(a|S_t,\theta)}]$$

$$= \mathbb{E}_\pi[\mathbb{E}_\pi[q_\pi(S_t,A_t)]\frac{\nabla\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}] \qquad \text{since } \mathbb{E}_\pi[\mathbb{E}_\pi[x]] = \mathbb{E}_\pi[x]$$

$$= \mathbb{E}_\pi[q_\pi(S_t,A_t)\frac{\nabla\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}]$$

$$= \mathbb{E}_\pi[G_t\frac{\nabla\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}]$$

$$= \mathbb{E}_\pi[G_t\nabla ln\pi(A_t|S_t,\theta)]$$

where G_t is the return.

- Note that the last equality is true because $\mathbb{E}_\pi[G_t|S_t,A_t] = q_\pi(S_t,A_t)$ and $\nabla ln x = \frac{\nabla x}{x}$.

REINFORCE

- The final expression in brackets can be used for the update. It is a quantity that can be sampled at each step.

- Think about it in a different way: you are moving around your objective and on average your correction will lead you close to your real objective (the maximum in this case).

- Since you repeat stochastically this correction you end up with a correction that is close to the expectation of the gradient. For this reason we can use this sample to instantiate the generic stochastic gradient ascent.

REINFORCE

- As in the stochastic gradient descent, in the stochastic gradient ascent, you repeat the “correction” many times in order to reach the maximum.

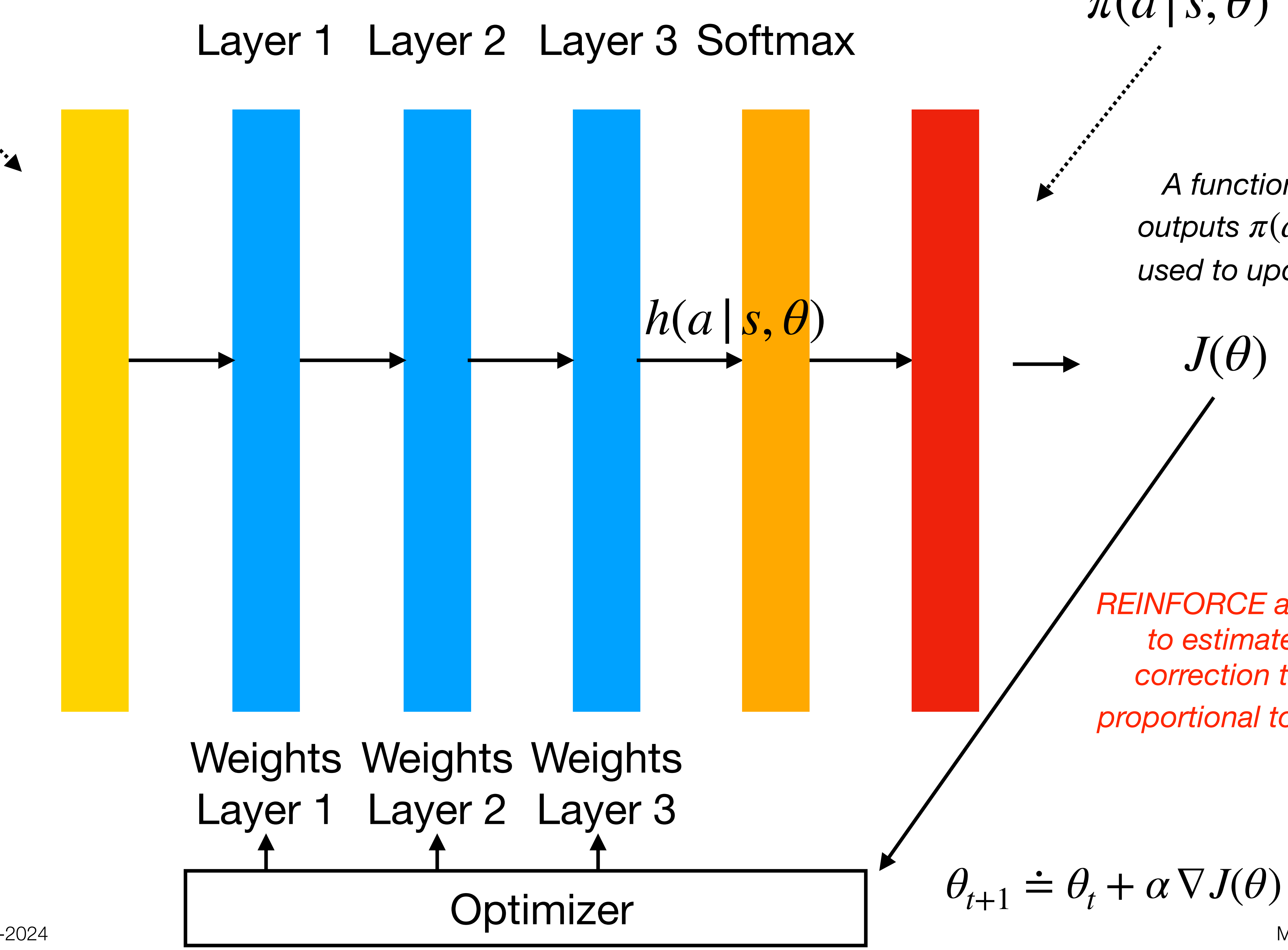
$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t)$$

- From the derivation above we have the following REINFORCE update:

$$\theta_{t+1} \doteq \theta_t + \alpha G_t \nabla ln\pi(A_t | S_t, \theta_t)$$

- Remember again that $\nabla J(\theta)$ is proportional to the update value, not equal, but we have the parameter α so it does not really matter.

Policy Networks



Inputs

s

Neural network function of weights θ

Layer 1 Layer 2 Layer 3 Softmax

Outputs

Probabilities of selecting an action

$\pi(a \mid s, \theta)$

A function of the outputs $\pi(a \mid s, \theta)$ is used to update $J(\theta)$

$h(a \mid s, \theta)$

$J(\theta)$

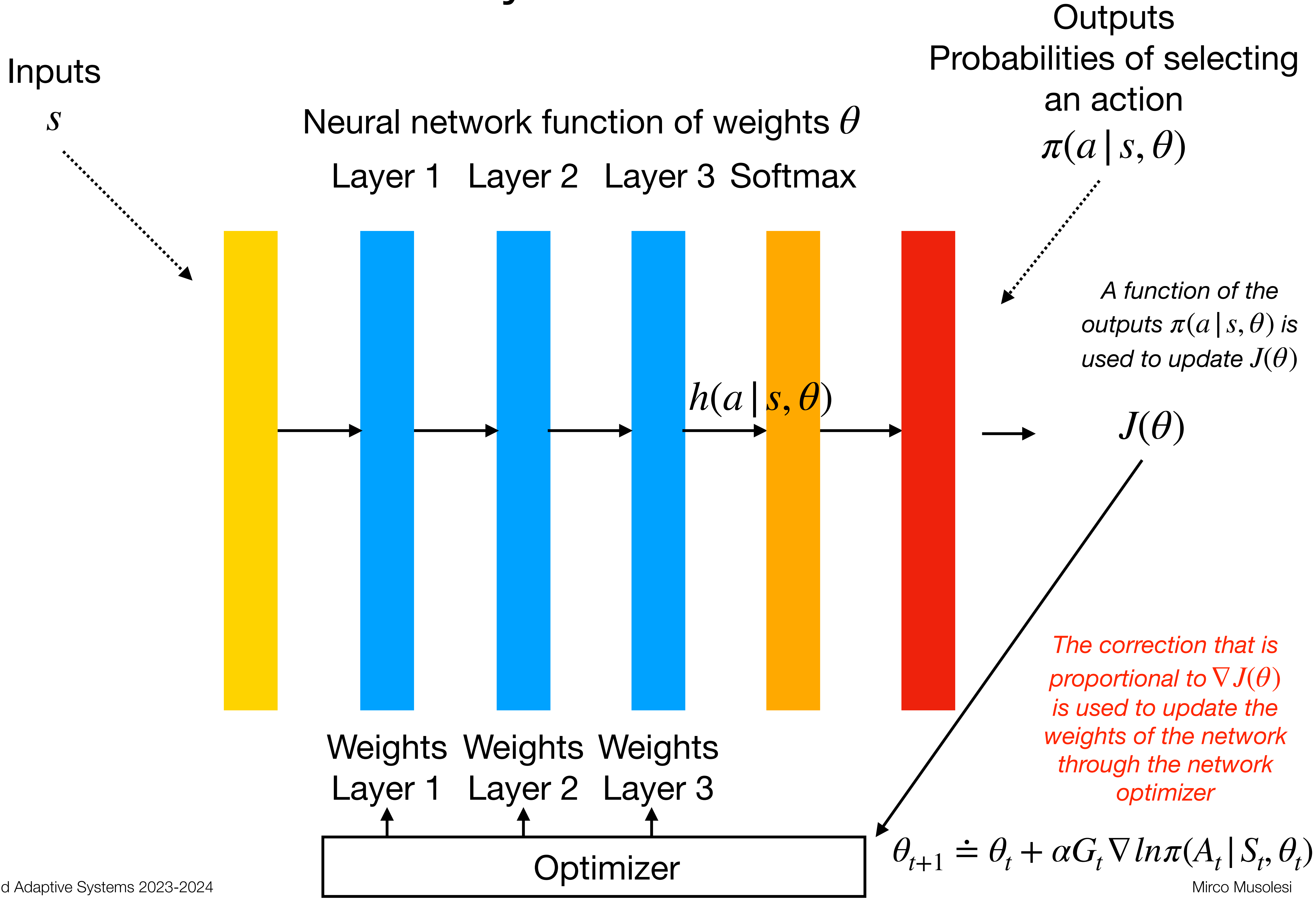
REINFORCE allows us to estimate the correction that is proportional to $\nabla J(\theta)$

