

COMS E6998 010

# Practical Deep Learning Systems Performance

**Lecture 8 10/05/20**

# Logistics

- Homework 4 posted Nov. 9. Due Nov. 22 by 11:59 PM.
- Homework 3 is due Nov. 7 by 11:59 PM
- Blog on seminar (5 points): due by end of this month
- Final project presentation, Friday Dec. 14, 16, 18.
- Quiz 3 will open Saturday Nov 14.

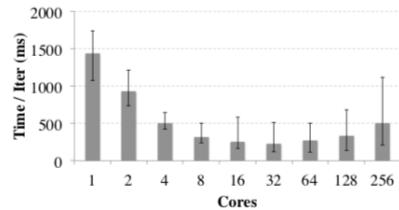
## Recall from last lecture

- Learn to transfer, pseudo labeling
- Automated labeling using KL divergence in transfer learning
- Performance modeling of DL, black box and white box techniques
- Execution time modeling of DL using neural networks
- Analytical modeling of DL in PALEO
- ML based runtime model of DL in AI Gauge

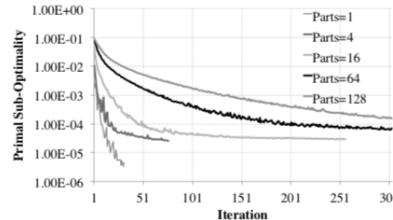
# Distributed ML on Compute Clusters

- Choose the right algorithm
- Choose the cluster size (how many nodes) and node configurations
- Nodes can be virtual servers or bare metal servers (dedicated machines with no resource sharing; no resource virtualization)

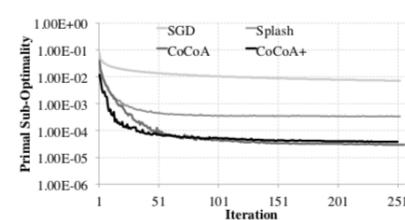
# Distributed ML Performance and Cluster Size



(a) Time per iteration as we vary the degree of parallelism used. The plot shows the mean across 50 iterations and the error bars show the 5th and 95th percentile



(b) Convergence of **CoCoA** as we vary the degree of parallelism used.



(c) Comparison of convergence rate of CoCoA, CoCoA+, SGD and Splash when using 16 cores.

M. Jaggi, V. Smith, et al. Communication-efficient distributed dual coordinate ascent. In *NIPS*, 2014

- Performance (time to converge to certain error threshold) depends on the cluster set up used for training
  - Data parallelism and strong scaling: compute time decreases with increase in the cluster size but communication time increases; per iteration time may first decrease but then increase
- Convergence rates (number of iterations to reach a certain error threshold) of algorithms is dependent on the size of the cluster use
  - With more nodes, batch size per node becomes smaller, the gradient estimates are noisier and it takes more iterations to converge

# Challenges with Distributed ML

- **Changing degree of parallelism affects performance and convergence**
- Computation vs. communication balance and convergence rates differ across algorithms (e.g., first-order methods vs. second-order methods)
  - Hard to predict which algorithm will be the most appropriate for a given cluster setup
- Convergence rate of an algorithm is also data dependent
- Choosing an appropriate algorithm and cluster size for a given task (dataset+problem) is challenging
  - Automated ML systems help in identifying the best algorithm (optimizes some algorithmic performance score) from a candidate list for a given task
  - Choosing the right cluster configuration to run an algorithm is difficult

# Hemingway

- **Goal:** Develop an interface where users can specify an end-to-end requirement and the system will automatically select the appropriate algorithm and degree of parallelism using predictive performance models.
- **Use cases:**
  - Given a relative error goal of  $\epsilon$ , choose the fastest algorithm and configuration
  - Given a target latency of  $t$  seconds choose an algorithm that will achieve the minimum training loss
- **Approach:** A black-box modeling applicable to different distributed ML algorithms

Pan et al. Hemingway: Modeling Distributed Optimization Algorithms. NIPS 2016

# Hemingway: Performance Models

- Build two models:
  - **System model:** captures the system level characteristics of how computation, communication change as we increase cluster sizes
  - **Convergence model:** captures how convergence rates change with cluster sizes
- System model:  $f(m) = \theta_0 + \theta_1 \times (\text{size}/m) + \theta_2 \times \log(m) + \theta_3 \times m$ 
  - iteration time with m node cluster
  - compute time
  - communication time
- Convergence model:  $g(i, m) = \sum_{j=1}^k \lambda_j \phi_j(i, m)$ 
  - score after i iterations with m node cluster
  - Regression with non-linear features;  
Coefficients  $\lambda_j$  determined by standard techniques like least square, LASSO etc.
- Combined model:  $h(t, m) = g(t/f(m), m)$

# Model decomposition helps

- Allows to train the two models independently and reuse them based on changes.
- Retrain the system model and reuse the convergence model
  - New machine types or networking hardware changes in a datacenter
- Retrain the convergence model and reuse the system model
  - Changes to the algorithm in terms of parameter tuning
- What about changes to batch size ? Will it require retraining both the models ?

# Hemingway: Experimental Evaluation

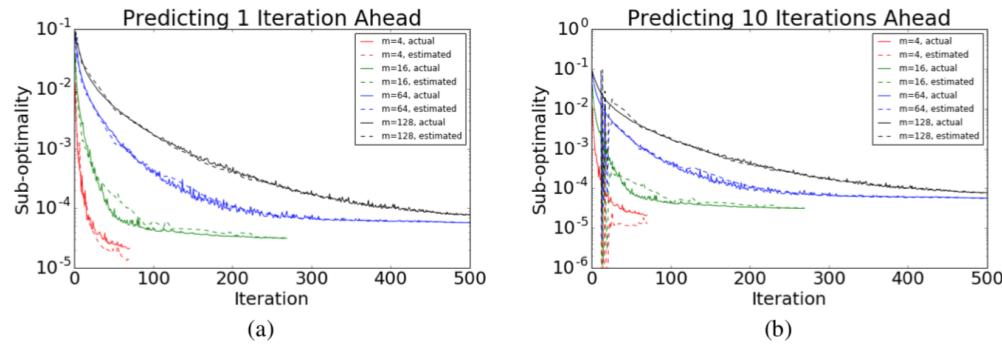


Figure 5: Convergence of CoCoA+ compared against our fitted model when predicting for future iterations.

