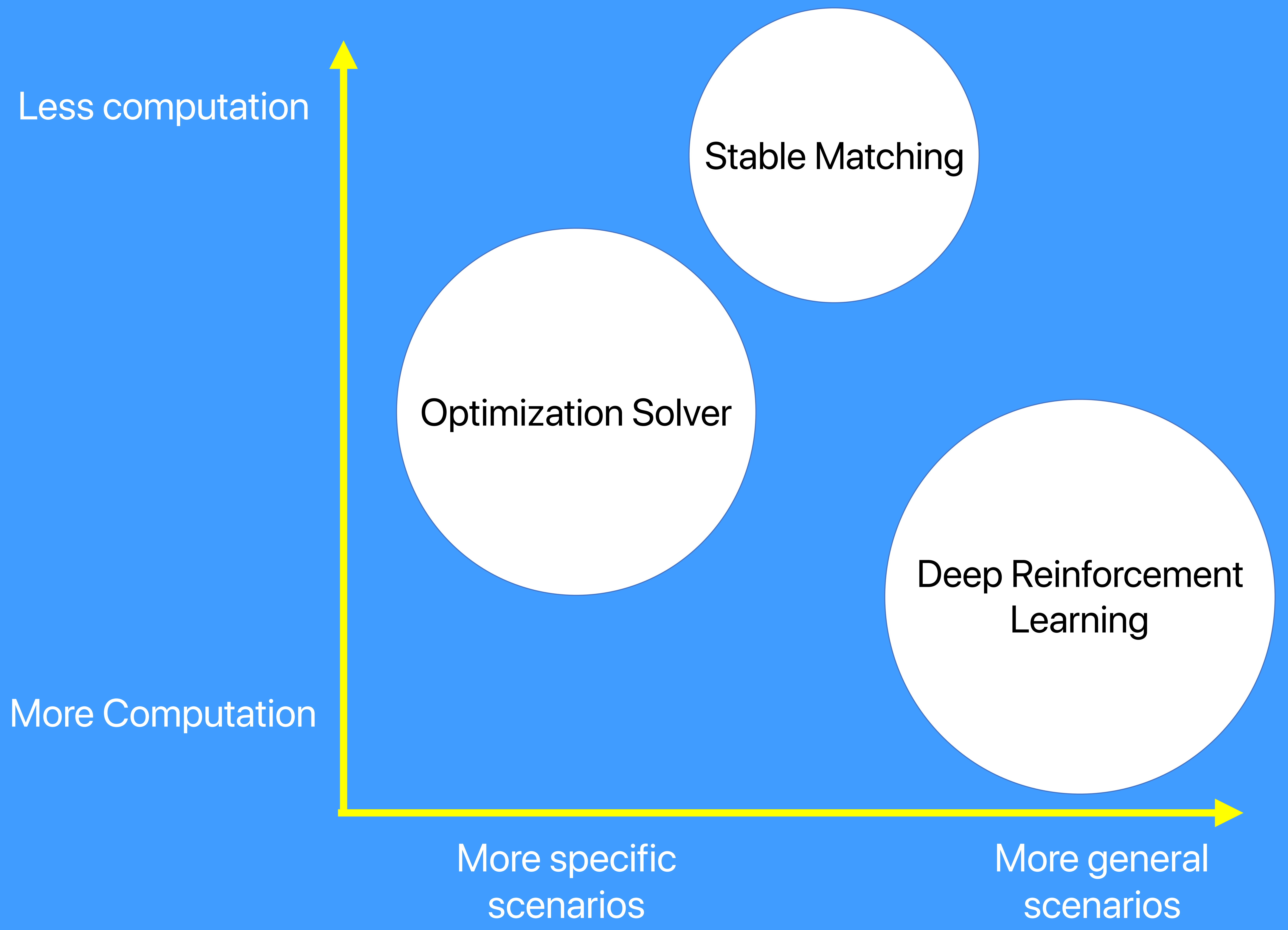
Resource Provisioning with using Deep Reinforcement Learning

