ST446 Distributed Computing for Big Data

Lecture 9

Scaling up distributed machine learning

Milan Vojnovic

https://github.com/lse-st446/lectures2021

Topics of this lecture

- Introduction

- Data parallel computation

- Scaling by reducing precision: quantized gradient descent

- Model parallel computation

Platforms for training machine learning models

- TensorFlow: https://www.tensorflow.org
- PyTorch: https://pytorch.org
- Cognitive Toolkit: https://www.microsoft.com/en-us/cognitive-toolkit
- …

- All these platforms allow for scaling up using distributed training

- Some examples:
  - TensorFlow: https://www.tensorflow.org/guide/distribute_strategy
  - PyTorch: https://pytorch.org/tutorials/intermediate/dist_tuto.html
  - Cognitive Toolkit: https://docs.microsoft.com/en-us/cognitive-toolkit/multiple-gpus-and-machines

- Popular application: training deep neural network models

Background: neural networks

- Single layer-perceptron:

output
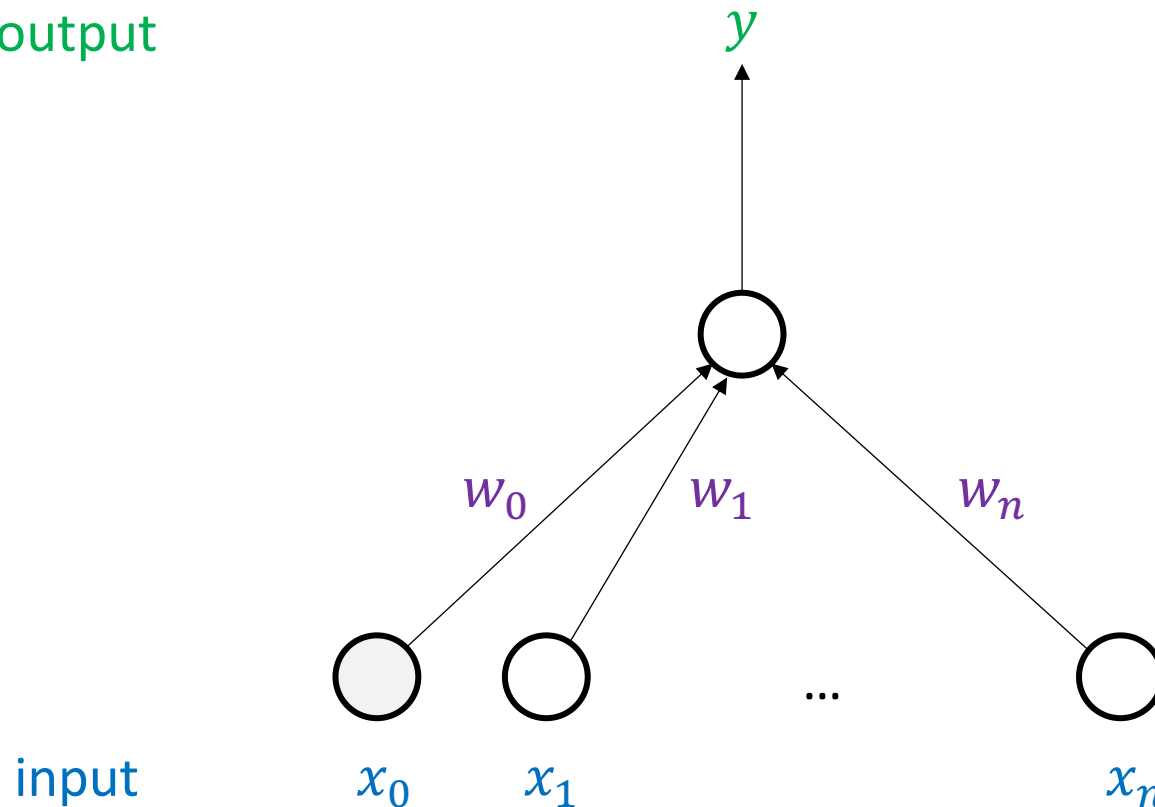
$y = a(\boldsymbol{x}^\top \boldsymbol{w} + b)$

activation function

w_0 w_1 w_n

$b := w_0 x_0$

...

$\boldsymbol{w} = (w_1, w_2, \ldots, w_n)^\top$

input x_0 x_1 x_n

- Example: logistic regression $a(x) = 1/(1 + e^{-x})$

Feedforward neural networks

- Many layers (depth)
- Many neurons in some layers (width)
- Many parameters (high dimensional models)

$\Rightarrow$ long training time
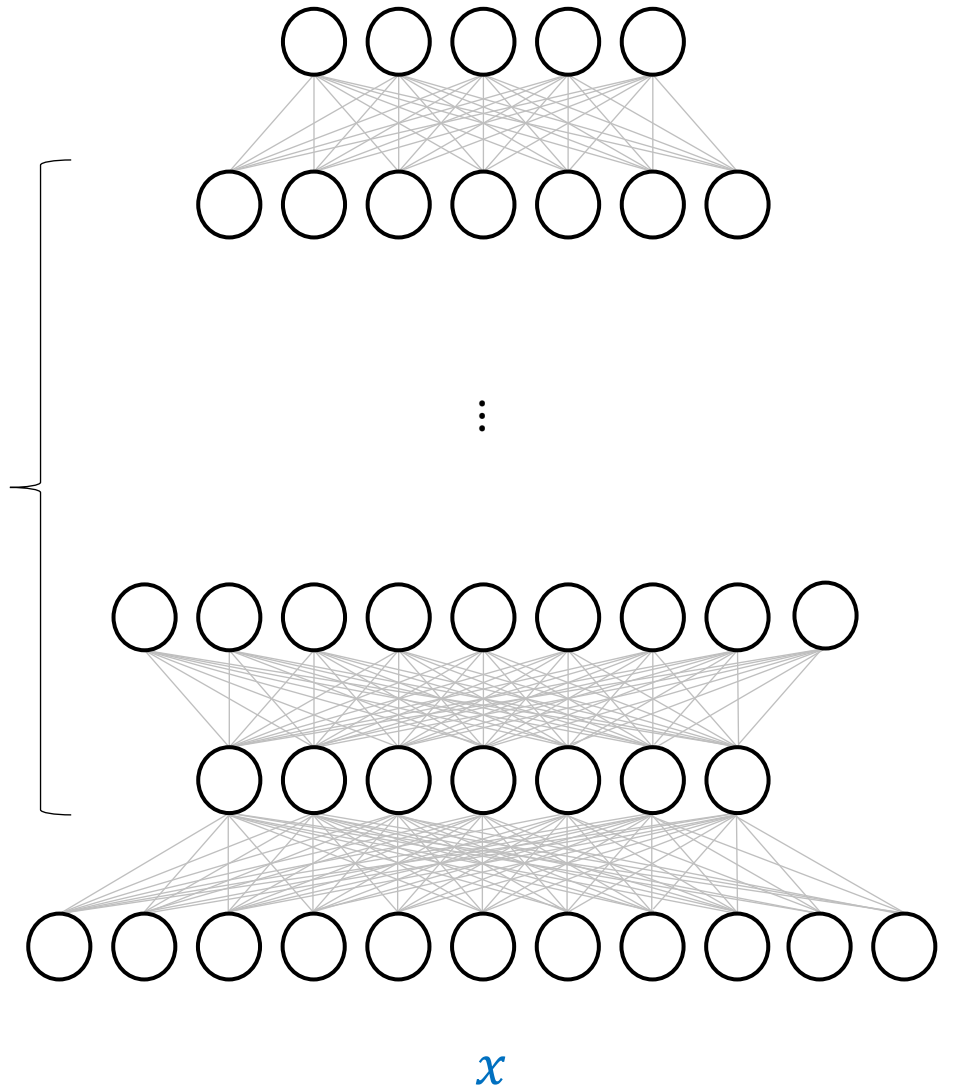
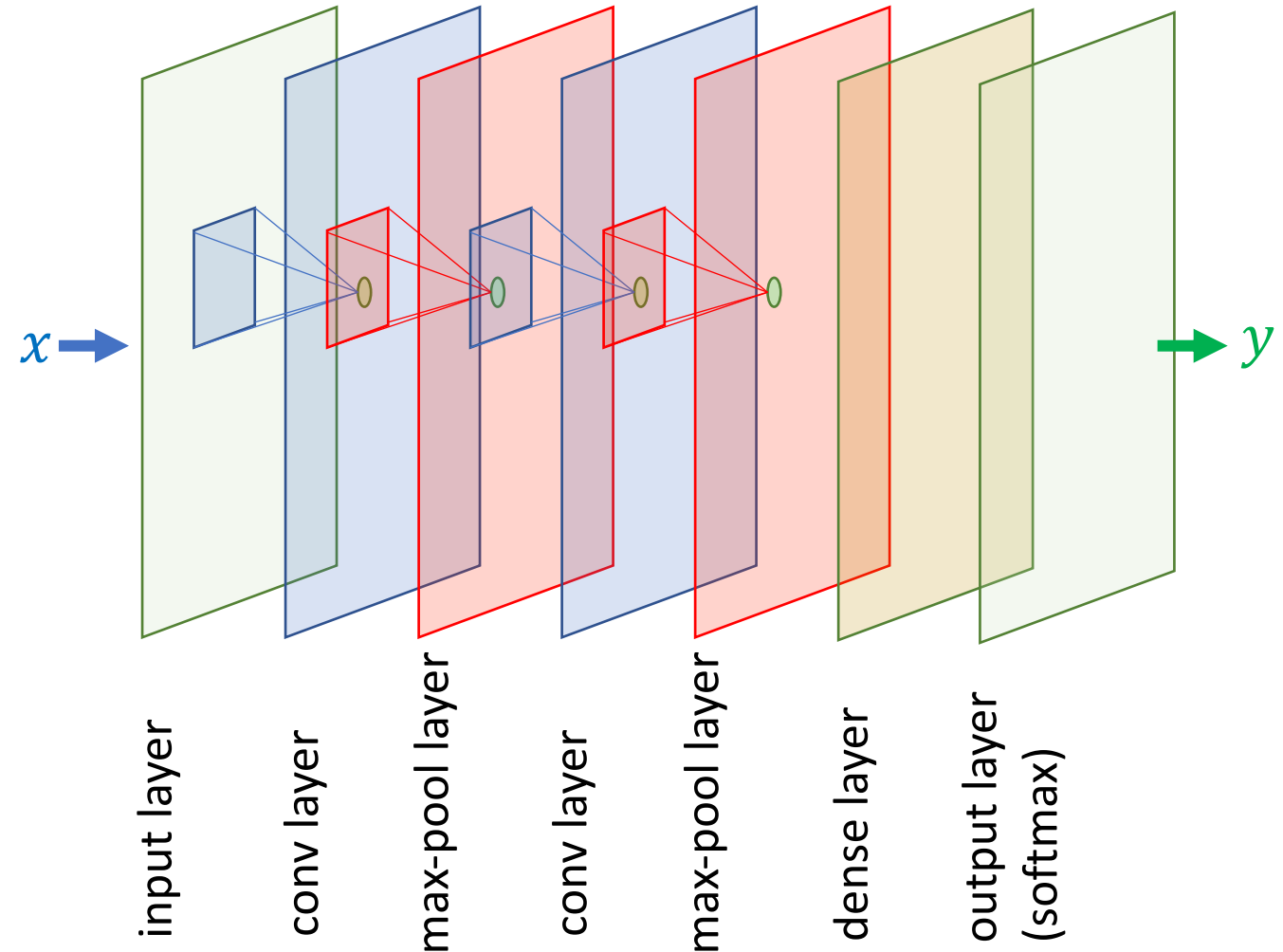- Need ways to scale up training !

output y

hidden layers

input

x

Convolutional neural networks

- Special type of a feedforward neural network architecture that achieved better than human accuracy for image recognition tasks

- A neural network is a convolutional neural network if it has at least one convolutional layer

- Convolutional layers have weights realizing convolutional filters

x → ... → y

input layer · conv layer · max-pool layer · conv layer · max-pool layer · dense layer · output layer (softmax)

To probe further on convolutional neural networks

- If you are taking ST449 Artificial Intelligence and Deep Learning this term, then you know already what convolutional neural networks are

- If not, you may consult a quick tutorial, e.g.

   Marin Goerner, Learn TensorFlow and deep learning without a PhD

Training machine learning models

- Training data examples: $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$
  - x_i is a feature vector in $\mathbf{R}^n$
  - y_i is a label / response variable, ex. for binary classification taking values in $\{-1, 1\}$

- Model: $x \mapsto f_w(x)$ (e.g. $f_w(x)$ defined by a neural network)

- Loss function minimization:

$$\ell(w) = \frac{1}{m} \sum_{i=1}^{m} \ell(w; x_i, y_i)$$

loss function

- Examples of loss functions:
  - Squared loss: $\ell(w; x, y) = \frac{1}{2}(y - f_w(x))^2$
  - Cross entropy: $\ell(w; x, y) = -y \log(f_w(x)) - (1-y) \log(1 - f_w(x))$

Stochastic gradient descent

- Key idea: use a gradient descent update rule with a stochastic gradient vector

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

where η_t is a step size (either constant or decaying) and

$\boldsymbol{g}^{(t)}$ is a stochastic gradient vector, assumed to be an unbiased estimator, i.e.

$$\mathbf{E}\big[\, \boldsymbol{g}^{(t)} \mid \boldsymbol{w}^{(t)} = \boldsymbol{w} \,\big] = \underbrace{\nabla \ell(\boldsymbol{w})}$$

gradient vector of the
loss function ℓ at $\boldsymbol{w}$

- Simple SGD: $\boldsymbol{g}^{(t)} := \nabla \ell(\boldsymbol{w}^{(t)}; \boldsymbol{x}_{I(t)}, y_{I(t)})$ where $I(t)$ is a random training example

- Minibatch SGD: $\boldsymbol{g}^{(t)} := \frac{1}{b} \sum_{i \in B_t} \nabla \ell(\boldsymbol{w}^{(t)}; \boldsymbol{x}_i, y_i)$ where B_t is a mini-batch of training examples with mini-batch size $|B_t| = b$

Training neural networks

- Using scalable iterative optimizers based on stochastic gradient descent and its variations
  - Acceleration by using momentum
  - Learning rate decay

- Other tricks
  - Random initial weights
  - Early stopping (for regularization)
  - Dropout: randomly dropping neurons (for regularization)

- Gradient vector computation: backpropagation
  - Forward computation: compute hidden-layer and output values for given input
 - Backward computation: compute gradients

Examples of training and test data sizes

- MNIST: handwritten digits (28x28 black-and-white images)
  - **60,000** training examples
  - **10,000** test examples
  - Training data size: **9.91MB**

- CIFAR (32x32 color images)
  - CIFAR-10: **60,000** examples, **10** classes, **6,000** images per class, **163MB** gz
  - CIFAR-100: **60,000** examples, **100** classes, **600** images per class, **161MB** gz

- ImageNet: image database organized according to the WordNet hierarchy
  - **14,197,122** examples