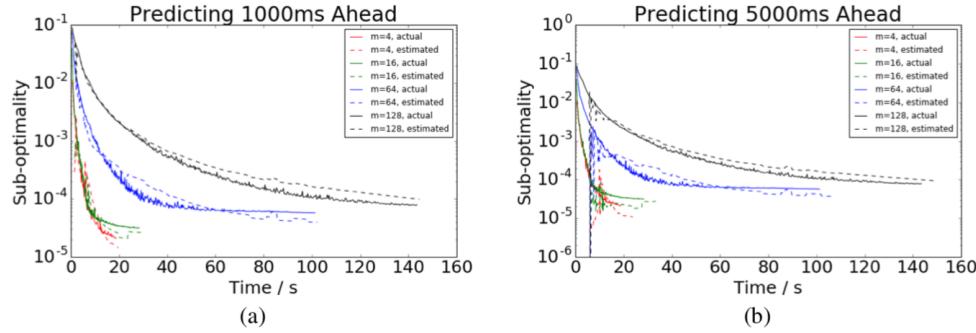


Figure 6: Convergence of CoCoA+ compared against our fitted model when predicting in future time.

# Hemingway in Action

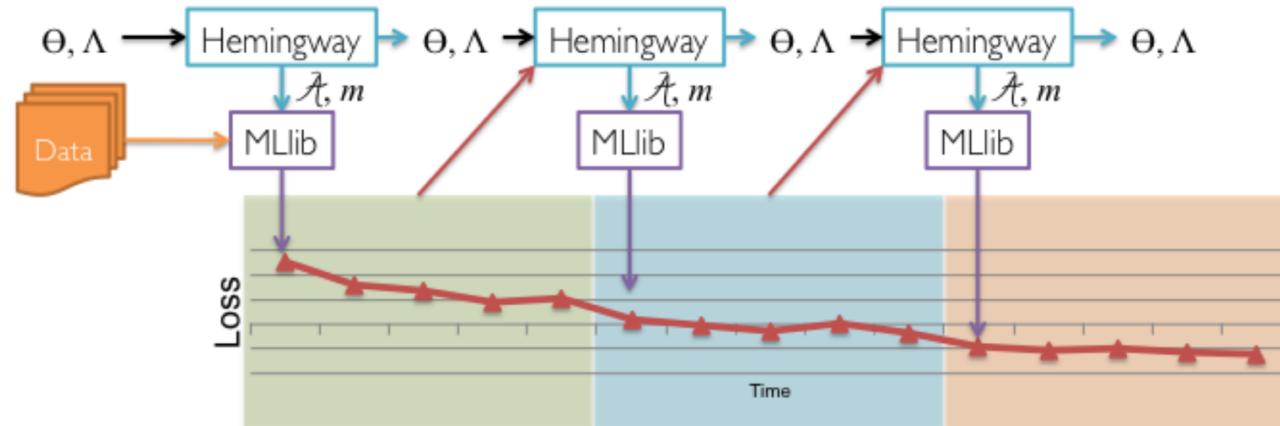


Figure 2: Idealized Example of using Hemingway. For each time frame, Hemingway takes as input the current estimated system model  $\Theta$  and convergence model  $\Lambda$ , and suggests the best algorithm  $\mathcal{A}$  and number of machines  $m$  to use for the next time frame. These are then fed into a machine learning framework, e.g. MLlib, which executes the convex optimization algorithm. Resultant losses are provided as input into Hemingway to update  $\Theta$  and  $\Lambda$ .

# Optimus

- **Goal:** To learn relation between resource configuration and the time a training job takes to achieve model convergence to make efficient scheduling decisions on Deep Learning cluster
- **Approach:**
  - Estimate number of remaining epochs needed to converge
    - Model training loss as a function of number of epochs
  - Estimate the time to complete per epoch as a function of hardware resources
    - Model time to finish per epoch as a function of number of learners

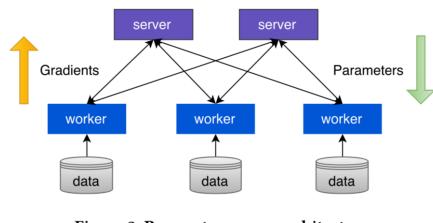
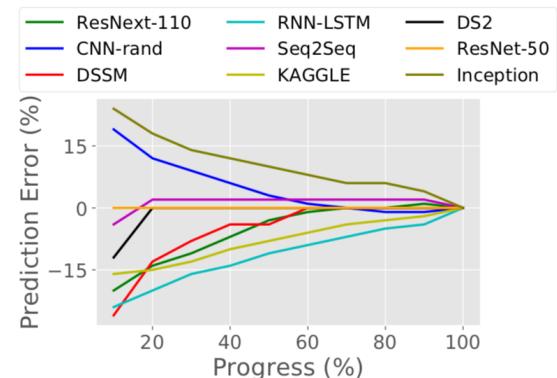
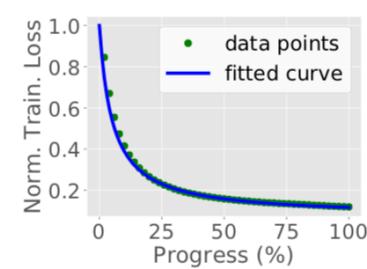
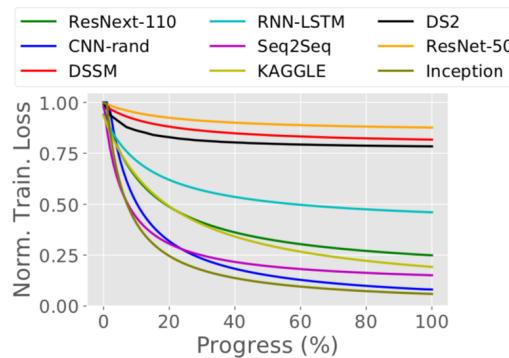