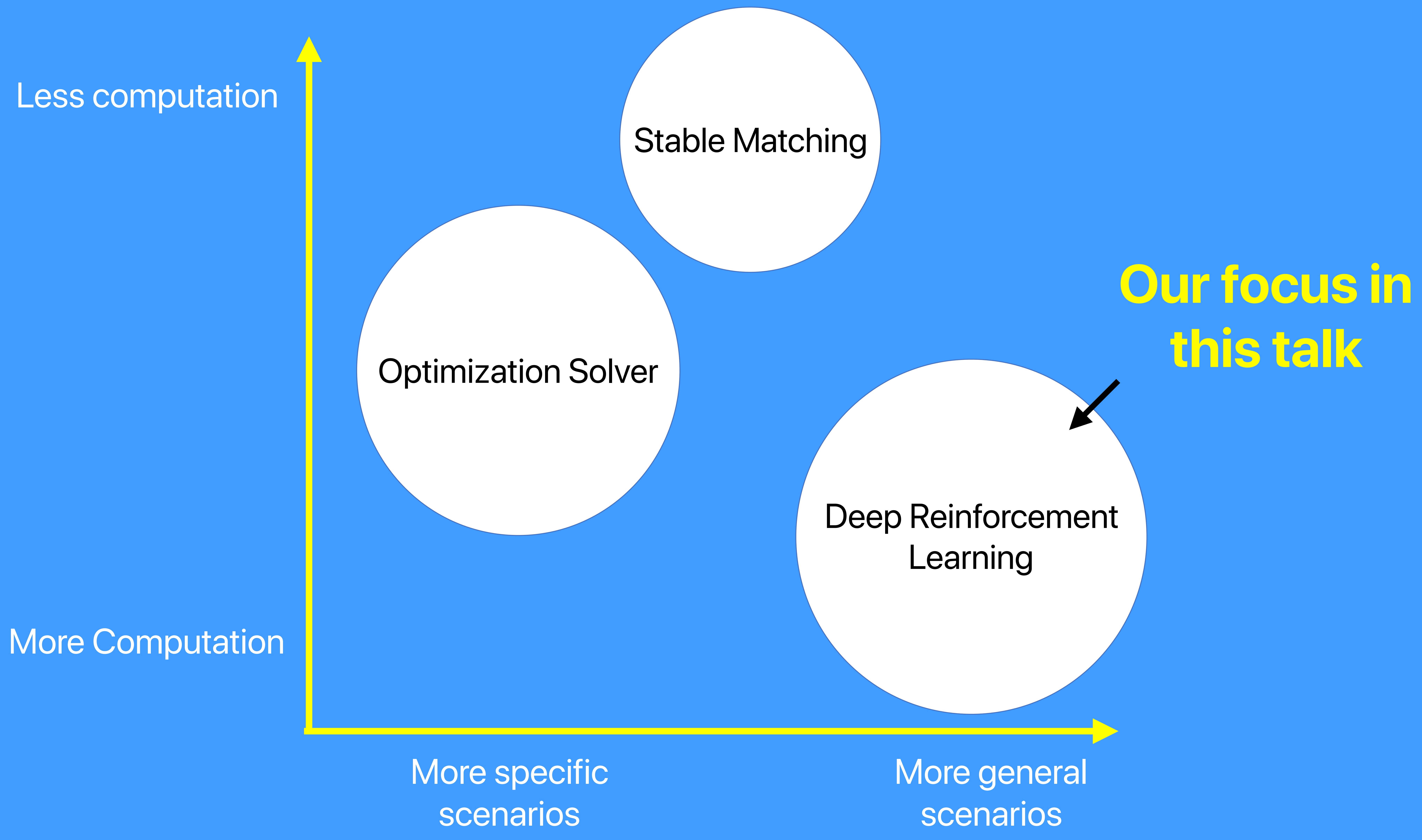
Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

IWQoS 2019, Phoenix, Arizona
June 24, 2019

Resource assignment problems in general: minimize the job completion time by **assigning resources**