High-dimensional models

- Image recognition:
  - AlexNet: **60M** parameters, **650K** neurons, **8** layers, **1000**-dim output space
  - Inception (GoogLeNet): **2.7K-1,388K** parameters, **22** layers
  - ResNet: **60M** parameters, **152** layers

- Speech:
  - LACEA (LSTM RNN): **65M** parameters, **22** layers

ImageNet accuracy versus model size
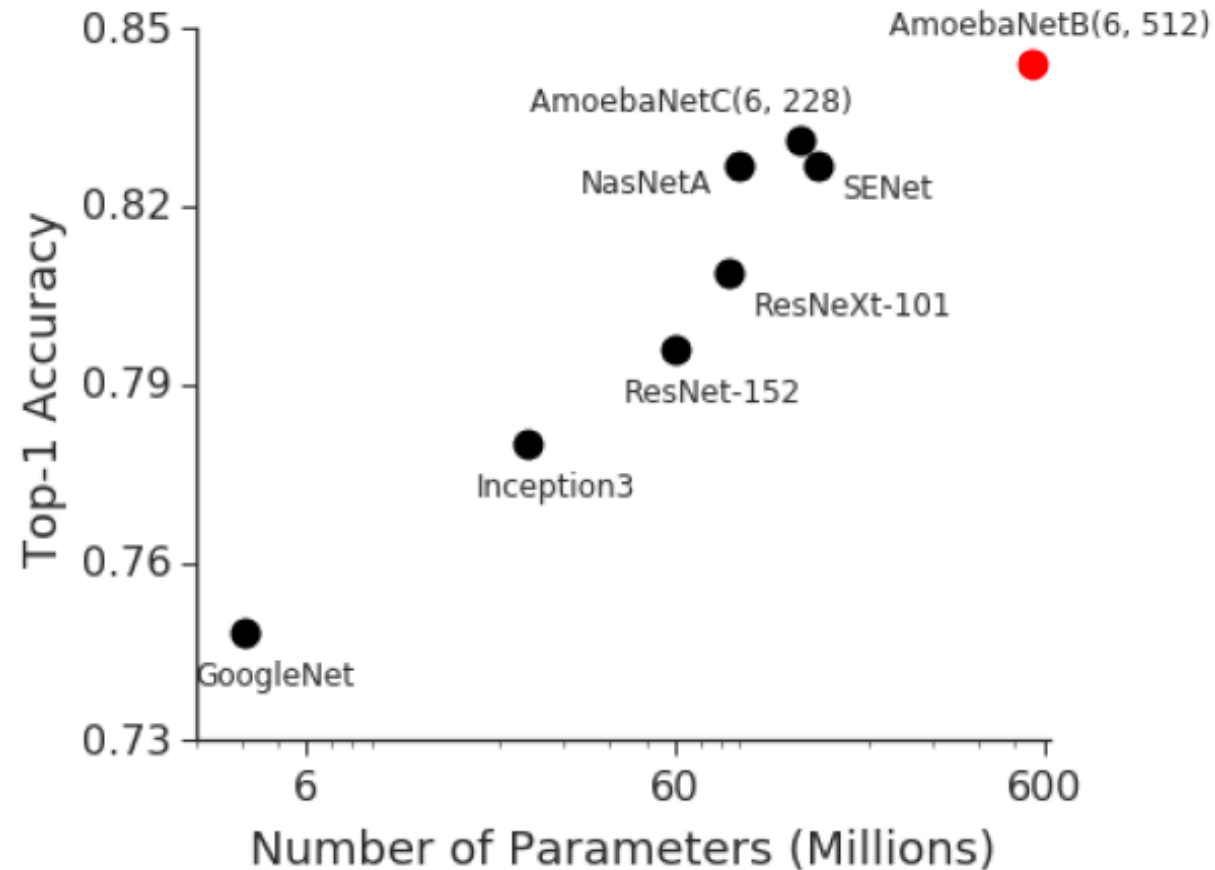
- Source: Google AI blog: Introducing GPipe, an Open Source Library for Efficiently Training Large-scale Neural Network Models, March 4, 2019

ImageNet competition trends

Team	Number of workers	Mini-batch size	Training time	Speedup	Top-1 accuracy (%)
He et al (2016)	8	256	29 hours	1.0	75.3
Goyal et al (2017)	256	8,192	1 hour	29	76.3
Akiba et al (2017)	1024	32,768	15 minutes	116	74.9
You et al (2018)	2048	32,768	14 minutes	124	74.9

Source: Ryota Tomioka, Introduction to Distributed Deep Learning, LSE Lecture, 5th March 2018
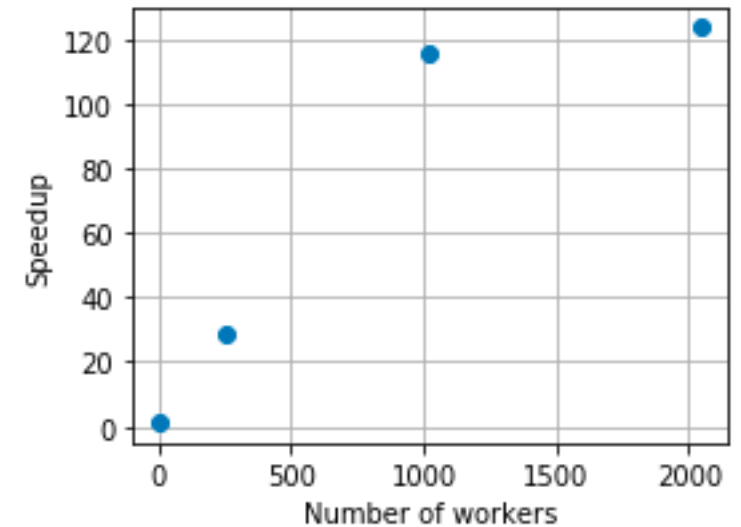
- Significant computation time speedup can be achieved by distributed training

Data parallel computation
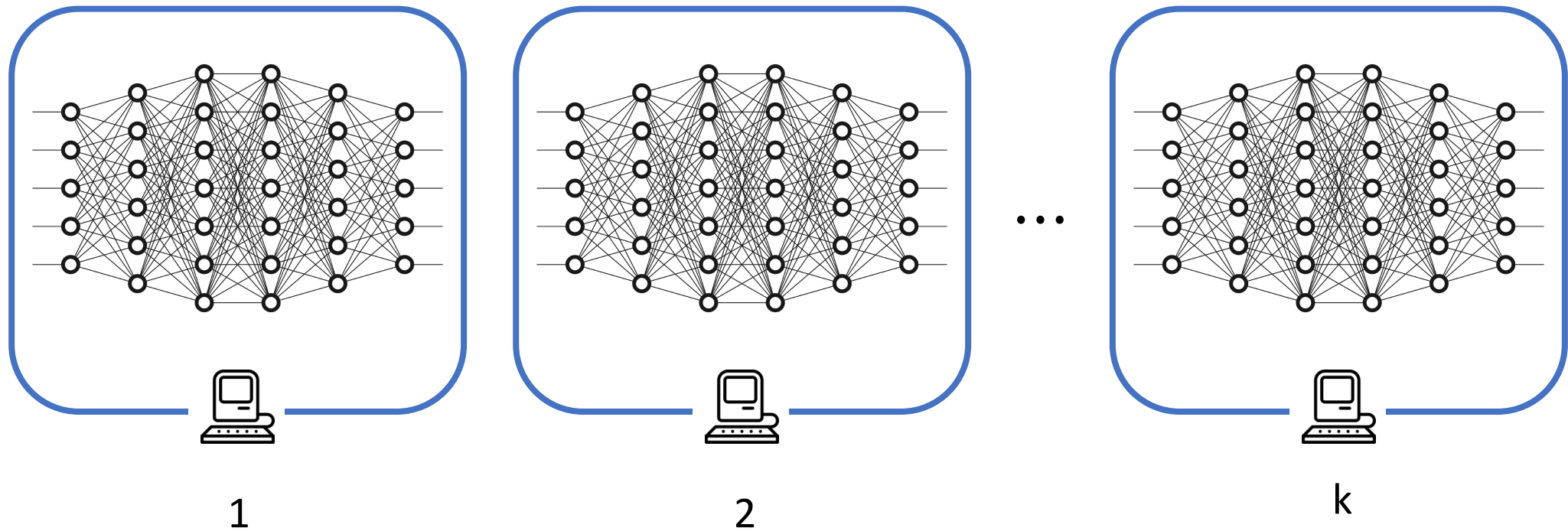
Sequential computation by using minibatch SGD

| 1 | 2 | 3 | 4 | | 5 | 6 | 7 | 8 | | 9 | 10 | 11 | 12 | ... |

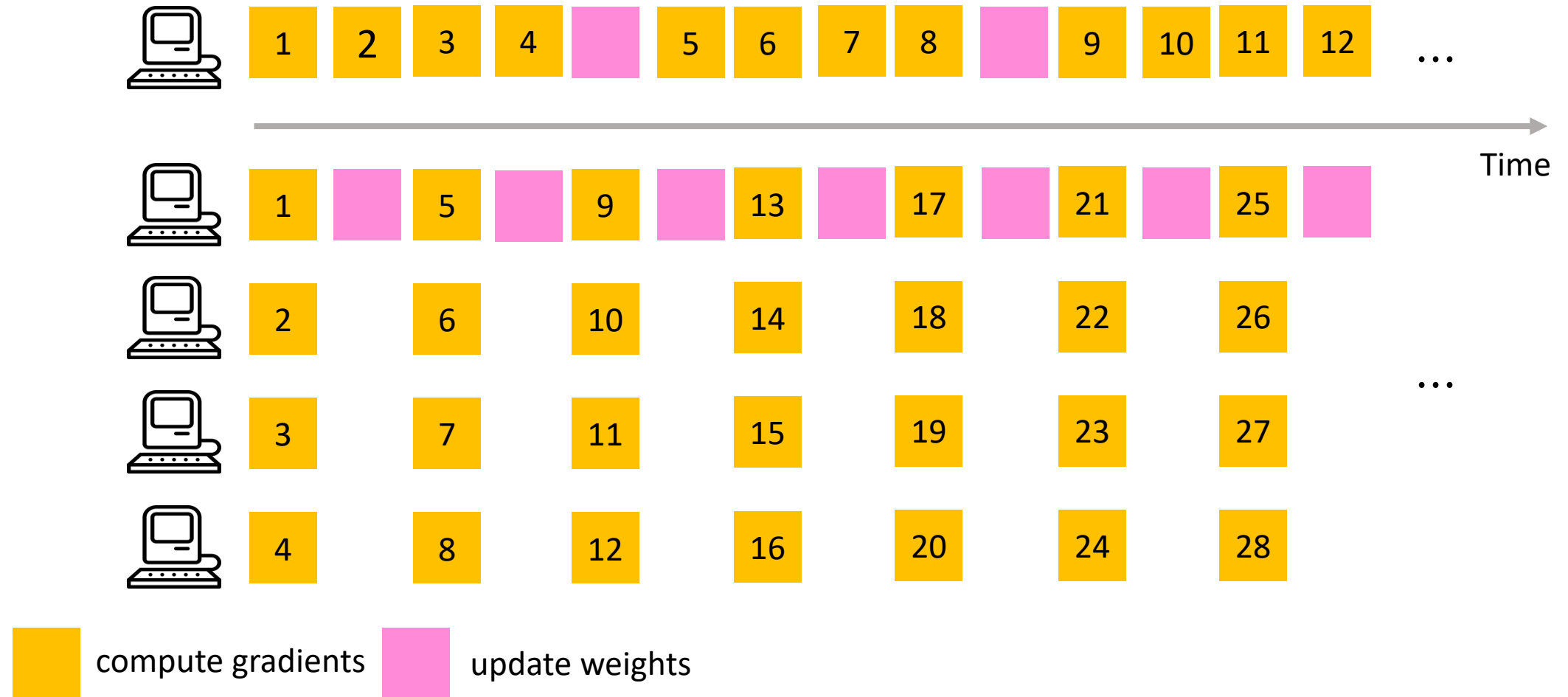Time

■ compute gradients ■ update weights

Data parallel computation

- Model replicated on each machine
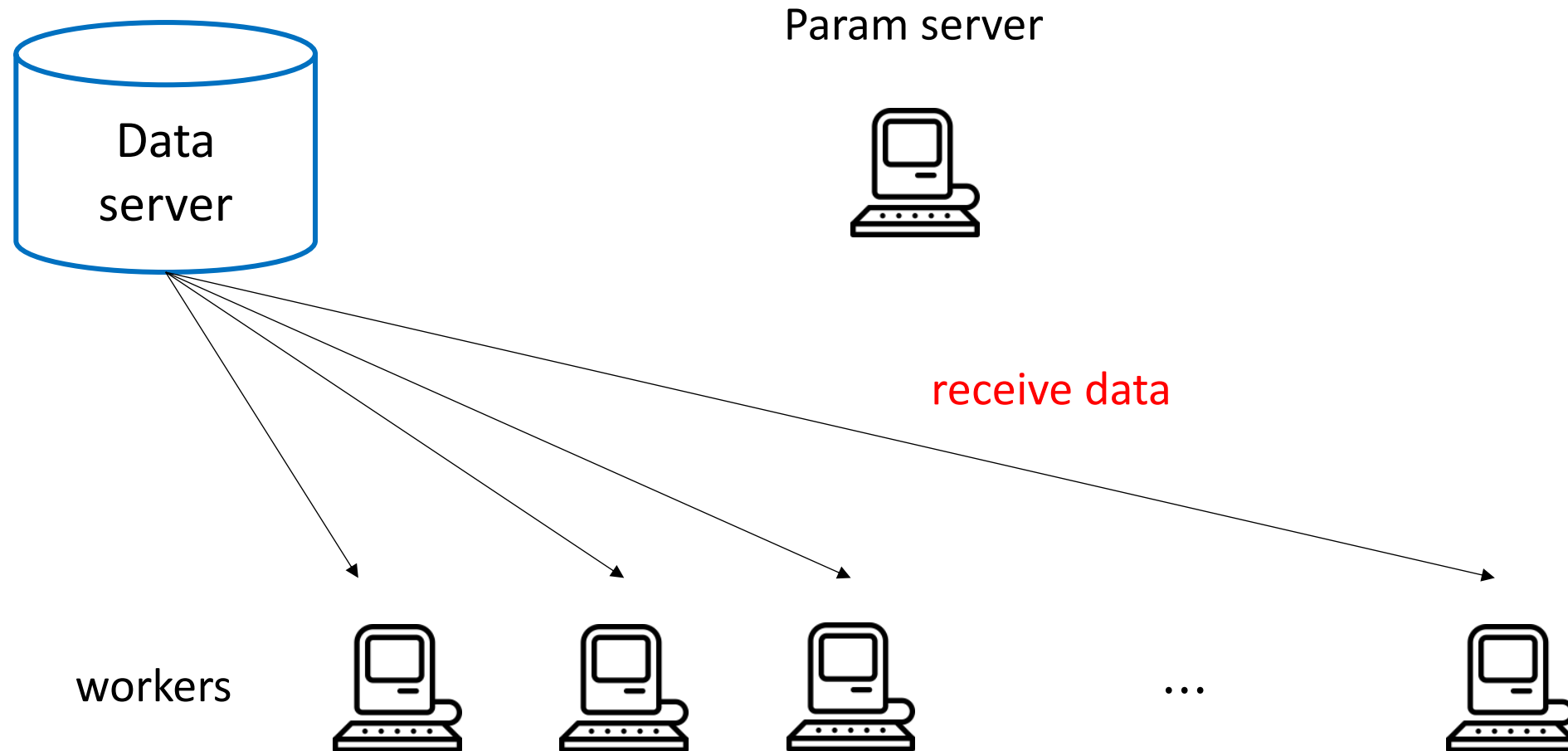- Training data split across machines

Minibatch SGD to data parallel training (cont'd)

- Gradient vector parallel computation speed of factor k by using k machines

Parameter server computation steps

Param server

Data server

receive data
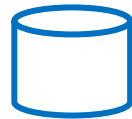
workers ...

