Neural Network Workflow Systems

CSE545 - Spring 2023
Stony Brook University

H. Andrew Schwartz

i.e.
PyTorch
TensorFlow

Big Data Analytics, The Class

Goal: Generalizations
A *model* or *summarization* of the data.

Data Workflow Frameworks

- Hadoop File System ✓
- Spark ✓
- Streaming ✓
- MapReduce ✓
- **Deep Learning Frameworks**

Analytics and Algorithms

- Similarity Search
- Hypothesis Testing
- Transformers/Self-Supervision
- Recommendation Systems
- Link Analysis

Limitations of Spark

Spark is fast for being so flexible

- Fast: RDDs in memory + Lazy evaluation: optimized chain of operations.
- Flexible: Many transformations -- can contain any custom code.

Limitations of Spark

Spark is fast for being so flexible

- Fast: RDDs in memory + Lazy evaluation: optimized chain of operations.
- Flexible: Many transformations -- can contain any custom code.

However:

- Hadoop MapReduce can still be better for extreme IO, data that will not fit in memory across cluster.

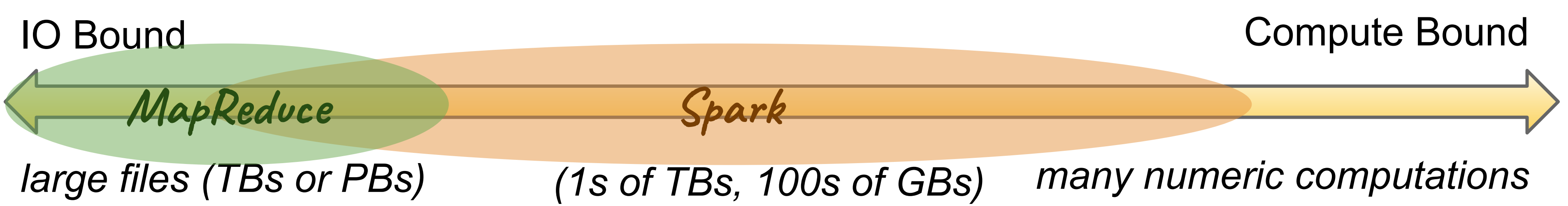
Limitations of Spark

Spark is fast for being so flexible

- Fast: RDDs in memory + Lazy evaluation: optimized chain of operations.
- Flexible: Many transformations -- can contain any custom code.

However:

- Hadoop MapReduce can still be better for extreme IO, data that will not fit in memory across cluster.



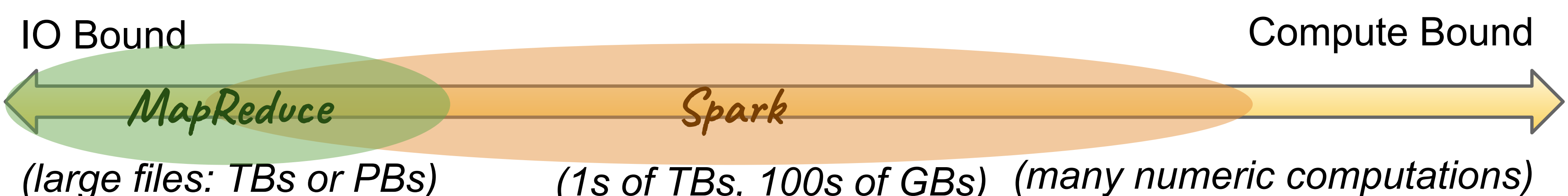
Limitations of Spark

Spark is fast for being so flexible

- Fast: RDDs in memory + Lazy evaluation: optimized chain of operations.
- Flexible: Many transformations -- can contain any custom code.

However:

- Hadoop MapReduce can still be better for extreme IO, data that will not fit in memory across cluster.
- Modern machine learning (esp. Deep learning), a common big data task, requires heavy numeric computation.



* this is the subjective approximation of the instructor as of February 2020. A lot of factors at play.

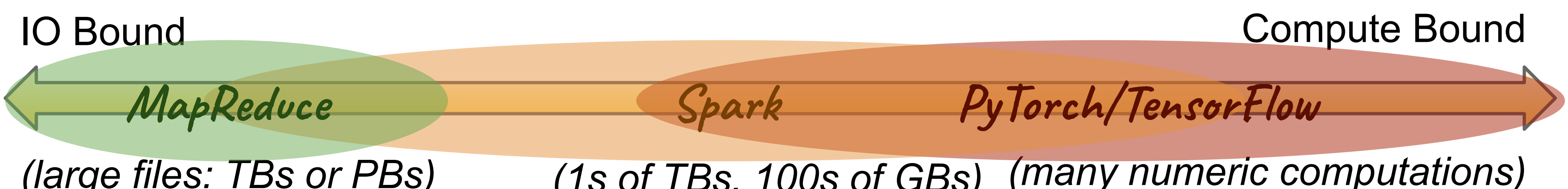
Limitations of Spark

Spark is fast for being so flexible

- Fast: RDDs in memory + Lazy evaluation: optimized chain of operations.
- Flexible: Many transformations -- can contain any custom code.

However:

- Hadoop MapReduce can still be better for extreme IO, data that will not fit in memory across cluster.
- Modern machine learning (esp. Deep learning), a common big data task, requires heavy numeric computation.



* this is the subjective approximation of the instructor as of February 2020. A lot of factors at play.

Learning Objectives

- Understand a neural network as transformations on tensors.
- Understand PyTorch as a data workflow system.
  - Know the key components of PyTorch
  - Understand the key concepts around *distributed* neural network processing.
- Execute basic pytorch on moderately large data.
- Establish a foundation to distribute deep learning models

What is PyTorch?

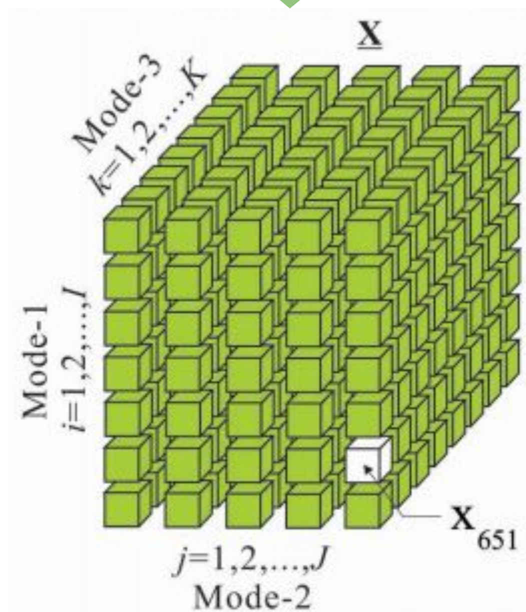
A workflow system catered to numerical computation.

One view: Like Spark, but uses *tensors* instead of *RDDs*.

What is a tensor?

A workflow system catered to numerical computation.

One view: Like Spark, but uses *tensors* instead of *RDDs*.



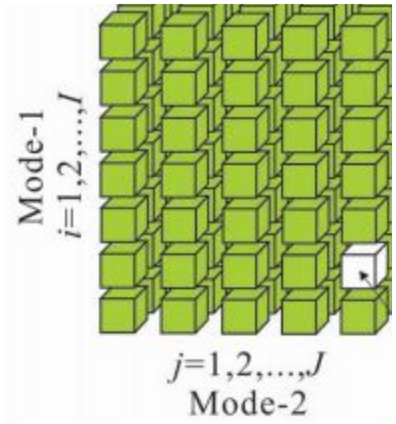
A multi-dimensional matrix

(i.stack.imgur.com)

What is a tensor?

A workflow system catered to numerical computation.

One view: Like Spark, but uses *tensors* instead of *RDDs*.



(i.stack.imgur.com)

A 2-d tensor is just a matrix.

1-d: vector

0-d: a constant / scalar

Note: Linguistic ambiguity:
Dimensions of a Tensor =/=
Dimensions of a Matrix

What is a tensor?

A workflow system catered to numerical computation.

One view: Like Spark, but uses *tensors* instead of *RDDs*.

Examples > 2-d :
Image definitions in terms of RGB per pixel
  Image[*row*][*column*][*rgb*]

Subject, Verb, Object representation of language:
  Counts[verb][subject][object]

What is a tensor?

A workflow system catered to numerical computation.

One view: Like Spark, but uses *tensors* instead of *RDDs*.

Technically, less abstract than *RDDs* which could hold tensors as well as many other data structures (dictionaries/HashMaps, Trees, ...etc…).

Then, why PyTorch?

Why Pytorch?

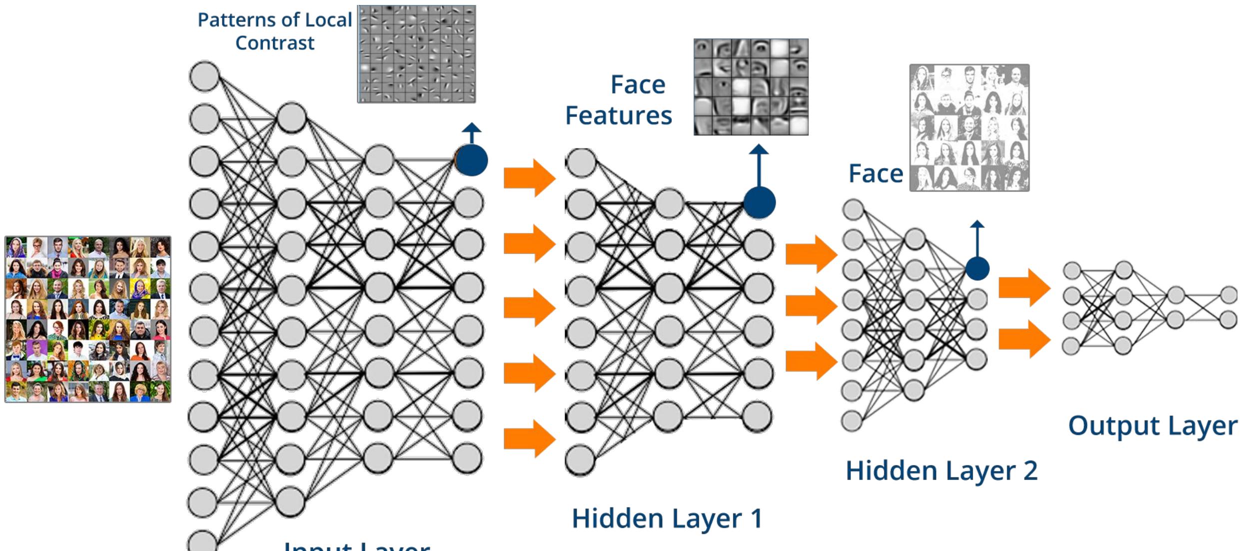
Efficient, high-level built-in **linear algebra** and **machine learning optimization** *operations* (i.e. transformations).

enables complex models, like deep learning

Why PyTorch?

Efficient, high-level built-in **linear algebra** and **machine learning optimization** *operations*.

enables complex models, deep neural networks



(Bakshi, 2016, "What is Deep Learning? Getting Started With Deep Learning")

From Linear Regression to Neural Nets

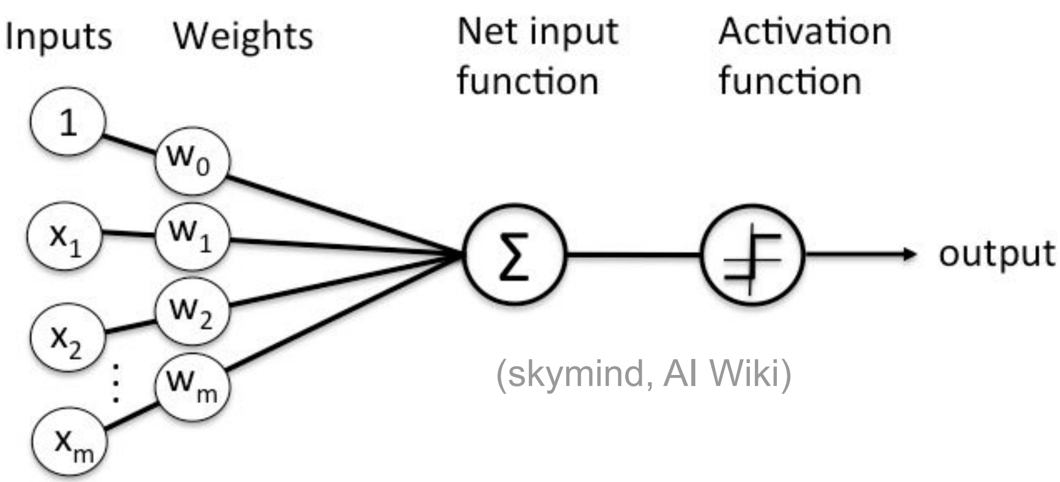
Linear Regression: $\hat{y} = wX$

Neural Network Nodes: $output = f(wX)$

From Linear Regression to Neural Nets

Linear Regression: $\hat{y} = wX$

Neural Network Nodes: $output = f(wX)$

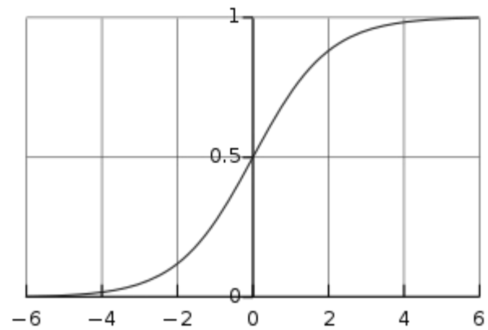


(skymind, AI Wiki)