Solving **resource assignment** problems



Solving **resource assignment** problems

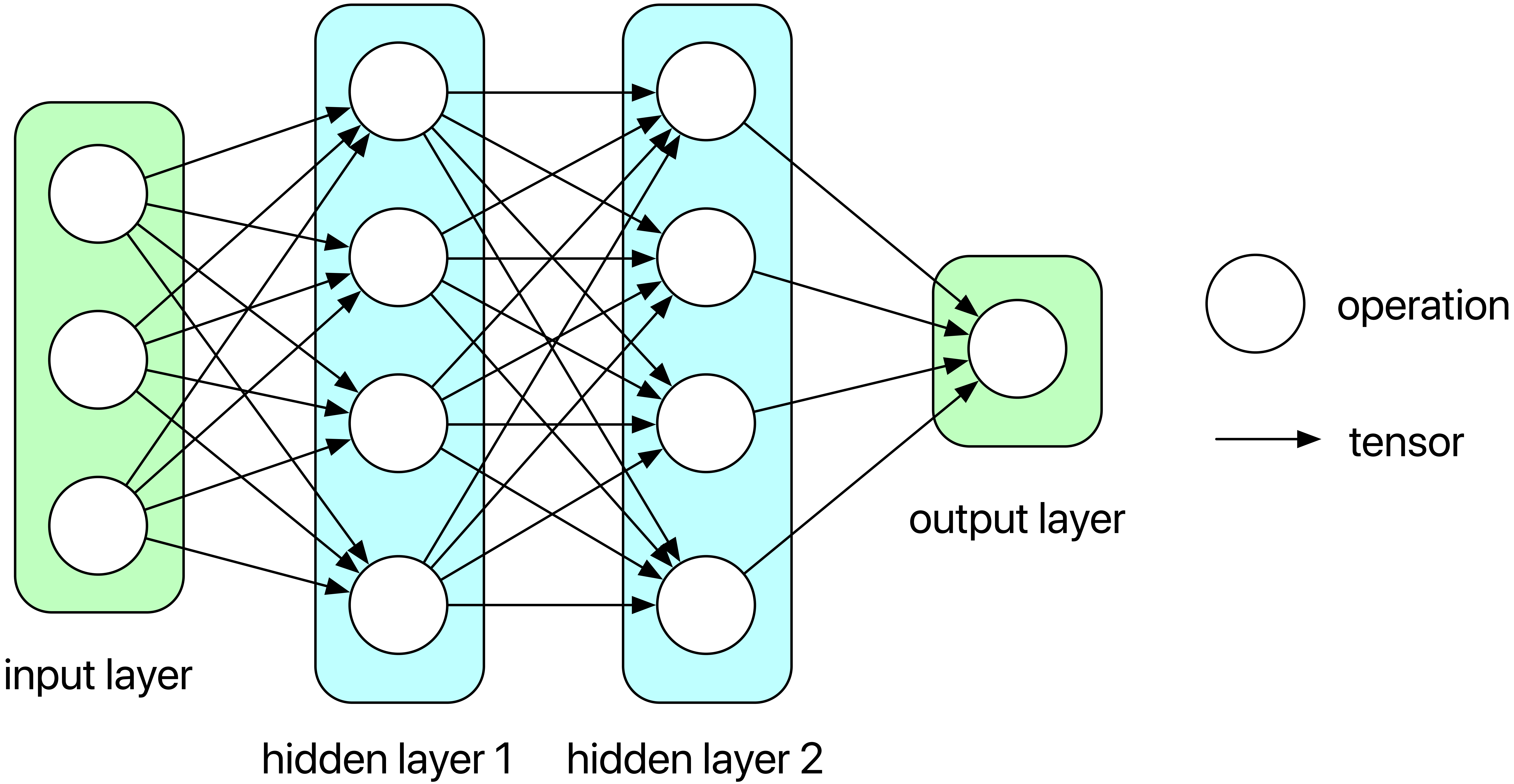
One running example today

Modern machine learning training workloads need a large amount of computation resources, and can take days to complete

Objective: complete the machine learning training workload as soon as possible

How? By optimally assigning devices — GPUs and CPUs — to neural network operations

Neural networks: computation graph

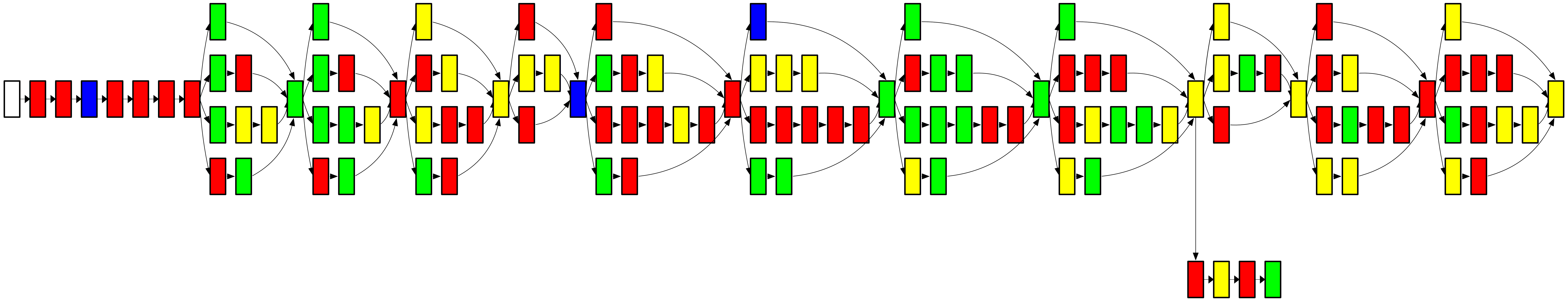


Same neural network, but in Python code

```
# activation function (use sigmoid)
f = lambda x: 1.0/(1.0 + np.exp(-x))

# random input vector of three numbers (3x1)
x = np.random.randn(3, 1)

# W1, W2, W3, b1, b2, b3 are learnable parameters
# calculate first hidden layer activations (4x1)
h1 = f(np.dot(W1, x) + b1)

# calculate second hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2)

# output neuron (1x1)
out = np.dot(W3, h2) + b3
```

Device placement with deep reinforcement learning

Objective: to find the **best** way to assign devices to operations to minimize training time