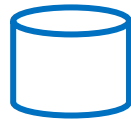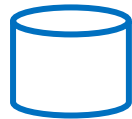# Parameter server computation steps (cont'd)

Param server

workers

...

Parameter server computation steps (cont'd)

Param server

workers

send initial weights
to worker nodes

...

Parameter server computation steps (cont'd)

Param server

workers ... compute gradients
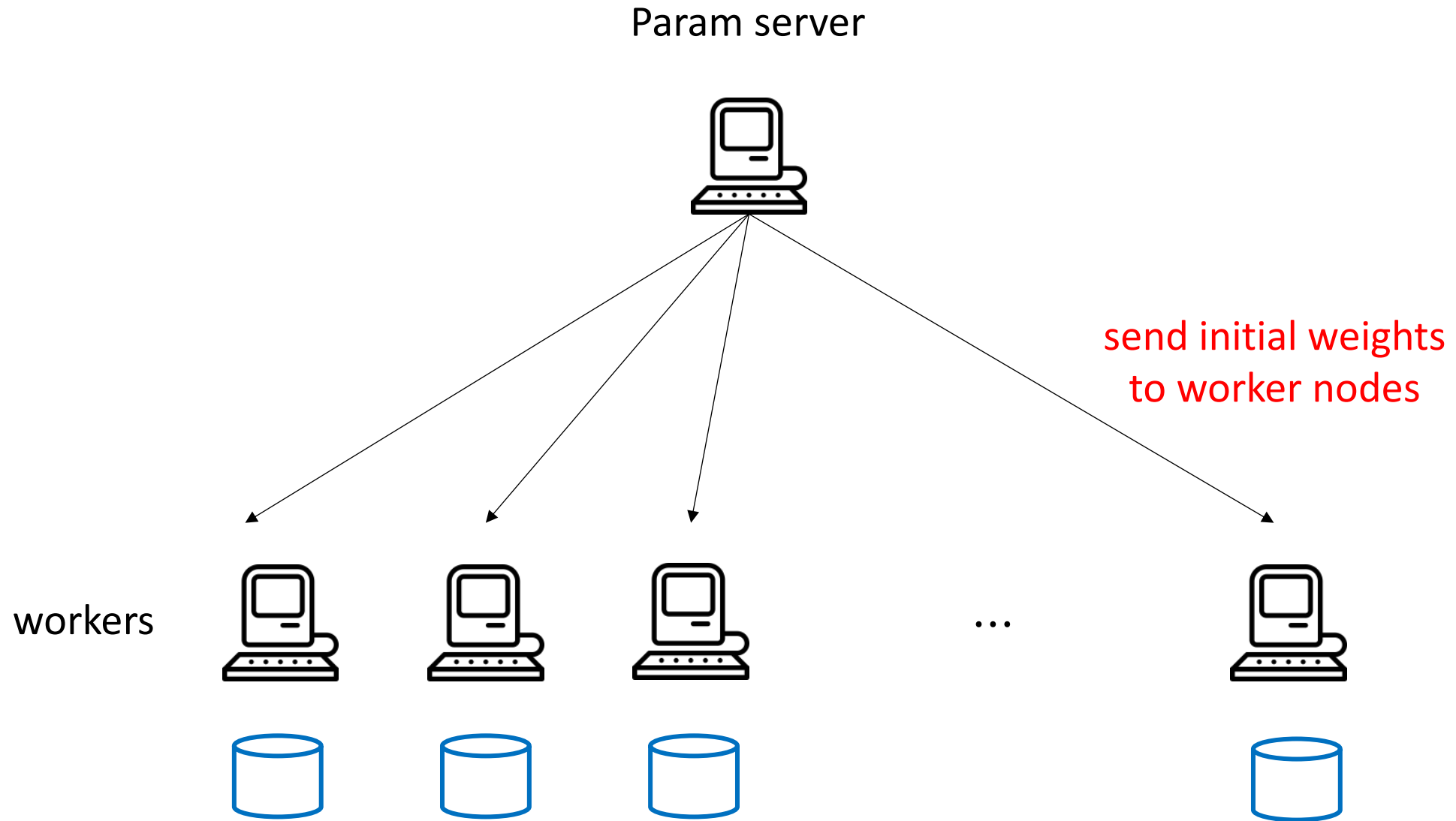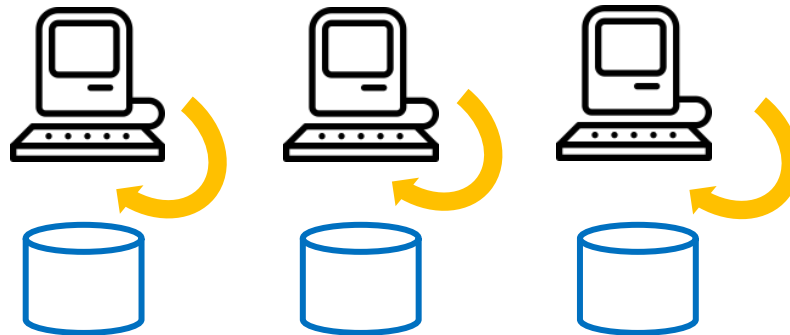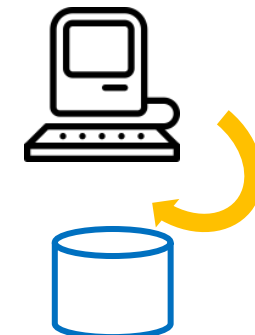
Parameter server computation steps (cont'd)
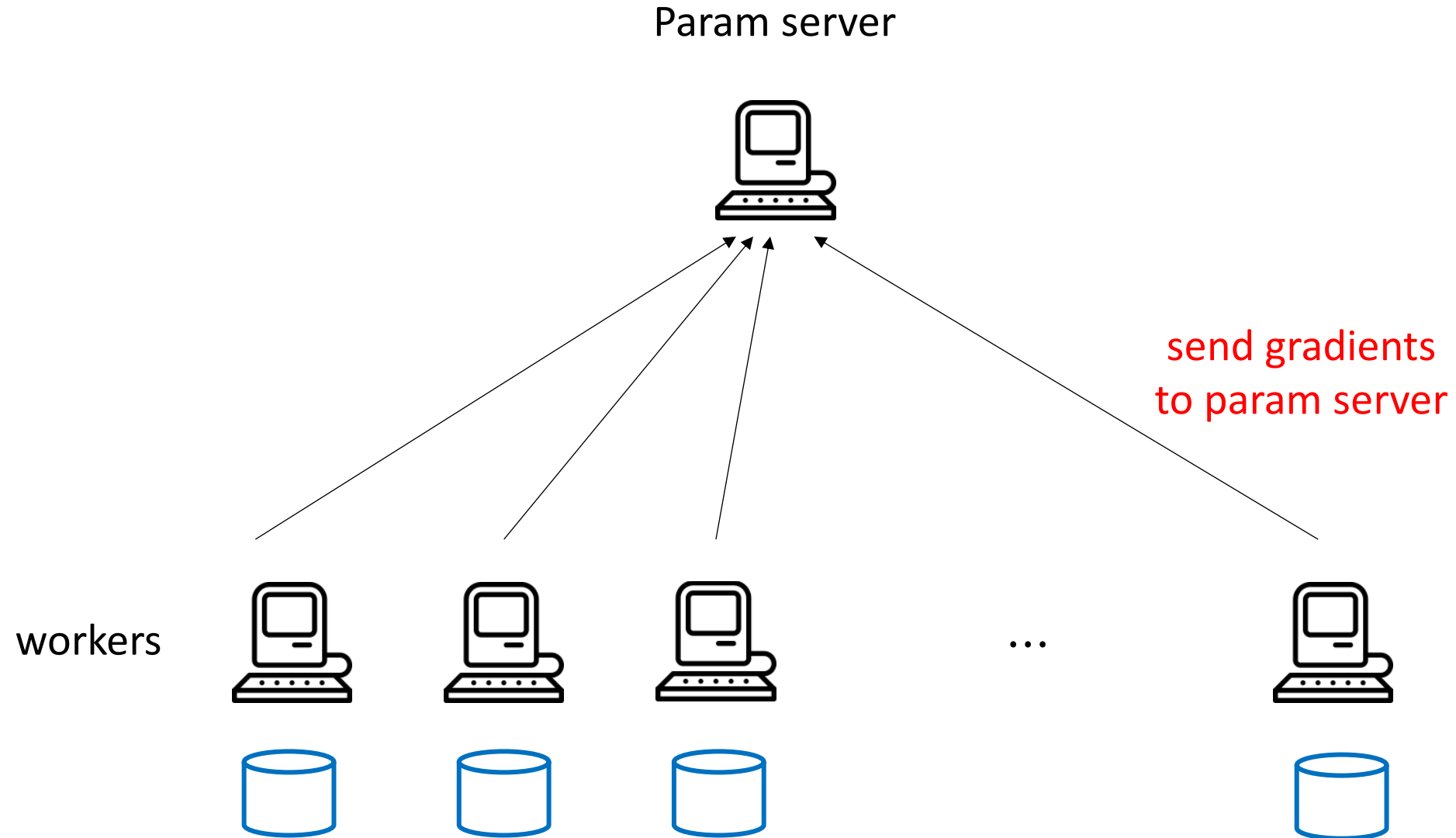
Param server

send gradients
to param server

workers

…

Parameter server computation steps (cont'd)

Param server

send weights
to worker nodes

workers

...

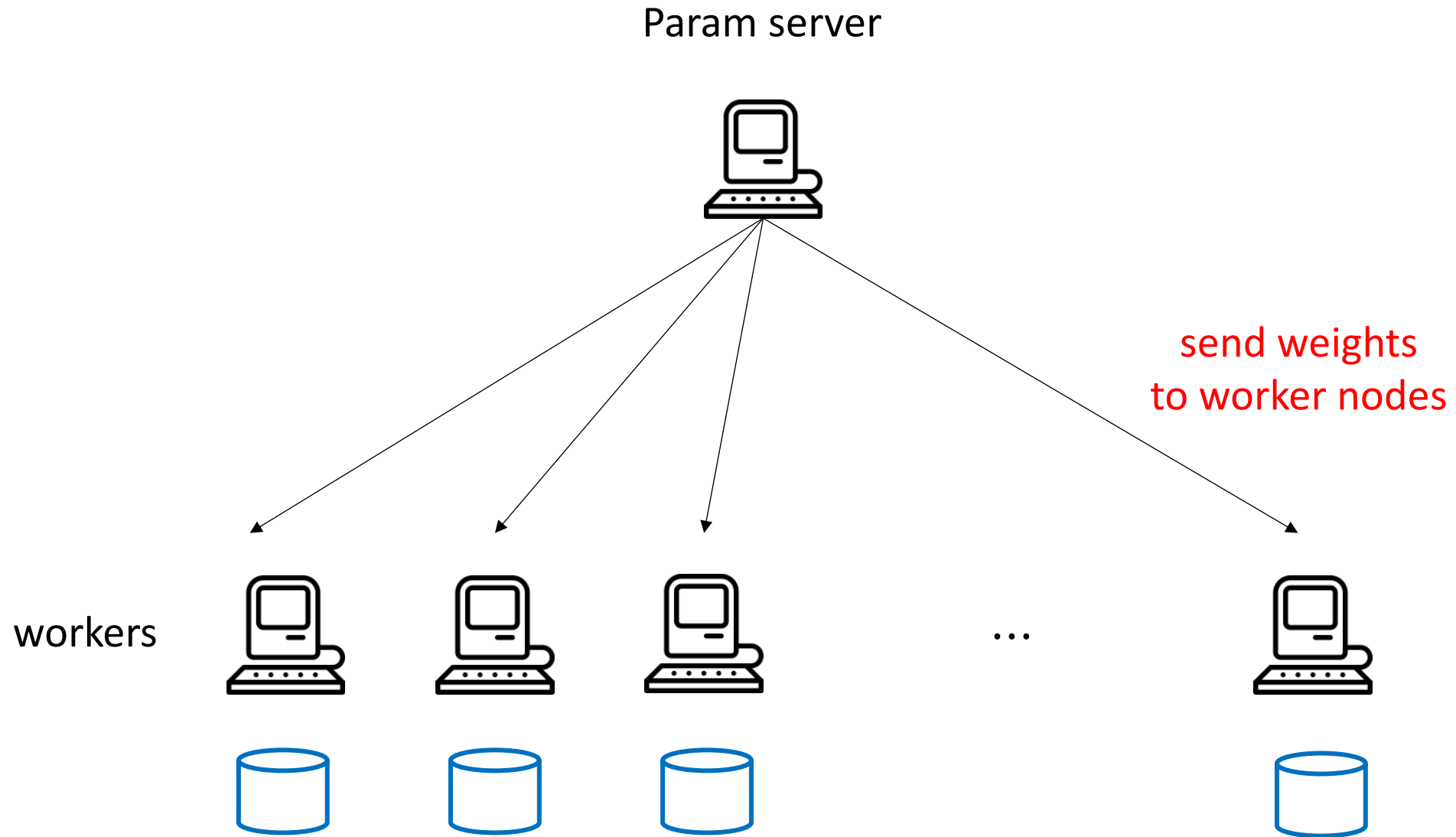# Parameter server computation steps (cont'd)