From Linear Regression to Neural Nets

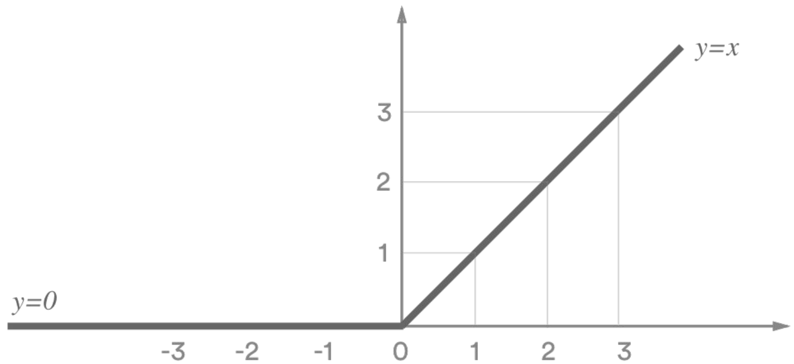
$z = wX$

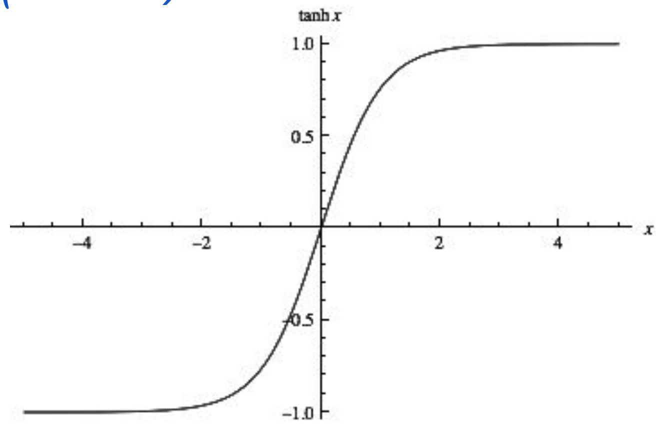
Logistic: $\sigma(z) = 1 / (1 + e^{-z})$



Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1)$

Rectified linear unit (ReLU): $ReLU(z) = \max(0, z)$

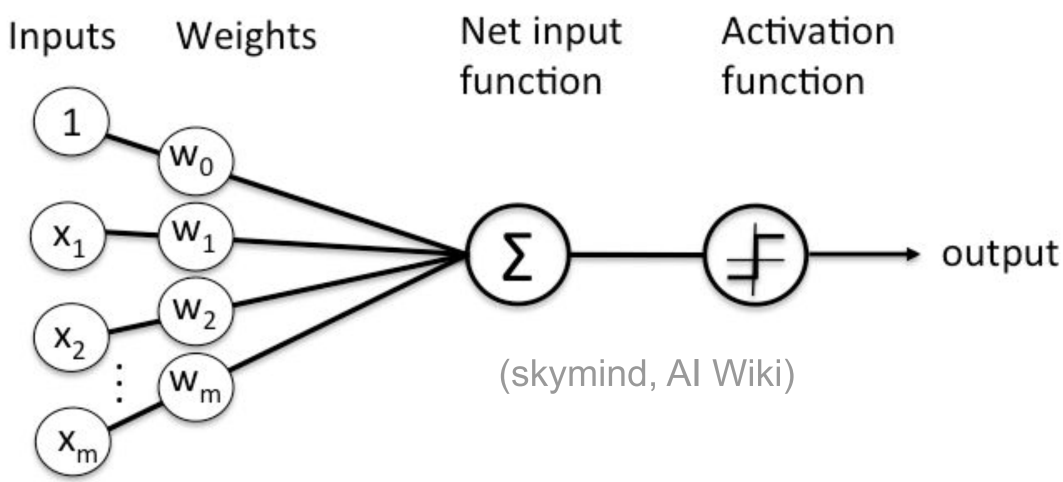




From Linear Regression to Neural Nets

Linear Regression: $y = wX$

Neural Network Nodes: $output = f(wX)$



(skymind, AI Wiki)

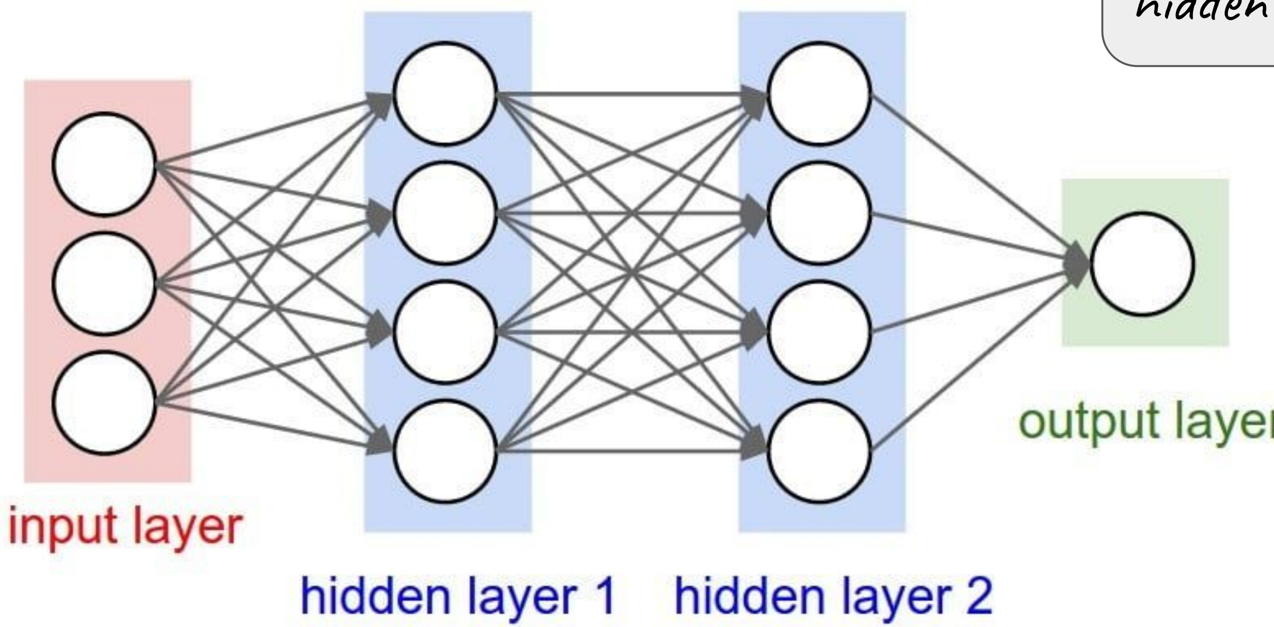
PyTorch

Efficient, high-level built-in **linear algebra** for deep neural network operations.

PyTorch

Efficient, high-level built-in **linear algebra** for **deep** neural network operations.

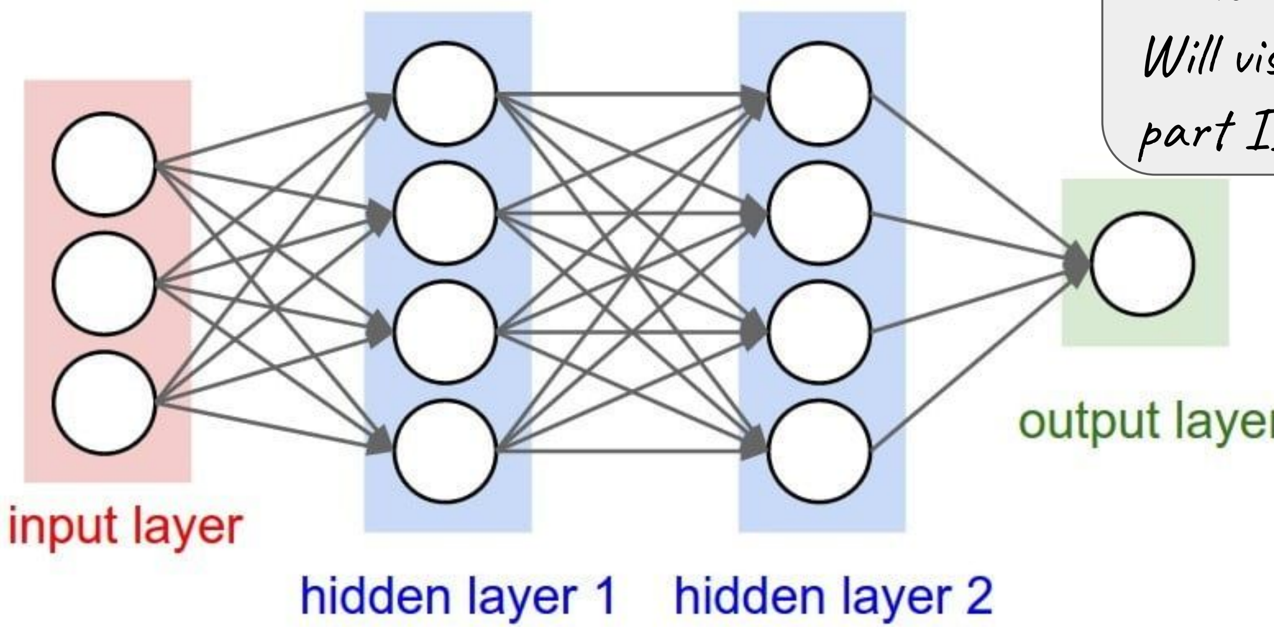
More than one hidden layer.



PyTorch

Efficient, high-level built-in **linear algebra** for **deep** neural network operations.

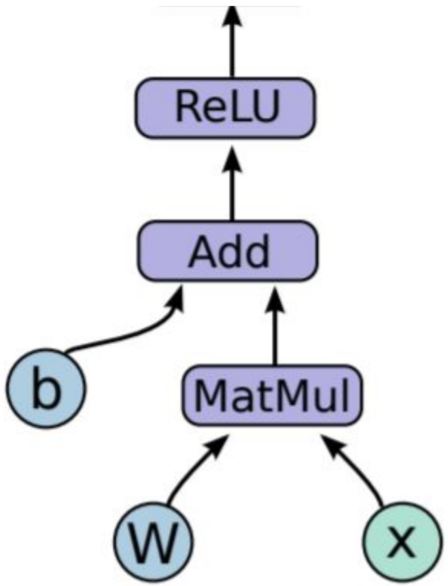
More than one hidden layer. Will visit in part II



PyTorch

Efficient, high-level built-in **linear algebra** for neural network operations.

Can be conceptualized as a graph of operations on tensors (matrices):



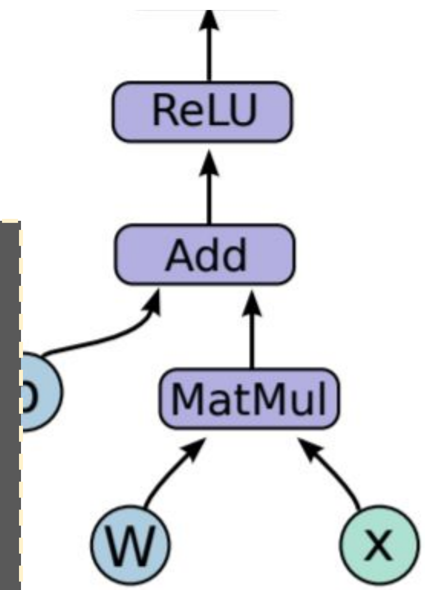
PyTorch

Efficient, high-level built-in **linear algebra** for neural network operations.

Can be conceptualized as a graph of operations on tensors (matrices):

```
import torch
from torch import nn #predefined nodes

x = torch.Tensor(input)
w= torch.random.randn(X.shape, 1) #weights
z = torch.matmul(x, beta)
yhat = nn.functional.relu(z)
loss = nn.MSELoss(yhat, torch.Tensor(y))
```



Linear Regression

Linear Regression: $\hat{y} = \beta X$

Objective: *Learn w, such that* $(y - \beta X)^2$ *is minimized*

Linear Regression

Linear Regression: $\hat{y} = \beta X$

Objective: *Learn w, such that* $(y - \beta X)^2$ *is minimized*

How do we solve for β?

Linear Regression

Linear Regression: $\hat{y} = \beta X$

Objective: *Learn w, such that $(y - \beta X)^2$ is minimized*

How do we solve for β?

1. Analytic Gradient: Differentiate the objective, solve the system of equations by equating it to 0

Linear Regression

Linear Regression: $\hat{y} = \beta X$

Objective: *Learn w, such that* $(y - \beta X)^2$ *is minimized*

How do we solve for β?

1. Analytic Gradient: Differentiate the objective, solve the system of equations by equating it to 0

$$\beta_{opt} = (X^T X)^{-1} X^T y$$

Linear Regression

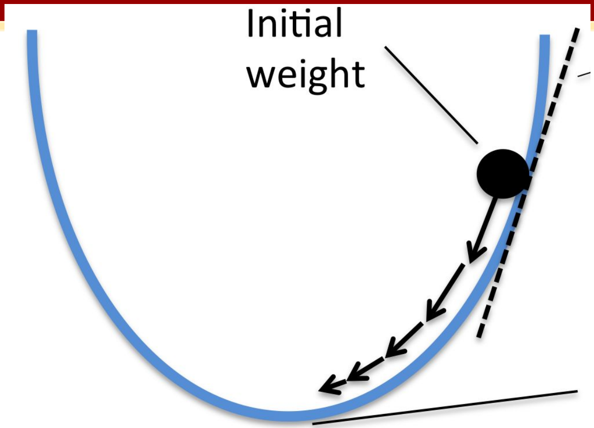
Linear Regression: $\hat{y} = \beta X$

Objective: *Learn w, such that $(y - \beta X)^2$ is minimized*

How do we solve for β?