Figure 3: Parameter server architecture

Y. Peng et al. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. Eurosys 2018

# Optimus: Training loss modeling

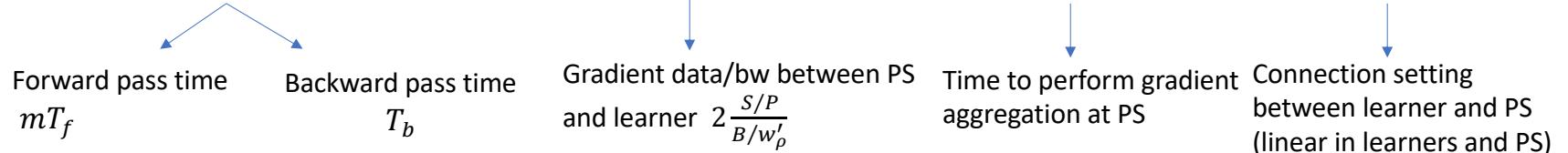
- Training loss model: 
$$l = \frac{1}{\beta_0 \cdot k + \beta_1} + \beta_2 \quad l : \text{training loss}$$
  $k : \text{number of epochs}$
- Online fitting (using a regression solver) after every epoch using the entire history
- Using the model and a convergence error threshold we can calculate the remaining number of training epochs needed



# Optimus: Resource-speed modeling

Modeling duration of one training step(iteration) on a learner  $T$

$$T_\rho = \text{Gradient compute at learner} + \text{Gradient comm. time} + \text{Gradient update time at PS } \rho + \text{Comm. overhead}$$



$$T_\rho = mT_f + T_b + 2 \frac{S/P}{B/w'_\rho} + \frac{T_{\text{update}} \cdot w'_\rho}{P} + \delta \cdot w + \delta' \cdot P$$

$$T = \max_\rho T_\rho$$

- Forward propagation time to process one sample:  $T_f$
- Forward propagation time to process a mini-batch:  $mT_f$
- Backward propagation time:  $T_b$
- Model/gradient size:  $S$
- Size of gradients transferred from a worker to a PS:  $S/P$
- Bandwidth at parameter server (PS):  $B$
- Number of workers that send gradients to a PS concurrently:  $w'$
- Bandwidth between a PS and a worker:  $B/w'$
- Communication time between a PS and a worker:  $2((S/P)/(B/w'))$
- Time to update parameters with size  $S$ :  $T_{\text{update}}$
- Parameter update time on PS:  $(T_{\text{update}} \cdot w')/P$
- Communication overhead:  $\delta \cdot w + \delta' \cdot P$
- Homogeneous workers; Load balanced parameter servers

# Optimus: Resource-speed modeling

- Asynchronous Training:

Where did  $w'$  go ?

$$\text{Throughput} = w \cdot T^{-1}$$

$$f(p, w) = w \cdot (\theta_0 + \theta_1 \cdot \frac{w}{p} + \theta_2 \cdot w + \theta_3 \cdot p)^{-1}$$

- Synchronous Training:

$$\text{Throughput} = T^{-1}$$

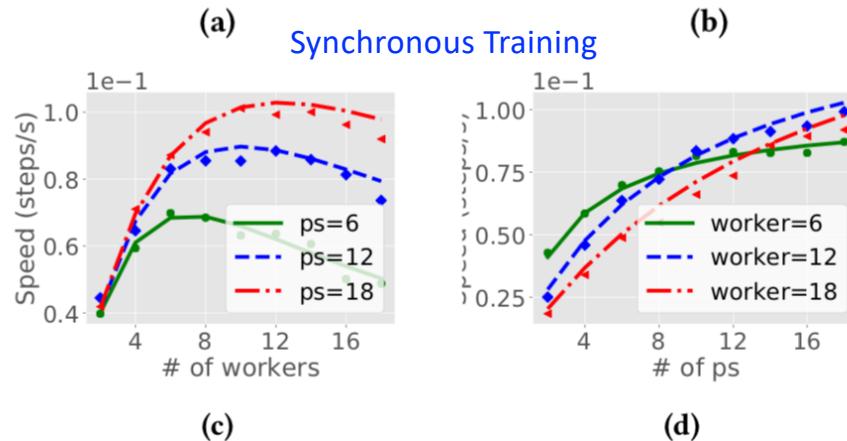
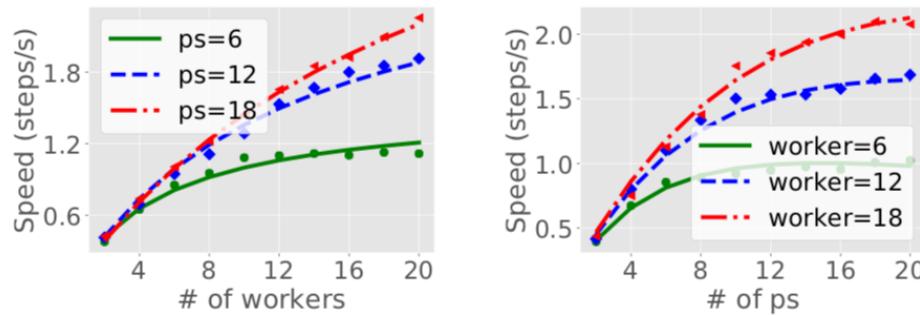
$$f(p, w) = (\theta_0 \cdot \frac{M}{w} + \theta_1 + \theta_2 \cdot \frac{w}{p} + \theta_3 \cdot w + \theta_4 \cdot p)^{-1}$$