Param server

update weights

workers

…
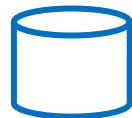
Computation and communication costs

Compute gradient   Exchange gradient   Update weights

Mini-batch 1

Mini-batch 2
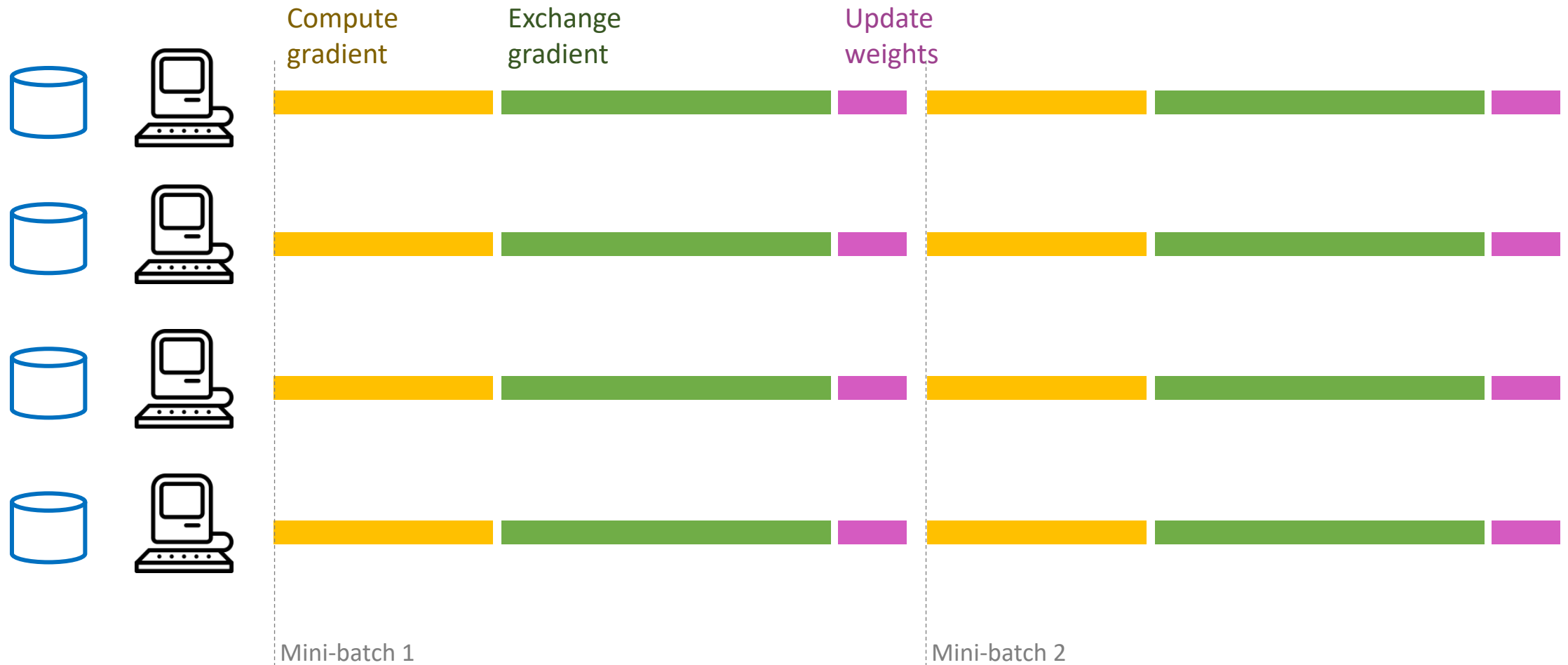
Communication cost reduction by gradient compression

Scaling by reduction precision: quantized SGD

Stochastic gradient vector compression

- Communication cost without compression: each worker node sends **F_n bits** to parameter server per gradient vector update
  - **F bits** for each element of the gradient vector
  - **F** is the number of bits used to for a floating-point representation
  - Ex. **64 bits** (double precision), **32 bits** (single precision), **16 bits** or **8 bits**

- Issue: **F_n bits** per gradient vector update of a worker can be too expensive for high-dimensional models!
  - Ex. Deep neural networks with millions of parameters

- Solution: use fewer bits for floating-point precision arithmetic
  - This is sensible as training machine learning models (ex deep neural networks) is tolerant to noise and reduced precision of numerical computations
  - A principle used by popular systems such as TensorFlow

TensorFlow example

TensorFlow:

Large-Scale Machine Learning on Heterogeneous Distributed Systems

(Preliminary White Paper, November 9, 2015)

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro,
Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow,
Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser,
Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray,
Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar,
Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals,
Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng
Google Research*

Abstract

TensorFlow [1] is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms. A computation expressed using TensorFlow can be executed with little or no change on a wide variety of heterogeneous systems, ranging from mobile devices such as phones and tablets up to large-scale distributed systems of hundreds of machines and thousands of computational devices such as GPU cards. The system is flexible and can be used to express a wide variety of algorithms, including training and inference

sequence prediction [47], move selection for Go [34], pedestrian detection [2], reinforcement learning [38], and other areas [17, 5]. In addition, often in close collaboration with the Google Brain team, more than 50 teams at Google and other Alphabet companies have deployed deep neural networks using DistBelief in a wide variety of products, including Google Search [11], our advertising products, our speech recognition systems [50, 6, 46], Google Photos [43], Google Maps and StreetView [19], Google Translate [18], YouTube, and many others.

5.5 Lossy Compression

Some machine learning algorithms, including those typically used for training neural networks, are tolerant of noise and reduced precision arithmetic. In a manner similar to the DistBelief system [14], we often use lossy compression of higher precision internal representations when sending data between devices (sometimes within the same machine but especially across machine boundaries). For example, we often insert special conversion nodes that convert 32-bit floating point representations into a 16-bit floating point representation (not the proposed IEEE 16-bit floating point standard, but rather just a 32-bit IEEE 794 float format, but with 16 bits less precision in the mantissa), and then convert back to a 32-bit representation on the other side of the communication channel (by just filling in zeroes for the lost portion

9

1-bit SGD vector compression

- Use only **1 bit** per element of the gradient vector (Seide et al 2014)
- Quantization $q(\boldsymbol{q})$ of gradient vector $\boldsymbol{g}$ defined by

$$q_i(\boldsymbol{q}) = \begin{cases} \bar{g}_+ & \text{if } g_i \geq 0 \\ \bar{g}_- & \text{if } g_i < 0 \end{cases} \text{ for } i = 1, 2, \ldots, n$$

where $\bar{g}_+$ and $\bar{g}_-$ are the mean values of positive and negative elements of $\boldsymbol{g}$, respectively

(define $\bar{g}_+ = 0$ if all elements of $\boldsymbol{g}$ are negative, and similarly for $\bar{g}_-$)

- This quantization scheme requires **1 bit** per element to encode the sign of the element, and **two floats** for the values of $\bar{g}_+$ and $\bar{g}_-$

1-bit SGD example

For the gradient vector

$$\boldsymbol{g} = (0.1, 0.15, -0.2, 20, 15.3, -1.222)^\top$$

$$\bar{g}_+ = \frac{0.1 + 0.15 + 20 + 15.3}{4} = 8.8875$$

$$\bar{g}_- = \frac{-0.2 - 1.222}{2} = -0.711$$

$$q(\boldsymbol{g}) = (8.8875, 8.8875, -0.711, 8.8875, -0.711)^\top$$

Need to encode signs $(1, 1, -1, 1, -1)$ and two numbers 8.8875 and 0.711

1-bith SGD pros and cons

- Pros: communication cost reduction
  - For communicating $q(\boldsymbol{g})$ need **n bits** for signs and **two floats** for $\bar{g}_+$ and $\bar{g}_-$
  - Communication cost per gradient update: **n** + **2F**
  - Compare this with no quantization where we need **Fn bits**

- Cons: an ad-hoc compression scheme
  - Biased quantized stochastic gradient vector may have adverse affects on convergence

- Problem: how can we design a compression for stochastic gradient vector such that communication cost is reduced but we also preserve convergence guarantees of SGD?
  - Several proposed solutions, we describe one next

SGD convergence theorem comes to rescue!

- Recall from lecture 7: Thm. Suppose
  - f is a convex and β-smooth loss function
  - $W \subseteq \mathrm{B}(R, \boldsymbol{w}^{(0)})$ where $\mathrm{B}(R, \boldsymbol{w}^{(0)})$ is a ball of radius R and center point $\boldsymbol{w}^{(0)}$
  - $\boldsymbol{w}^\star \in W$ minimizes loss function
  - There exists $\sigma \geq 0$ such that stochastic gradient vector $\boldsymbol{g}(\boldsymbol{w})$ at $\boldsymbol{w}$ satisfies

$$\mathbf{E}[\|\boldsymbol{g}(\boldsymbol{w}) - \nabla f(\boldsymbol{w})\|^2] \leq \sigma^2 \text{ for all } \boldsymbol{w} \in W$$

Then, for the SGD algorithm with step size $\eta_t = 1/(\beta + (\frac{\sigma}{R\sqrt{2}})\sqrt{t})$, it holds:
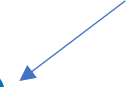
$$\mathbf{E}\left[f\left(\frac{1}{t}\sum_{s=1}^{t} \boldsymbol{w}_s\right)\right] - f(\boldsymbol{w}^\star) \leq \frac{\sqrt{2}R\sigma}{\sqrt{t}} + \frac{\beta R^2}{t}$$

- $\Rightarrow$ we just need to construct a stochastic gradient vector $\boldsymbol{g}(\boldsymbol{w})$ that
  - Is unbiased $\mathbf{E}[\boldsymbol{g}(\boldsymbol{w}^{(t)})|\boldsymbol{w}^{(t)} = \boldsymbol{w}] = \nabla f(\boldsymbol{w})$
  - Has small variance parameter σ^2
  - Can be efficiently compressed

A simple example of quantized SGD

- Quantized gradient defined as

"sparsifier"

$$q_i(\boldsymbol{g}) = \|\boldsymbol{g}\| \operatorname{sgn}(g_i)\, \xi_i(\boldsymbol{g})$$

where $\xi_i(\boldsymbol{g})$ are independent Bernoulli random variables with, if $\boldsymbol{g} \neq \boldsymbol{0}$,

$$\mathbf{P}[\xi_i(\boldsymbol{g}) = 1] = 1 - \mathbf{P}[\xi_i(\boldsymbol{g}) = 0] = \frac{|g_i|}{\|\boldsymbol{g}\|}$$

and $\mathbf{P}[\xi_i(\boldsymbol{g}) = 0] = 1$ otherwise

- Use this with a loss-free compression scheme for communicating gradient vector updates from worker nodes to parameter server
  - Easy to implement in a distributed system
  - No need for machines to maintain any shared state
  - Worker nodes can perform quantization independently on their own

A simple example of quantized SGD (cont'd)
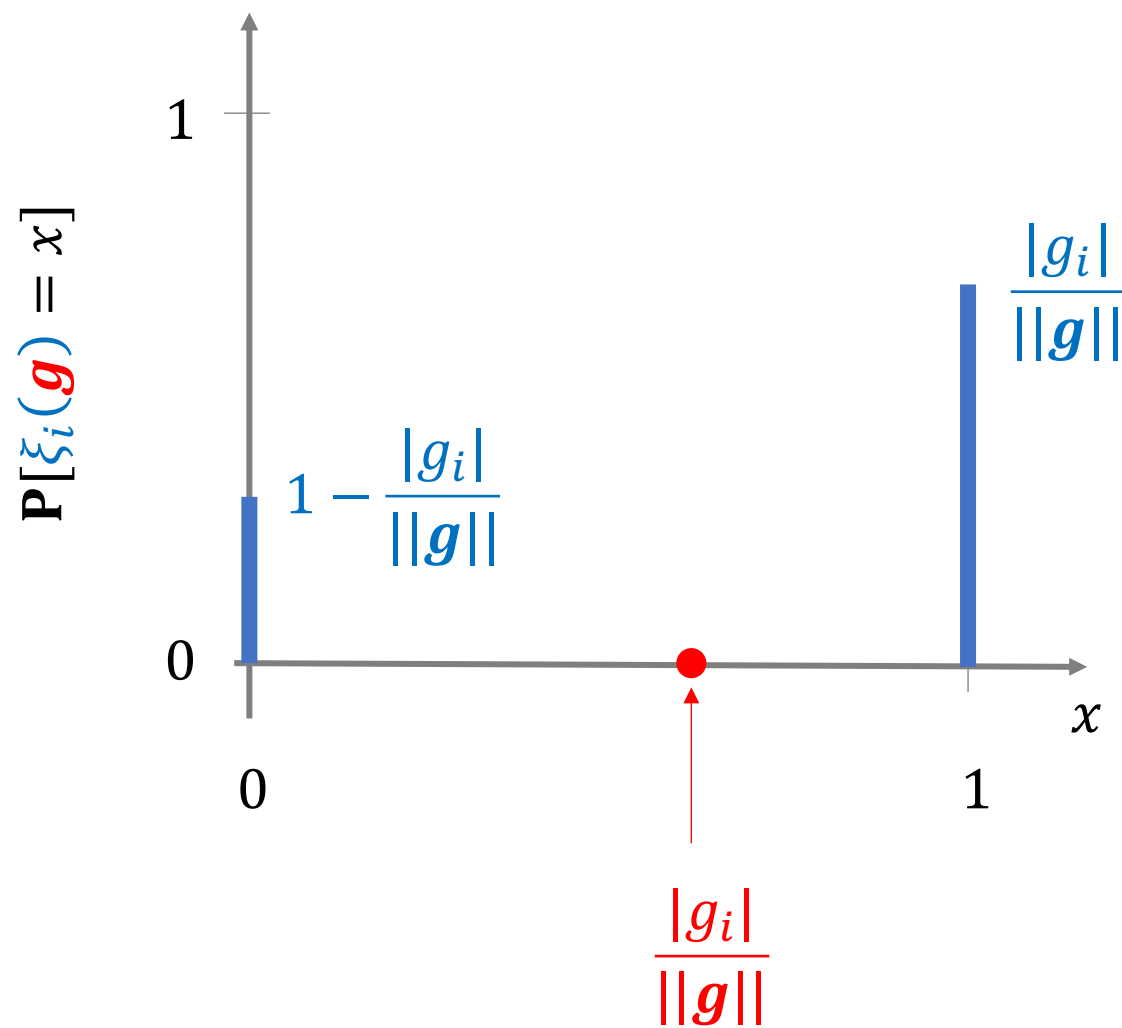
Property 1: unbiasedness

- P1: If a stochastic gradient vector is unbiased, so is the quantized stochastic gradient vector

For any unbiased stochastic gradient vector $\boldsymbol{g}$, i.e. $\boldsymbol{g}$ such that $\mathbf{E}[\boldsymbol{g}] = \nabla f(\boldsymbol{w})$, we have

$$\mathbf{E}[q_i(\boldsymbol{g})] = \mathbf{E}[\|\boldsymbol{g}\| \operatorname{sgn}(g_i)\xi_i(\boldsymbol{g})]$$

$$= \mathbf{E}\big[\|\boldsymbol{g}\| \operatorname{sgn}(g_i)\, \mathbf{P}[\, \xi_i(\boldsymbol{g}) = 1 \mid \boldsymbol{g} \,]\big]$$

$$= \mathbf{E}\left[\|\boldsymbol{g}\| \operatorname{sgn}(g_i)\frac{|g_i|}{\|\boldsymbol{g}\|}\right]$$

$$= \mathbf{E}[\operatorname{sgn}(g_i)\,|g_i|]$$

$$= \mathbf{E}[g_i]$$

$$= \frac{\partial}{\partial w_i} f(\boldsymbol{w})$$

Hence, $\mathbf{E}[q(\boldsymbol{g})] = \nabla f(\boldsymbol{w})$ (quantized stochastic gradient vector is unbiased)