1. Analytic Gradient: Differentiate the objective, solve the system of equations by equating it to 0
2. Numerical Gradient: Start at a random point and move in the direction of minima until optima is reached

Linear Regression



Linear Regression: $\hat{y} = \beta X$

Objective: *Learn w, such that* $(y - \beta X)^2$ *is minimized*

How do we solve for β?

1. Analytic Gradient: Differentiate the objective, solve the system of equations by equating it to 0
2. **Numerical Gradient: Start at a random point and move in the direction of minima until optima is reached**

Numerical Gradient Approach

Linear Regression: Trying to find “betas” that minimize:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i}^{N}(y_i - \hat{y}_i)^2\}$$

Numerical Gradient Approach

Linear Regression: Trying to find “betas” that minimize:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i}^{N}(y_i - \hat{y}_i)^2\}$$

matrix multiply

$$\hat{y}_i = X_i\beta$$

Numerical Gradient Approach

Linear Regression: Trying to find "betas" that minimize:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i}^{N}(y_i - \hat{y}_i)^2\}$$

matrix multiply

$$\hat{y}_i = X_i\beta$$

Thus:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i=0}^{N}(y_i - X_i\beta)^2\}$$

Numerical Gradient Approach

Linear Regression: Trying to find “betas” that minimize:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i}^{N}(y_i - \hat{y}_i)^2\}$$

$\hat{y}_i = X_i\beta$ Thus: $$\hat{\beta} = argmin_{\beta}\{\sum_{i=0}^{N}(y_i - X_i\beta)^2\}$$

How to update? $\beta_{new} = \beta_{prev} - \alpha * \text{grad}$

Numerical Gradient Approach

Linear Regression: Trying to find "betas" that minimize:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i}^{N}(y_i - \hat{y}_i)^2\}$$

$\hat{y}_i = X_i\beta$ Thus: $$\hat{\beta} = argmin_{\beta}\{\sum_{i=0}^{N}(y_i - X_i\beta)^2\}$$

How to update? $\beta_{new} = \beta_{prev} - \alpha * \text{grad}$

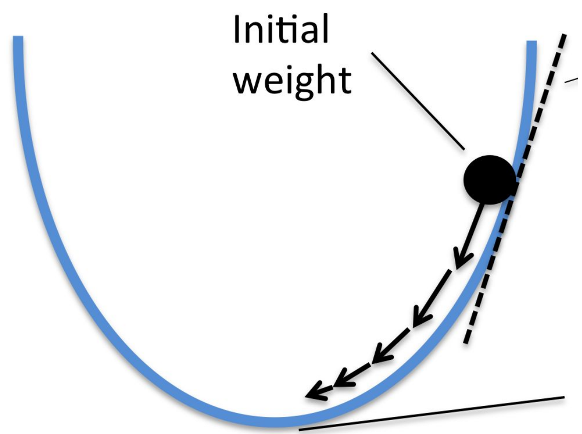
α: Learning Rate

Numerical Gradient Approach

Linear Regression: Trying to find "betas" that minimize:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i}^{N}(y_i - \hat{y}_i)^2\}$$

$\hat{y}_i = X_i\beta$ Thus: $\hat{\beta} = argmin_{\beta}\{\sum_{i=0}^{N}(y_i -$



How to update? $\beta_{new} = \beta_{prev} - \alpha * \text{grad}$

α: Learning Rate

Numerical Gradient Approach

Linear Regression: Trying to find "betas" that minimize:

Gradient Descent: $\beta_{new} = \beta_{prev} - \alpha * \mathrm{grad}$

Numerical Gradient Approach

Linear Regression: Trying to find “betas” that minimize:

Gradient Descent: $\beta_{new} = \beta_{prev} - \alpha * \text{grad}$

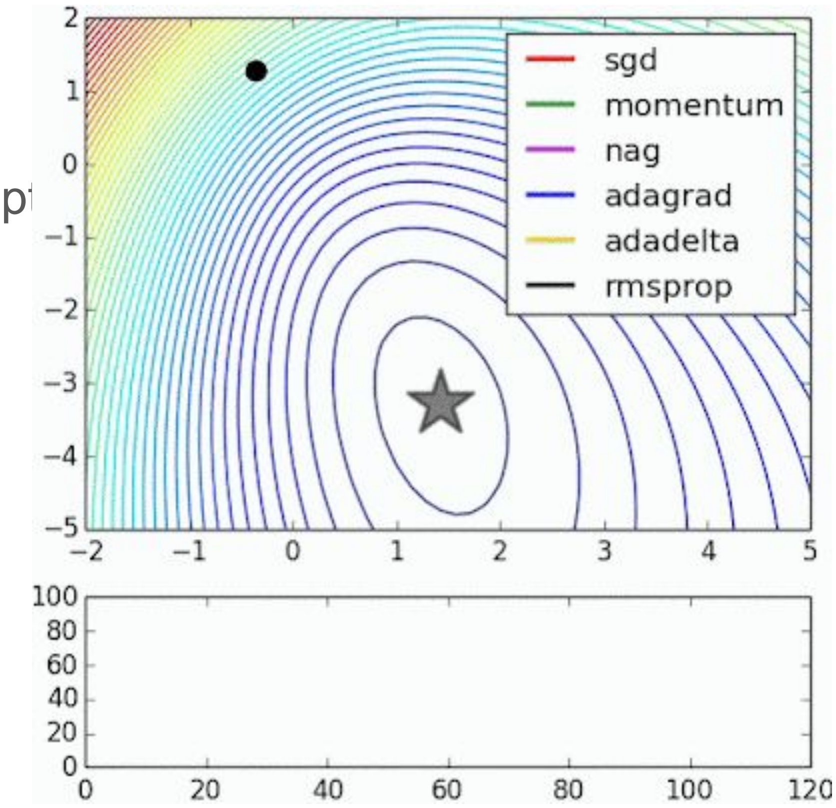
But there are other gradient descent based optimization methods which are better*

Numerical Gradient Approach

Linear Regression: Trying to find “betas” that minimize:

Gradient Descent: $\beta_{new} = \beta_{prev} - \alpha * \text{grad}$

But there are other gradient descent based opt



Animation: Alec Radford

Linear Regression as DAG

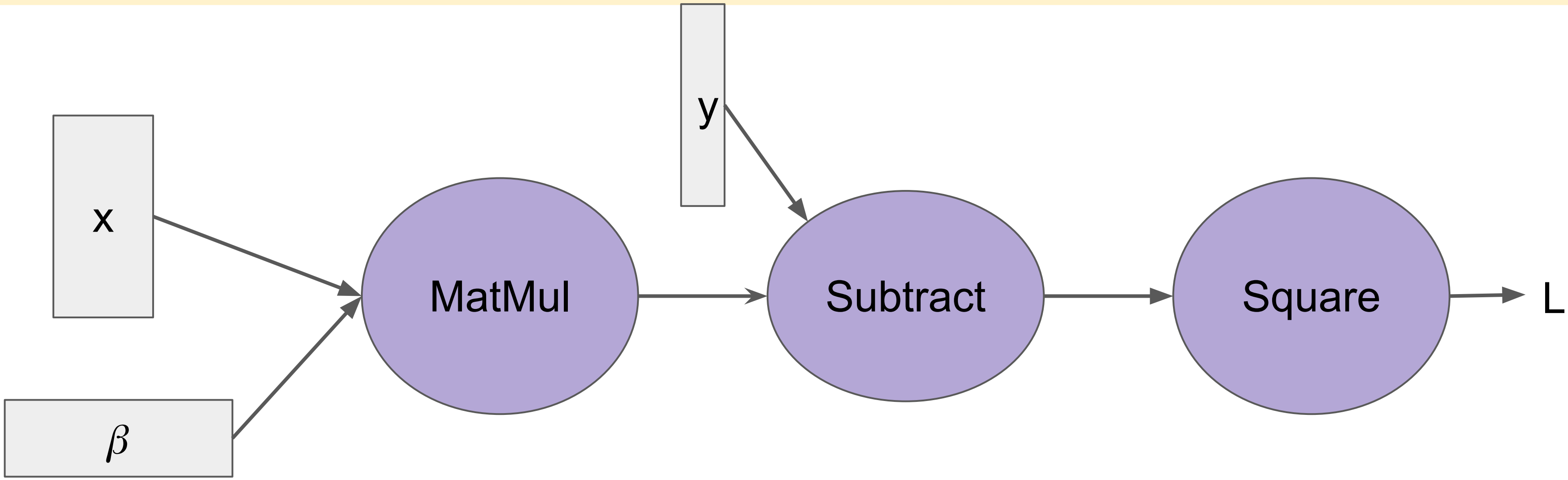
How do Machine learning/ Deep learning frameworks represent these models?

Linear Regression as DAG

How do Machine learning/ Deep learning frameworks represent these models?

Computational Graph!

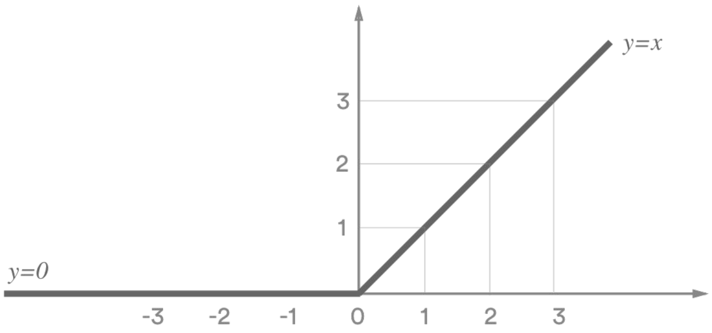
Linear Regression as DAG



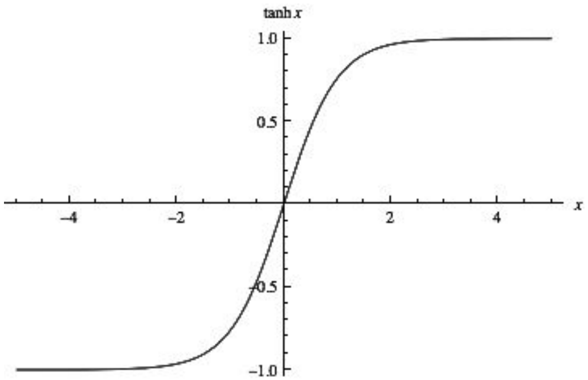
$$L = (y - \beta x)^2$$

Activations

Rectified linear unit (ReLU): $ReLU(z) = \max(0, z)$

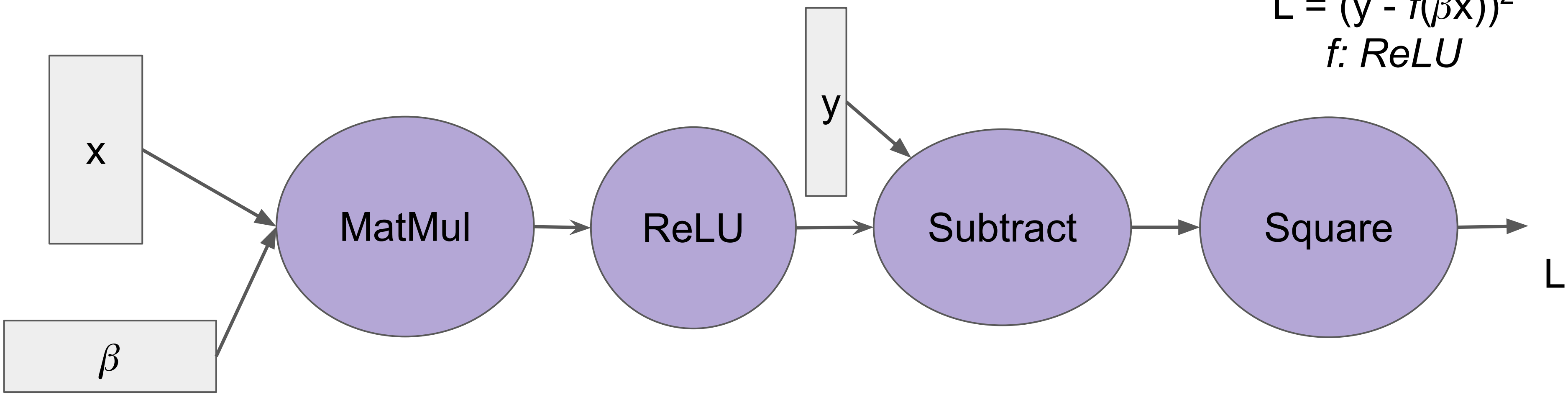


Hyperbolic tangent: $tanh(z) = (e^{2z} - 1) / (e^{2z} + 1)$



Linear Regression as DAG

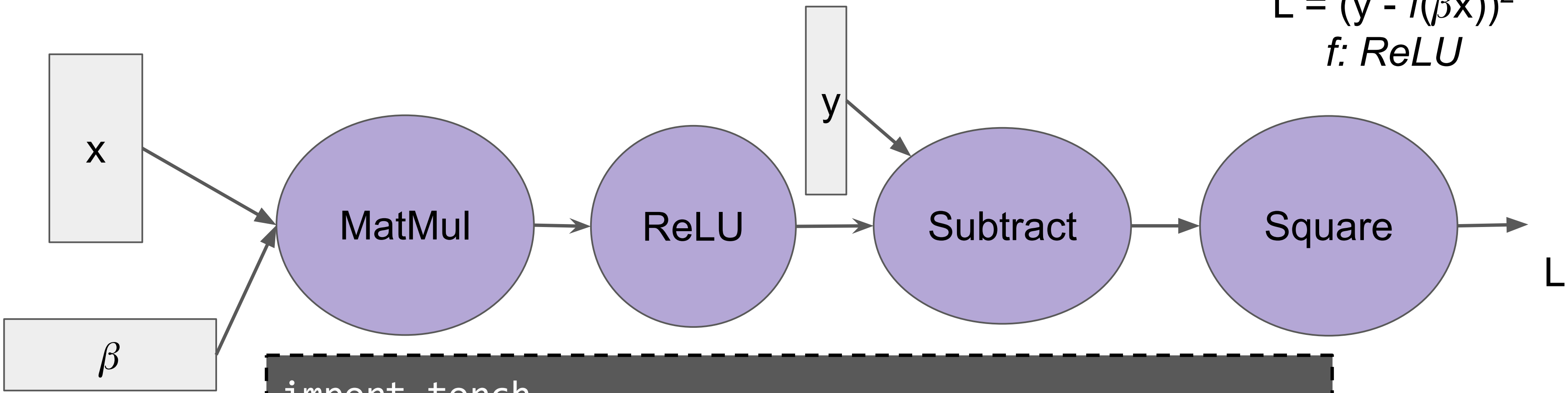
$L = (y - f(\beta x))^2$
f: ReLU



Linear Regression as DAG

$$L = (y - f(\beta x))^2$$

f: ReLU




```
import torch
from torch import nn

x = torch.Tensor(input_features)
y = torch.Tensor(input_scores)
beta = torch.random.randn(X.shape, 1)
z = torch.matmul(x, beta)
yhat = nn.functional.relu(z)
loss = nn.MSELoss(yhat, y)
```