Total batch size:  $M$

$$\text{Batch size per learner: } m = \frac{M}{w}$$

# Optimus: Predictive Performance for Strong Scaling

Asynchronous Training



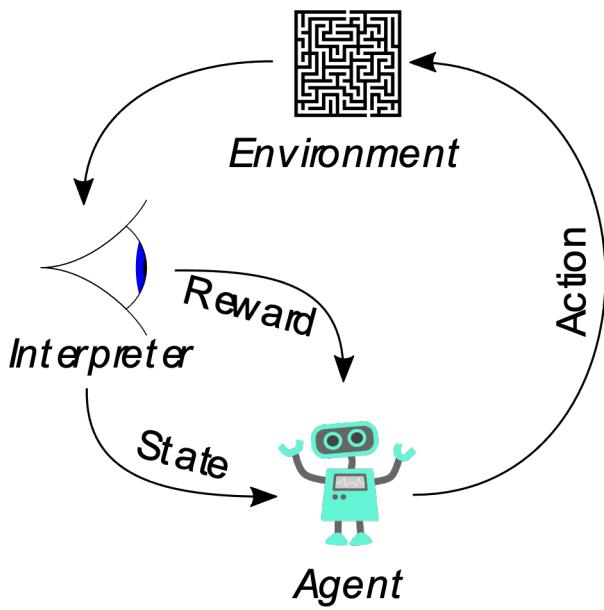
- (a) due to communication overhead, there is a trend of diminishing return where adding more parameter servers or workers does not improve the training speed much
- (b) for synchronous training, more workers may lead to lower training speed.

Table 2: Coefficients in speed functions

	$\theta_1$	$\theta_2$	$\theta_3$	$\theta_4$	$\theta_5$	Residual sum of squares for fitting
Async	2.83	3.92	0.00	0.11	-	0.10
Sync	1.02	2.78	4.92	0.00	0.02	0.00

Less than 10% prediction error

# Reinforcement Learning (RL)



Agent reacts sequentially with an environment with the goal to maximize cumulative reward

Agent tries to learn efficient representations of the environment from high-dimensional sensory input

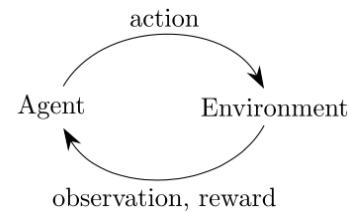
Environment is only partially observed and interpreted

Reinforcement Learning Challenges:

1. Credit-assignment problem: Contribution of each action for the reward is not known
2. Rewards can be probabilistic : can only be estimated approximately in data-driven manner
3. Model generalization is difficult: very large number of states
4. Exploration vs exploitation tradeoff
5. Inability to gather sufficient data in real settings

# Terminology

- Episodic Tasks
- Continuous Tasks
- Exploration
- Exploitation
- $\epsilon$  – *greedy* policy



# RL Formulation

- Action  $a_t$  is selected using some policy  $\pi$

$\pi$  is a mapping from states  $s_t$  to actions  $a_t$ .

$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$  cumulative discounted reward at t

- Action-value function  $Q^\pi(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$ ,  $s_{t+1}, r_t$

Expected return for taking action a in state s and then following  $\pi$

- Optimal action-value function:

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a)$$

Maximum action value for state s and action achievable by any policy.

- Optimal action-value functions obeys Bellman's equation:

$$Q^*(s, a) = \mathbb{E} \left[ r + \gamma \max_{a'} Q^*(s', a') \right]$$

Reward r can be dependent on (s,a,s')  
Probability distribution on the state s' reachable by taking action a in s

# Q-learning

- In reinforcement learning we represent the (optimal) action-value function using a function approximator which we try to learn
  - A neural network (Q-network) can be used as function approximator

- The parameters  $\theta$  of the *Q-network* are optimized so as to approximately solve the Bellman equation.

# DQN Algorithm

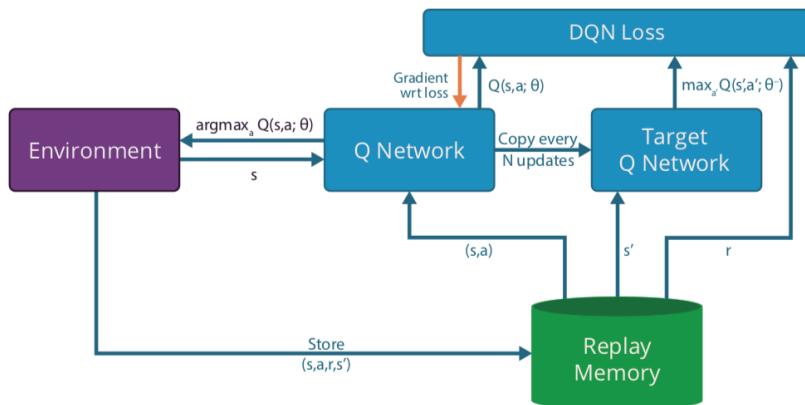


Figure 1. The DQN algorithm is composed of three main components, the Q-network ( $Q(s, a; \theta)$ ) that defines the behavior policy, the target Q-network ( $Q(s, a; \theta^-)$ ) that is used to generate target Q values for the DQN loss term and the replay memory that the agent uses to sample random transitions for training the Q-network.

DQN differs from Q-Learning in two ways:

1. DQN uses experience replay

At each time-step  $t$  during an agent's interaction with the environment it stores the experience tuple in replay memory

$$e_t = (s_t, a_t, r_t, s_{t+1}) \quad \text{Experience tuple}$$

$$D_t = \{e_1, \dots, e_t\} \quad \text{Replay memory}$$

2. Use of 2 separate Q-networks

$$Q(s, a; \theta) \text{ and } Q(s, a; \theta^-)$$

Q-network
Target Q-network

$$L_i(\theta_i) = \mathbb{E} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

# DQN Algorithm

**Algorithm 1:** deep Q-learning with experience replay.

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

        With probability  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$

        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$

        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$

        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

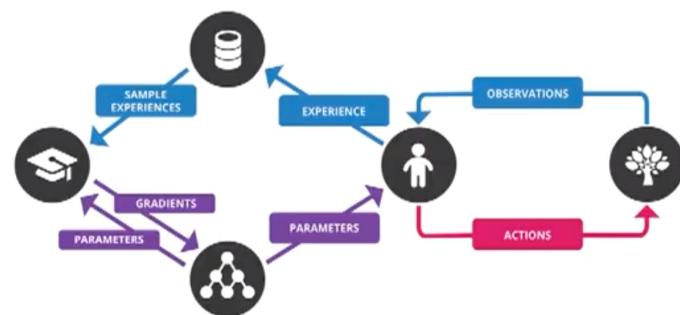
$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$