Property 2: variance bound

- P2: If a stochastic gradient vector $\boldsymbol{g}$ has second-moment bound σ^2, i.e. $\mathbf{E}[\|\boldsymbol{g}\|^2] \leq \sigma^2$, then $q(\boldsymbol{g})$ has second-moment bound $\sqrt{n}\sigma^2$

$$\mathbf{E}[\|q(\boldsymbol{g})\|^2] = \sum_{i=1}^{n} \mathbf{E}[\|\boldsymbol{g}\|^2 \mathbf{E}[\xi_i(\boldsymbol{g})^2 \mid \boldsymbol{g}]]$$

$$= \sum_{i=1}^{n} \mathbf{E}\left[\|\boldsymbol{g}\|^2 \frac{|g_i|}{\|\boldsymbol{g}\|}\right]$$

$$= \mathbf{E}[\|\boldsymbol{g}\| \sum_{i=1}^{n} |g_i|]$$

$$= \sqrt{n}\mathbf{E}[\|\boldsymbol{g}\|^2] \qquad \text{(Cauchy-Schwarz inequality: } |\boldsymbol{a}^\top \boldsymbol{b}| \leq \|\boldsymbol{a}\|\|\boldsymbol{b}\|)$$

$$\leq \sqrt{n}\sigma^2$$

- Note: for any random vector $\boldsymbol{v}$, second moment bound $\mathbf{E}[\|\boldsymbol{v}\|^2] \leq \sigma^2$ implies the variance bound $\mathbf{E}[\|\boldsymbol{v} - \mathbf{E}[\boldsymbol{v}]\|^2] \leq \sigma^2$

Property 3: sparsity

- **P3:** Expected number of non-zero quantized gradient elements is $\leq \sqrt{n}$

$$\mathbf{E}[\|\xi(\boldsymbol{g})\|_0] = \mathbf{E}[|\{i \in \{1, 2, \dots, n\}: \xi_i(\boldsymbol{g}) = 1\}|]$$

$$= \sum_{i=1}^n \mathbf{P}[\xi_i(\boldsymbol{g}) = 1]$$

$$= \sum_{i=1}^n \frac{|g_i|}{\|\boldsymbol{g}\|}$$

$$\leq \sqrt{n} \qquad \text{(again Cauchy-Schwartz inequality)}$$

- High sparsity: in expectation, at most $\sqrt{n}$ non-zero elements out of n

Expected communication cost

- Claim: expected number of bits to encode the quantized gradient vector is at most

$$\sqrt{n}(\log_2(n) + \log_2(2e)) + F \textbf{ bits}$$

- Breakdown:
    - F **bits** to encode $\|\boldsymbol{g}\|$
    - $\log_2\left(\binom{n}{\|\xi(\boldsymbol{g})\|_0}\right)$ **bits** to encode x positions of non-zero elements
    - $\|\xi(\boldsymbol{g})\|_0$ **bits** to encode the sign of non-zero coordinates

- For high-dimensional problems (large n), substantial communication efficiency gain
    - $O(\sqrt{n}\log(n))$ **bits** instead of order n **bits** with no compression

Expected communication cost (cont'd)

- Proof of the claim

$$\mathbf{E}\left[\log_2\left(\binom{n}{\|\xi(\boldsymbol{g})\|_0}\right)\right] = \mathbf{E}\left[\log_2\left(\binom{n}{\|\xi(\boldsymbol{g})\|_0}\right)\mathbf{1}_{\{\|\xi(\boldsymbol{g})\|_0>0\}}\right]$$

$$\leq \log_2(ne)\mathbf{E}\left[\|\xi(\boldsymbol{g})\|_0\mathbf{1}_{\{\|\xi(\boldsymbol{g})\|_0>0\}}\right] \qquad \text{(basic fact } \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k\text{)}$$

$$= \log_2(ne)\,\mathbf{E}[\|\xi(\boldsymbol{g})\|_0]$$

$$\leq \log_2(ne)\sqrt{n} \qquad\qquad\qquad\qquad \text{(P3)}$$

Summing together with $\mathbf{E}[\|\xi(\boldsymbol{g})\|_0] \leq \sqrt{n}$ **bits** and F **bits** yields the total expected cost

41

Controllable quantization

- Simple quantization scheme presented so far quantizes each gradient element to one of two quantization values or sets the gradient element to zero
  - This yields the variance bound increased by factor $\sqrt{n}$
  - We would like to have a control on the granularity of quantization

- Generalization to quantization with $s+1$ quantization values, for $s \geq 1$:

$$q_i(\boldsymbol{g}; s) = \|\boldsymbol{g}\| \operatorname{sgn}(g_i) \xi_i(\boldsymbol{g}; s) \text{ for } i = 1, 2, \ldots, n$$

where $\xi_i(\boldsymbol{g}; s)$ are independent random variables with, if $\boldsymbol{g} \neq \boldsymbol{0}$,

$$\mathbf{P}\left[\xi_i(\boldsymbol{g}) = \frac{l+1}{s}\right] = 1 - \left[\xi_i(\boldsymbol{g}) = \frac{l}{s}\right] = s\frac{|g_i|}{\|\boldsymbol{g}\|} - l, \text{ if } \frac{l}{s} \leq \frac{|g_i|}{\|\boldsymbol{g}\|} \leq \frac{l+1}{s}$$

and $\mathbf{P}[\xi_i(\boldsymbol{g}) = 0] = 1$ otherwise

- See next slide for a graphical illustration

Controllable quantization (cont'd)
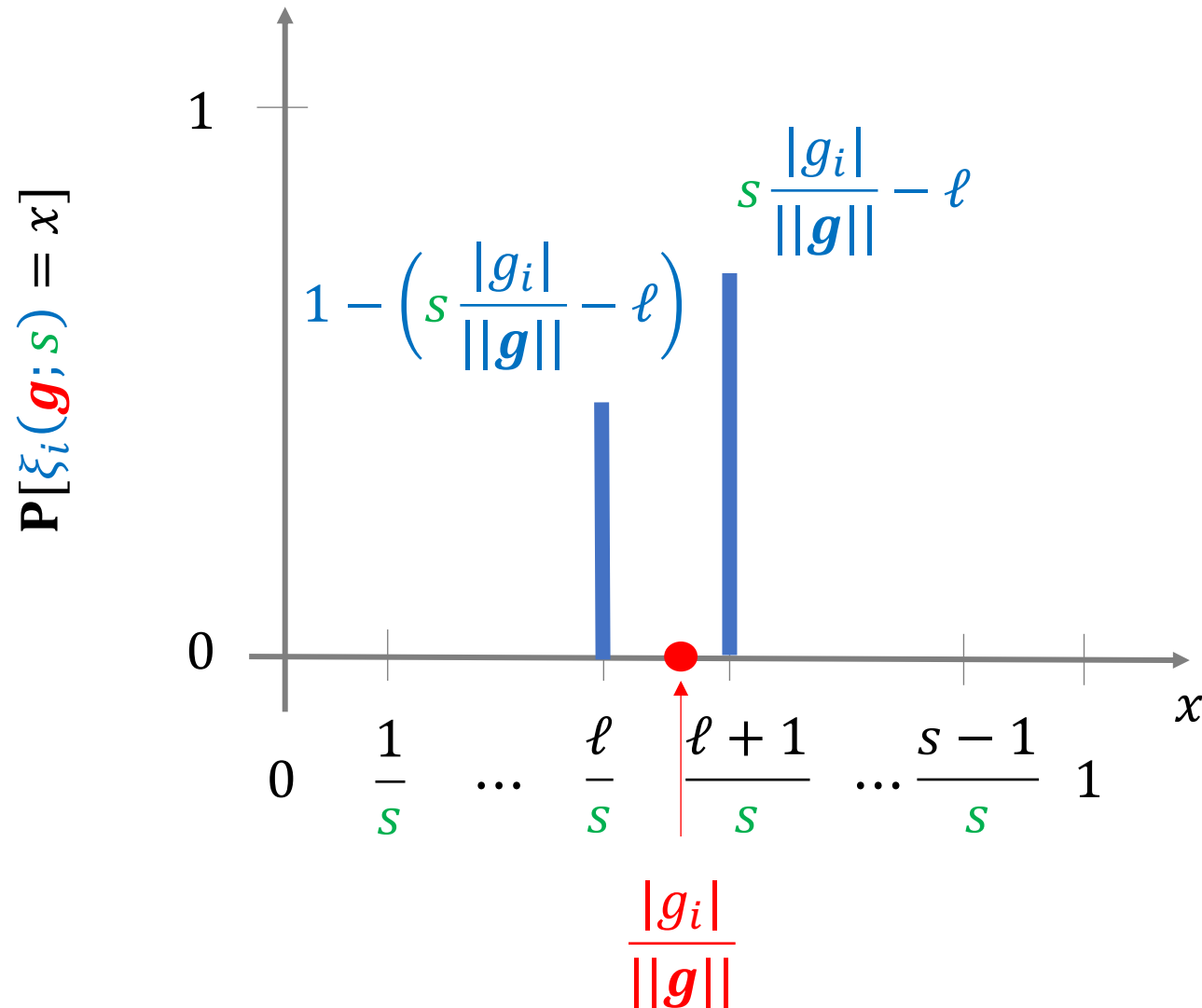
Controllable quantization (cont'd)

- Second moment bound: suppose $\mathbf{E}[\|g\|^2] \leq \sigma^2$, then

$$\mathbf{E}[\|q(g; s)\|^2] \leq \left(1 + \min\left\{\left(\frac{\sqrt{n}}{s}\right)^2, \frac{\sqrt{n}}{s}\right\}\right)\sigma^2$$

- Expected number of bits: to encode the quantized gradient vector need at most

$$(s^2 + \sqrt{n})\left(\frac{3}{2}\log\left(\frac{2(s^2+n)}{s^2+\sqrt{n}}\right) + 3\right) + F \text{ bits}$$

- The larger s is the smaller the second moment bound
- The large s is the larger the communication cost

Proof for the above claims can be found in Alistarh et al 2017

Iteration-communication trade-off

iterations

$\sqrt{n}$

1

1

$\sqrt{n}$

s

bits per iteration

n

$\sqrt{n}$

1

$\sqrt{n}$

s
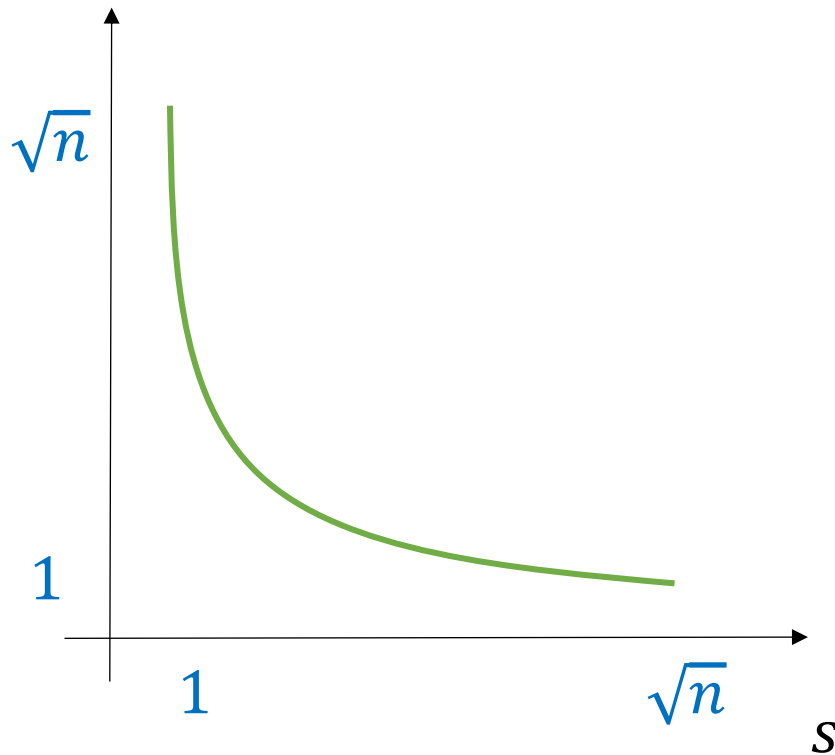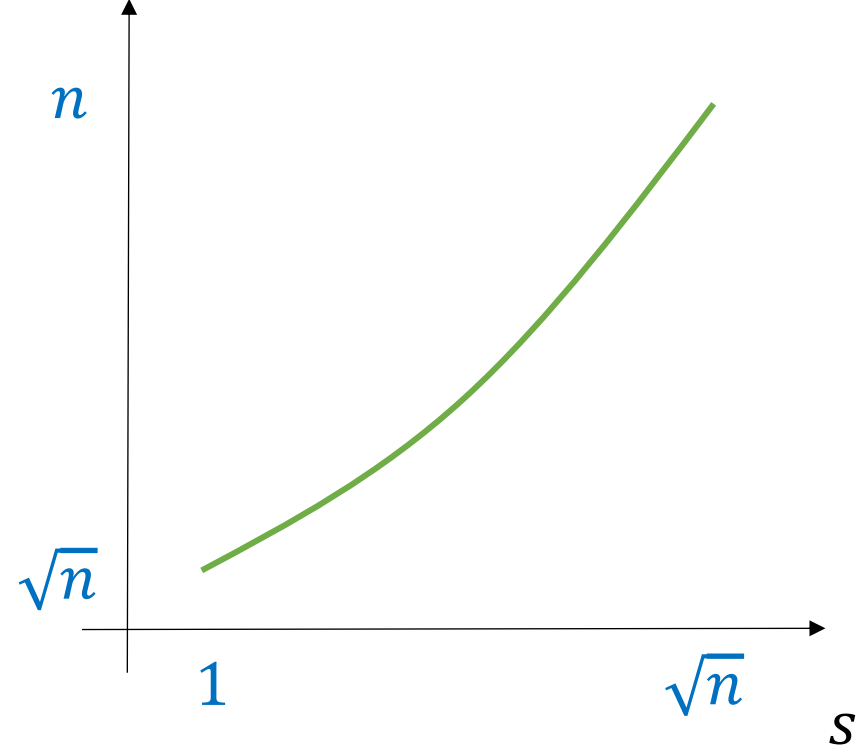
- By SGD convergence theorem, the number of iterations approximately proportional to variance / second-moment bound