Weights Layer 1 Weights Layer 2 Weights Layer 3

Optimizer

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla J(\theta)$$

Policy Networks



A function of the outputs $\pi(a \mid s, \theta)$ is used to update $J(\theta)$

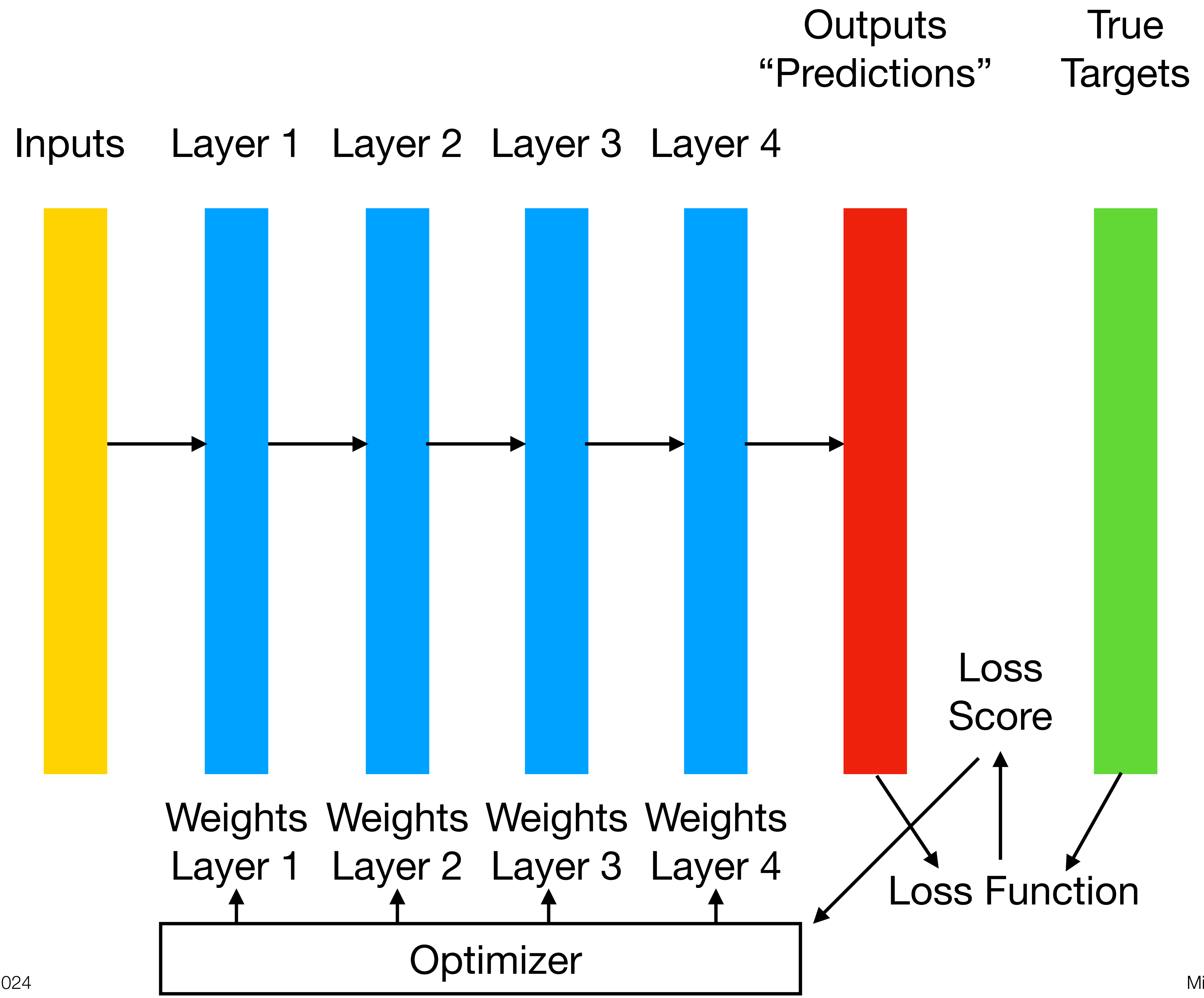
The correction that is proportional to $\nabla J(\theta)$ is used to update the weights of the network through the network optimizer

$$\theta_{t+1} \doteq \theta_t + \alpha G_t \nabla ln \pi(A_t \mid S_t, \theta_t)$$

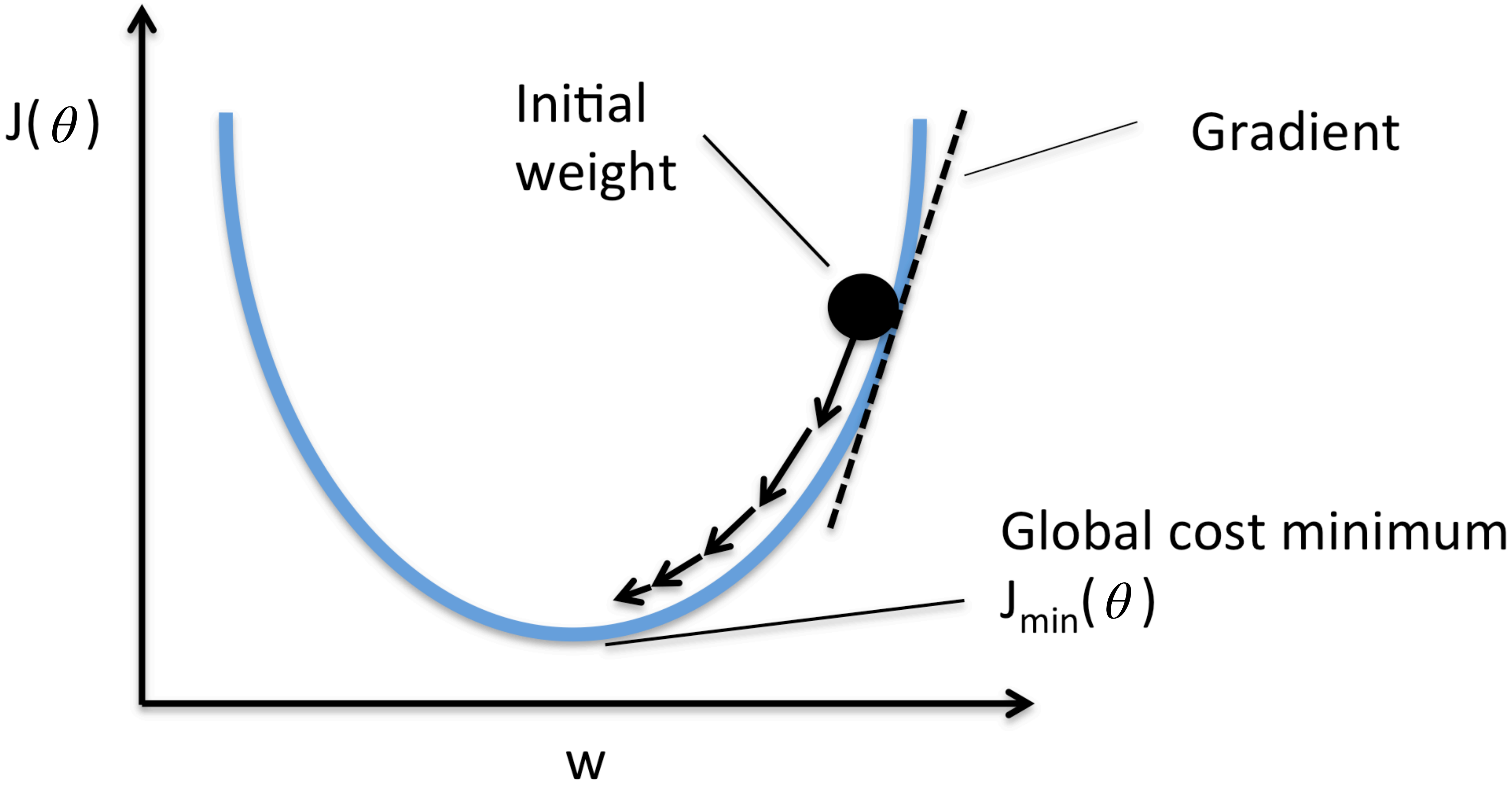
REINFORCE

```
Input: a differentiable policy parametrisation $\pi(a|s,\theta)$

Algorithm parameter: step size $\alpha > 0$

Initialise policy parameter $\theta \in \mathbb{R}^{d'}$

Loop forever (for each episode):

  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi$

  Loop for each step of the episode $t = 0,1,...,T-1$:

    $G_t \leftarrow \sum_{k=t+1}^{T} R_k$

    $\theta \leftarrow \theta + \alpha G_t \nabla ln\pi(A_t|S_t,\theta)$
```

Deep Neural Networks



Gradient-based Optimisation



Credit: Sebastian Raschka

Stochastic Gradient ***Descent***

- Recall the stochastic gradient *descent:*
  - Draw a batch of training example $\mathbf{x}$ and corresponding targets $\mathbf{y}_{target}$.
  - Run the network on $\mathbf{x}$ (forward pass) to obtain predictions $\mathbf{y}_{pred}$.
  - Computer the loss of the network on the batch, a measure of the mismatch between $\mathbf{y}_{pred}$ and $\mathbf{y}_{target}$.
  - Compute the gradient of the loss with regard to the network's parameters (backward pass).

Stochastic Gradient ***Descent***

- Move the parameters in the opposite direction from the gradient with:

$$\theta_j \leftarrow \theta_j - \Delta\theta_j = \theta_j - \eta \frac{\partial J}{\partial \theta_j}$$

where J is the loss (cost) function.

- If you have a batch of samples of dimension k:

$$\theta_j \leftarrow \theta_j - \Delta\theta_j = \theta_j - \eta \ average(\frac{\partial J_k}{\partial \theta_j})$$

for all the k samples of the batch.