        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

        Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**



# Why Experience Replay Helps ?

- Prevents instability in Q-learning
- Reuse of data
- Removes correlation in input batch samples
  - Prevents overfitting

# GORILA

- Asynchronous training of reinforcement learning agents in a distributed setting
- Gorila agent parallelizes the training procedure by separating out learners, actors and parameter server
- Gorila components:
  - Actors (multiple)
  - Experience Replay Memory (local and global)
  - Learners (multiple)
  - Parameter server (sharded)
- Considerable flexibility in the number of ways an RL agent may be parallelized
  - Give examples

# GORILA Architecture

scalable distributed training

Using sharded parameter server

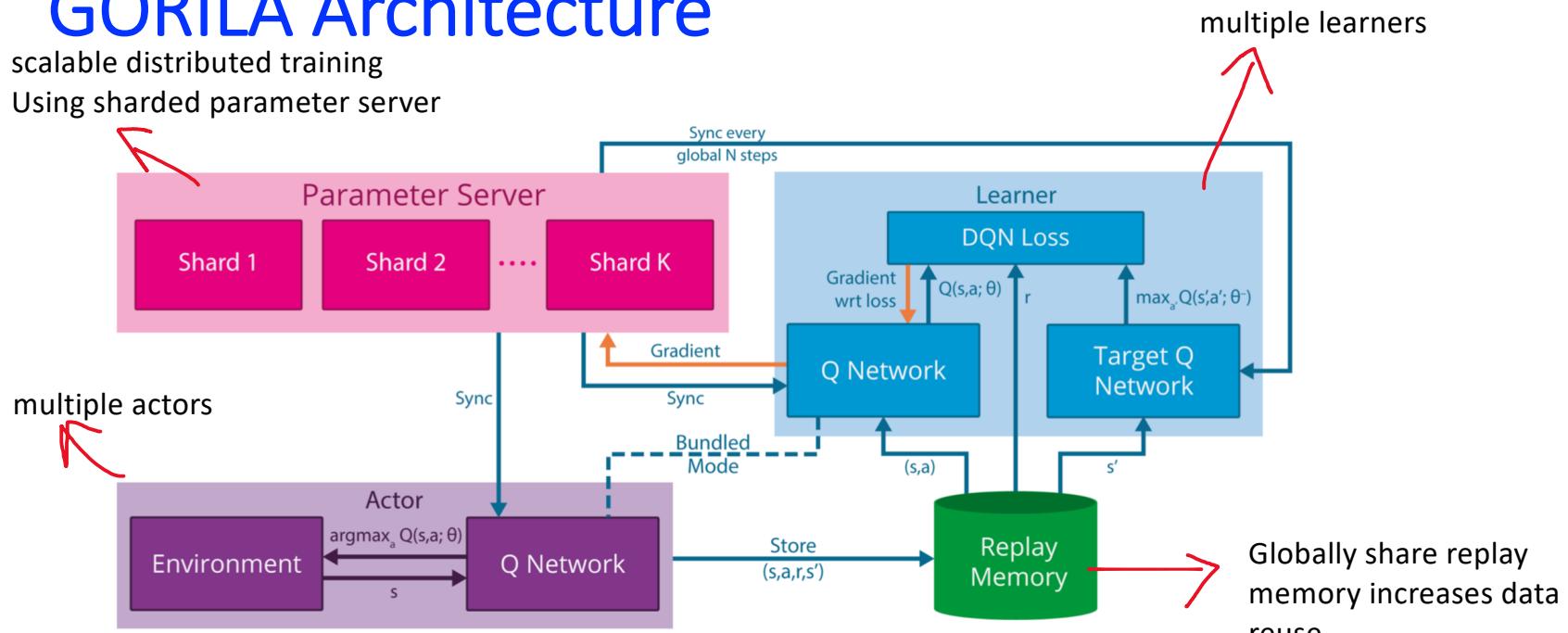


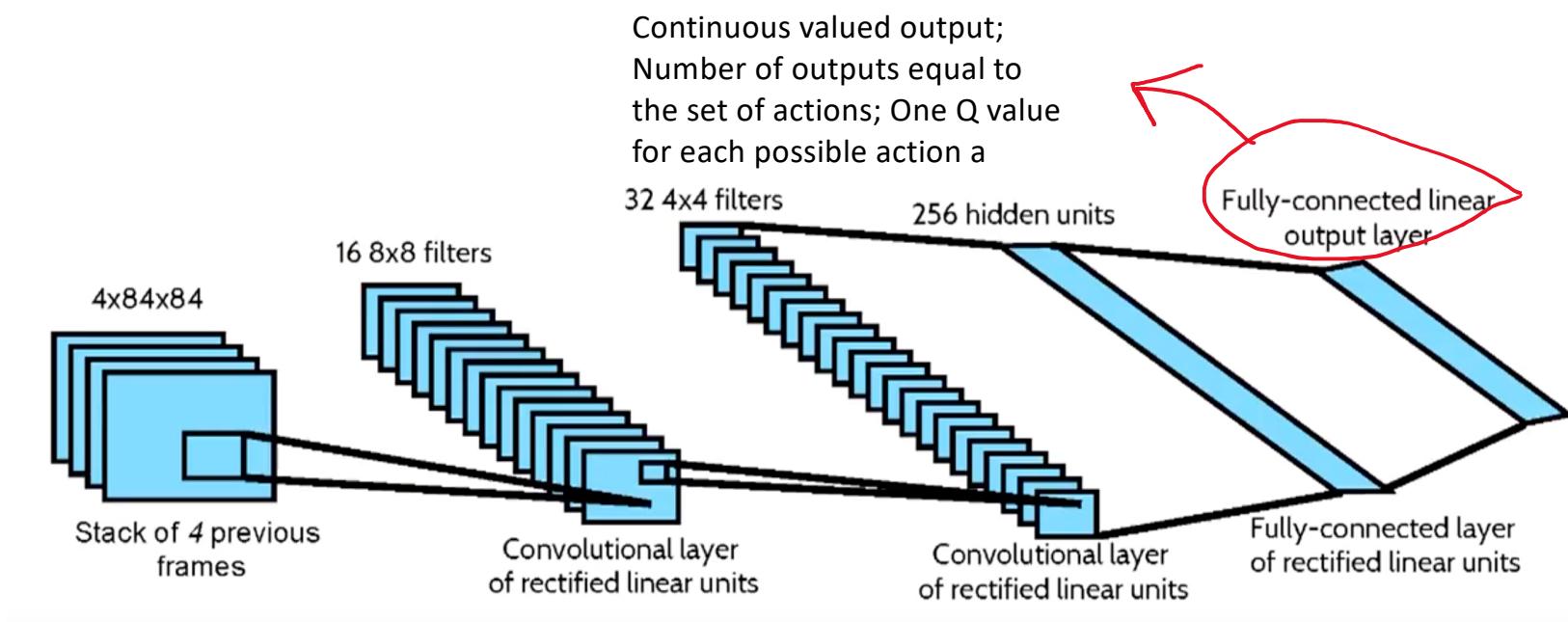
Figure 2. The Gorila agent parallelises the training procedure by separating out learners, actors and parameter server. In a single experiment, several learner processes exist and they continuously send the gradients to parameter server and receive updated parameters. At the same time, independent actors can also in parallel accumulate experience and update their Q-networks from the parameter server.