- No free lunch: smaller number of bits per iteration requires a larger number of iterations
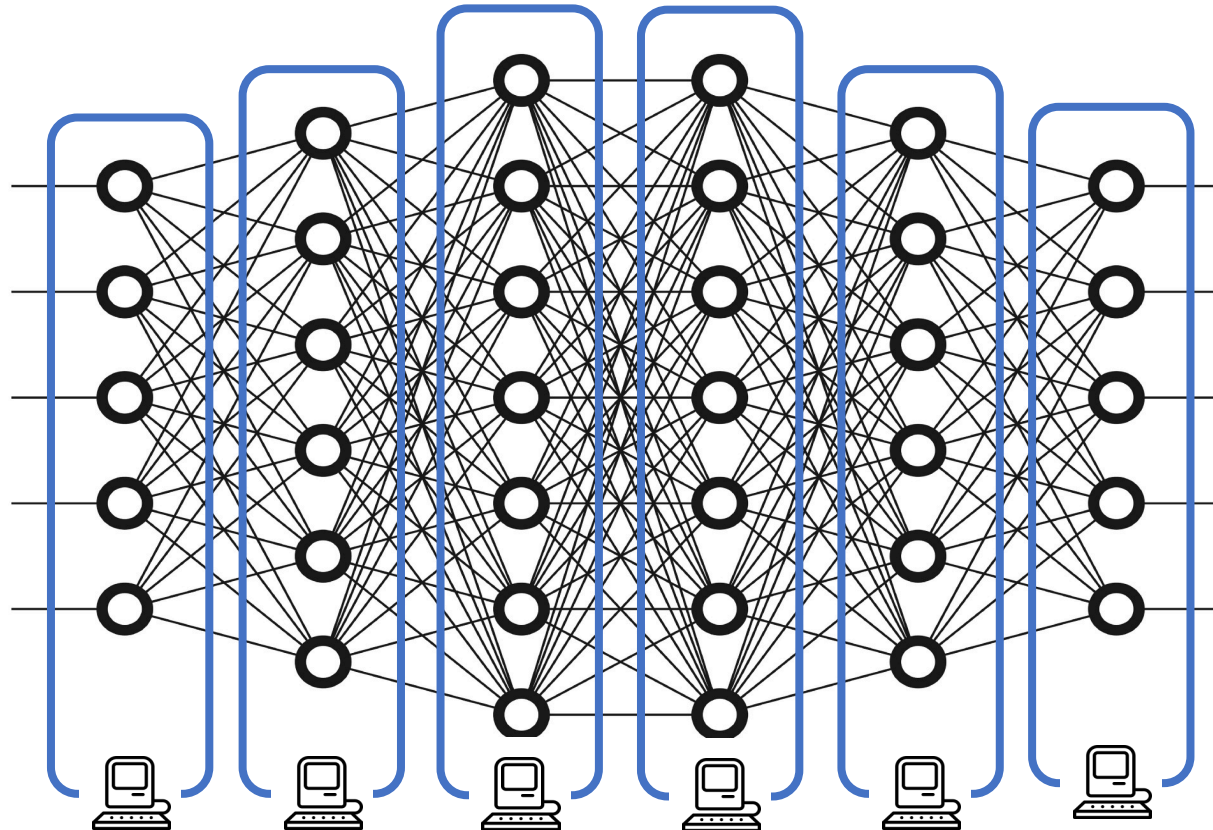
Experimental results
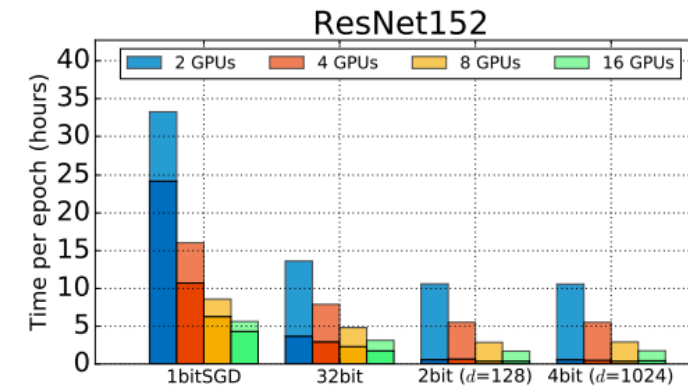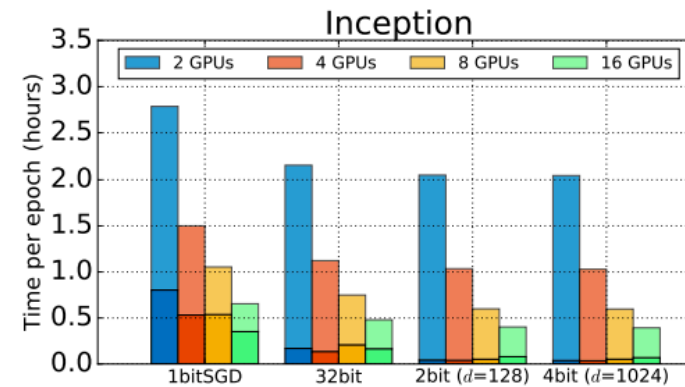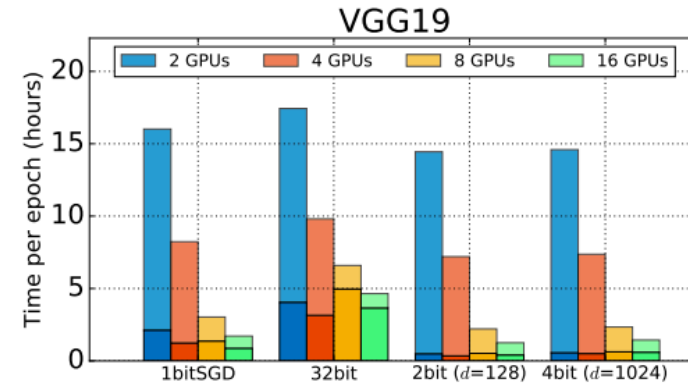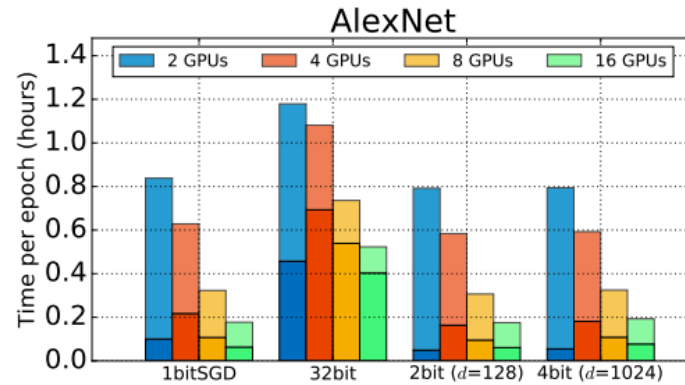
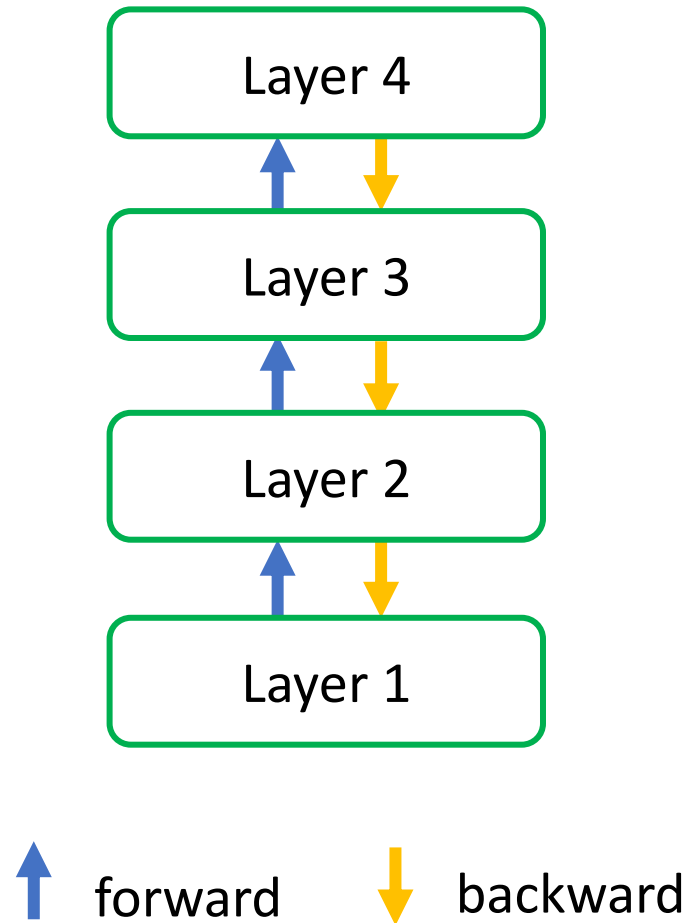- Source: Alistarh et al (2017)

Model parallel computation

Model parallel computation

- High-dimensional models have many parameters and operations

- Scale up by partitioning the model across machines
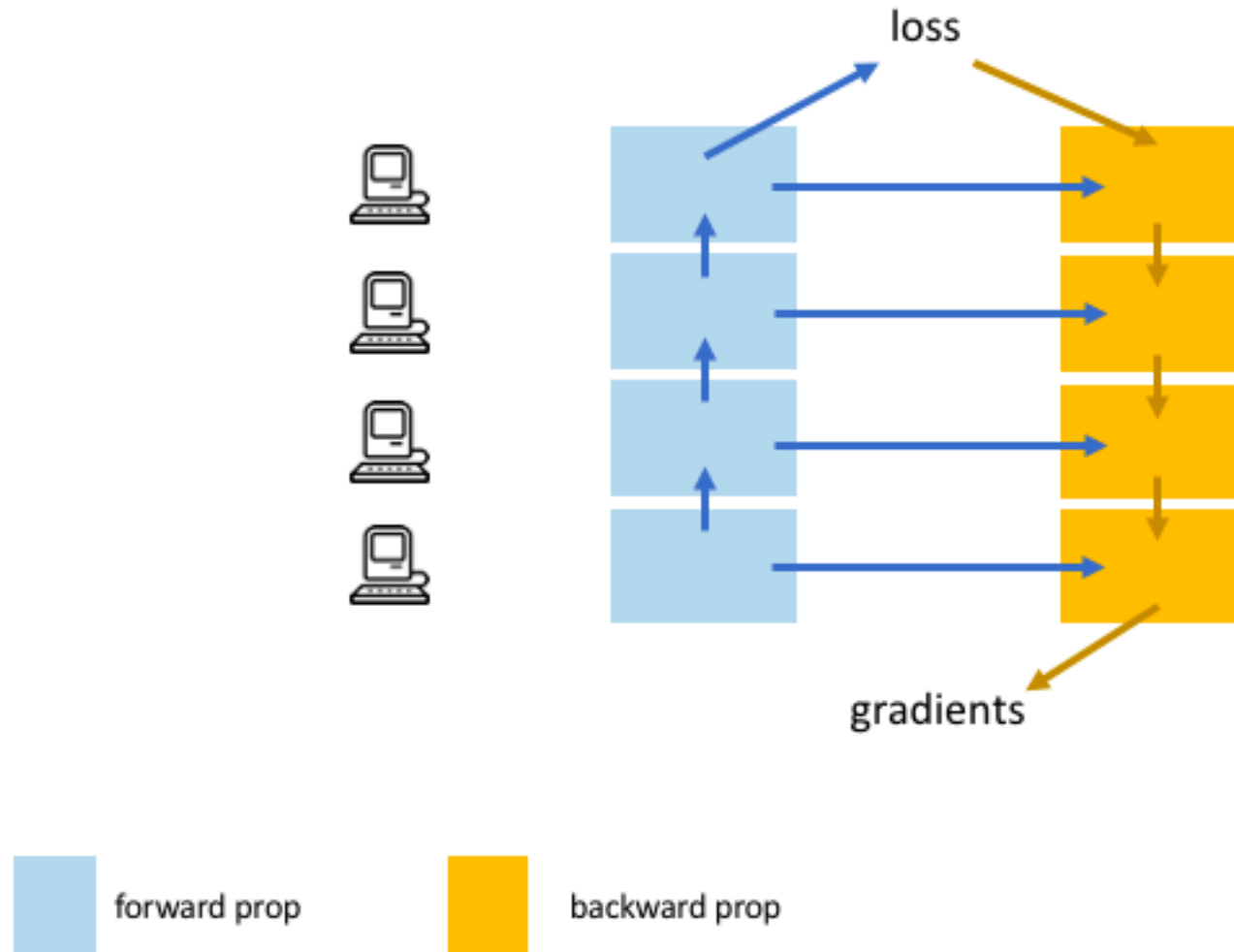
Model parallel computation example