PyTorch Demo

Native Linear Regression Implementation (Link)

Torch.nn Linear Regression Implementation (Link)

Ingredients of PyTorch

torch.Tensor
useful attributes:
`dtype`: data type (`'torch.float32'`)
`shape`: tensor size
`device`: where to store

operations *(`torch.`)*
computation on tensors, e.g. :
`+, *, .floor, .abs`
`.sum, .max, .mean,`
`.matmul, .unique`

building blocks *(`torch.nn`)*
predefined layers; e.g.:
`.Linear, .ReLu,`
`.MSELoss, .Transformer`
`.CrossEntropyLoss`

nn.Module
`__init__`
`forward`
(graph)

Ingredients of PyTorch

```
class ToyModel(nn.Module): #Pytorch: graph example
    def __init__(self):
        #initialize all nn objects:
        super(ToyModel, self).__init__()
        self.net1 = torch.nn.Linear(10, 10)
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 1)

    def forward(self, x):
        #define graph
        x = self.relu(self.net1(x))
        return self.net2(x)
```

perations ***(*****torch. *****)***

computation on tensors, e.g. :

`+, *, .floor, .abs`

`.sum, .max, .mean,`

`.matmul, .unique`

building blocks ***(***torch.nn***)***

predefined layers; e.g.:

`.Linear, .ReLu,`

`.MSELoss, .Transformer`

`.CrossEntropyLoss`

nn.Module

`__init__`

`forward`

(graph)

Ingredients of PyTorch

```
class ToyModel(nn.Module): #Pytorch: graph example
    def __init__(self):
        #initialize all nn objects:
        super(ToyModel, self).__init__()
        self.net1 = torch.nn.Linear(10, 10)
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 1)

    def forward(self, x):
        #define graph
        x = self.relu(self.net1(x))
        return self.net2(x)
…
tm =ToyModel()
#training loop
for i in range(num_iters):
    …
    y_pred = tm(x)
    nn.MSELoss(y_pred, y)
    …
```

***perations** (`torch.`)*

computation on tensors, e.g. :

`+`, `*`, `.floor`, `.abs`

`.sum`, `.max`, `.mean`,

`.matmul`, `.unique`

nn.Module

`__init__`

`forward`

(graph)

PyTorch

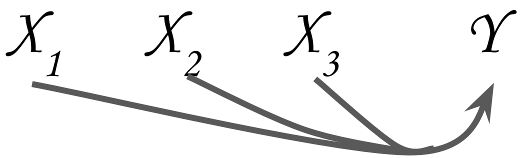
Typical use-case: (Supervised Machine Learning)

Determine weights, $\mathcal{W}$, of a function, f, such that $|\boldsymbol{\varepsilon}|$ is minimized: $f(\mathcal{X}|\mathcal{W}) = \mathcal{Y} + \boldsymbol{\varepsilon}$

PyTorch

Typical use-case:

Determine weights, $\mathcal{W}$, of a function, f, such that $|\boldsymbol{\varepsilon}|$ is minimized: $f(X|W) = Y + \boldsymbol{\varepsilon}$



PyTorch

Typical use-case:

Determine weights, $\mathcal{W}$, of a function, f, such that $|\boldsymbol{\varepsilon}|$ is minimized: $f(X|W) = Y + \boldsymbol{\varepsilon}$

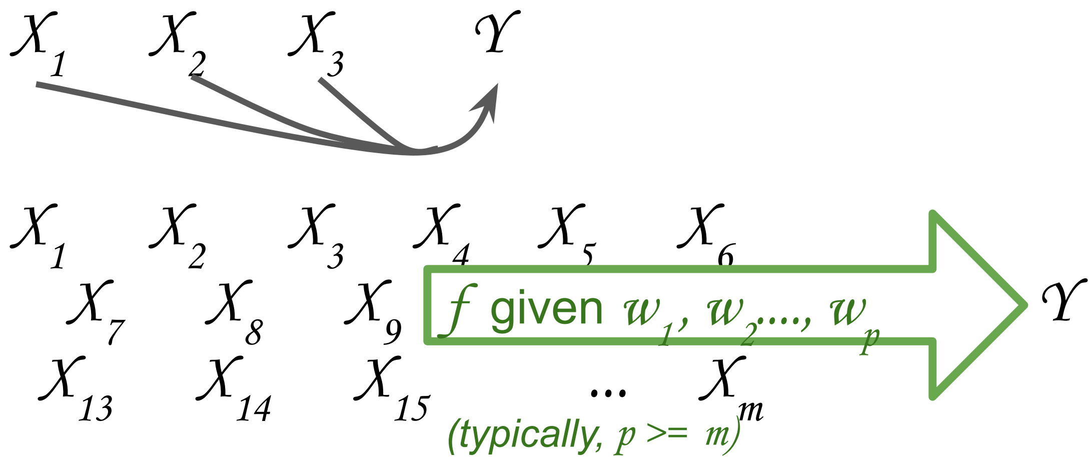
$$X_1 \quad X_2 \quad X_3 \qquad Y$$

$$X_1 \quad X_2 \quad X_3 \quad X_4 \quad X_5 \quad X_6$$
$$X_7 \quad X_8 \quad X_9 \quad X_{10} \quad X_{11} \quad X_{12} \qquad\qquad Y$$
$$X_{13} \quad X_{14} \quad X_{15} \quad \dots \quad X_m$$

PyTorch

Typical use-case:

Determine weights, $\mathcal{W}$, of a function, f, such that $|\varepsilon|$ is minimized: $f(X|W) = Y + \varepsilon$



PyTorch

$$f(X|W) = \hat{Y}$$
$$Y = (X|W) + \varepsilon$$
$$Y = \hat{Y} + \varepsilon$$
$$\varepsilon = \hat{Y} - Y$$

Typical use-case:

Determine weights, $\mathcal{W}$, of a function, f, such that $|\varepsilon|$ is minimized

ε

X_1 X_2 X_3 Y

X_1 X_2 X_3 X_4 X_5 X_6
X_7 X_8 X_9
X_{13} X_{14} X_{15} … X_m

f given $w_1, w_2, \ldots, w_p$ → Y

(typically, $p >= m$)

PyTorch

Typical use-case:

Determine weights, $\mathcal{W}$, of a function, f, such that $|\varepsilon|$ is minimized

ε

$$F(X|W) = \hat{Y}$$
$$Y = F(X|W) + \varepsilon$$
$$Y = \hat{Y} + \varepsilon$$
$$\varepsilon = \hat{Y} - Y$$

X_1 X_2 X_3 Y

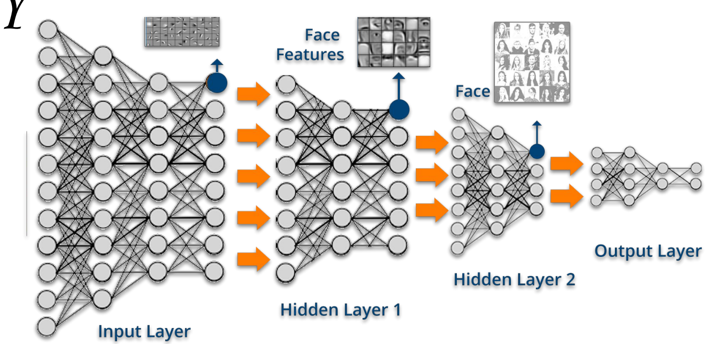
Typically, very complex!

X_1 X_2 X_3 X_4 X_5 X_6
X_7 X_8 X_9
X_{13} X_{14} X_{15} … X_m

f given $w_1, w_2, \ldots, w_p$

(typically, $p >= m$)

Y



PyTorch

$$F(X|W) = \hat{Y}$$
$$Y = (X|W) + \varepsilon$$
$$Y = \hat{Y} + \varepsilon$$
$$\varepsilon = \hat{Y} - Y$$

Typical use-case:

Determine weights, $\mathcal{W}$, of a function, f, such that $|\varepsilon|$ is minimized

ε

$\mathcal{W}$ determined through *gradient descent*:

back propagating error across the network that defines f

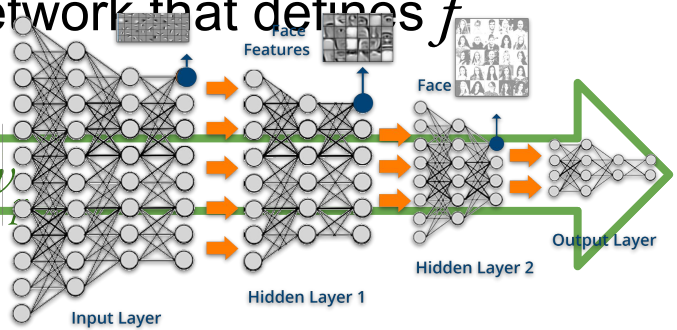
$X_1^{(1)}$ X_2 X_3 X_4 X_5 X_6

X_7 X_8 X_9

X_{13} X_{14} X_{15} … X_m

f given $w_1, w_2, \ldots, w$

(typically, $p >= m$)



$Y^{(1)}$

PyTorch

$$F(X|W) = \hat{Y}$$
$$Y = (X|W) + \varepsilon$$
$$Y = \hat{Y} + \varepsilon$$
$$\varepsilon = \hat{Y} - Y$$

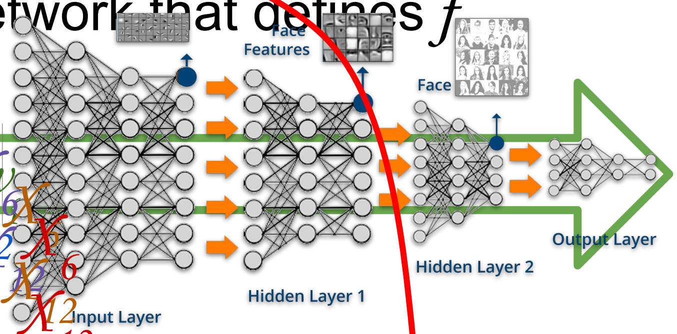
Typical use-case:

Determine weights, $\mathcal{W}$, of a function, f, such that $|\varepsilon|$ is minimized