# GORILA: Experimental Evaluation

- 49 Atari 2600 games
- An agent must learn to play the games directly from  $210 \times 160$  RGB video input
- Only the changes in the score provided as rewards
- $210 \times 160$  RGB images preprocessed by downsampling them to  $84 \times 84$
- Input consists of 4 image frames

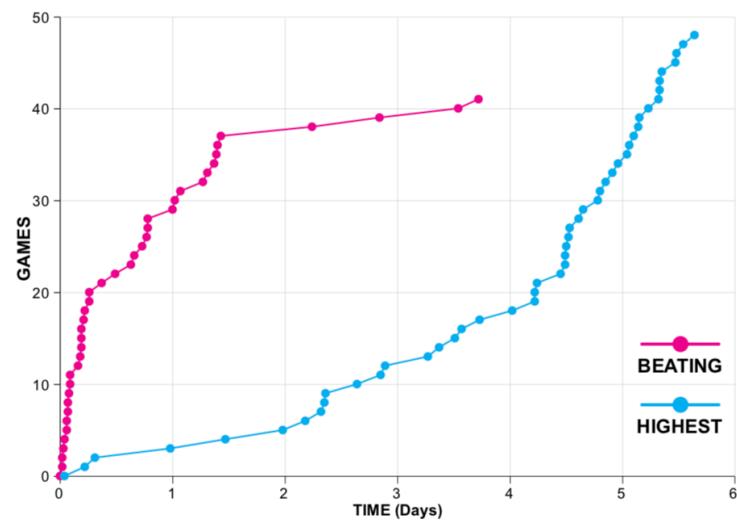
In all experiments, Gorila DQN used:  $N_{param} = 31$  and  $N_{learn} = N_{act} = 100$ . We use the bundled mode. Replay memory size  $D = 1$  million frames and used  $\epsilon$ -greedy as the behaviour policy with  $\epsilon$  annealed from 1 to 0.1 over the first one million global updates. Each learner syncs the parameters  $\theta^-$  of its target network after every 60K parameter updates performed in the parameter server.

# ATARI Convolutional Neural Network



One output per action: expected action-value for that action  $Q(s,a)$

# GORILA: Experimental Evaluation



Explain the graph and significance of Gorila DQN over single DQN; Recall the cause of speed-up

Figure 5. The time required by Gorila DQN to surpass single DQN performance (red curve) and to reach its peak performance (blue curve).

Nair et al. Massively Parallel Methods for Deep Reinforcement Learning. 2015

# Prioritized Experience Replay

- Replying all transitions with equal probability is highly suboptimal
- Experience replay may also help to prevent overfitting by allowing the agent to learn from data generated by previous versions of the policy.
- Focus learning on the most ‘surprising’ experiences
- Biased Sampling
  - Reward signal may be sparse and the data distribution depends on the agent’s policy

# Ape-X

- Distributed Prioritized Experience Replay

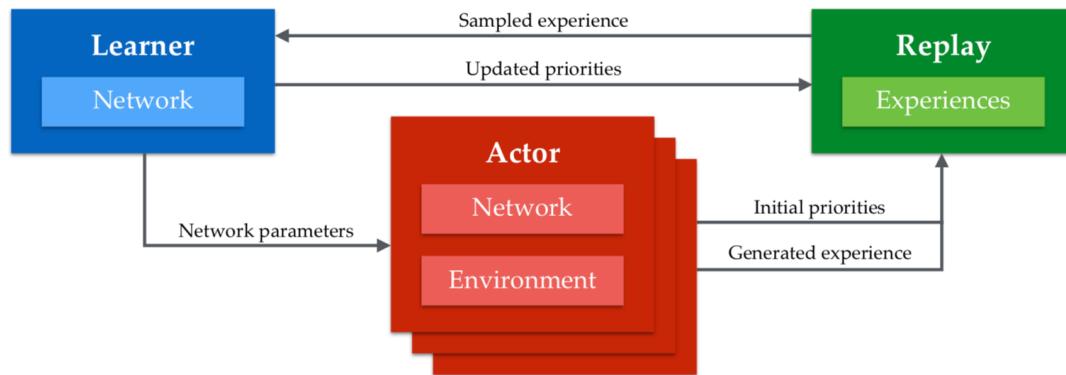


Figure 1: The Ape-X architecture in a nutshell: multiple actors, each with its own instance of the environment, generate experience, add it to a shared experience replay memory, and compute initial priorities for the data. The (single) learner samples from this memory and updates the network and the priorities of the experience in the memory. The actors' networks are periodically updated with the latest network parameters from the learner.