Stochastic Gradient ***Ascent***

- Similarly in our case, we move the parameters in the same direction as the gradient with:

$$\theta_j \leftarrow \theta_j + \Delta\theta_j = \theta_j + \eta \frac{\partial J}{\partial \theta_j}$$

where J is the loss (cost) function.

- If you have a batch of samples of dimension k you can think about moving towards your actual objective using this formula:

$$\theta_j \leftarrow \theta_j + \Delta\theta_j = \theta_j + \eta \; average \left(\frac{\partial J_k}{\partial \theta_j} \right)$$

for all the k samples of the batch.

Implementing REINFORCE with Artificial Neural Networks/Deep Learning

- We said that an artificial neural network can be used to implement REINFORCE. But how can we do this in practice in a package like TensorFlow or Keras?
- The key problem is to select a loss function that can be mapped to REINFORCE.
  - We can talk about *compatibility* between Artificial Neural Networks and REINFORCE in a sense (see also theory in Williams 1992).
- From a practical point of view, we want to find a way of exploiting back-propagation for updating the weights using the “machinery” that is offered for example by the existing frameworks.
- We want to adapt a “machinery” that is built for stochastic gradient descent to a stochastic gradient ascent. As you can imagine the trick would be to do a correction in the opposite direction.

Implementing REINFORCE with Artificial Neural Networks/Deep Learning

- Let us consider again our formula: $\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t)$

- Our loss function is $J(\theta)$. Our goal is to correct each weight by $\nabla J(\theta)$ using the correction $\nabla J(\theta)$ for each weight.

- The value can be considered against the value 0, i.e., the difference $G_t ln \pi - 0$.

- However, since this is stochastic gradient *ascent* we will take the opposite of this quantity. $J'(\theta)$, the loss function we are going to use in Tensorflow, is the opposite of $J(\theta)$.

- More formally:

$$J'(\theta) = -(G_t ln \pi - 0) = -G_t ln \pi$$

Implementing REINFORCE with Artificial Neural Networks/Deep Learning

- And of course the gradient of this quantity is

$$\nabla J'(\theta) = - G_t \nabla ln\pi$$

- Essentially, for example in Keras (TensorFlow) it would be sufficient to set the loss function to a custom loss function equal to $-G_t ln\pi$.
- Then Keras will take care of the stochastic gradient ascent, i.e., optimisation problem.

Issues with REINFORCE

- As a stochastic gradient method, REINFORCE has good theoretical convergence properties.

- By construction, the expected update over an episode in the same direction as the performance gradient.

- This assures an improvement in expected performance for sufficient small α and convergence to a local optimum under standard stochastic approximation conditions for decreasing α.

- However, since it is a Monte Carlo method, REINFORCE suffers from high variance and this might lead to slow learning.

- One way of dealing with this problem is to use baselines and actor-critic methods.

REINFORCE with Baseline

- The policy gradient theorem can be generalised to include a comparison of the action value to an arbitrary baseline $b(s)$:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s,a) - b(s)) \nabla \pi(a \mid s, \theta)$$

- The baseline can be any function, even a random variable, as long as it does not vary with a.

- The equation remains valid because the subtracted quantity is zero:

$$\sum_a b(s) \nabla \pi(a \mid s, \theta) = b(s) \nabla \sum_a \pi(a \mid s, \theta) = b(s) \nabla 1 = 0$$

REINFORCE with Baseline

- The policy gradient theorem with baseline can be used to derive an update rule using similar steps.

- The update for REINFORCE with baseline is as follows:

$$\theta_{t+1} \doteq \theta_t + \alpha(G_t - b(S_t))\frac{\nabla \pi(A_t | S_t, \theta_t)}{\pi(A_t | S_t, \theta_t)}$$
$$= \theta_t + \alpha(G_t - b(S_t))\nabla ln\pi(A_t | S_t, \theta_t)$$

- The use of the baseline leaves the expected value unchanged, but it can have a large effect on its variance.

- The value of $b(s)$ can be a random number but it is not ideal.
  - In some states all actions have high values and we need a high baseline to differentiate the higher valued actions from the less highly values ones. In other states all actions will have low values and a low baseline is appropriate.

REINFORCE with Baseline

- A natural choice is to learn the state value at the same time;

$$b(S_t) = \hat{v}(S_t, \mathbf{w})$$

where $\mathbf{w} \in \mathbb{R}^m$ is a weight vector learned using, for example, another deep neural network.

- Because REINFORCE is a Monte Carlo method it makes to use a Monte Carlo method also for learning the state-value weights $\mathbf{w}$.

- This method will be based on two step-size denoted α^θ and $\alpha^\mathbf{w}$.

REINFORCE with Baseline

```
Input: a differentiable policy parametrisation $\pi(a \mid s, \theta)$ and a differentiable state-value function parametrisation $\hat{v}(s, \mathbf{w})$.

Algorithms parameters: steps size $\alpha^{\theta} > 0$ and $\alpha^{\mathbf{w}} > 0$

Initialise policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$

Loop forever (for each episode):

  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi$

  Loop for each step of the episode $t = 0,1,...,T-1$:

    $G_t \leftarrow \sum_{k=t+1}^{T} R_k$

    $\delta \leftarrow G_t - \hat{v}(S_t, \mathbf{w})$

    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t, \mathbf{w})$

    $\theta \leftarrow \theta + \alpha^{\theta}(G_t - \hat{v}(S_t, \mathbf{w})) \nabla ln \pi(A_t \mid S_t, \theta)$
```

REINFORCE with Baseline

```
Input: a differentiable policy parametrisation $\pi(a \mid s, \theta)$m a differentiable state-value function parametrisation $\hat{v}(s, \mathbf{w})$.

Algorithms parameters: steps size $\alpha^{\theta} > 0$ and $\alpha^{\mathbf{w}} > 0$

Initialise policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$

Loop forever (for each episode):

  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi$

  Loop for each step of the episode $t = 0,1,\ldots,T-1$:

    $G_t \leftarrow \sum_{k=t+1}^{T} R_k$

    $\delta \leftarrow G_t - \hat{v}(S_t, \mathbf{w})$          In the vanilla REINFORCE, i.e., REINFORCE without baseline, $\delta = G$.

    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t, \mathbf{w})$

    $\theta \leftarrow \theta + \alpha^{\theta}(G_t - \hat{v}(S_t, \mathbf{w})) \nabla ln \pi(A_t \mid S_t, \theta)$
```

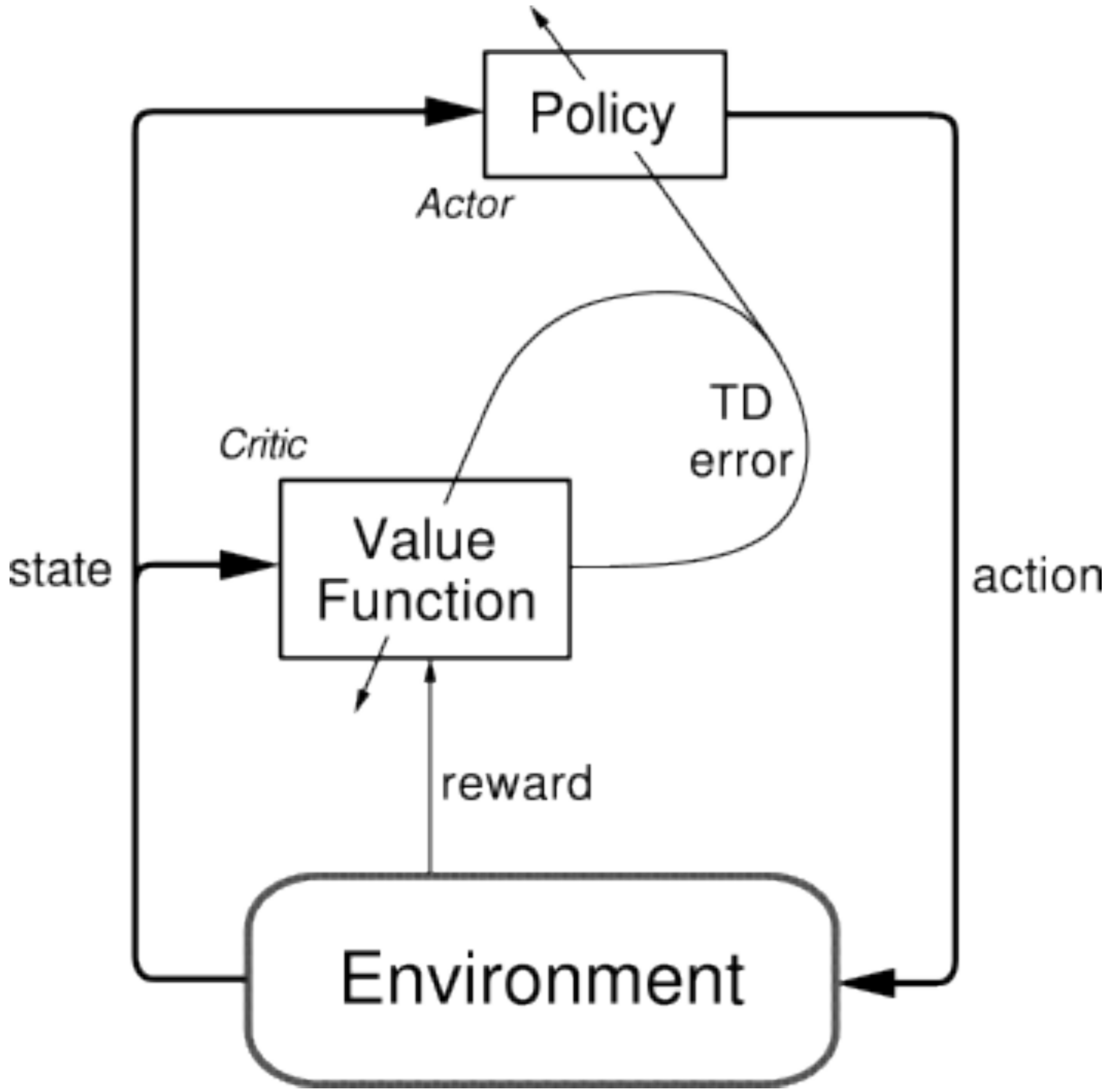
Actor-Critic Methods

- Methods that learn approximations to both policy and value functions are often called *actor-critic methods*, where *actor* is a reference to the learned policy and *critic* refers to the learned value function.
- We do not consider REINFORCE with baseline an *actor-critic method*, even if it learns both a policy and state-value functions.
- The reason is that its state-value function is used only as a baseline and not as a critic and not for updating the value estimate for a state from the estimated values of subsequent states (i.e., bootstrapping).
- This is an important distinction, since only through bootstrapping we introduce bias and an asymptotic dependence on the quality of the function approximation.
- Recall that the bias introduced through bootstrapping is often beneficial since it reduces variance and accelerates learning.