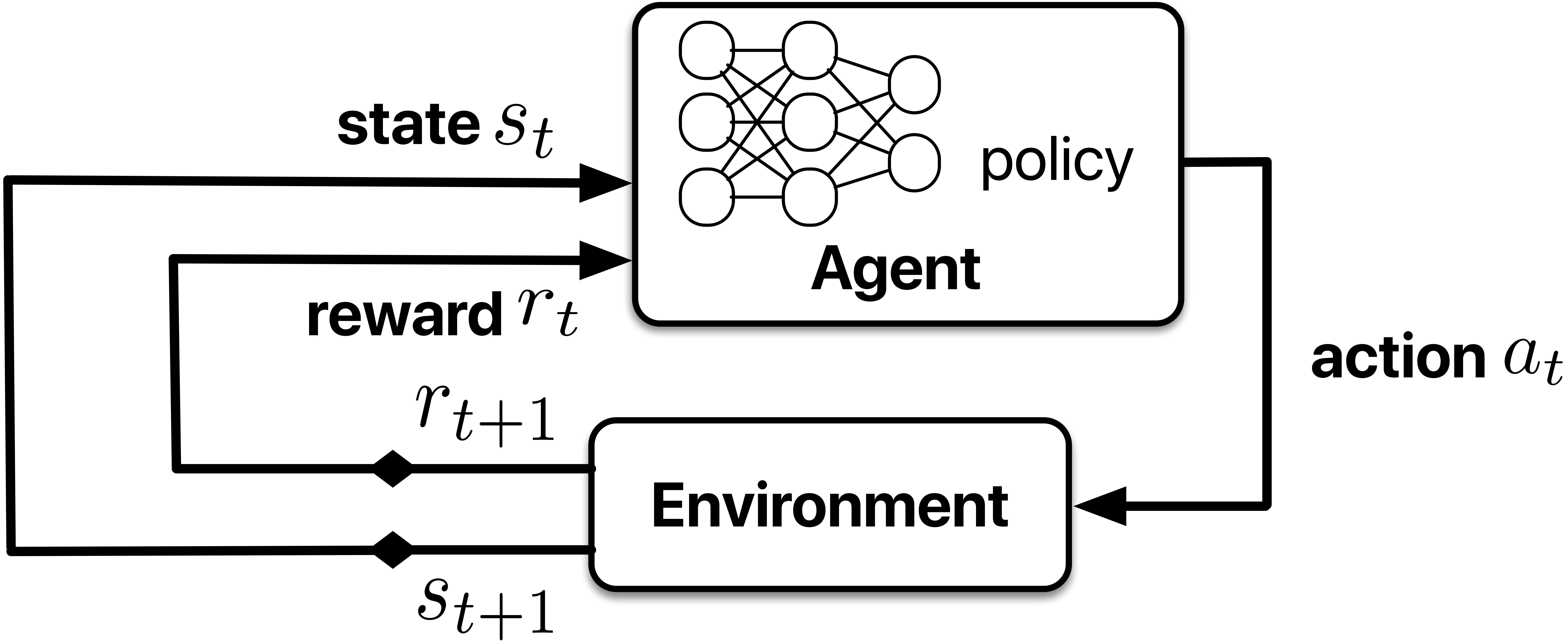
Mirhoseini et al. (Google Inc.), “**Device placement optimization with reinforcement learning,**” in Proc. **ICML 2017**.

But what is deep reinforcement learning?

Recent success stories from DeepMind

- Deep Q-Network: Atari 2600 games (February 2015)
- AlphaGo (3:1 win over Lee Sedol, October 2016)
- AlphaGo Zero (100:0 win over AlphaGo, October 2017)
  - Learned from scratch using self-play with deep reinforcement learning and Monte-Carlo Tree Search