ε

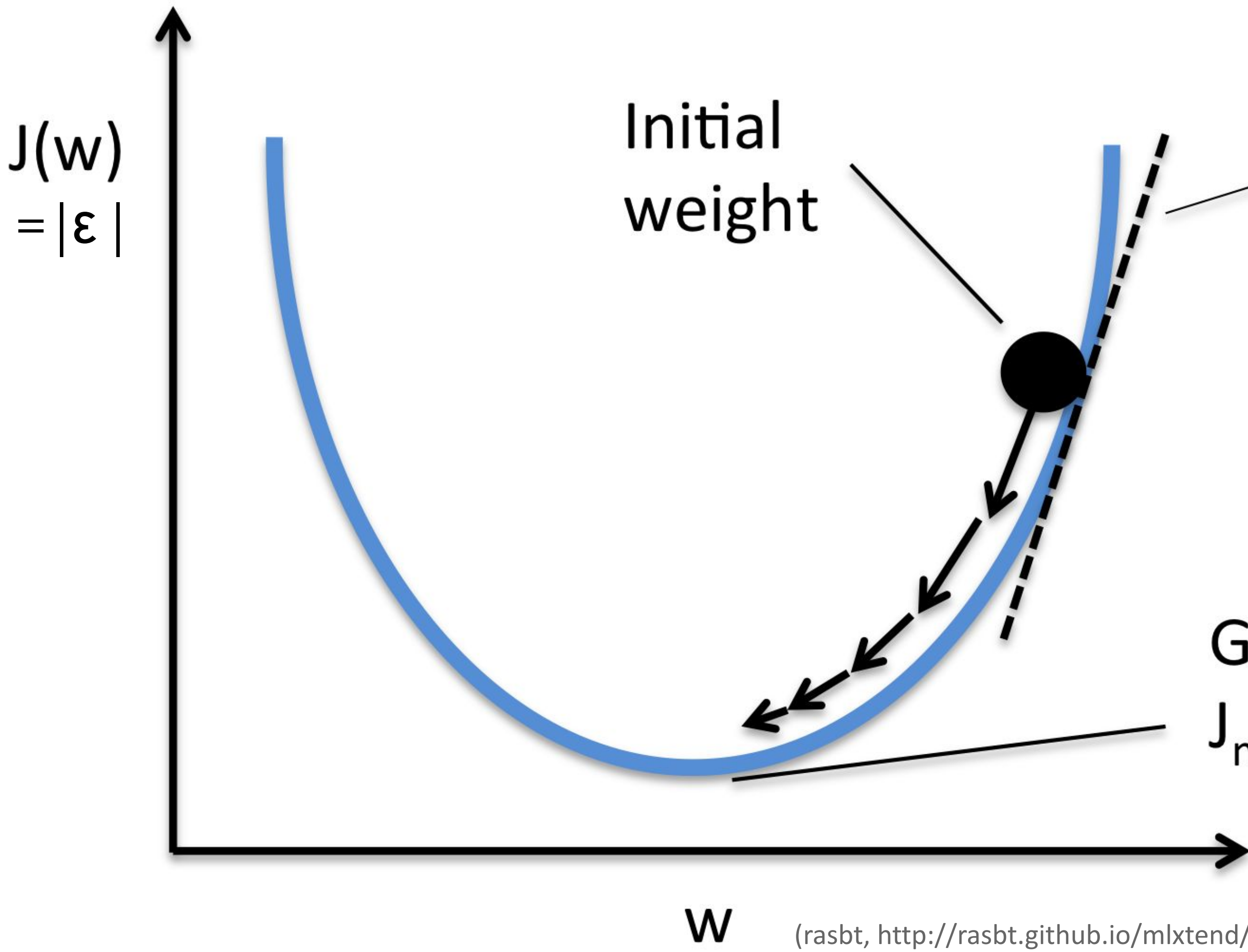
$\mathcal{W}$ determined through *gradient descent*:

back propagating error across the network that defines f



minimizes ε on ***N*** training examples

Weights Derived from Gradients



(rasbt, http://rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/)

Weights Derived from Gradients

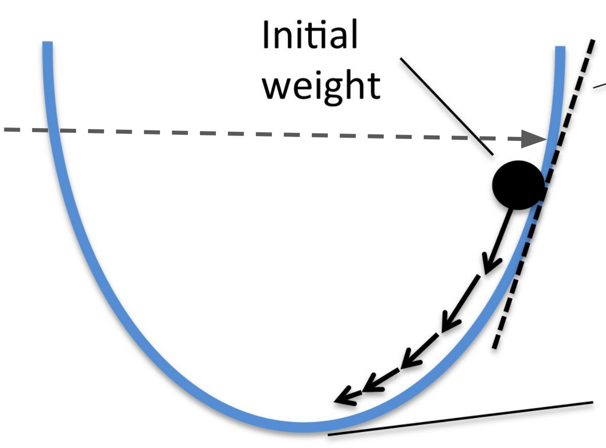
Linear Regression: Trying to find “betas” that minimize:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i}^{N}(y_i - \hat{y}_i)^2\}$$

$\hat{y}_i = X_i\beta$ Thus: $$\hat{\beta} = argmin_{\beta}\{\sum_{i=0}^{N}(y_i - X_i\beta)^2\}$$

How to update? $\beta_{new} = \beta_{prev} - \alpha * \mathrm{grad}$

(for gradient descent) “learning rate”



Weights Derived from Gradients

Linear Regression: Trying to find "betas" that minimize:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i}^{N}(y_i - \hat{y}_i)^2\}$$

matrix multiply

$$\hat{y}_i = X_i\beta$$

Thus:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i=0}^{N}(y_i - X_i\beta)^2\}$$

In standard linear equation:

$$y = mx + b$$

$$\text{let } x' = x + [1, 1, ..., 1]_N^T$$

$$\text{then, } y = mx'$$

(if we add a column of 1s, mx + b is just matmul(m, x))

How to train GPT3?

Time to train Bert Large (330 M) on K80, which is 530 times smaller than GPT3

# GPUs	Training Time (minutes)	Per-GPU Scaling Efficiency
1	399	1.00

Dave Troiano, 2020

How to train GPT3?

Time to train Bert Large (330 M) on K80, which is 530 times smaller than GPT3

# GPUs	Training Time (minutes)	Per-GPU Scaling Efficiency
1	399	1.00

For the same amount of data, GPT3 can be trained in 212k mins = 3533 hours = 147 days*

[Dave Troiano, 2020](#)

**GPT3 wont fit into the memory of a single K80*

How to train GPT3?

Time to train Bert Large (330 M) on K80, which is 530 times smaller than GPT3

# GPUs	Training Time (minutes)	Per-GPU Scaling Efficiency
1	399	1.00
2	214	0.93
4	118	0.85
8	61	0.82

Dave Troiano, 2020

Options for distribution

1. **Distribute copies of entire dataset**
    a. Train over all with different hyperparameters
    b. Train different folds per worker node.

Options for distribution

1. **Distribute copies of entire dataset**
   a. Train over all with different parameters
   b. Train different folds per worker node.

2. **Distribute data**
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

Options for distribution

1. **Distribute copies of entire dataset**
    a. Train over all with different parameters
    b. Train different folds per worker node.

2. **Distribute data**
    a. Each node finds parameters for subset of data
    b. Needs mechanism for updating parameters
        i. Centralized parameter server
        ii. Distributed All-Reduce

3. **Distribute model or individual operations** (e.g. matrix multiply)

Options for distribution

1. **Distribute copies of entire dataset**
   a. Train over all with different parameters
   b. Train different folds per worker node.

   Pro: Easy; Good for compute-bound; Con: Requires data fit in worker memories

2. **Distribute data**
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

   Pro: Flexible to all situations; Con: Optimizing for subset is suboptimal

3. **Distribute model or individual operations** (e.g. matrix multiply)

   Pro: Parameters can be localized Con: High communication for transferring Intermediar data.

Options for distribution

Done often in practice. Not talked about much because it's mostly as easy as it sounds.

1. **Distribute copies of entire dataset**
   a. Train over all with different parameters
   b. Train different folds per worker node.

   Pro: Easy; Good for compute-bound; Con: Requires data fit in worker memories

2. **Distribute data**
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

   Pro: Flexible to all situations; Con: Optimizing for subset is suboptimal

3. **Distribute model or individual operations** (e.g. matrix multiply)

   Pro: Parameters can be localized Con: High communication for transferring Intermediar data.

Options for distribution

Done often in practice. Not talked about much because it's mostly as easy as it sounds.

1. **Distribute copies of entire dataset**
   a. Train over all with different parameters
   b. Train different folds per worker node.

   Pro: Easy; Good for compute-bound; Con: Requires data fit in worker memories

2. **Distribute data**
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

   Preferred method for big data or very complex models (i.e. models with many internal parameters).

   Pro: Flexible to all situations; Con: Optimizing for subset is suboptimal

3. **Distribute model or individual operations** (e.g. matrix multiply)

   Pro: Parameters can be localized Con: High communication for transferring Intermediar data.

Options for distribution

Done often in practice. Not talked about much because it's mostly as easy as it sounds.

1. **Distribute copies of entire dataset**
   a. Train over all with different parameters
   b. Train different folds per worker node.

   Pro: Easy; Good for compute-bound; Con: Requires data fit in worker memories

2. **Distribute data**
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

   Data Parellelism

   Preferred method for big data or very complex models (i.e. models with many internal parameters).

   Pro: Flexible to all situations; Con: Optimizing for subset is suboptimal

3. **Distribute model or individual operations** (e.g. matrix multiply)

   Model Parellelism

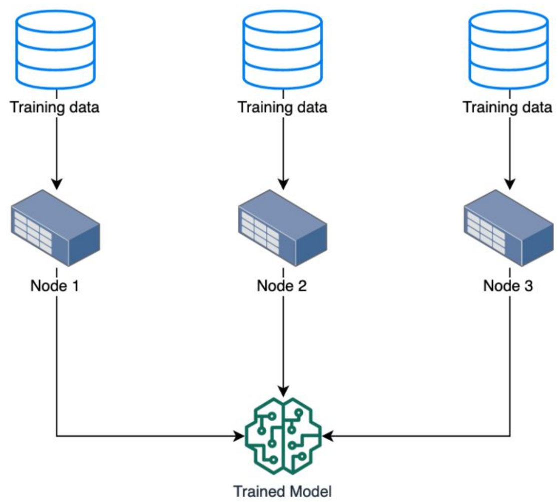
   Pro: Parameters can be localized Con: High communication for transferring Intermediar data.

Distributed Training

- Parallelism :
  - Data Parallelism
  - Model Parallelism
  - Hybrid

Distributed PyTorch Training

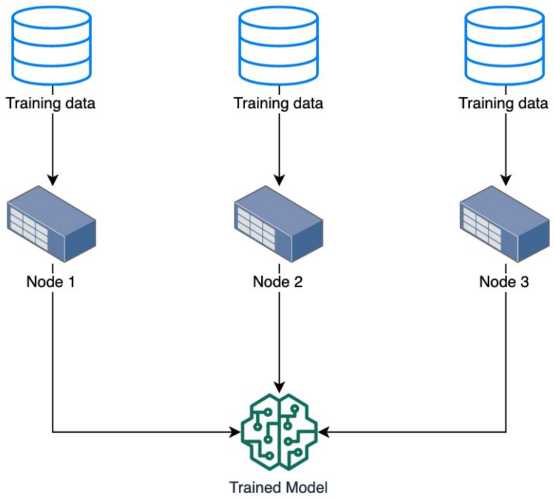
- Data Parallelism: Scatter dataset into parts across different workers to train on subsets and sync gradients



Data Parallelism

Distributed PyTorch Training

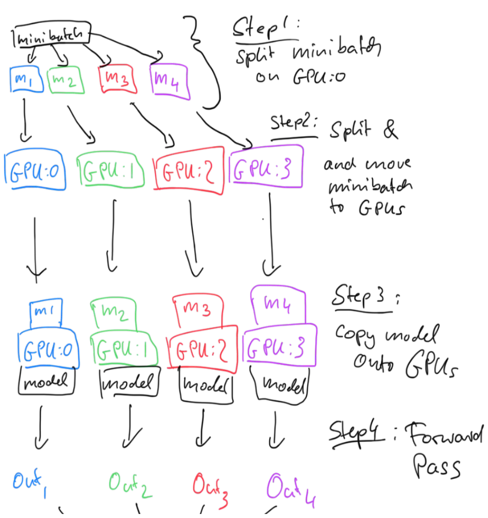
- Data Parallelism: Scatter dataset into parts across different workers to train on subsets and sync gradients
- Modes of Data Parallelism :
  - DataParallel
  - DistributedDataParallel

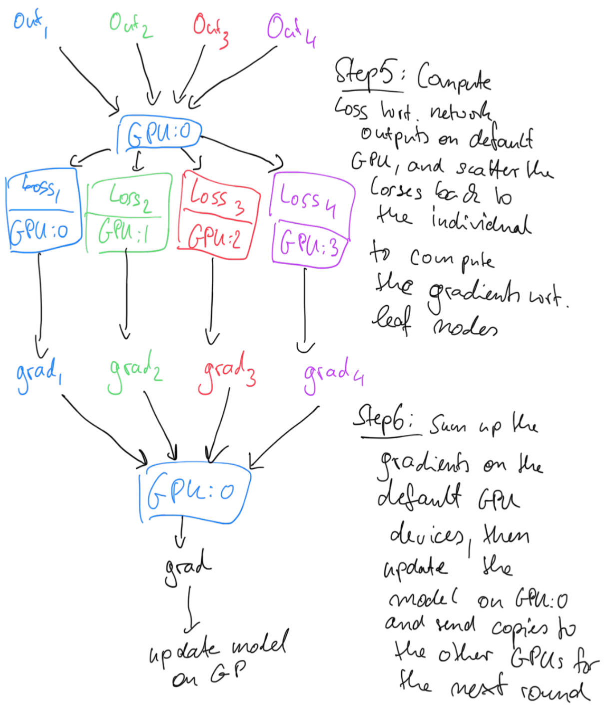


Data Parallelism

Distributed PyTorch Training

Data Parallel: How it works?