- Backpropagation: computing gradients requires a forward and a backward propagation
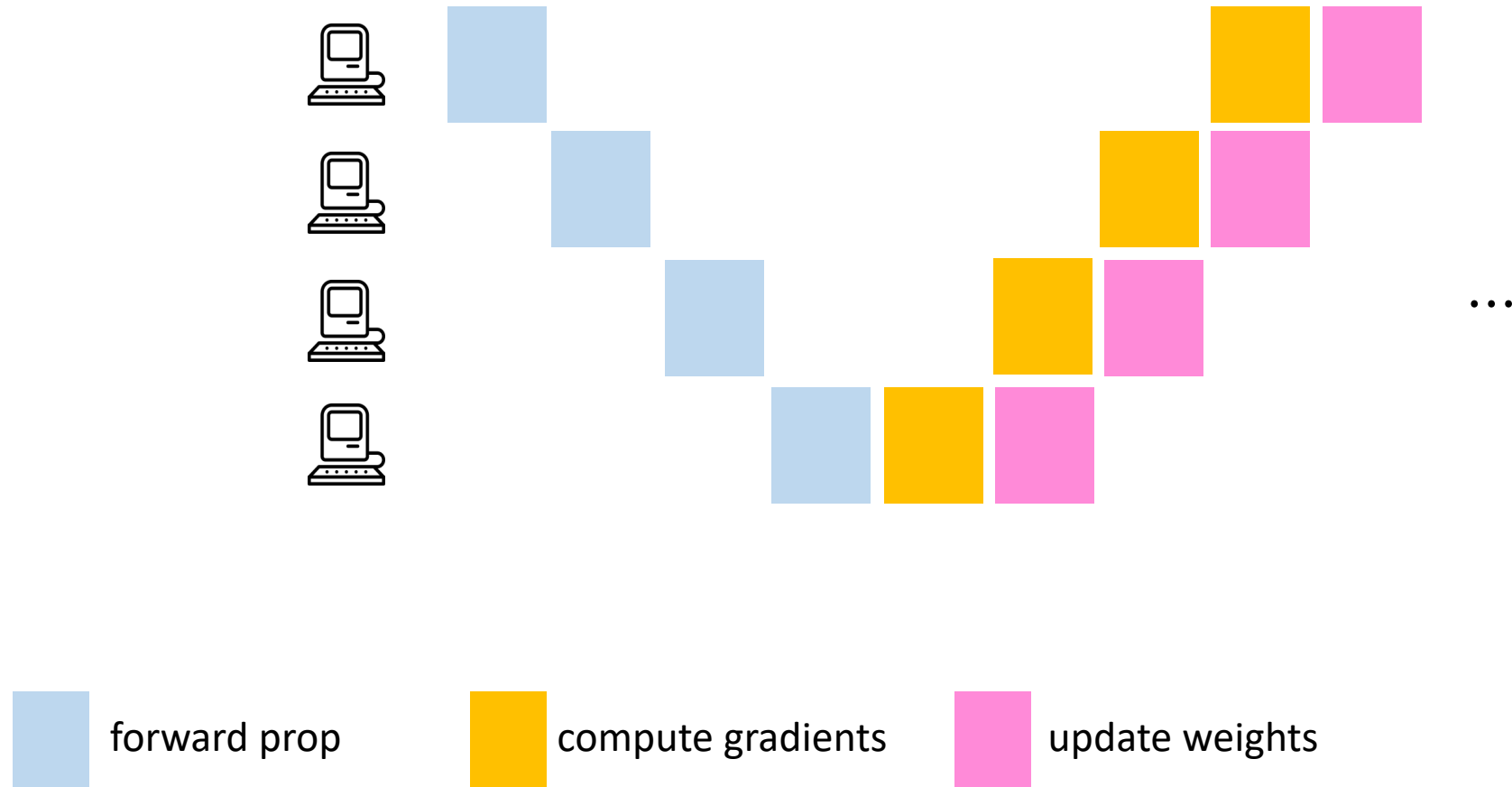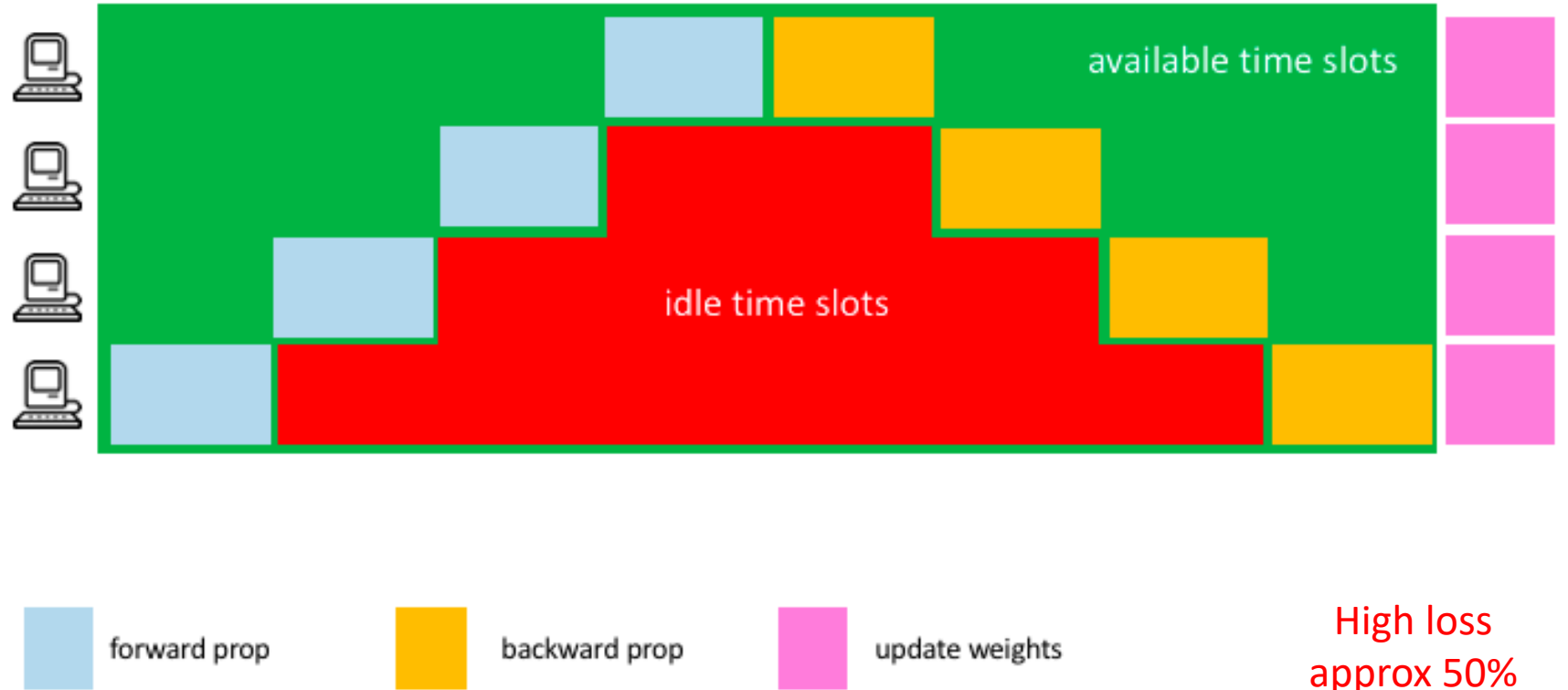
Model parallel computation example (cont'd)

forward ↑   backward ↓
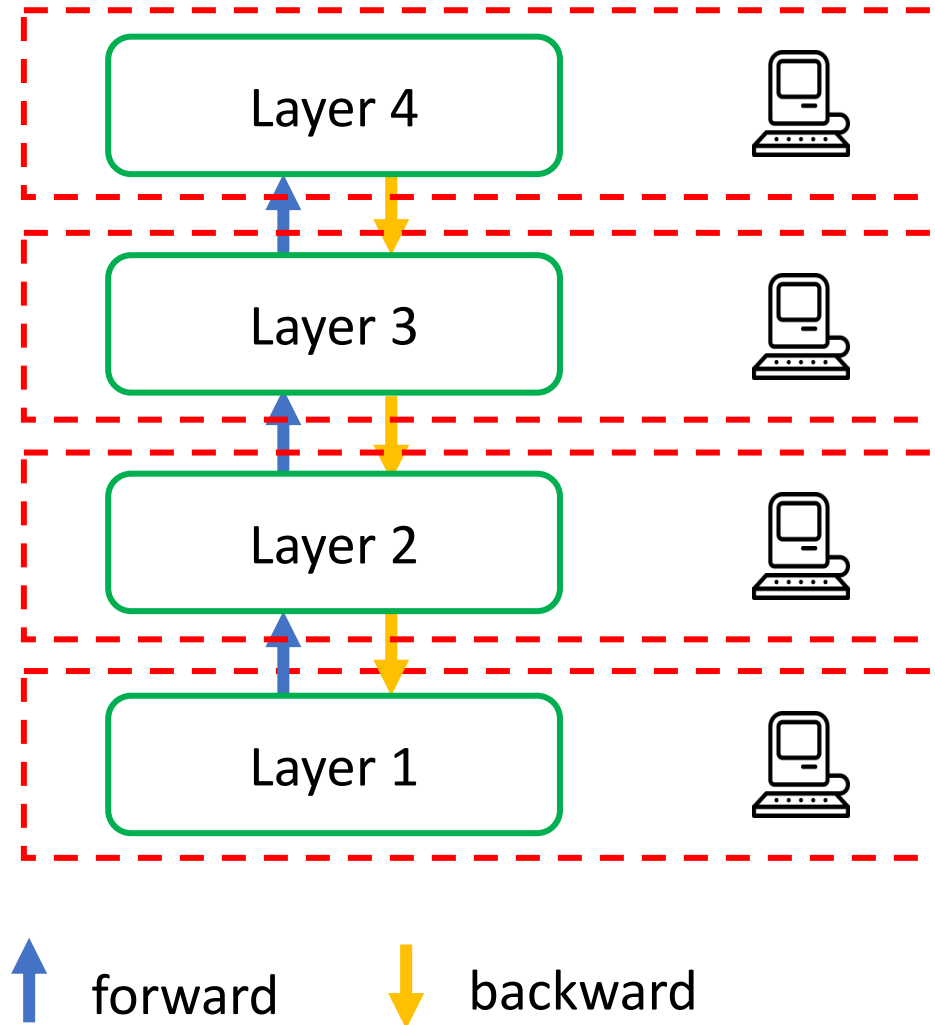
Computing gradients

Model parallel computation: a simple approach

forward prop   compute gradients   update weights

- Parallelization limited by sequential nature of computations
- Precedence constraints: computation in a layer can be performed only after computation in some other layer has completed

Loss factor

available time slots

idle time slots

- forward prop
- backward prop
- update weights

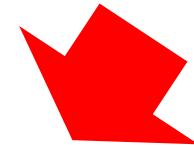High loss approx 50% for large number of machines

- $k :=$ number of devices
- $b :=$ mini-batch size

$$\text{Loss factor} = \frac{\text{number of idle time slots}}{\text{number of available time slots}}$$
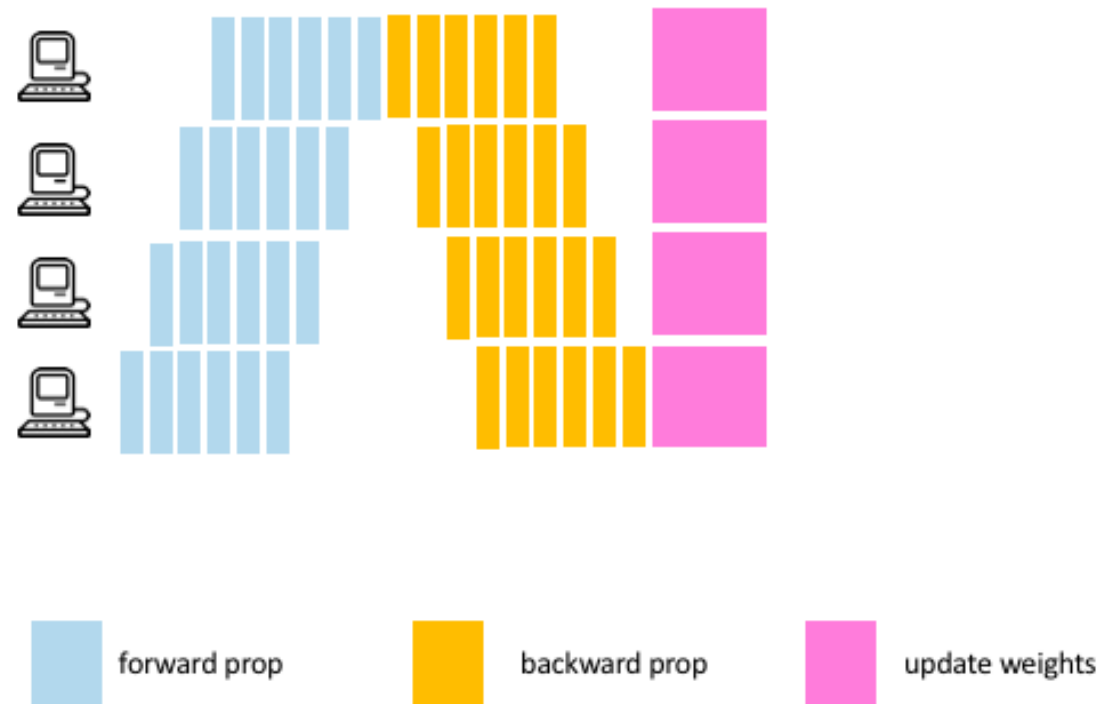
$$= \frac{2b(1 + 2 + \cdots + k - 1)}{2bk^2} = \frac{1}{2}\left(1 - \frac{1}{k}\right)$$
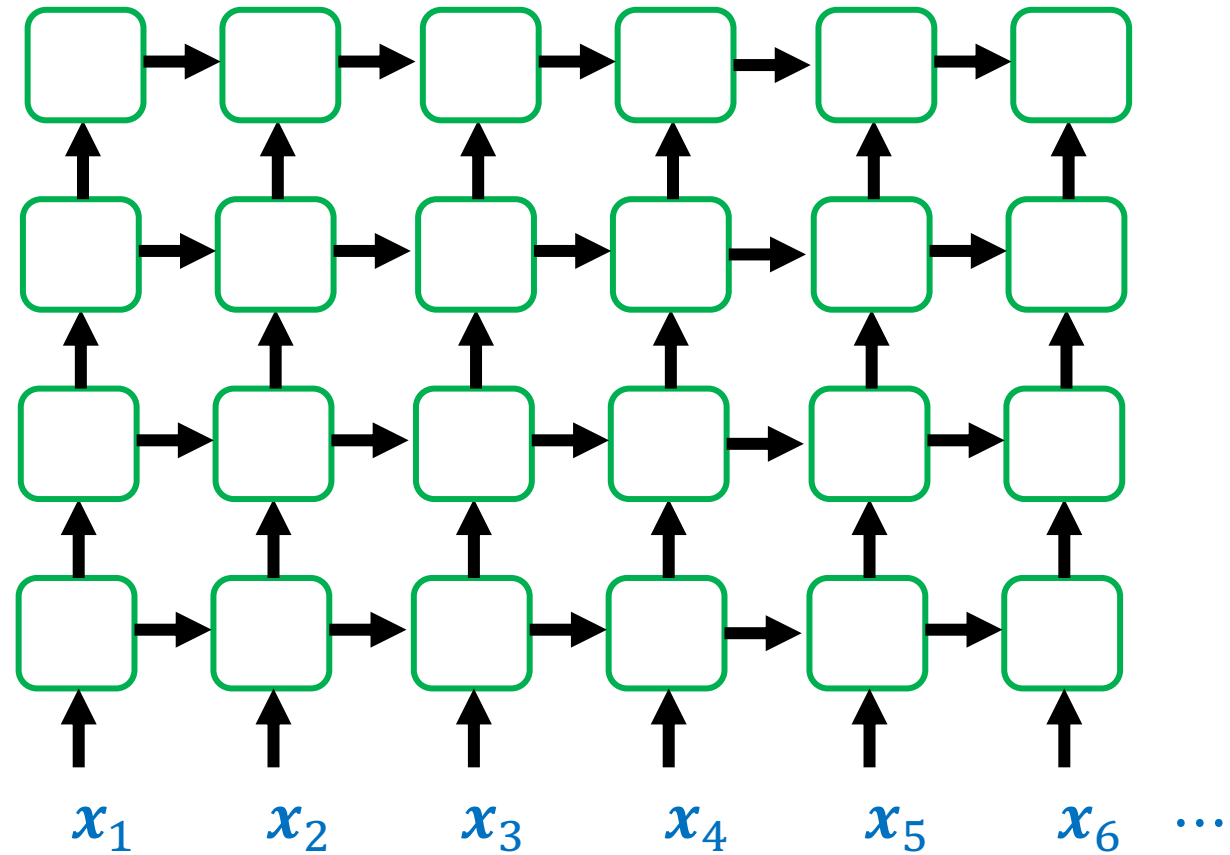
Improving efficiency by pipelining