Let’s start from the beginning

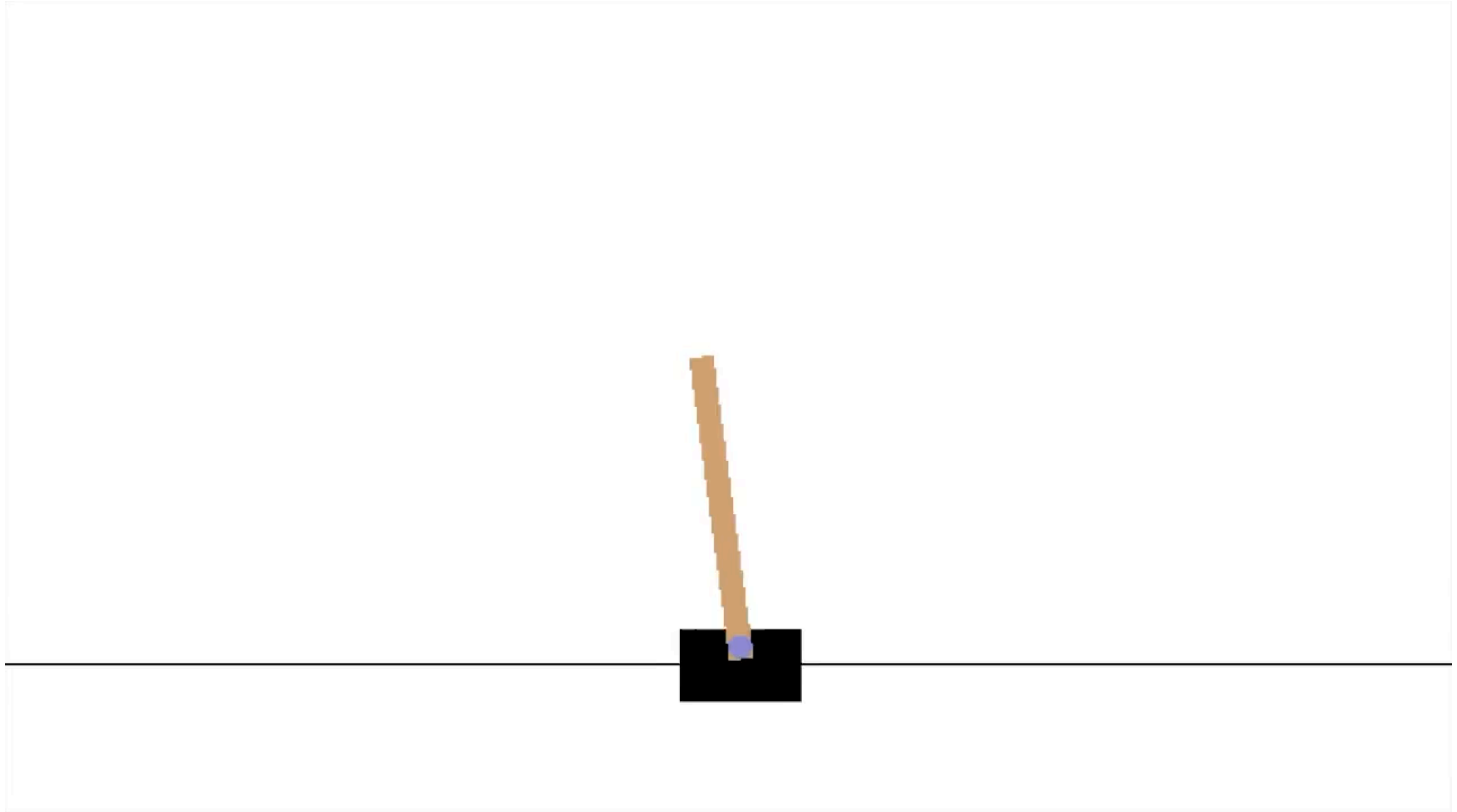

state s_t
policy
Agent
reward r_t
action a_t
r_{t+1}
Environment
s_{t+1}

Reinforcement learning is "semi-supervised," with not much more guidance than a reward from the environment

Toy example: Cartpole

States observed: `[position of cart, velocity of cart, angle of pole, pole velocity at tip]`

Possible actions: `[push cart to the left, push cart to the right]`



Cartpole: random agent

```
import gym
env = gym.make("CartPole-v0")
total_reward = 0.0
total_steps = 0
obs = env.reset()

# start taking random actions
while True:
  action = env.action_space.sample()
  observation, reward, done, _ = env.step(action)
  total_reward += reward
  total_steps += 1
  if done:
    break
```

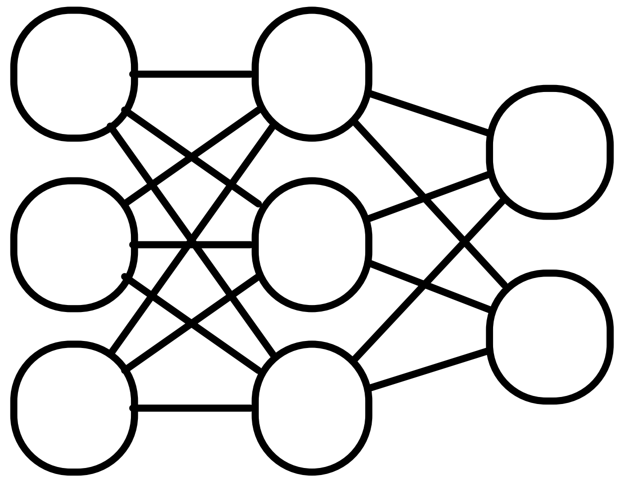
```
(pytorch) $ python cartpole_random.py
```

Our first algorithm: the cross-entropy method

Model free: directly connects observations to actions, without building a model of the environment or the reward

Our first algorithm: the cross-entropy method

state (observation) s →



neural network

→ policy $\pi(a|s)$

Performing random sampling using this probability distribution adds some random exploration to our agent — a good thing!

→ a probability distribution over actions

Cross-entropy: training

- Play **N** episodes using our current model and environment
- Calculate the total reward for every episode and decide on a reward boundary (say, 70%)
- Throw away all episodes with a reward below the boundary
- Train on the remaining **elite** episodes
- Repeat until we are satisfied with the result

Cross-entropy: an iterative algorithm

$$\pi_{i+1}(a|s) = \arg\min_{\pi_{i+1}} - \mathbb{E}_{z\sim\pi_i(a|s)}[R(z) \geq \phi_i] \log \pi_{i+1}(a|s)$$

Basic idea: we sample episodes using our current policy (starting with some random initial policy), and minimize the **negative log-likelihood** of the most successful samples using our policy

- happens to the be same as minimizing the **cross-entropy** (and the **Kullback-Leibler (K-L) divergence** that quantifies the distance between two probability distributions)

Defining our neural network using PyTorch

```python
import torch.nn as nn

class Net(nn.Module):
    def __init__(self, obs_size, hidden_size, n_actions):
        super(Net, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, n_actions)
        )

    def forward(self, x):
        return self.net(x)
```

Starting a loop to interact with the environment

```
while True:
    obs_v = torch.FloatTensor([obs])
    act_probs_v = sm(net(obs_v))
    act_probs = act_probs_v.data.numpy()[0]

    action = np.random.choice(len(act_probs),
                              p=act_probs)

    next_obs, reward, is_done, _ = env.step(action)
    episode_reward += reward
```