# Ape-X Performance

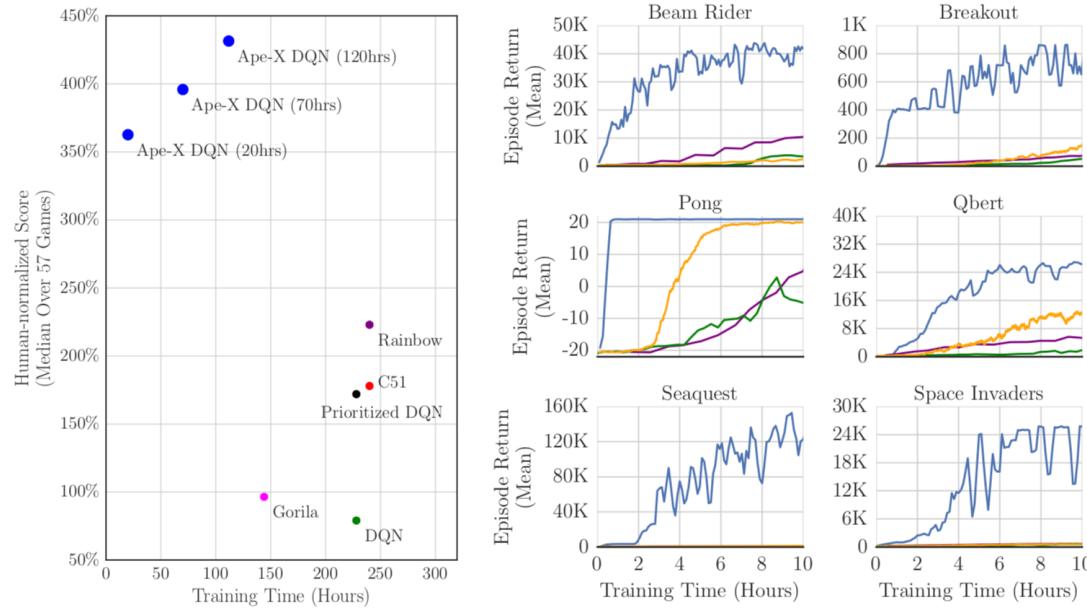


Figure 2: Left: Atari results aggregated across 57 games, evaluated from random no-op starts. Right: Atari training curves for selected games, against baselines. Blue: Ape-X DQN with 360 actors; Orange: A3C; Purple: Rainbow; Green: DQN. See appendix for longer runs over all games.

Horgan et al. DISTRIBUTED PRIORITIZED EXPERIENCE REPLAY. ICLR 2018

# Ape-X Performance

Algorithm	Training Time	Environment Frames	Resources (per game)	Median (no-op starts)	Median (human starts)
Ape-X DQN	5 days	22800M	376 cores, 1 GPU <sup>a</sup>	<b>434%</b>	<b>358%</b>
Rainbow	10 days	200M	1 GPU	223%	153%
Distributional (C51)	10 days	200M	1 GPU	178%	125%
A3C	4 days	—	16 cores	—	117%
Prioritized Dueling DQN	9.5 days	200M	1 GPU	172%	115%
Gorila DQN <sup>c</sup>	~4 days	—	unknown <sup>b</sup>	96%	78%
UNREAL <sup>d</sup>	—	250M	16 cores	331% <sup>d</sup>	250% <sup>d</sup>

Table 1: Median normalized scores across 57 Atari games. <sup>a</sup> Tesla P100. <sup>b</sup> >100 CPUs, with a mixed number of cores per CPU machine. <sup>c</sup> Only evaluated on 49 games. <sup>d</sup> Hyper-parameters were tuned per game.

# Ape-X Scaling: Number of Actors

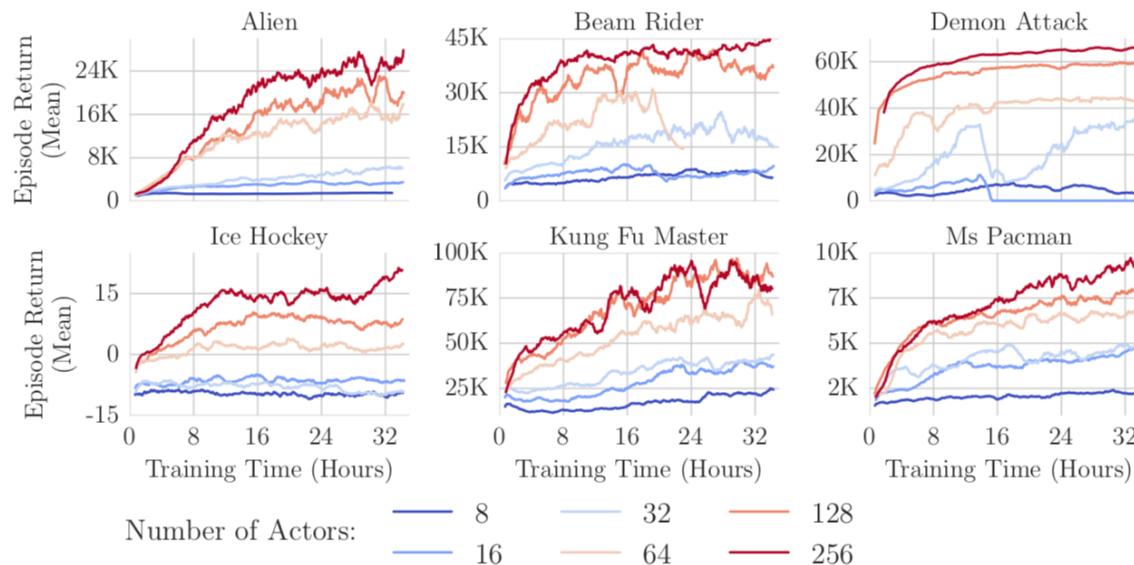


Figure 4: Scaling the number of actors. Performance consistently improves as we scale the number of actors from 8 to 256, note that the number of learning updates performed does not depend on the number of actors.