- Splitting minibatches into microbatches

forward prop   backward prop   update weights

Loss factor with pipelining

- $k :=$ number of devices

- $b :=$ minibatch size

- $m :=$ number of microbaches
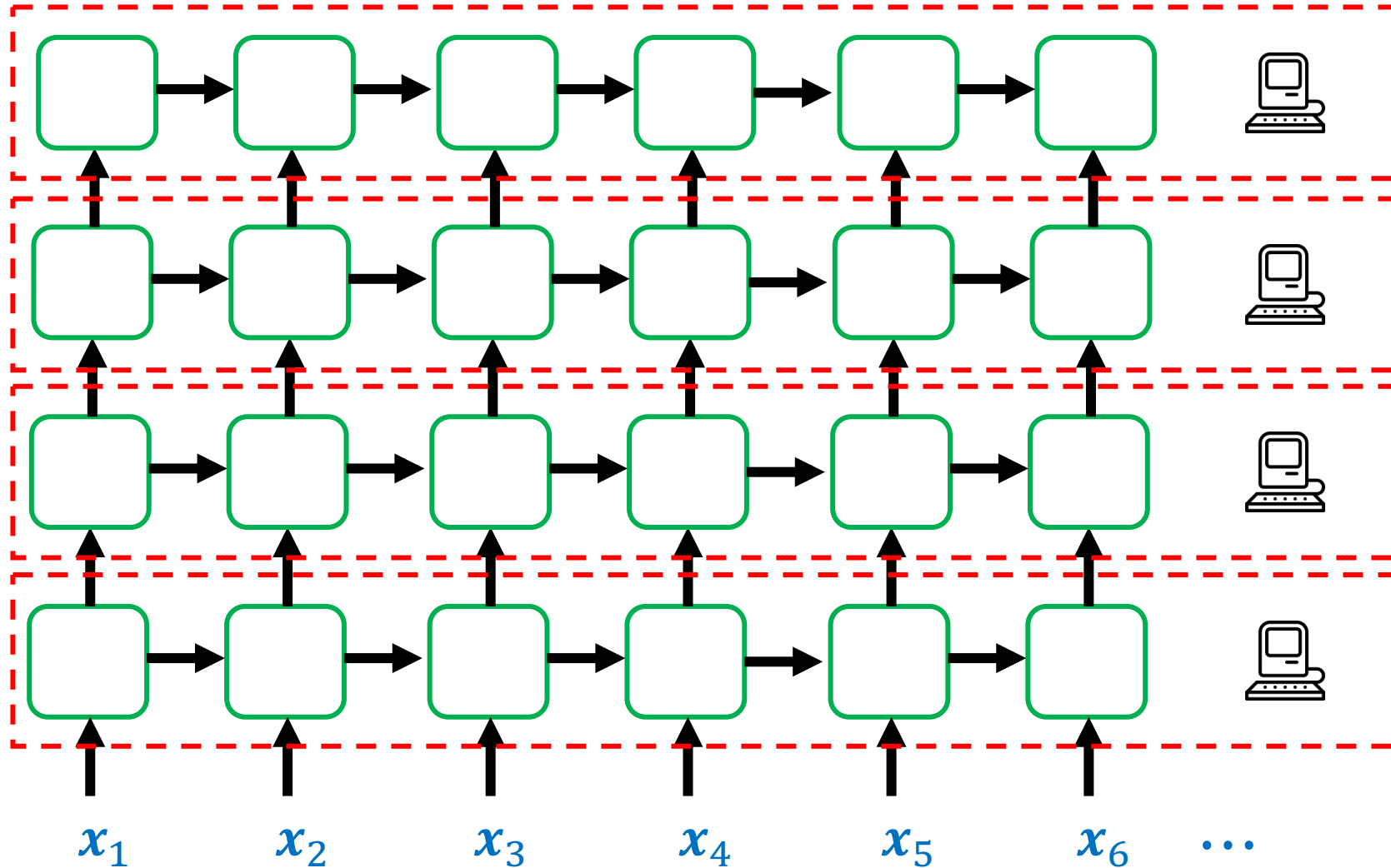

forward prop   backward prop   update weights
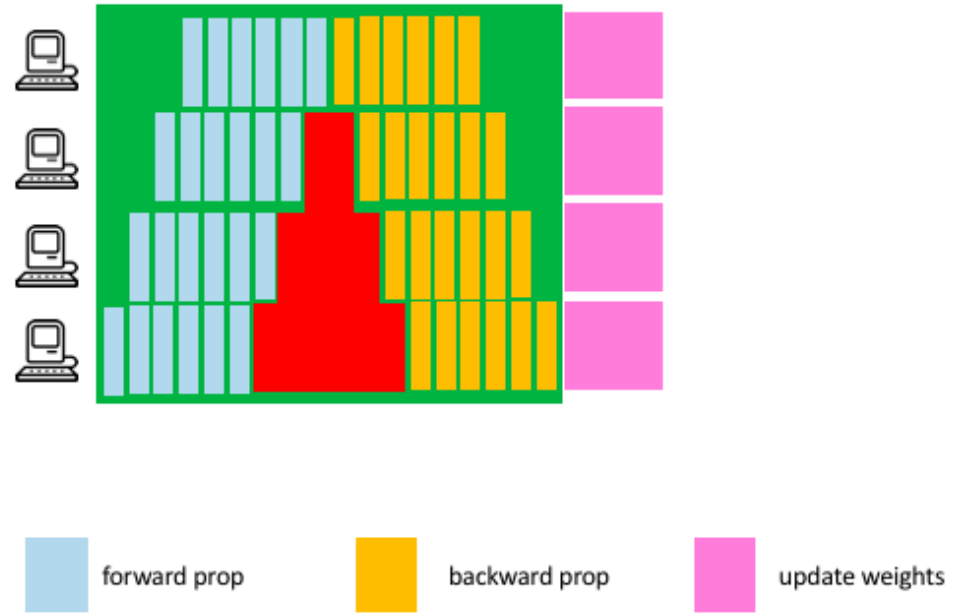
$$\text{Loss factor} = \frac{\text{number of idle time slots}}{\text{number of available time slots}}$$

$$= \frac{2\dfrac{b}{m}(1 + 2 + \cdots + k - 1)}{2\left(b + (k-1)\dfrac{b}{m}\right)k} = \frac{1}{2}\left(1 - \frac{m}{k - 1 + m}\right) \qquad \downarrow \text{ with } m$$
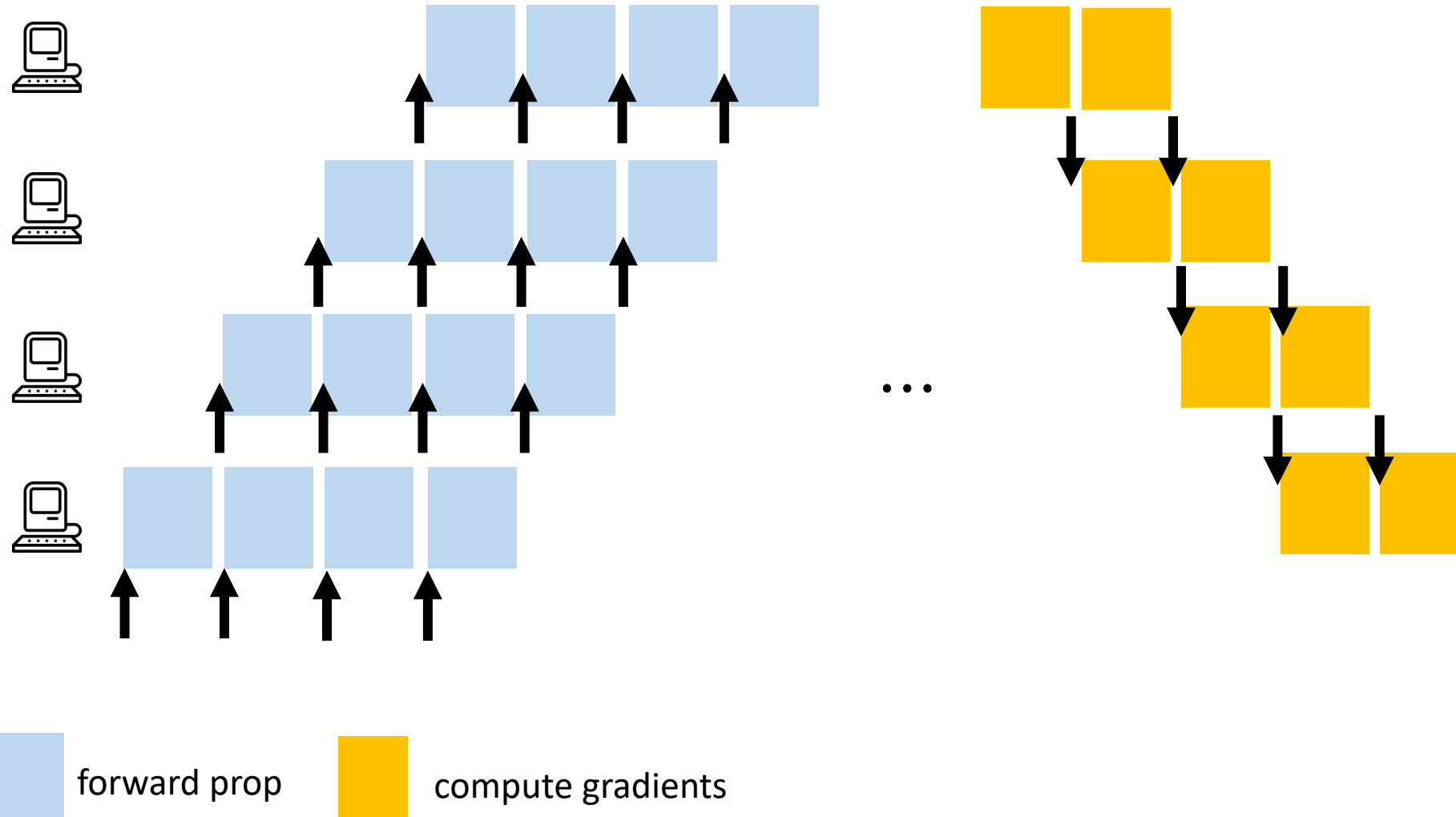
Recurrent neural networks

$$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \cdots$$

Recurrent neural networks (cont'd)

x_1 x_2 x_3 x_4 x_5 x_6 ...

Recurrent neural networks (cont'd)

forward prop · compute gradients
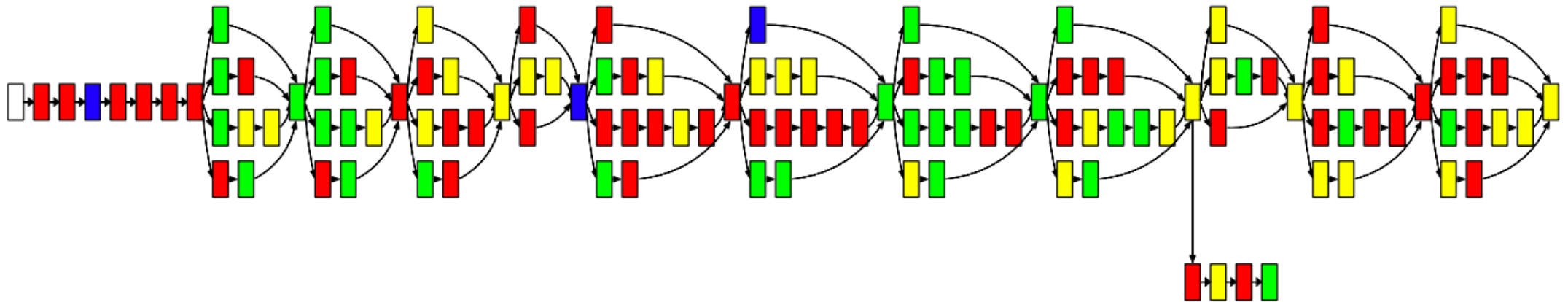
Device placement problem

- Placing operators (ex. linear transformations, activation functions) of a neural network onto devices (ex. CPU, GPU, …) for training is a hard problem

- Recent proposals use reinforcement learning for this problem
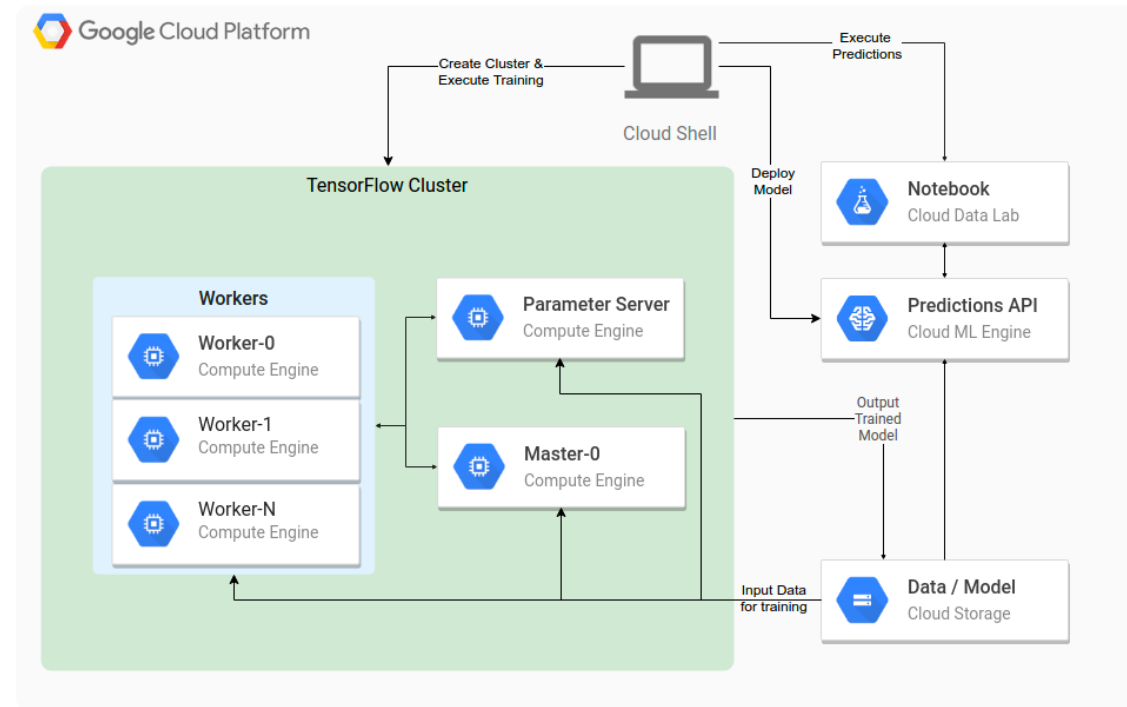

- Source: Mirhoseini et al (2017)

References

- Li et al, Scaling Distributed Machine Learning with the Parameter Server, OSDI 2014

- Abadi et al, TensorFlow: A system for large-scale machine learning, OSDI 2016

- Seide et al, 1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs, Interspeech 2014

- Alistarh et al, QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding, NIPS 2017, QSGD-TF

- Huang et al, GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism, NeurIPS 2019

- Vogels et al, PowerSGD: Practical low-rank gradient compression for distributed optimization, NeurIPS 2019, PowerSGD

References (cont'd)

- Akiba et al (2017), Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes, Arxiv

- Chen and Huo (2016), Scalable Training of Deep Learning Machines by Incremental Block Training with Intra-block Parallel Optimization and Blockwise Model-Update Filtering, ICASSP

- Goyal et al (2016), Accurate, large minibatch SGD: training ImagenNet in 1 hour, Arxiv

- He et al (2016), Deep residual learning for image recognition, IEEE CVPR

- Jaggi et al (2014), Communication-Efficient Distributed Dual Coordinate Ascent, NIPS

- You et al (2017), Large batch training of convolutional networks, Arxiv

- You et al (2018), Scaling SGD Batch Size to 32K for ImageNet Training, Technical Report

- You et al (2018), ImageNet Training in Minutes, Arxiv

Seminar 9

- Training neural networks using TensorFlow on Google Cloud Platform

https://github.com/lse-st446/lectures2021/blob/master/Week10/class/README.md