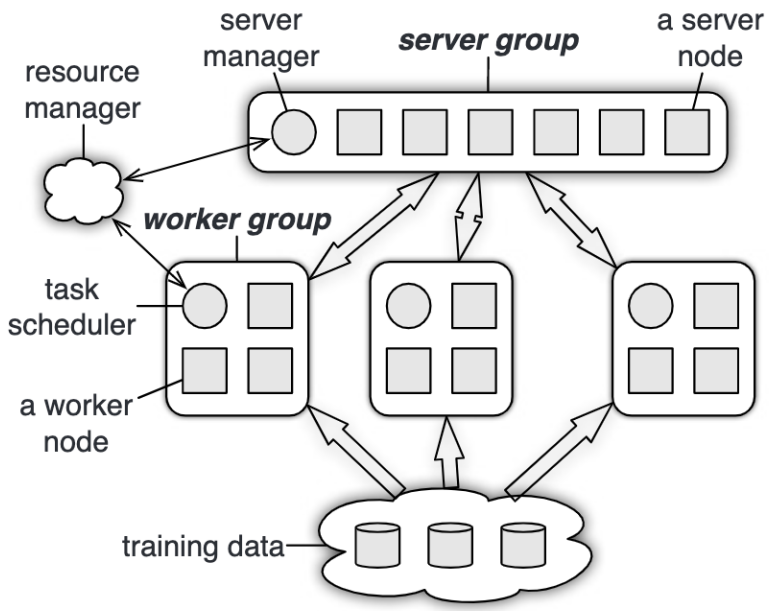

Figure 4: Architecture of a parameter server communicating with several groups of workers.

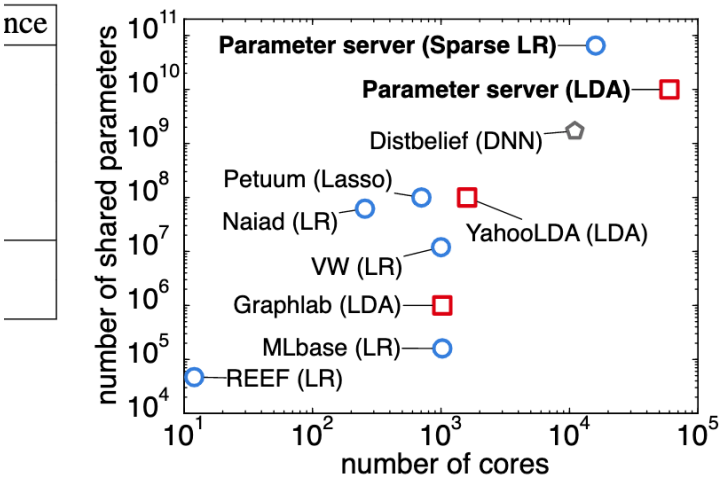

Figure 1: Comparison of the public largest machine learning experiments each system performed. Problems are color-coded as follows: Blue circles — sparse logistic regression; red squares — latent variable graphical models; grey pentagons — deep networks.

Above figures borrowed from
"Scaling Distributed Machine Learning with the Parameter Server", OSDI'14

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big)$
  $\widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

return θ_t

Adam optimization (https://pytorch.org/docs/stable/generated/torch.optim.Adam.html)

3 Design principles

PyTorch's success stems from weaving previous ideas into a design that balances speed and ease of use. There are four main principles behind our choices:

Be Pythonic Data scientists are familiar with the Python language, its programming model, and its tools. PyTorch should be a first-class member of that ecosystem. It follows the commonly established design goals of keeping interfaces simple and consistent, ideally with one idiomatic way of doing things. It also integrates naturally with standard plotting, debugging, and data processing tools.

Put researchers first PyTorch strives to make writing models, data loaders, and optimizers as easy and productive as possible. The complexity inherent to machine learning should be handled internally by the PyTorch library and hidden behind intuitive APIs free of side-effects and unexpected performance cliffs.

Provide pragmatic performance To be useful, PyTorch needs to deliver compelling performance, although not at the expense of simplicity and ease of use. Trading 10% of speed for a significantly simpler to use model is acceptable; 100% is not. Therefore, its *implementation* accepts added complexity in order to deliver that performance. Additionally, providing tools that allow researchers to manually control the execution of their code will empower them to find their own performance improvements independent of those that the library provides automatically.

Worse is better [26] Given a fixed amount of engineering resources, and all else being equal, the time saved by keeping the internal implementation of PyTorch simple can be used to implement additional features, adapt to new situations, and keep up with the fast pace of progress in the field of AI. Therefore it is better to have a simple but slightly incomplete solution than a comprehensive but complex and hard to maintain design.

Above paragraphs borrowed from
"PyTorch: An Imperative Style, High-Performance Deep Learning Library", NeurIPS'19