Actor-Critic Methods

- REINFORCE with baseline is unbiased and converges asymptotically to a local minimum, but as all the the other Monte Carlo methods tends to learn slowly (high variance in the estimates).
- We have seen that with temporal distance methods we can remove these problems.
- For this reason, we use actor-critic methods with a bootstrapping critic, i.e., we update the value estimate for a state from the estimated values of subsequent state.
- We consider one-step actor-critic methods.
  - These are analogs to TD(0), Sarsa and Q-learning.

Actor-Critic Methods



One-step Actor-Critic Method

- The one-step actor-critic method replaces the full return of REINFORCE with the one-step return (and use a learned state-value function as baseline) as follows:

$$\theta_{t+1} \leftarrow \theta_t + \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)}$$

$$\theta_{t+1} \leftarrow \theta_t + \alpha(R_{t+1} + \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)}$$

$$\theta_{t+1} \leftarrow \theta_t + \alpha\delta_t\frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)}$$

$$\theta_{t+1} \leftarrow \theta_t + \alpha\delta_t \nabla ln\pi(A_t | S_t, \theta)$$

One-step Actor Methods

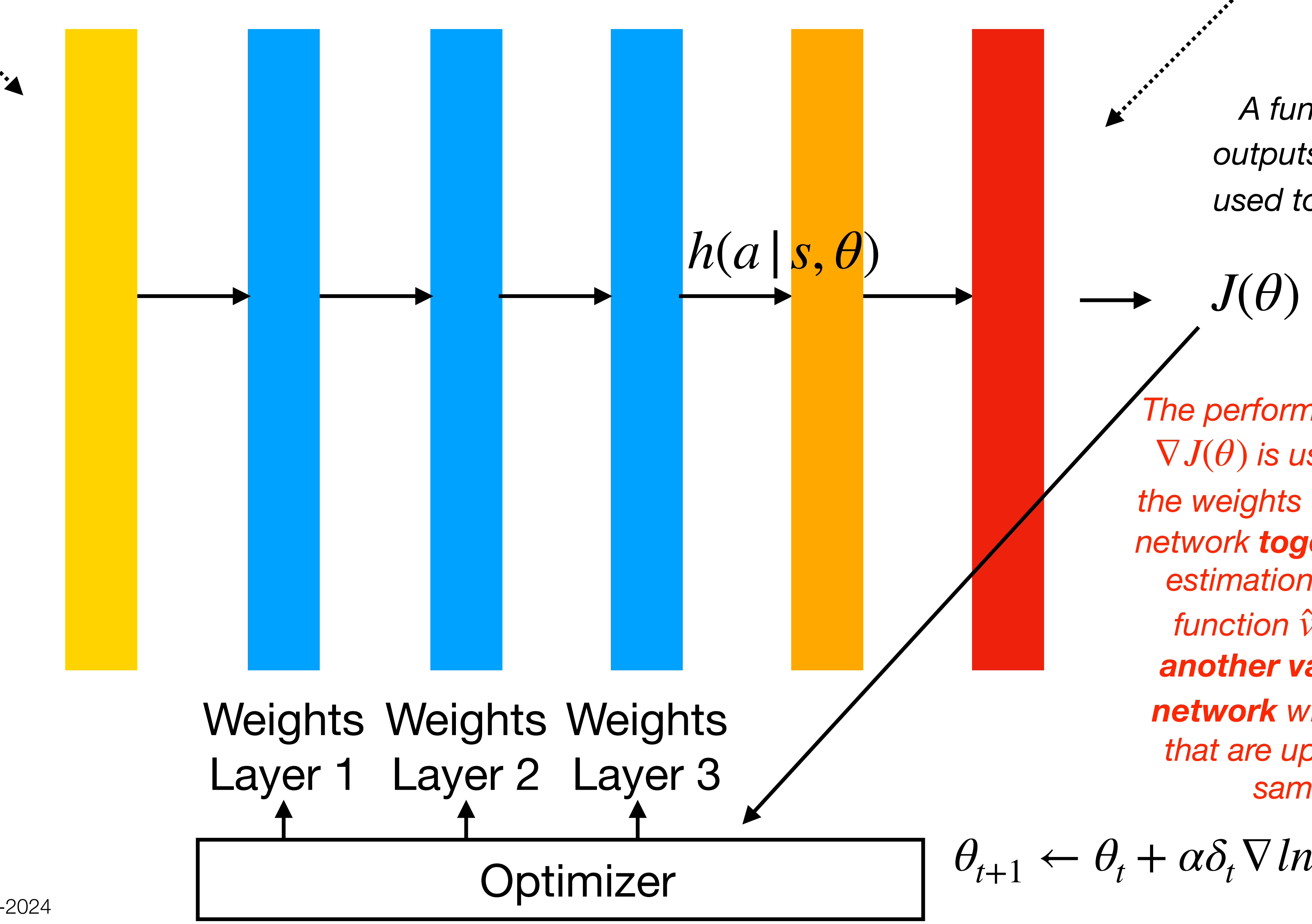
- The value of the state is usually estimated with *another network* for example with a semi-gradient TD(0) method (other methods are possible).

- Please note that the weights $\mathbf{w}$ of this other network are learned at the same time, but independently.

- The network that learns the policy (that with weights θ) in our example is the actor network (i.e, it used to act).

- The network that learns the values (that with weights $\mathbf{w}$) is called critic network (i.e., it is used to “judge” the actions of the actor, which is implemented through the actor network).

Policy Network with One-step Actor Critic



Inputs

s

Neural network function of weights θ

Layer 1 Layer 2 Layer 3 Softmax

Outputs

Probabilities of selecting an action

$\pi(a \mid s, \theta)$

A function of the outputs $\pi(a \mid s, \theta)$ is used to update $J(\theta)$

$h(a \mid s, \theta)$

$J(\theta)$

This is an actor network (the output is the policy)

*The performance gradient $\nabla J(\theta)$ is used to update the weights θ of the neural network **together with** the estimation of the value function $\hat{v}(s \mid \mathbf{w})$ from **another value (neural) network** with weights $\mathbf{w}$ that are updated at the same time*

Weights Layer 1 Weights Layer 2 Weights Layer 3

Optimizer

$$\theta_{t+1} \leftarrow \theta_t + \alpha \delta_t \nabla ln \pi(A_t \mid S_t, \theta)$$

One-step Actor-Critic Method

```
Input: a differentiable policy parametrisation $\pi(a|s, \theta)$, a differentiable state-value function parametrisation $\hat{v}(s, \mathbf{w})$

Parameters: step sizes $\alpha^{\theta} > 0$ and $\alpha^{w} > 0$

Initialise policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$

Loop forever (for each episode):

   Initialise $S$ (first state of episode)

   Loop while $S$ is not terminal

     Select A using policy $\pi$

     Take action A, observe $S', R$

     $\delta \leftarrow R + \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$    (if $S'$ is terminal, then $\hat{v}(S', \mathbf{w}) \doteq 0$)

     $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$

     $\theta \leftarrow \theta + \alpha^{\theta} \delta \nabla ln \pi(A|S, \theta)$

     $S \leftarrow S'$
```

Advantage Actor Critic (A2C)

- We can stabilise learning further by using the advantage function as critic instead of the action value function.

- In practice instead of using

$$\delta = R_{t+1} + \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$$

we will use

$$\delta = \hat{Q}(S_t, A_t, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$$

- The latter expression is usually called the *advantage function*.

Advantage Actor Critic (A2C)

- Te idea is that the advantage function calculates how better the selection the specific action A_t at state S_t is compared to the average value of the state S_t. In a sense, we are subtracting the mean value of the state from the value of the state-action pair.

- In practice, we do not need to quantify how good an action is in absolute sense, but only how much it is better than others on average.

- It is a relative advantage that we are interested in.