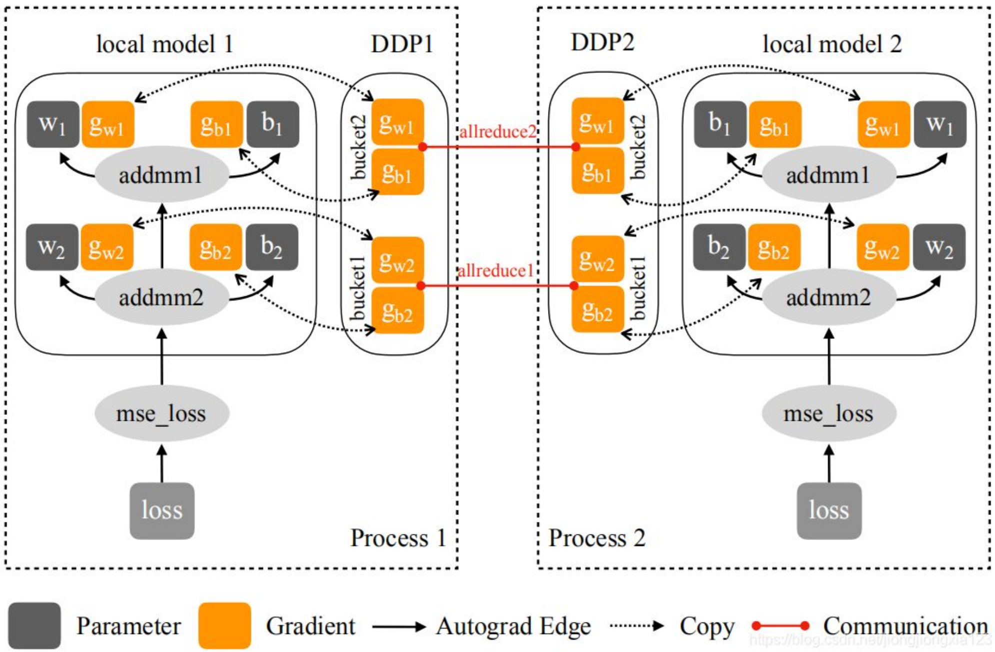
https://erickguan.me/2019/pytorch-parallel-model

Distributed PyTorch Training

- Data Parallel
  - Most simple form of parallelism with minimal code change
  - Downside: Slower form of parallelism - involves inter node communication 3x per training step

Distributed PyTorch Training

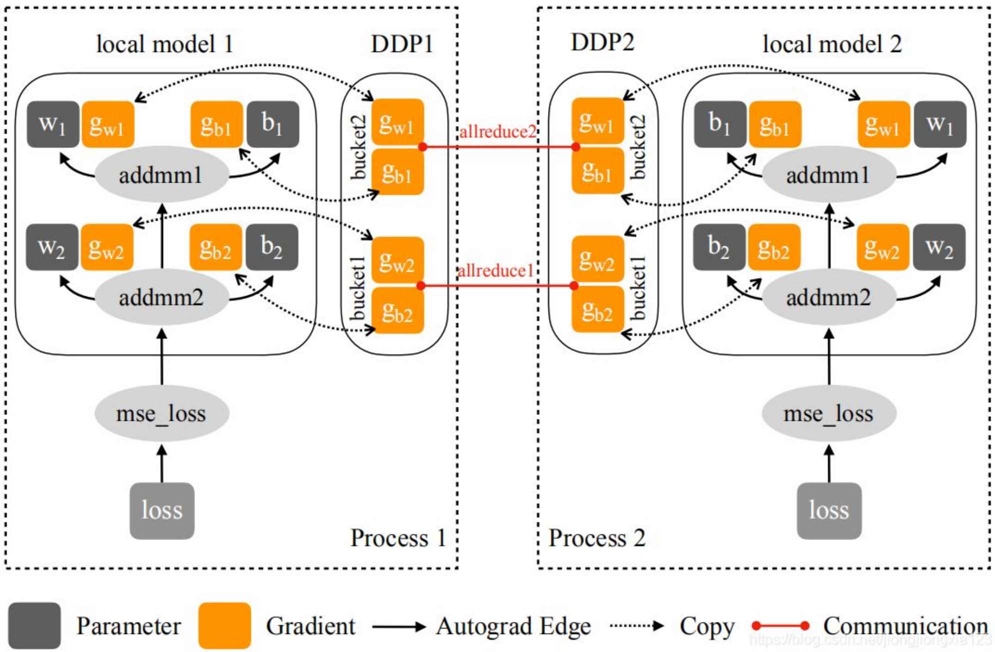
DistributedDataParallel: How it works?

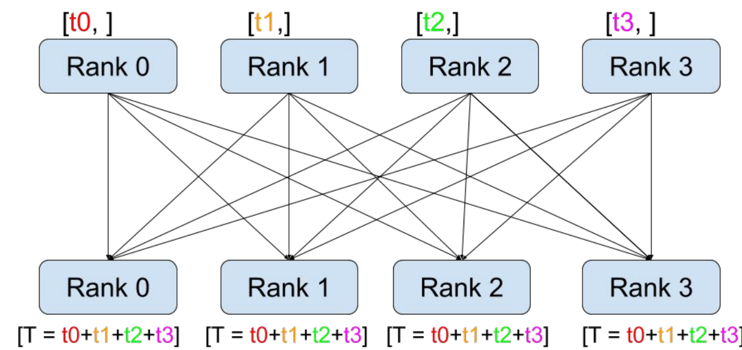


(Li et al., 2020)

Distributed PyTorch Training

- DistributedDataParallel

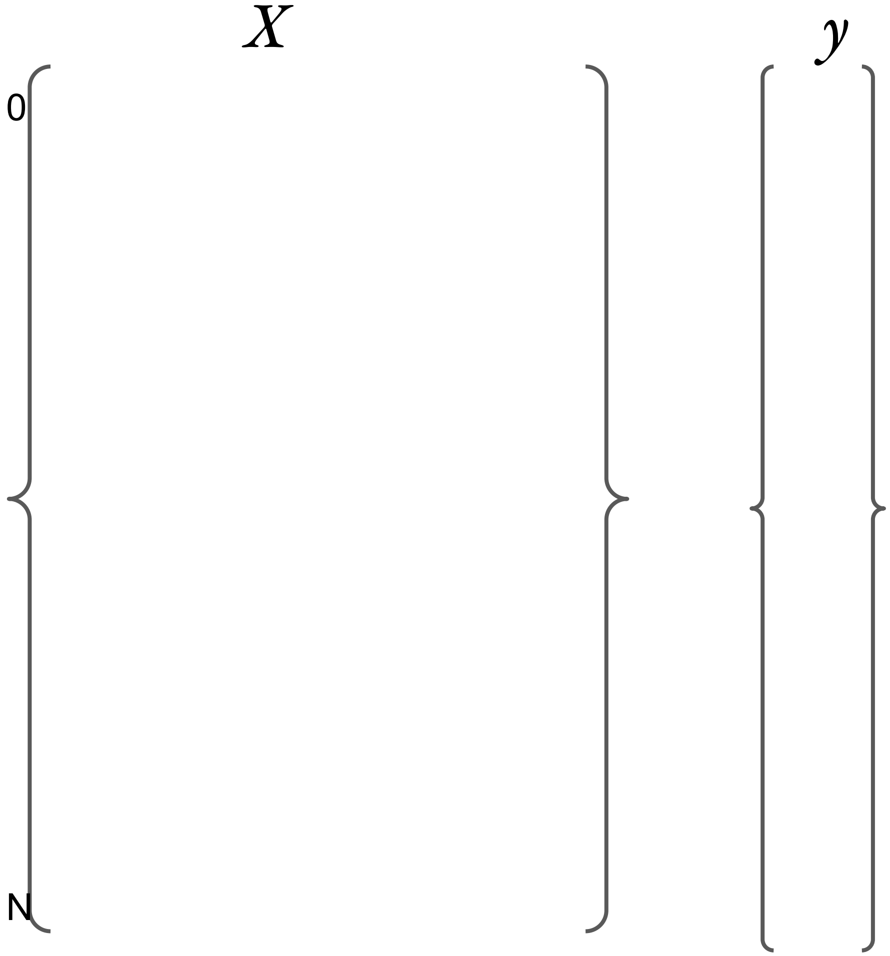




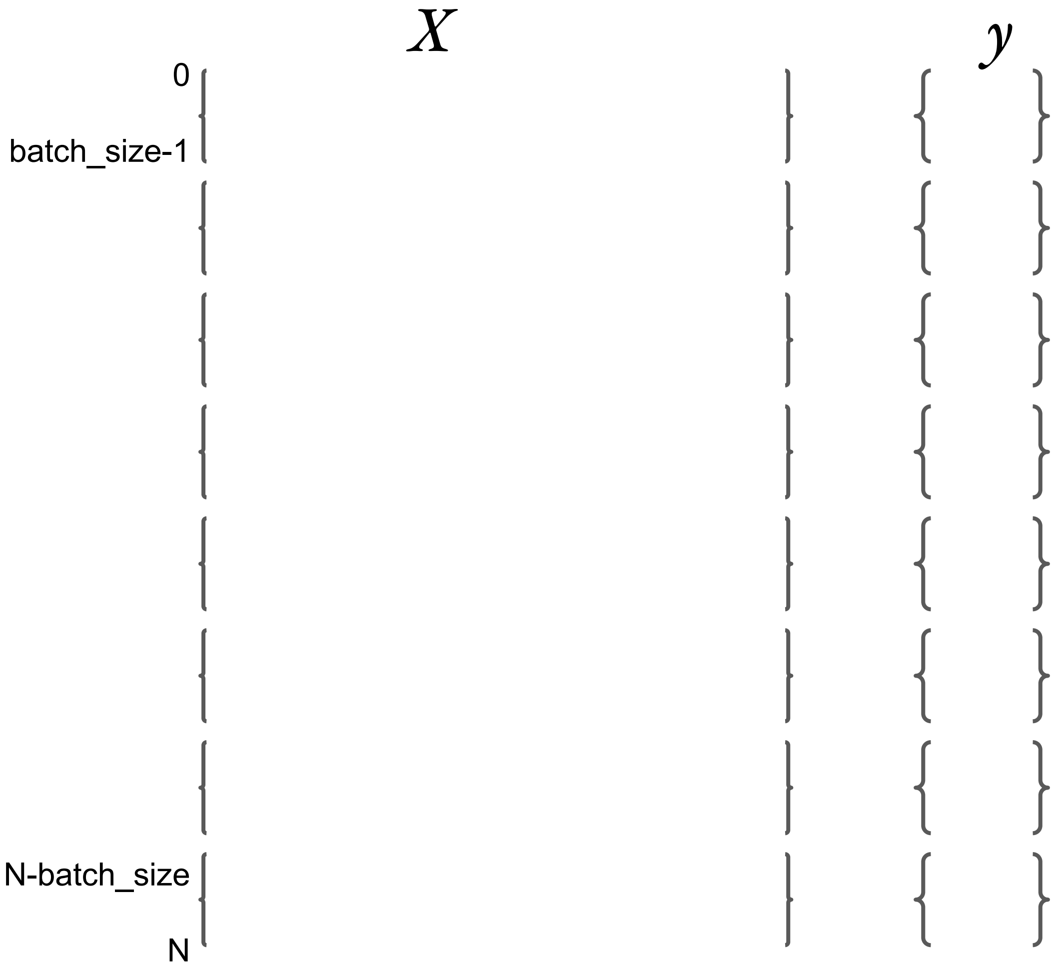
AllReduce

(Li et al., 2020)