# Ape-X Scaling: Replay Capacity

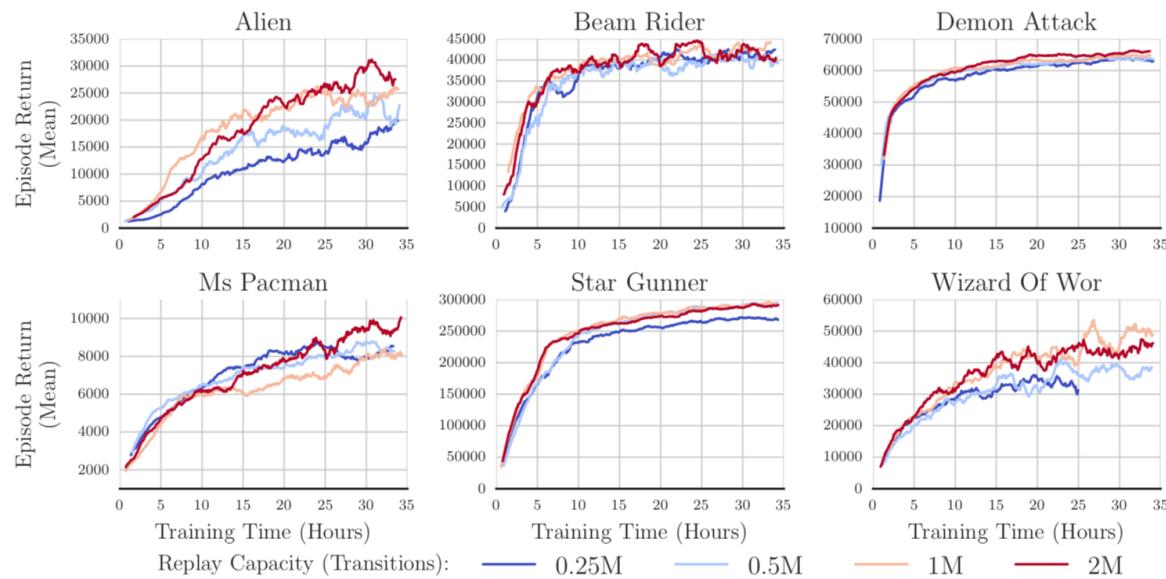


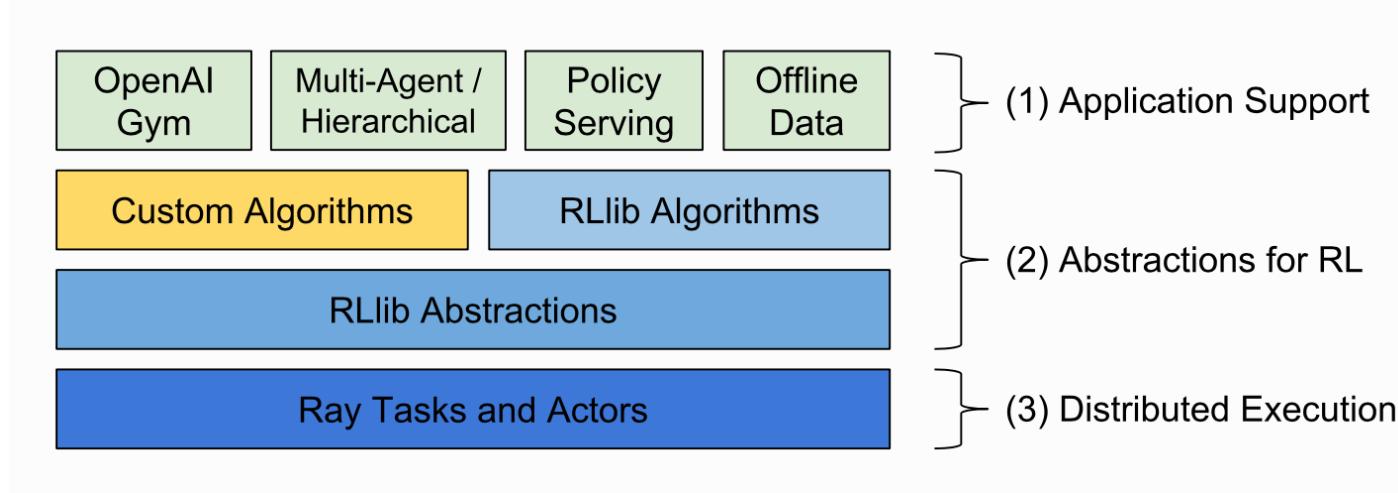
Figure 5: Varying the capacity of the replay. Agents with larger replay memories perform better on most games. Each curve corresponds to a single run, smoothed over 20 points. The curve for Wizard Of Wor with replay size 250K is incomplete because training diverged; we did not observe this with the other replay sizes.

# Deep RL Simulators

- OpenAI Gym
  - <https://gym.openai.com/docs/>
  - A toolkit for developing and comparing reinforcement learning algorithms.
  - **Gym** provides an environment to implement any reinforcement learning algorithms.
  - Developers can write agents in TensorFlow
  - Limitation: no multi-agent support
- Full 3D Environments
  - Unity ML Agents: <https://unity3d.com/machine-learning>

# RLLib

- <https://ray.readthedocs.io/en/latest/rllib.html>
- Open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLLib natively supports TensorFlow, TensorFlow Eager, and PyTorch.



---

# Seminar Reading List

- **DL resource estimation**
  - Pan et al. [Hemingway: Modeling Distributed Optimization Algorithms](#). NIPS 2016
  - Shi et al. [Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs](#). 2018
  - Peng et al. [Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters](#). Eurosys 2017.
- **Deep Reinforcement Learning**
  - Mnih et al. [Playing Atari with Deep Reinforcement Learning](#). 2013
  - Mnih et al. [Human-level control through deep reinforcement learning](#). Nature. 2015
  - Nair et al. [Massively Parallel Methods for Deep Reinforcement Learning](#). 2015
  - Mnih et al. [Asynchronous Methods for Deep Reinforcement Learning](#). 2016
  - Horgan et al. [DISTRIBUTED PRIORITIZED EXPERIENCE REPLAY](#). ICLR 2018
  - Espeholt et al. [IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures](#). 2018

## Links, Blogs

- [RLLib: Scalable Reinforcement Learning](#)
- [Get to grips on the latest AI power in Unity](#)