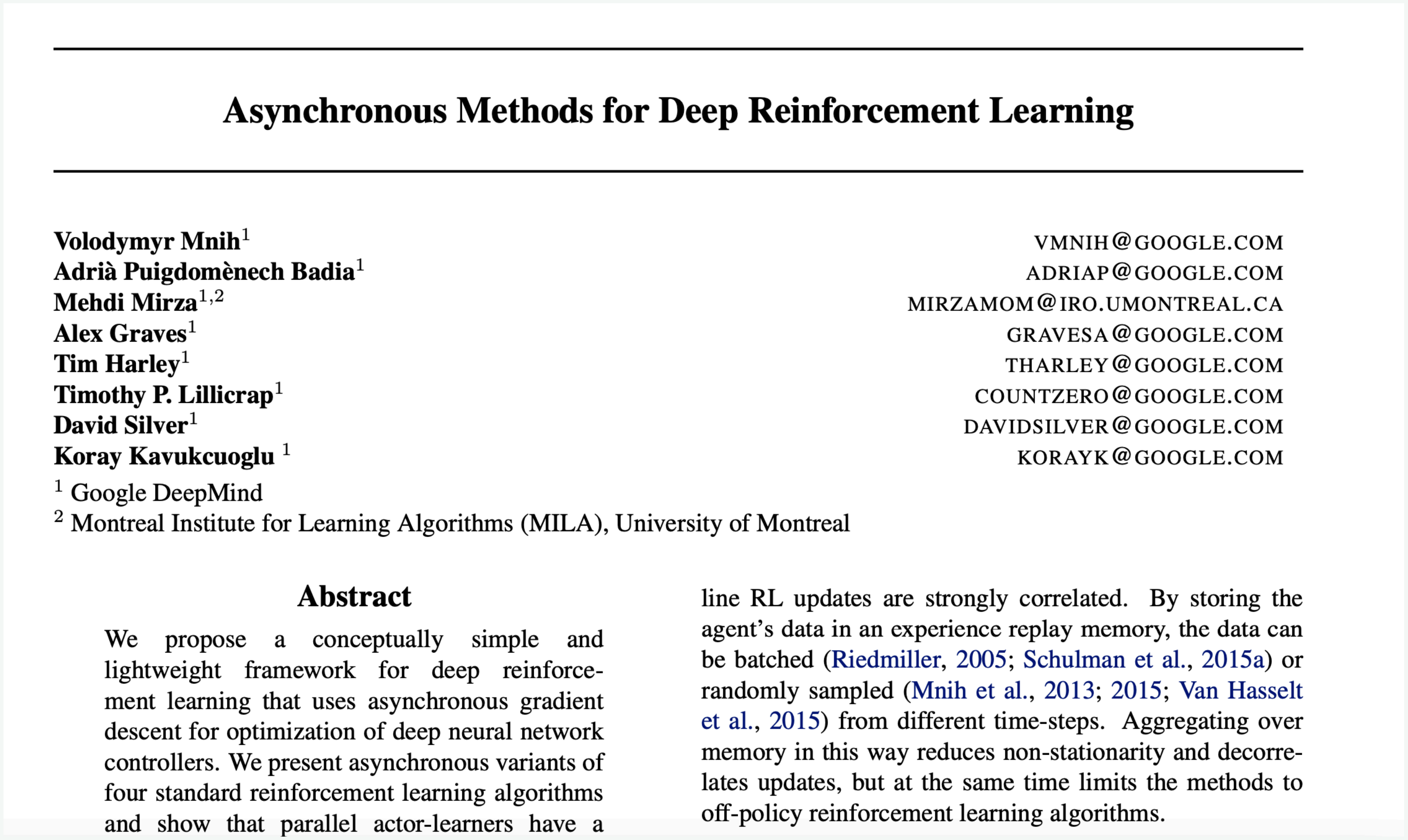
- Different types of advantage functions and approximate advantage functions have been proposed and evaluated in the past years.

- The only (important) condition is that they have to lead to the same expected value for the policy gradient, despite having different variances.

Asynchronous Advantage Actor-Critic (A3C)

- Asynchronous variants have been proposed in order to stabilise the process.
- It has been shown that that parallel actor-learners have a *stabilising* effect on training.
- This has also benefit in terms of performance, since it allows parallel execution.
- This method is usually called Asynchronous Advantage Actor-Critic.
- We are not going to discuss the details of this implementation. If you are interested please check the relevant paper.
- High-scalable implementation of the A3C algorithm has been proposed such as the IMPALA architecture.

Asynchronous Advantage Actor-Critic (A3C)



Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih[1] VMNIH@GOOGLE.COM
Adrià Puigdomènech Badia[1] ADRIAP@GOOGLE.COM
Mehdi Mirza[1,2] MIRZAMOM@IRO.UMONTREAL.CA
Alex Graves[1] GRAVESA@GOOGLE.COM
Tim Harley[1] THARLEY@GOOGLE.COM
Timothy P. Lillicrap[1] COUNTZERO@GOOGLE.COM
David Silver[1] DAVIDSILVER@GOOGLE.COM
Koray Kavukcuoglu [1] KORAYK@GOOGLE.COM

[1] Google DeepMind
[2] Montreal Institute for Learning Algorithms (MILA), University of Montreal

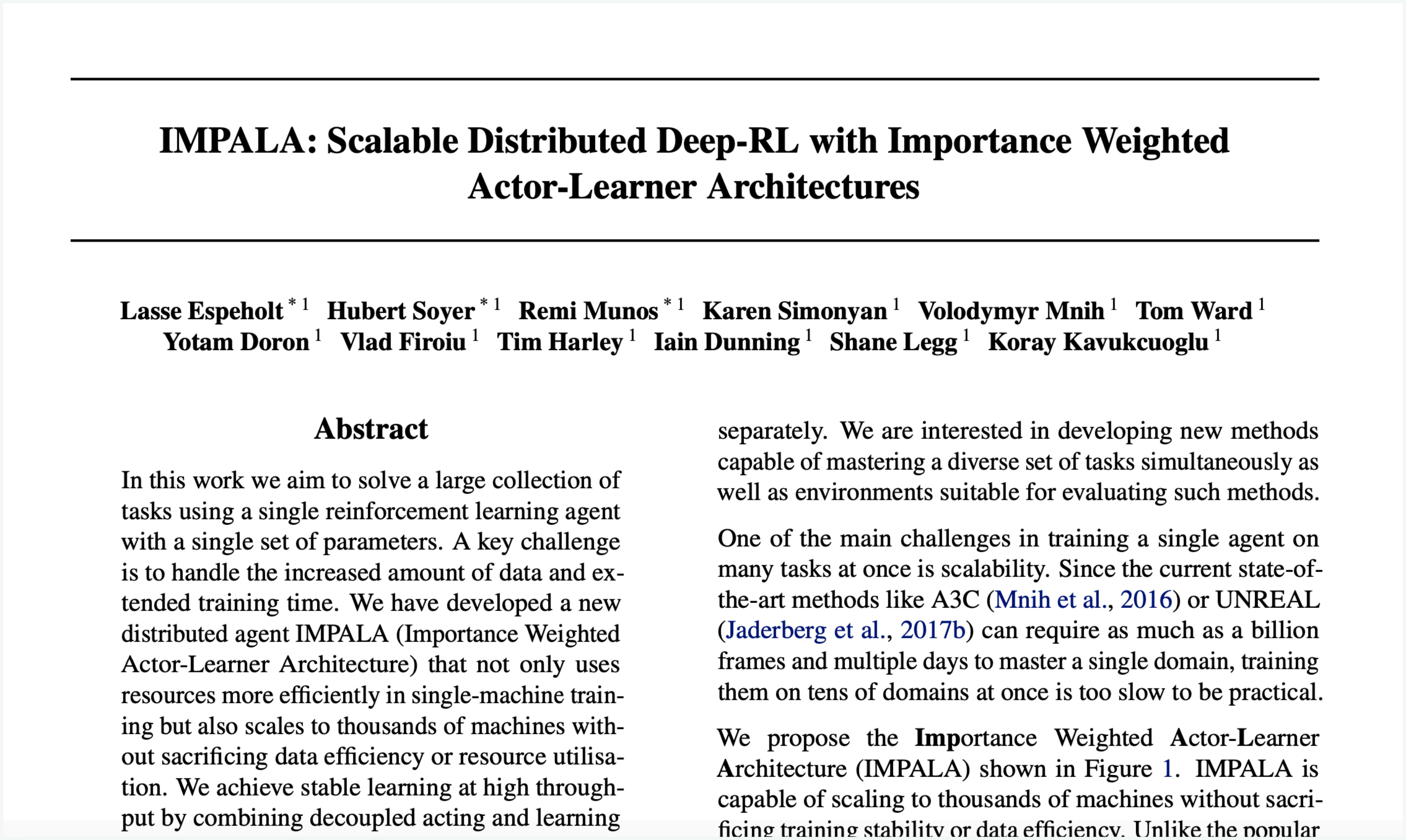
Abstract

We propose a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers. We present asynchronous variants of four standard reinforcement learning algorithms and show that parallel actor-learners have a

line RL updates are strongly correlated. By storing the agent's data in an experience replay memory, the data can be batched (Riedmiller, 2005; Schulman et al., 2015a) or randomly sampled (Mnih et al., 2013; 2015; Van Hasselt et al., 2015) from different time-steps. Aggregating over memory in this way reduces non-stationarity and decorrelates updates, but at the same time limits the methods to off-policy reinforcement learning algorithms.

Published at ICML 2016

IMPALA



IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures

Lasse Espeholt [*1] Hubert Soyer [*1] Remi Munos [*1] Karen Simonyan [1] Volodymyr Mnih [1] Tom Ward [1]
Yotam Doron [1] Vlad Firoiu [1] Tim Harley [1] Iain Dunning [1] Shane Legg [1] Koray Kavukcuoglu [1]

Abstract

In this work we aim to solve a large collection of tasks using a single reinforcement learning agent with a single set of parameters. A key challenge is to handle the increased amount of data and extended training time. We have developed a new distributed agent IMPALA (Importance Weighted Actor-Learner Architecture) that not only uses resources more efficiently in single-machine training but also scales to thousands of machines without sacrificing data efficiency or resource utilisation. We achieve stable learning at high throughput by combining decoupled acting and learning

separately. We are interested in developing new methods capable of mastering a diverse set of tasks simultaneously as well as environments suitable for evaluating such methods.

One of the main challenges in training a single agent on many tasks at once is scalability. Since the current state-of-the-art methods like A3C (Mnih et al., 2016) or UNREAL (Jaderberg et al., 2017b) can require as much as a billion frames and multiple days to master a single domain, training them on tens of domains at once is too slow to be practical.