Distributing Data

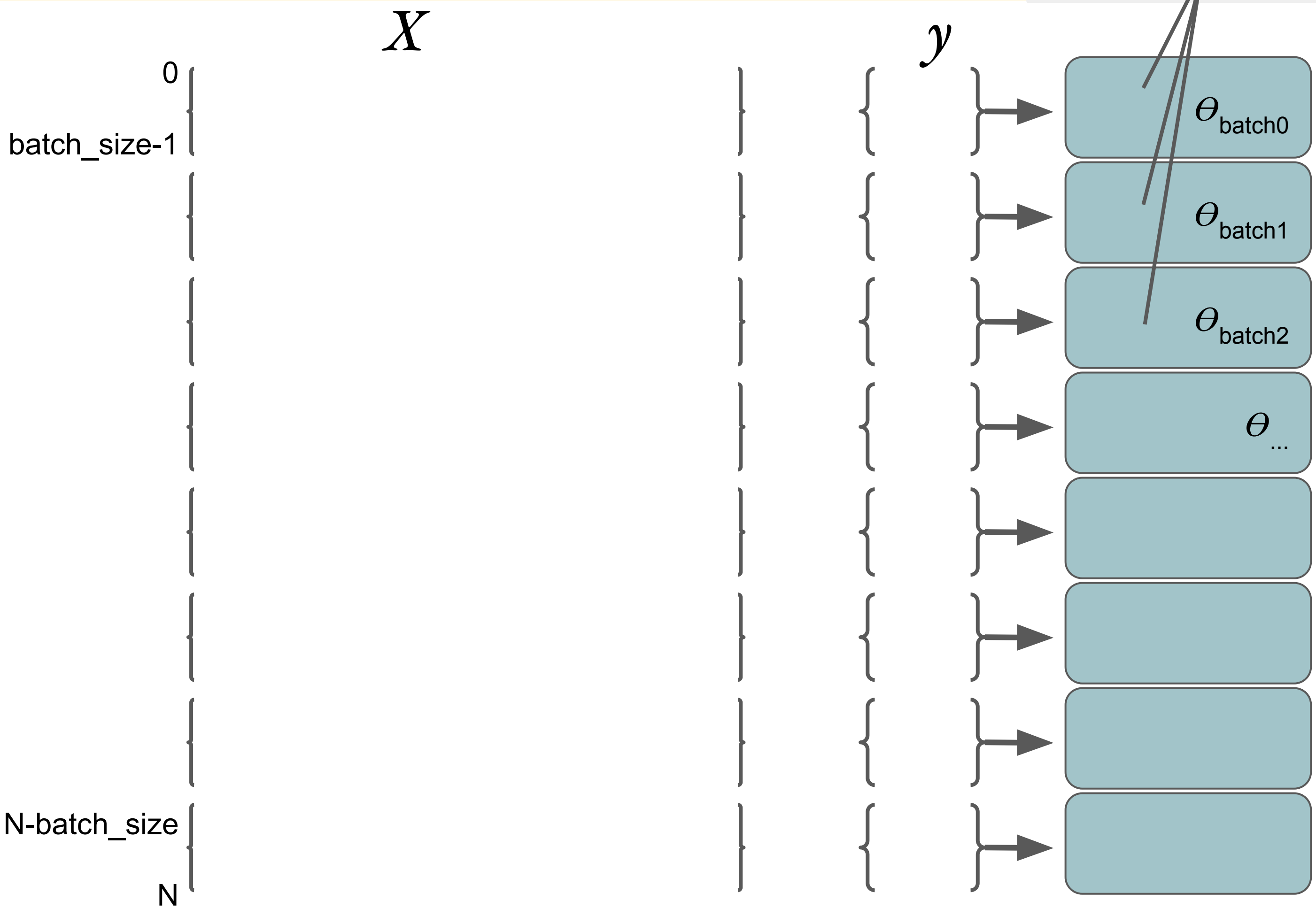


Distributing Data

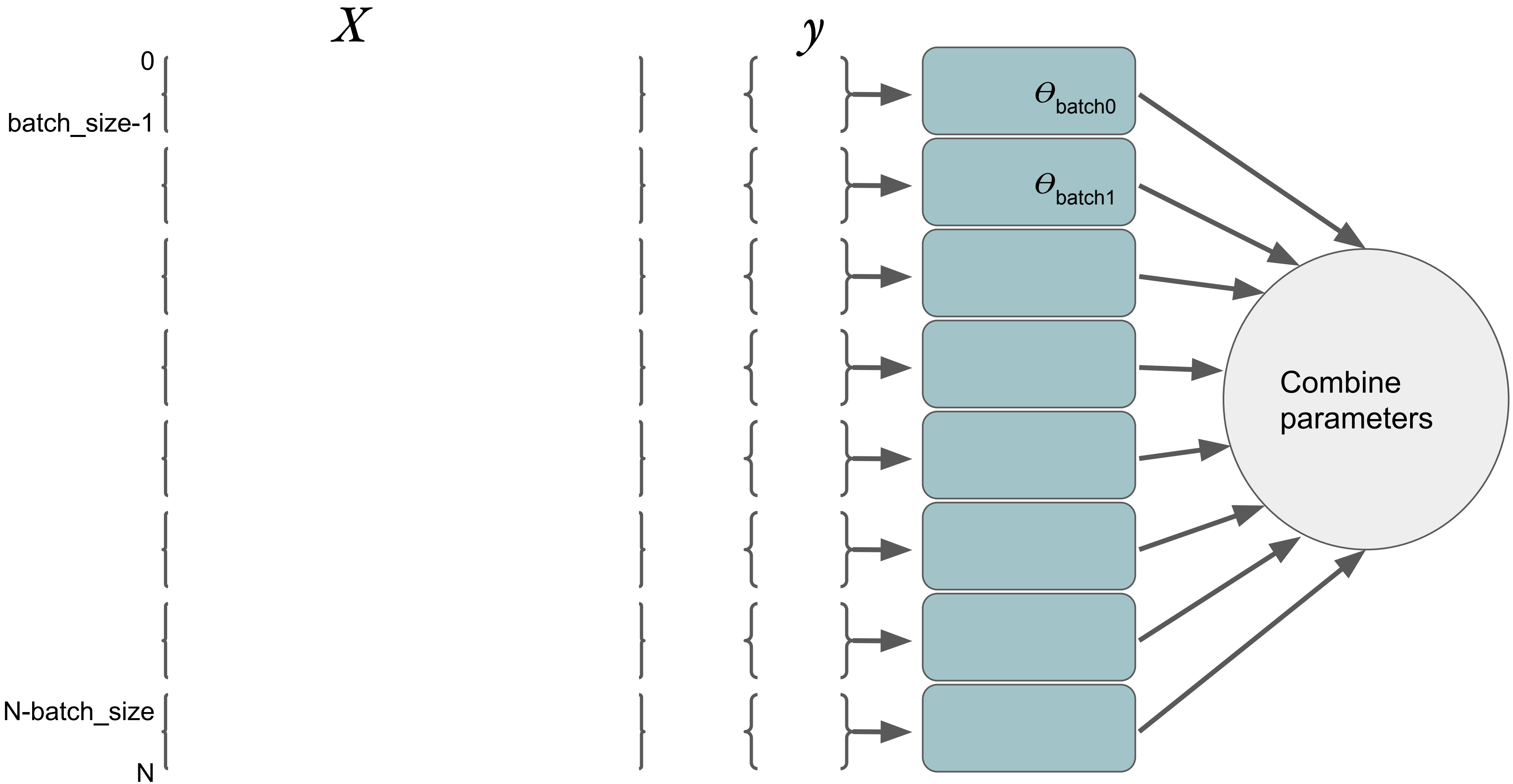


Distributing Data

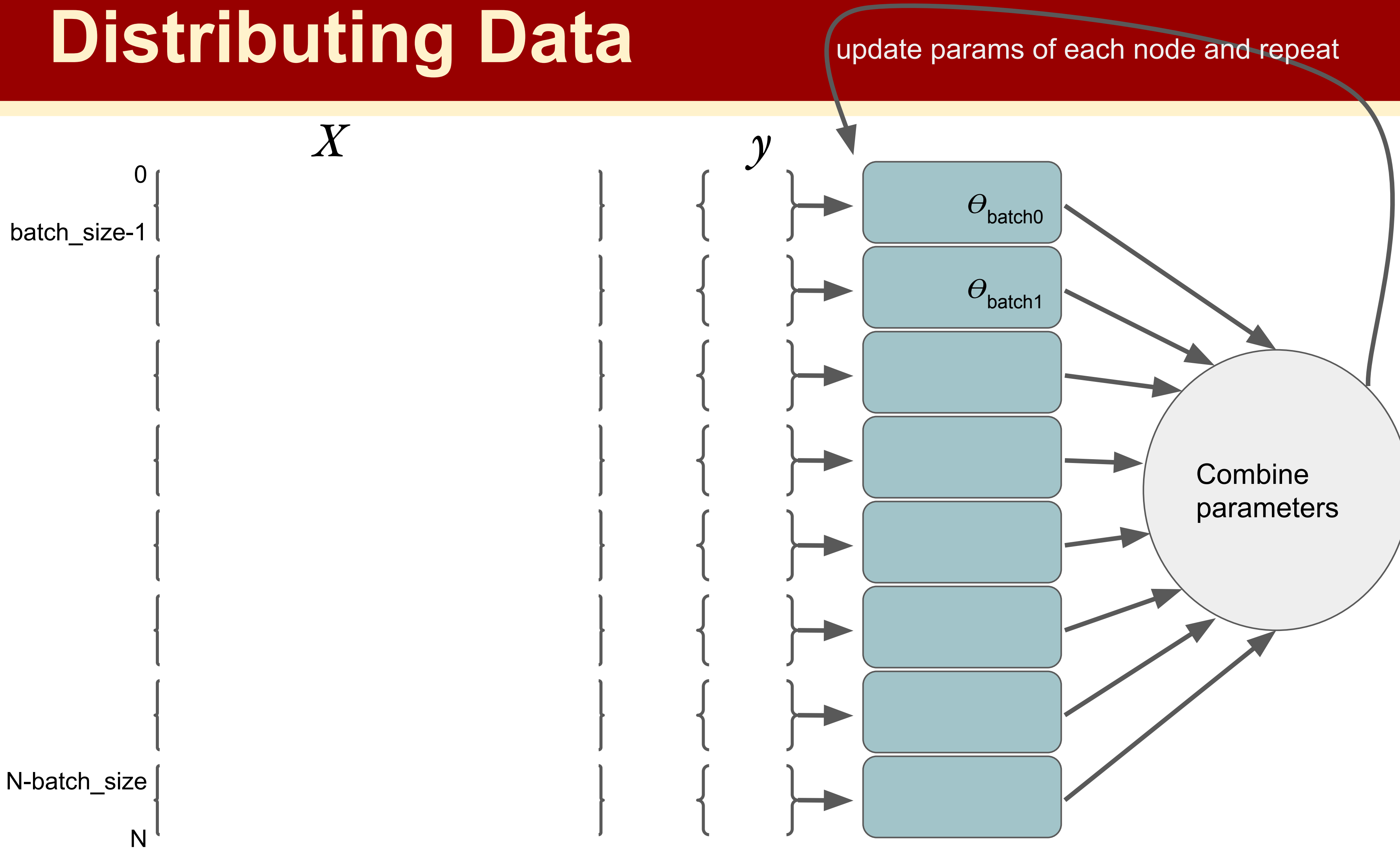
learn parameters (i.e. weights), given graph with cost function and *optimizer*



Distributing Data



Distributing Data



Distributed PyTorch Training

- DistributedDataParallel ([Li et al., 2020](#))
  - Efficient form of parallelism but involves a little extra code change*
  - Performs AllReduce on the computed gradients across all nodes and machines

** Extra code change if you are implementing using Pytorch. It has been made extremely simple by*

Distributed PyTorch Training

- DistributedDataParallel (Li et al., 2020)
  - Efficient form of parallelism but involves a little extra code change*
  - Performs AllReduce on the computed gradients across all nodes and machines
  - Downside: Python pickles all objects while spawning multiple processes (which happens in DDP). Code might crash if an object is not pickle-able

* *Extra code change if you are implementing using Pytorch. It has been made extremely simple by*

Options for distribution: *PyTorch*

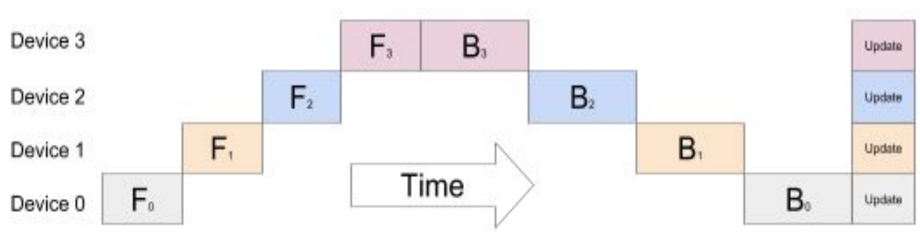
Model Parallelism: Distribute layer(s) of the model into different machines/GPUs to train a very large network.

Options for distribution: *PyTorch*

- Model Parallelism: Distribute layer(s) of the model into different machines/GPUs to train a very large network.
- Model Parallelism
  - Naive Model Parallelism
  - Pipelined Parallelism

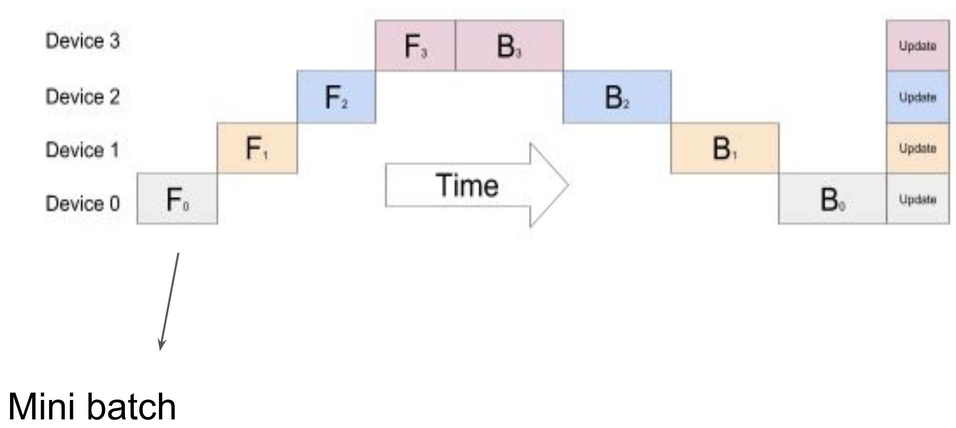
Options for distribution: *PyTorch*

- Naive Model Parallelism



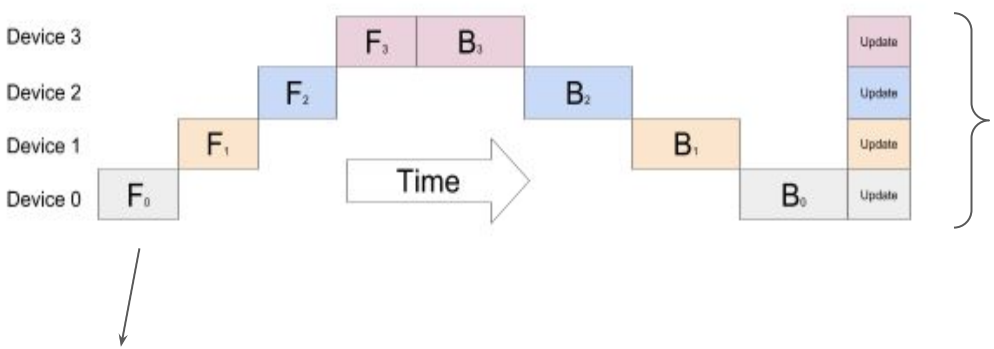
Options for distribution: *PyTorch*

- Naive Model Parallelism



Options for distribution: *PyTorch*

- Naive Model Parallelism



Synchronous Gradient updates at the end of a batch

Mini batch

Distributed PyTorch Training

- Naive Model Parallelism

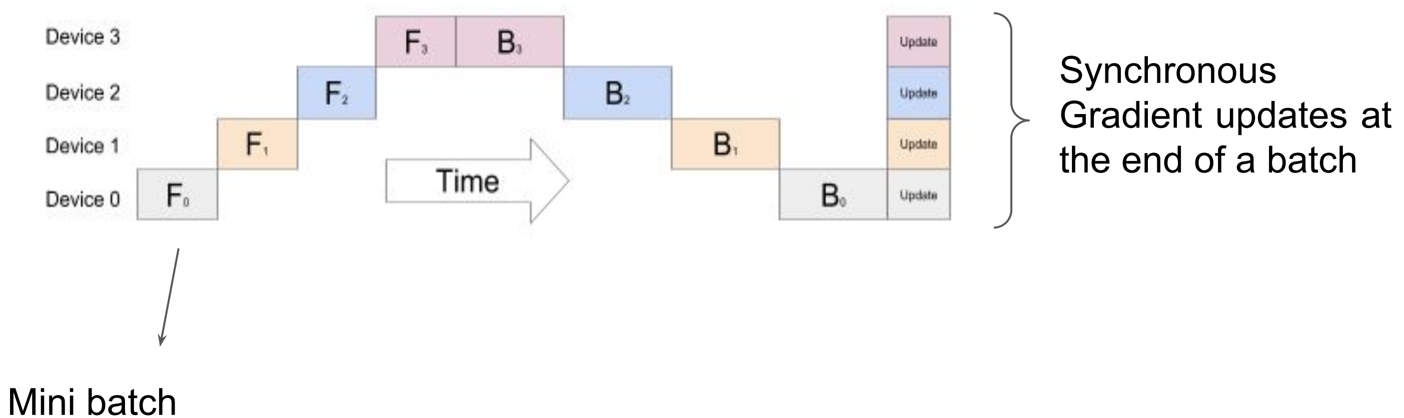

```
class ToyModel(nn.Module): #Pytorch: model_parallel_tutorial
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = torch.nn.Linear(10, 10).to('cuda:0')
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 5).to('cuda:1')

    def forward(self, x):
        x = self.relu(self.net1(x.to('cuda:0')))
        return self.net2(x.to('cuda:1'))
```