Training our neural network

```
env = gym.make("CartPole-v0")
obs_size = env.observation_space.shape[0]
n_actions = env.action_space.n

net = Net(obs_size, HIDDEN_SIZE, n_actions)
objective = nn.CrossEntropyLoss()
optimizer = optim.Adam(params=net.parameters(), lr=0.01)

for iter_no, batch in enumerate(iterate_batches(env, net,
BATCH_SIZE)):
  obs_v, acts_v, reward_b, reward_m = filter_batch(batch, PERCENTILE)
  optimizer.zero_grad()
  action_scores_v = net(obs_v)
  loss_v = objective(action_scores_v, acts_v)
  loss_v.backward()
  optimizer.step()
```

combines **softmax** (exp) and **cross-entropy** (log) into one function for numerical stability

```
(pytorch) $ python cartpole_cross_entropy.py
```

Our second algorithm: policy gradient

- A policy gradient method, called REINFORCE, was used by Google's ICML 2017 paper to solve the device placement problem
- The cross-entropy method uses the **elite** episodes with high rewards, and discards the bad episodes with low rewards
- But can we use a more **fine-grained separation** of episodes?
  - Perhaps an episode with a total reward of 100 should contribute more than another episode with a total reward of 50?

REINFORCE: Monte-Carlo Policy Gradient

- Play N episodes using our current model and environment
- For every step *t* in every episode *k*, calculate a discounted total reward for subsequent steps:

$$Q_{k,t} \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

  - An episode with rewards `[1, 1, 1, 1]` now becomes `[3.9404, 2.9701, 1.99, 1.0]` ($\gamma = 0.99$)

REINFORCE: Monte-Carlo Policy Gradient

Update the neural network weights to minimize the loss function:

$$-\mathbb{E}_{k,t}\big[\underbrace{Q_{k,t}}_{\text{discounted total reward}} \log \underbrace{\pi(a_{k,t}|s_{k,t};\theta)}_{\text{policy function}}\big]$$

Training our neural network

```
optimizer.zero_grad()
states_v = torch.FloatTensor(batch_states)
batch_actions_t = torch.LongTensor(batch_actions)
batch_qvals_v = torch.FloatTensor(batch_qvals)

logits_v = net(states_v)
log_prob_v = F.log_softmax(logits_v,
                           dim=1)
log_prob_actions_v = batch_qvals_v *
   log_prob_v[range(len(batch_actions)), batch_actions_t]
loss_v = -log_prob_actions_v.mean()
loss_v.backward()
optimizer.step()
```

combines **softmax** (exp) and **log** into one function for numerical stability

```
(pytorch) $ python cartpole_reinforce.py
```

We are not done yet: Proximal Policy Optimization

- Motivation: improve the **stability** of policy updates during training
  - We wish to train as fast as we can, making large steps in Stochastic Gradient Descent (SGD) updates
  - But since our policy is very nonlinear, a large update can ruin the policy we've just learned — making a bad update once may not be recoverable later
  - One can make tiny steps in SGD updates, but this will slow down convergence

Lesson learned: distance in parameter space $\neq$ distance in policy space

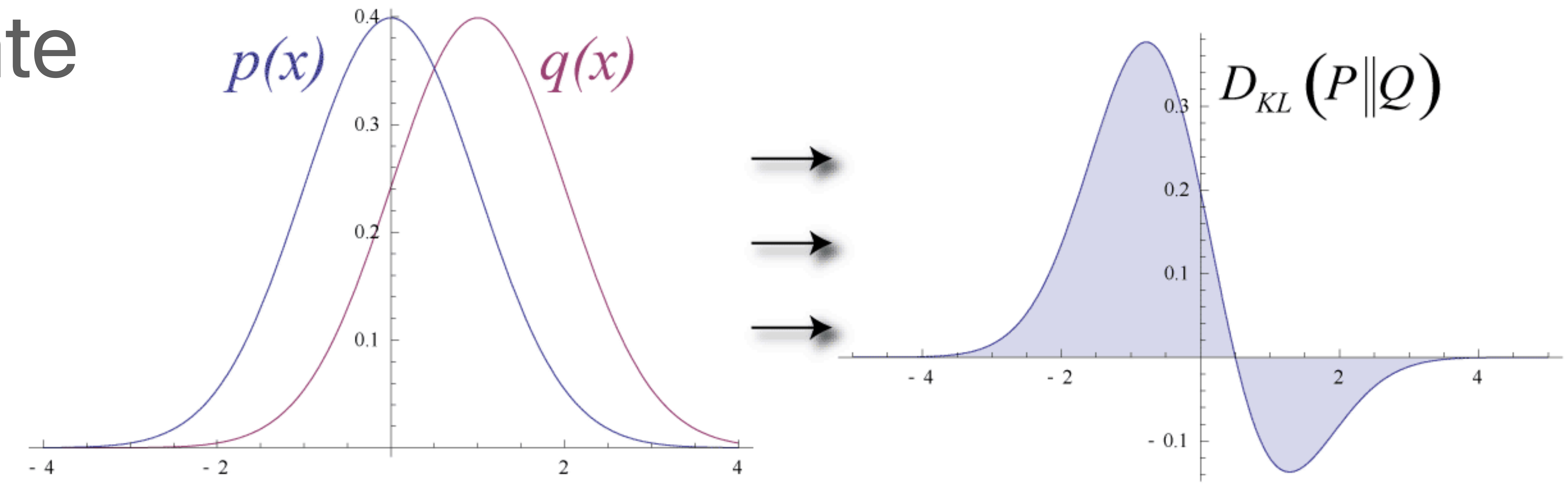
Proximal Policy Optimization

Update the neural network weights to minimize the loss function:

$$-\mathbb{E}_{k,t}[Q_{k,t}\frac{\pi(a_{k,t}|s_{k,t};\theta)}{\pi(a_{k,t}|s_{k,t};\theta_{\text{old}})} - \beta D_{\text{KL}}[\pi(\cdot|s_{k,t},\theta) \parallel \pi(\cdot|s_{k,t},\theta_{\text{old}})]$$

policy parameters before the update

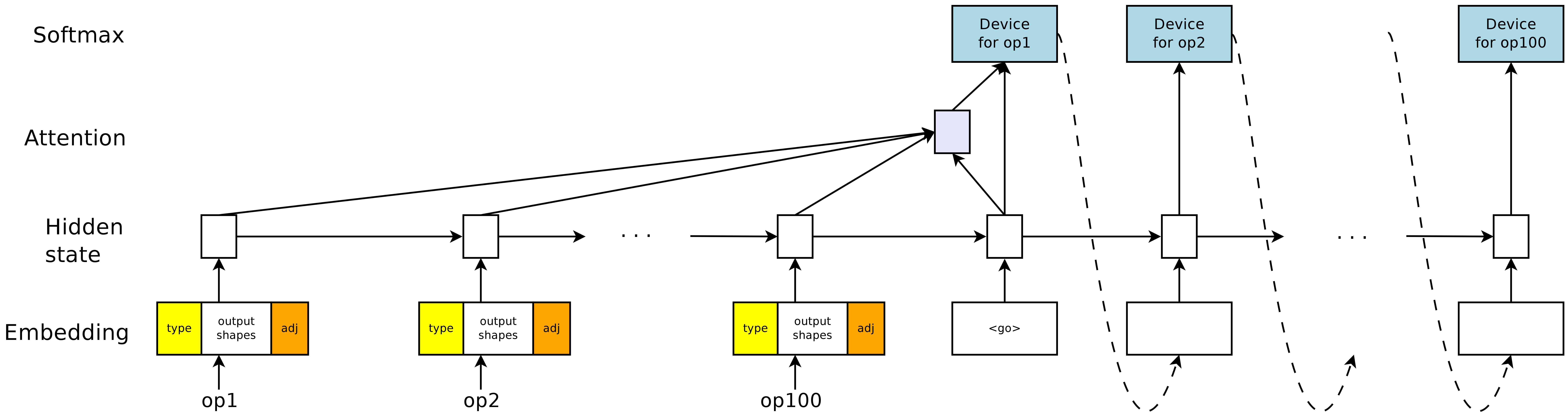
Kullback-Leibler (K-L) divergence



source: Wikipedia

Back to our device placement problem using deep reinforcement learning

ICML 2017: a variant of REINFORCE policy gradient



Neural network model: a sequence-to-sequence model with an attention layer

Reward: square root of the average running time over several iterations

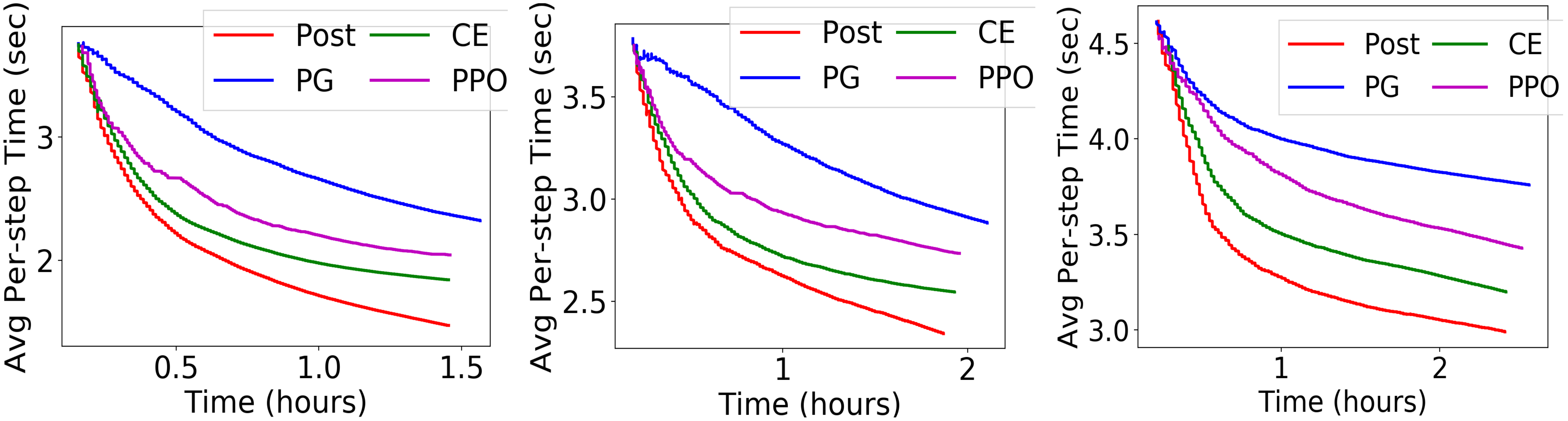
Our recent work: PPO + cross-entropy

Reward: $\bar{R} - R$

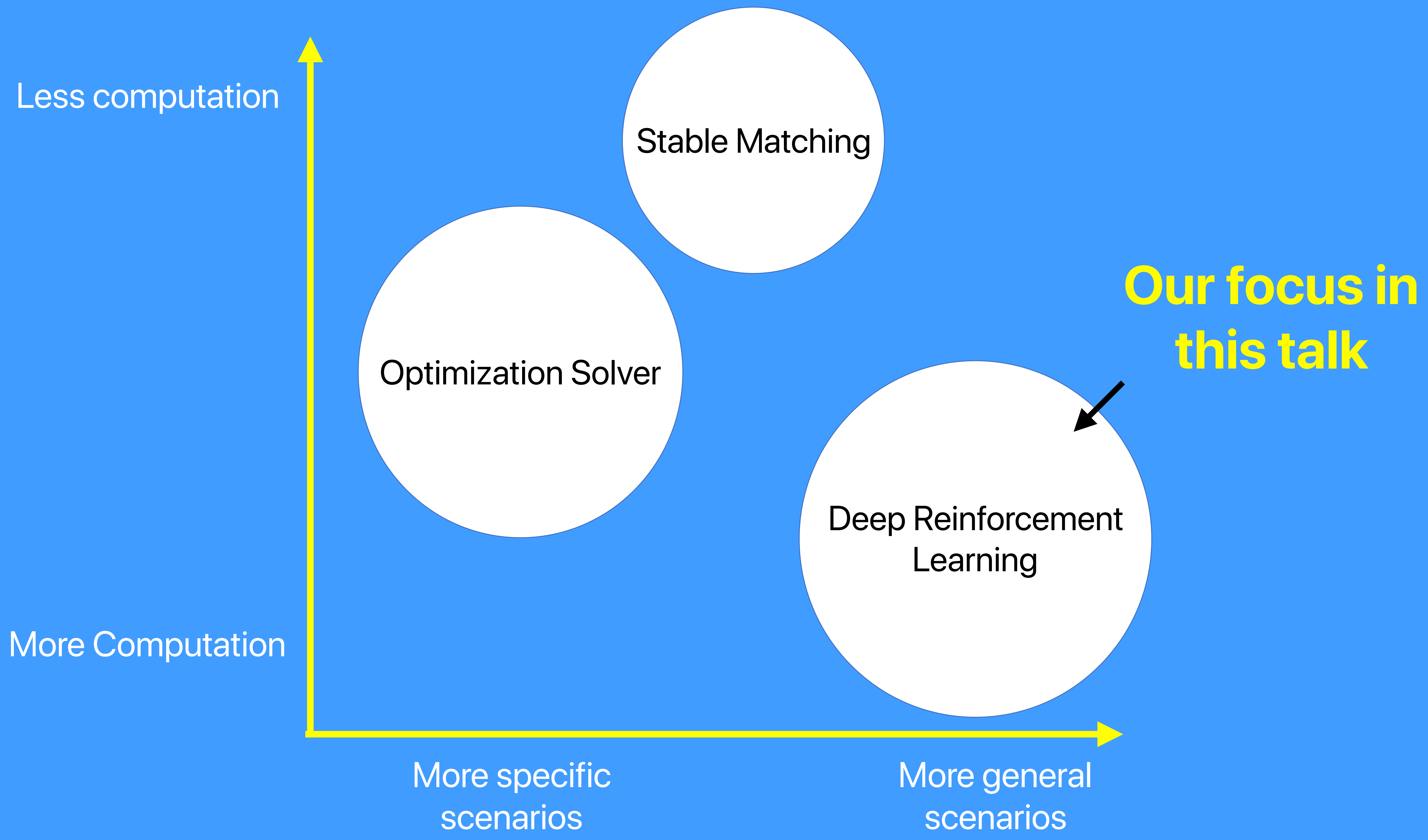
Every *K* steps: update the policy using PPO

Every *N* steps: update the policy using cross-entropy

Training progress over time



(a) ResNet-50 on 2 GPUs (b) Inception-V3 on 4 GPUs (c) ResNet-101 on 8 GPUs



Deep RL may perform quite well in resource assignment problems

Using DRL to solve real-world problems

- Using DRL for traffic optimization in datacenter networks
- Congestion control
- Scheduling distributed machine learning workloads in clusters
- Federated learning

Problems with reinforcement learning

- Sample inefficiency: learning a policy usually needs more samples than you think it will
- It’s difficult to design the reward function

Slides and source code:

iqua.ece.toronto.edu/~bli/iwqos-talk-baochunli.zip

Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

IWQoS 2019, Phoenix, Arizona
June 24, 2019