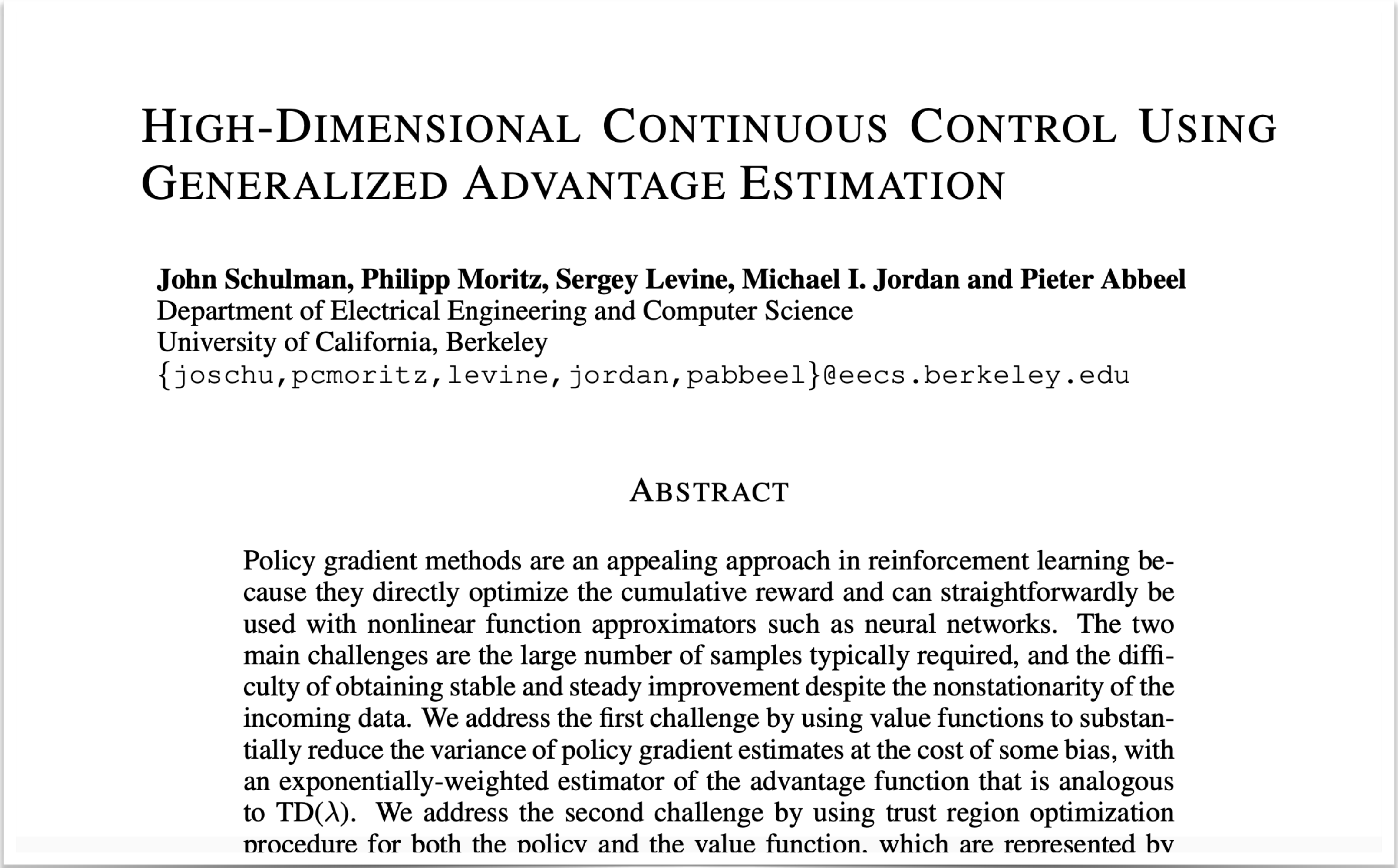
We propose the **Imp**ortance Weighted **A**ctor-**L**earner **A**rchitecture (IMPALA) shown in Figure 1. IMPALA is capable of scaling to thousands of machines without sacrificing training stability or data efficiency. Unlike the popular

Published at ICML 2018

Approximating Advantage Functions

- Methods have been developed for approximating the advantage functions.
- An important method is the *Generalized Advantage Estimation (GAE)*, which is based on the idea of using an exponentially-weighted estimator of the advantage function.
- The GAE method is at the basis of two practical algorithms for policy approximation:
    - Trust Region Policy Optimization (TRPO)
    - Proximal Policy Optimization (PPO)

Generalised Advantage Estimation (GAE)



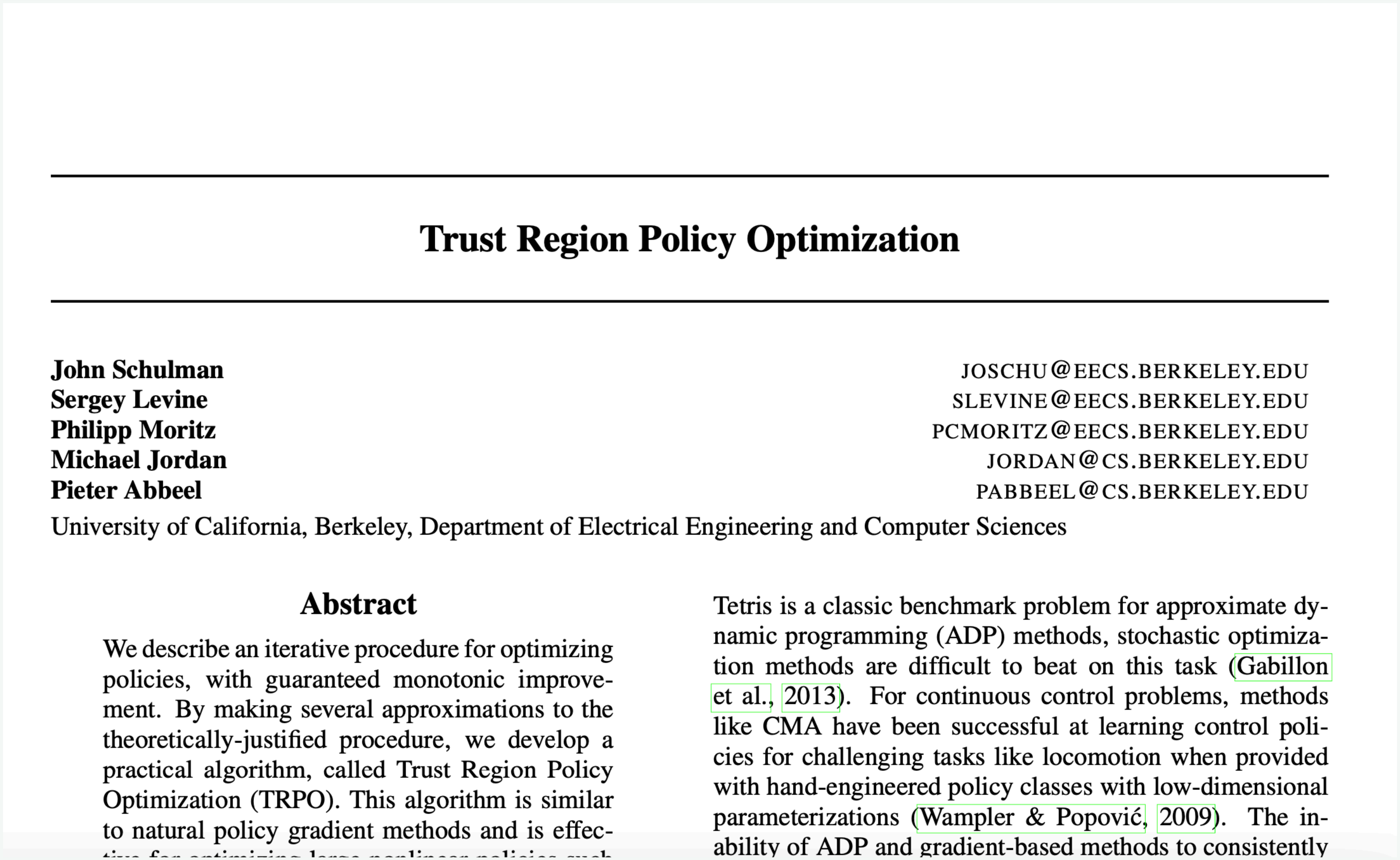
High-Dimensional Continuous Control Using Generalized Advantage Estimation

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan and Pieter Abbeel
Department of Electrical Engineering and Computer Science
University of California, Berkeley
{joschu,pcmoritz,levine,jordan,pabbeel}@eecs.berkeley.edu

Abstract

Policy gradient methods are an appealing approach in reinforcement learning because they directly optimize the cumulative reward and can straightforwardly be used with nonlinear function approximators such as neural networks. The two main challenges are the large number of samples typically required, and the difficulty of obtaining stable and steady improvement despite the nonstationarity of the incoming data. We address the first challenge by using value functions to substantially reduce the variance of policy gradient estimates at the cost of some bias, with an exponentially-weighted estimator of the advantage function that is analogous to TD(λ). We address the second challenge by using trust region optimization procedure for both the policy and the value function, which are represented by

Trust Region Policy Optimization (TRPO)