nous
updates at
of a batch

Distributed PyTorch Training

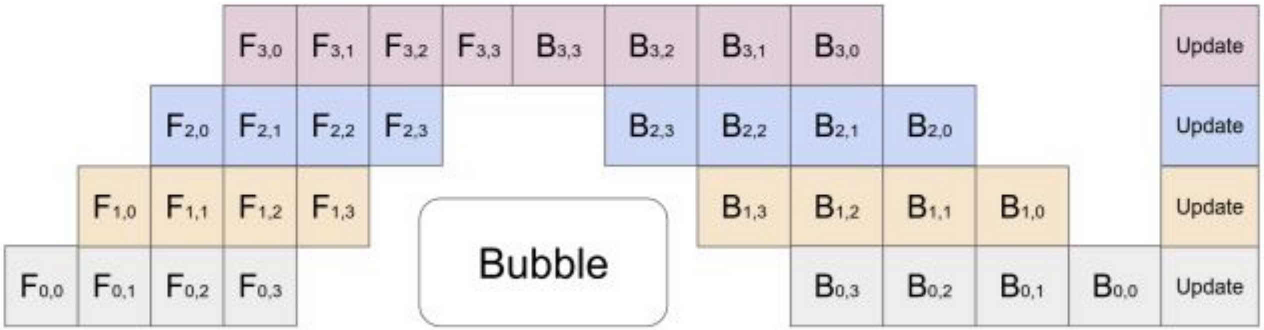
- Naive Model Parallelism



Severe under utilization of resources due to sequential dependency of the network

Distributed PyTorch Training

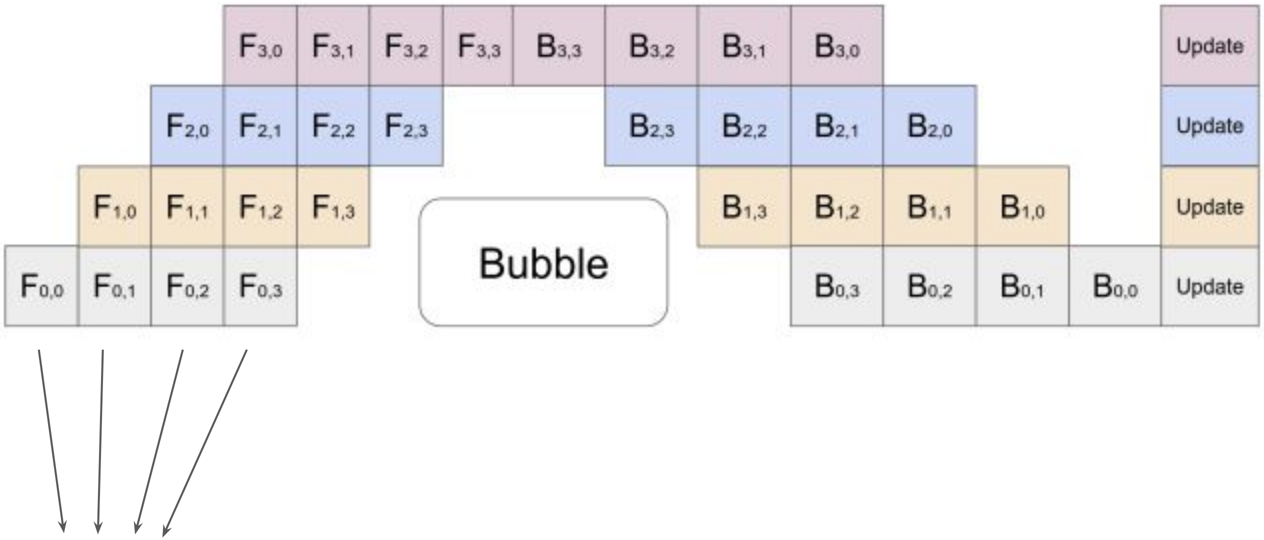
- Pipelined Parallelism



Huang et al., 2019

Distributed PyTorch Training

- Pipelined Parallelism

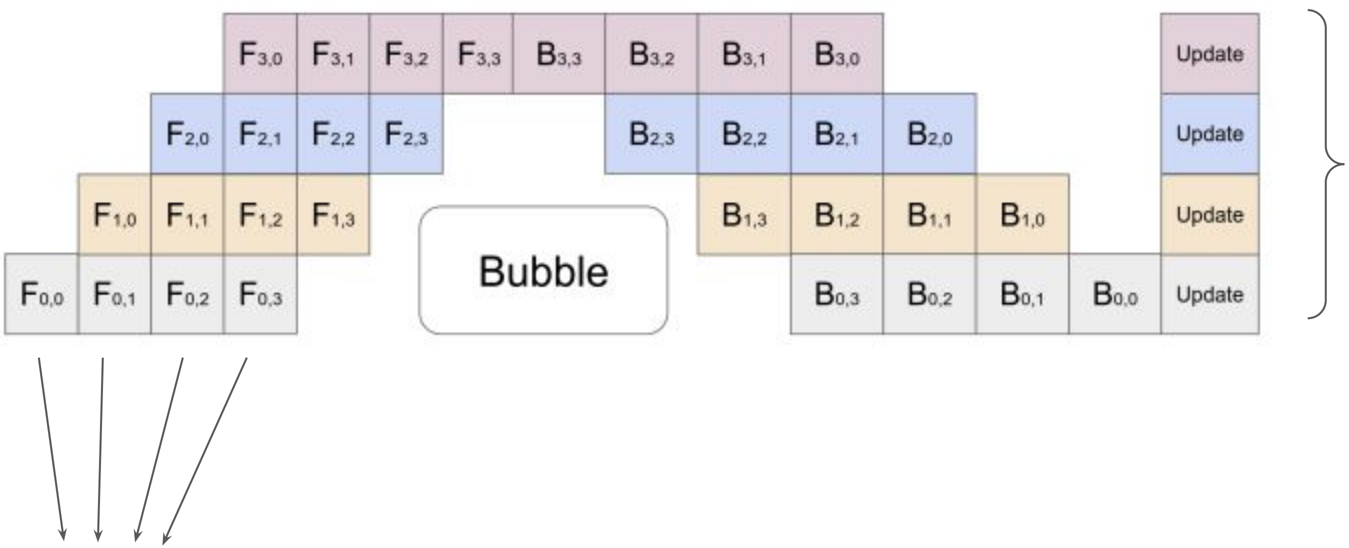


Mini batch split into
micro batches

[Huang et al., 2019](#)

Distributed PyTorch Training

- Pipelined Parallelism



Synchronous Gradient updates at the end of a batch

Mini batch split into micro batches

[Huang et al., 2019]

Distributed PyTorch Training

- Pipelined Parallelism

Synchronous Gradient updates at the end of a batch

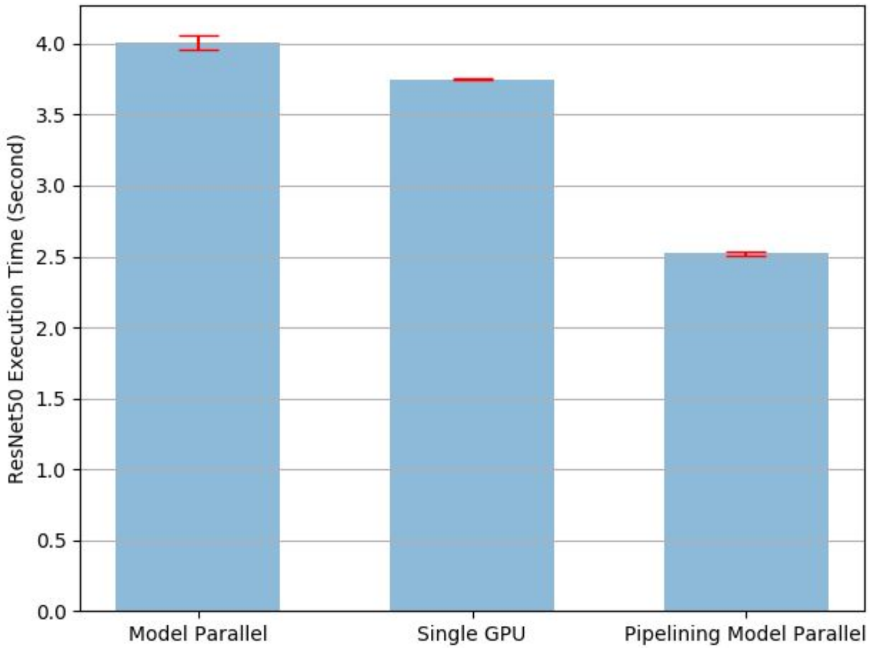
Mini batch split into micro batches

Provides high utilization of workers while ensuring reliable + stable training

[Huang et al., 2019]

Distributed PyTorch Training

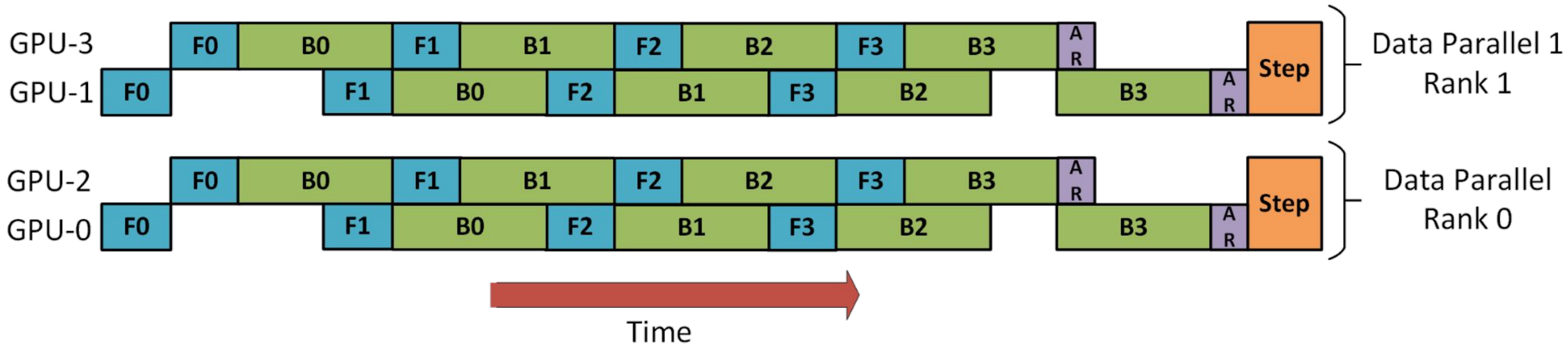
- Pipelined Parallelism



PyTorch: [Model Parallel best practices]

Distributed PyTorch Training

- Hybrid
  - DeepSpeed (Rasley et al., 2020)



Horovod: PyTorch 🤝 PySpark

Horovod is a distributed deep learning training framework.

Horovod helps scaling single GPU (worker) into multi-GPU or even multi-host training without no code change

Horovod on spark: "provides a convenient wrapper around Horovod that makes running distributed training jobs in Spark clusters easy"

Distributed Hardware:

- Locally: Across processors (cpus, gpus, tpus)
- Across a Cluster: Multiple machine with multiple processors

Parallelisms:

- Data Parallelism: All nodes doing same thing on different subsets of data
- Graph/Model Parallelism: Different portions of model on different devices

Model Updates:

- Asynchronous Parameter Server
- Synchronous AllReduce (doesn’t work with Model Parallelism)

Summary

- PyTorch is workflow system, where records are always tensors
  - *operations* applied to tensors
- Optimized for numerical / linear algebra
  - automatically finds gradients
  - specification of devices
- “Easily” distributes
  - Data Parallelism
  - Model Parallelism
  - Updating Parameters: AllReduce