Trust Region Policy Optimization

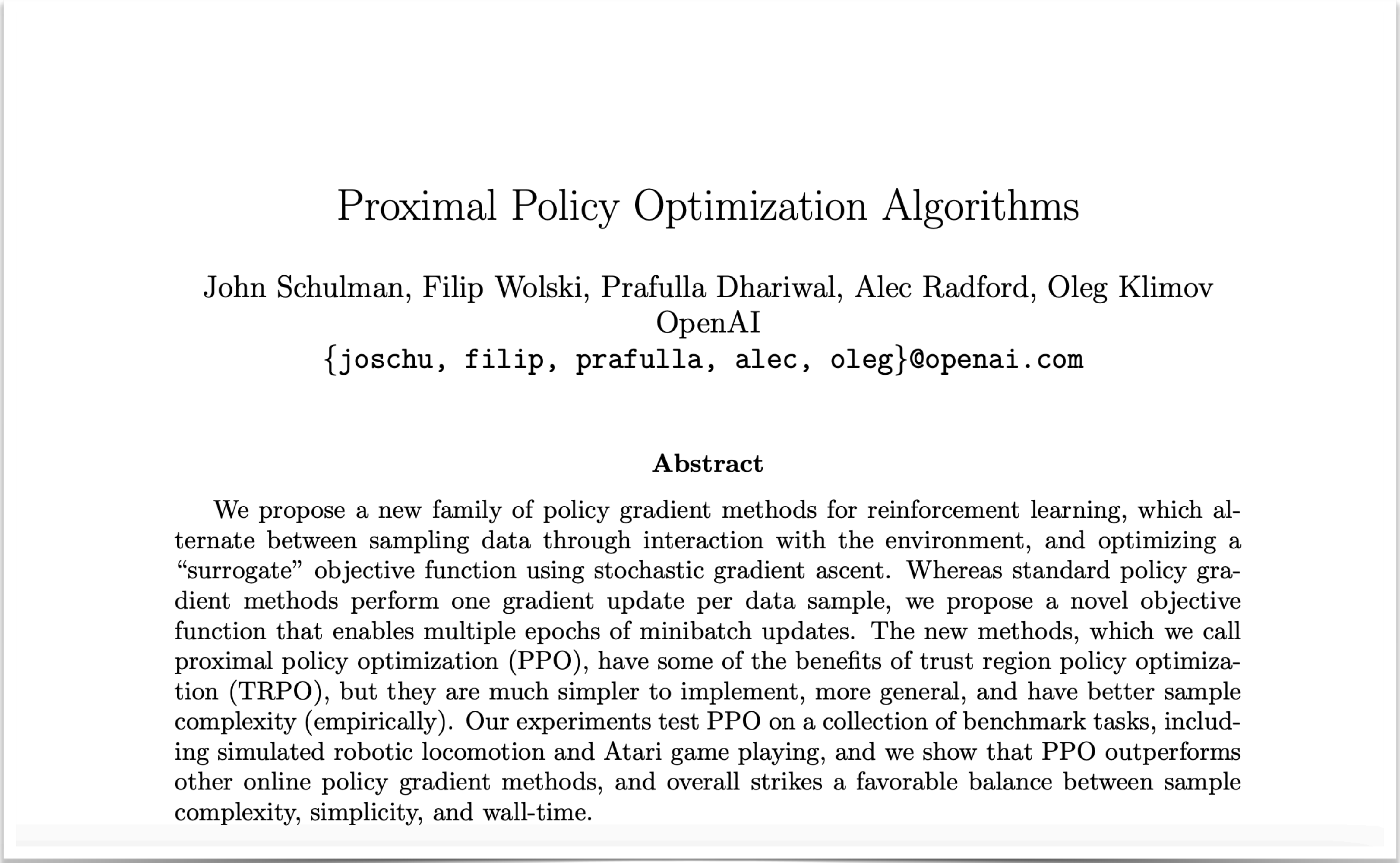
John Schulman JOSCHU@EECS.BERKELEY.EDU
Sergey Levine SLEVINE@EECS.BERKELEY.EDU
Philipp Moritz PCMORITZ@EECS.BERKELEY.EDU
Michael Jordan JORDAN@CS.BERKELEY.EDU
Pieter Abbeel PABBEEL@CS.BERKELEY.EDU
University of California, Berkeley, Department of Electrical Engineering and Computer Sciences

Abstract

We describe an iterative procedure for optimizing policies, with guaranteed monotonic improvement. By making several approximations to the theoretically-justified procedure, we develop a practical algorithm, called Trust Region Policy Optimization (TRPO). This algorithm is similar to natural policy gradient methods and is effective for optimizing large nonlinear policies such

Tetris is a classic benchmark problem for approximate dynamic programming (ADP) methods, stochastic optimization methods are difficult to beat on this task (Gabillon et al., 2013). For continuous control problems, methods like CMA have been successful at learning control policies for challenging tasks like locomotion when provided with hand-engineered policy classes with low-dimensional parameterizations (Wampler & Popović, 2009). The inability of ADP and gradient-based methods to consistently

Proximal Policy Optimization (PPO)



Proximal Policy Optimization Algorithms

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov
OpenAI
{joschu, filip, prafulla, alec, oleg}@openai.com

Abstract

We propose a new family of policy gradient methods for reinforcement learning, which alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. Whereas standard policy gradient methods perform one gradient update per data sample, we propose a novel objective function that enables multiple epochs of minibatch updates. The new methods, which we call proximal policy optimization (PPO), have some of the benefits of trust region policy optimization (TRPO), but they are much simpler to implement, more general, and have better sample complexity (empirically). Our experiments test PPO on a collection of benchmark tasks, including simulated robotic locomotion and Atari game playing, and we show that PPO outperforms other online policy gradient methods, and overall strikes a favorable balance between sample complexity, simplicity, and wall-time.

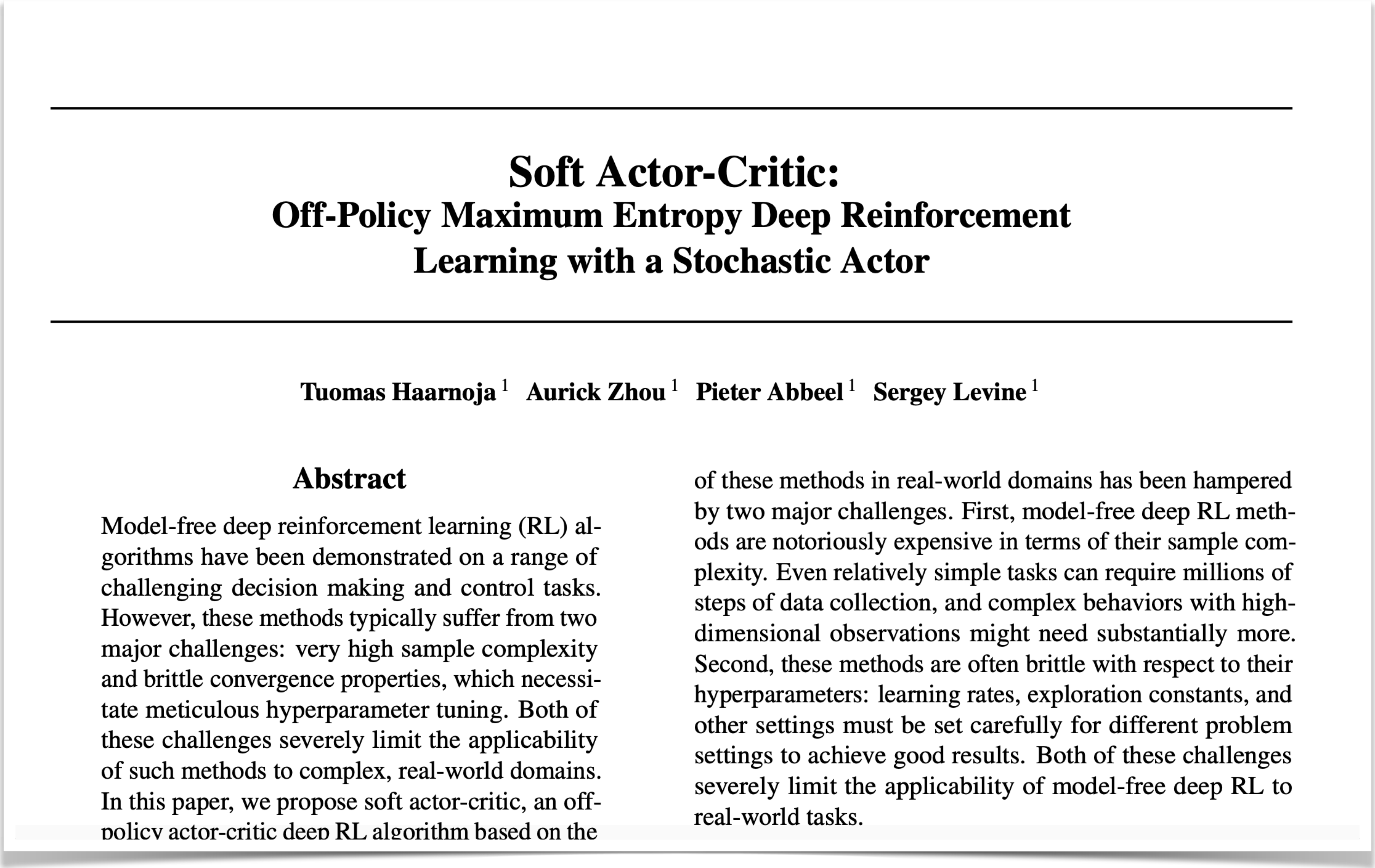
Soft-Actor Critic (SAC)

- Key (open) problems:
  - Very high sample complexity;
  - Brittleness of the convergence.
- One solution is *regularisation*.
- The Soft-Actor Critic algorithm has been introduced with this goal.
- The key idea is to regularise the expression using an entropy term. The actor aims at maximising the expected reward while also maximising entropy (i.e., exploration).
- We will consider a revised objective function as follows:

$$J(\pi) = \sum_{t=0}^{T} E_{\pi}[R_t + \alpha \mathscr{H}(\pi)]$$

where $\mathscr{H}(\pi)$ is the entropy of the policy over the distribution of the marginals of the trajectory distribution (induced by the policy π). α is called the *temperature parameter*.

Soft-Actor Critic (SAC)



Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor

Tuomas Haarnoja[1] Aurick Zhou[1] Pieter Abbeel[1] Sergey Levine[1]

Abstract

Model-free deep reinforcement learning (RL) algorithms have been demonstrated on a range of challenging decision making and control tasks. However, these methods typically suffer from two major challenges: very high sample complexity and brittle convergence properties, which necessitate meticulous hyperparameter tuning. Both of these challenges severely limit the applicability of such methods to complex, real-world domains. In this paper, we propose soft actor-critic, an off-policy actor-critic deep RL algorithm based on the

of these methods in real-world domains has been hampered by two major challenges. First, model-free deep RL methods are notoriously expensive in terms of their sample complexity. Even relatively simple tasks can require millions of steps of data collection, and complex behaviors with high-dimensional observations might need substantially more. Second, these methods are often brittle with respect to their hyperparameters: learning rates, exploration constants, and other settings must be set carefully for different problem settings to achieve good results. Both of these challenges severely limit the applicability of model-free deep RL to real-world tasks.

Published at ICML 2018

Continuous Action Space

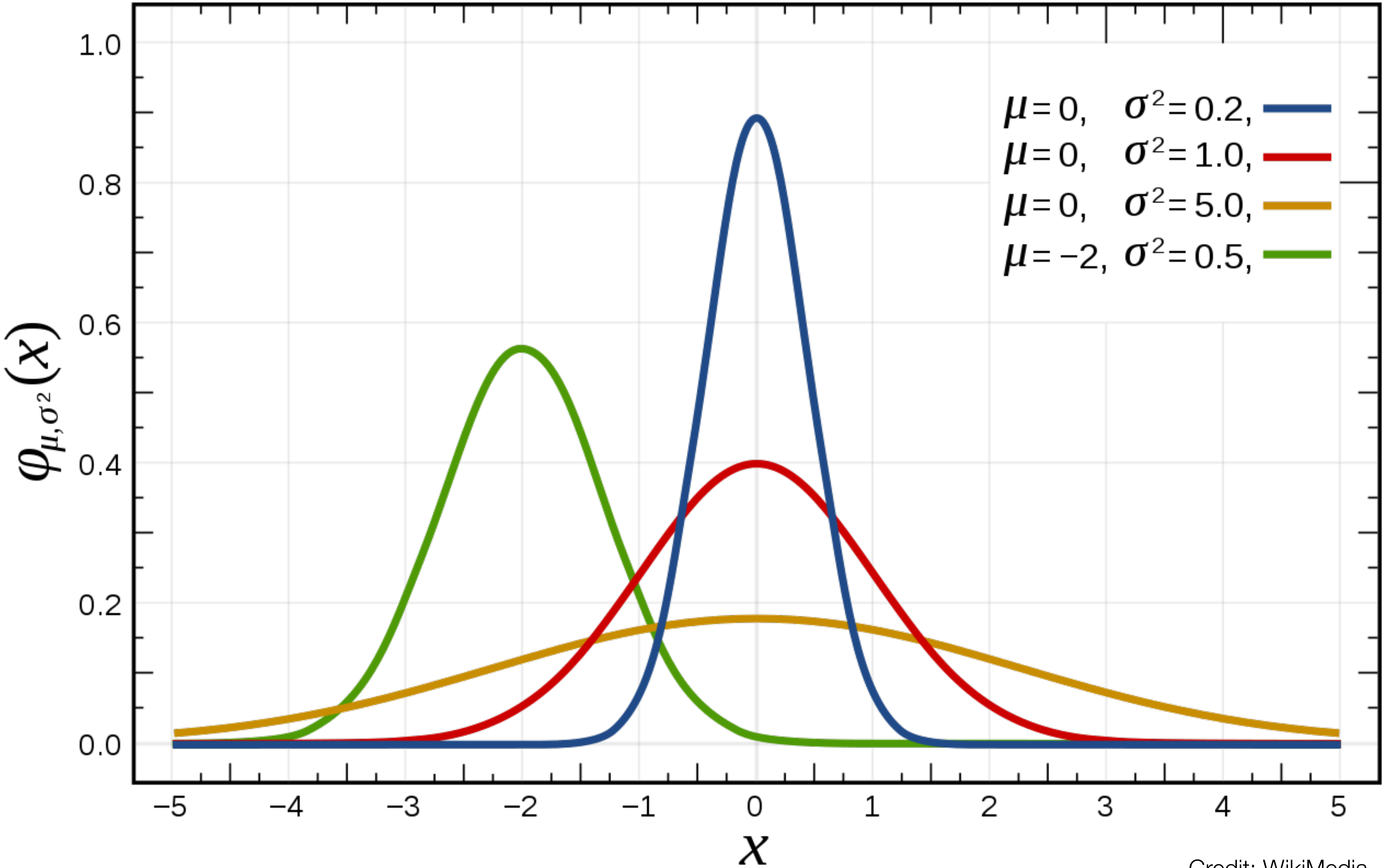
- In this lecture, we have examined the case of discrete actions, but it is worth noting that the policy networks can be used to learn continuous actions (think about moving the wheel of a car).

- In that case the output is not a *discrete* probability distribution $\pi(a \mid s, \theta)$ on the state space (i.e., at the end a set of discrete values), but a *continuous* one.

- In that case we will not learn the discrete probability for each actions, but the parameters of a probability distribution.

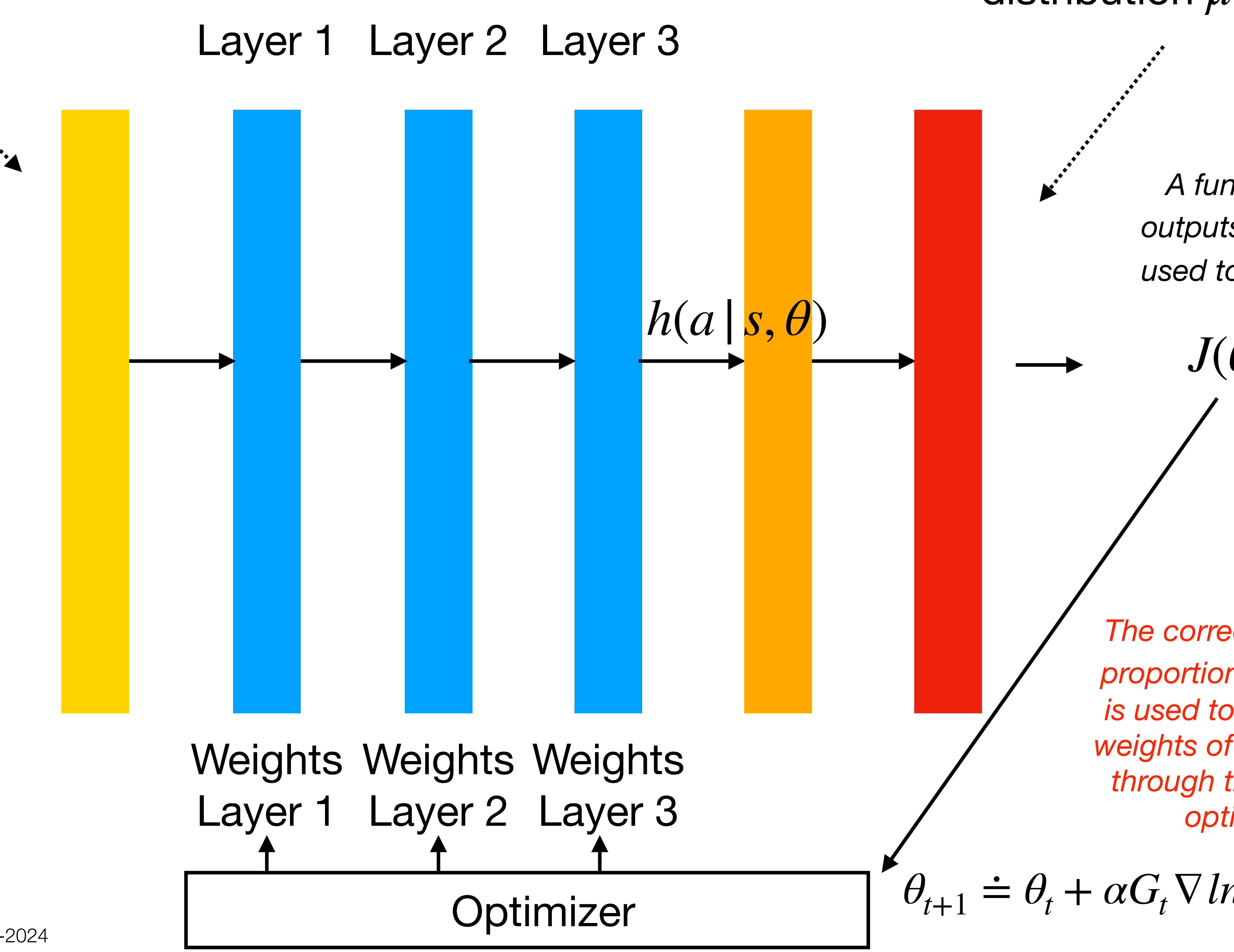
- For example, we can use a Gaussian distribution and learn the parameters $mean(s)$ and $variance(s)$.

- Then, we will sample from that distribution for extracting the action to be taken.



Credit: WikiMedia

Policy Networks



Inputs

s

Neural network function of weights θ

Layer 1 Layer 2 Layer 3

Outputs
Parameter of the
probability
distribution μ and σ

A function of the outputs $\pi(a|s,\theta)$ is used to update $J(\theta)$

$h(a|s,\theta)$

$J(\theta)$

The correction that is proportional to $\nabla J(\theta)$ is used to update the weights of the network through the network optimizer

Weights Layer 1 Weights Layer 2 Weights Layer 3

Optimizer

$$\theta_{t+1} \doteq \theta_t + \alpha G_t \nabla ln\pi(A_t|S_t,\theta_t)$$

References

- Chapter 13 of Barto and Sutton. Introduction to Reinforcement Learning. Second Edition. MIT Press 2018.
- For A2C, A3C, GAE, SAC, TRPO and PPO, please refer to the original papers.