CS6216 Advanced Topics in Machine Learning (Systems)

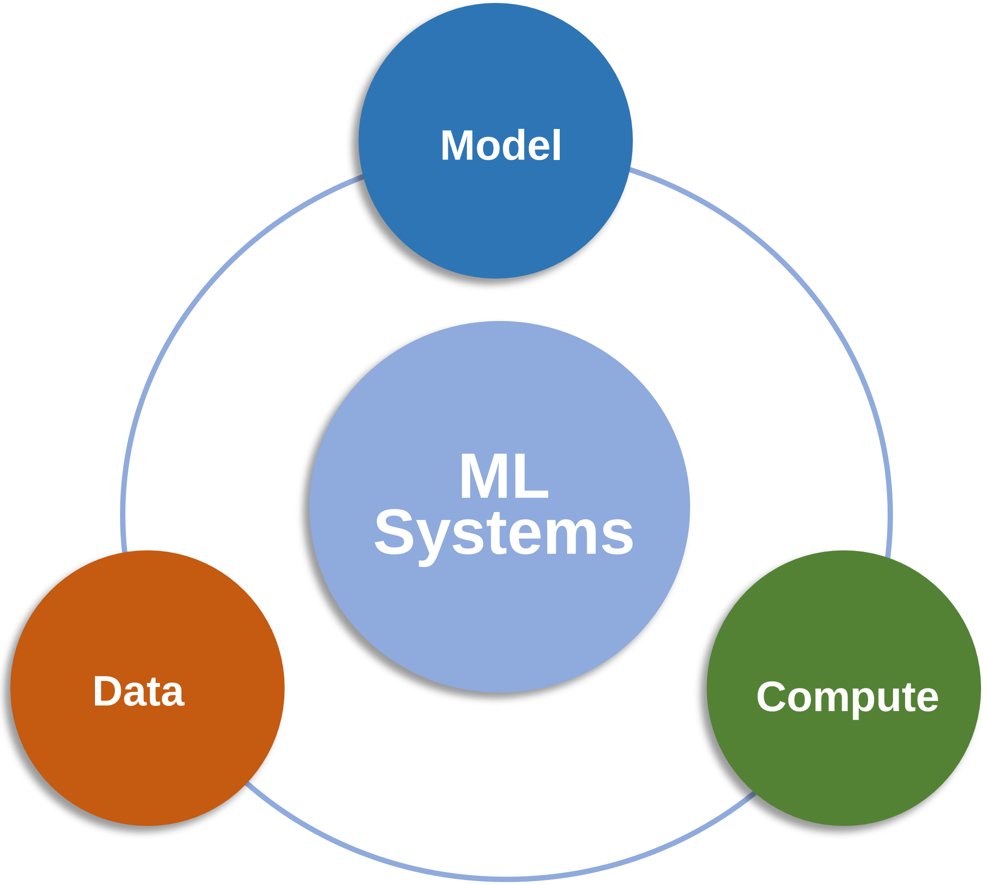
MLsys Foundations

Yao LU

21 Aug 2024

National University of Singapore
School of Computing

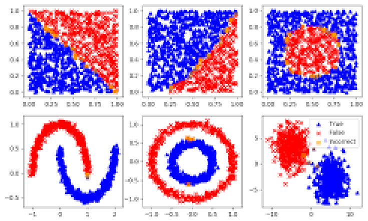
ML Systems Overview



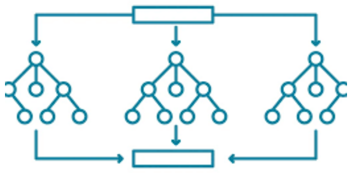
- Three components
- ML tasks
  - Training / tuning
  - Inference

Models

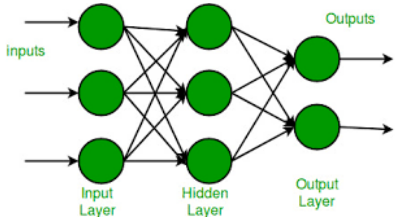
- What are models?



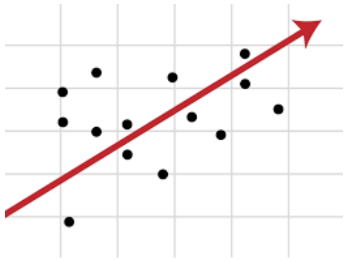
Clustering



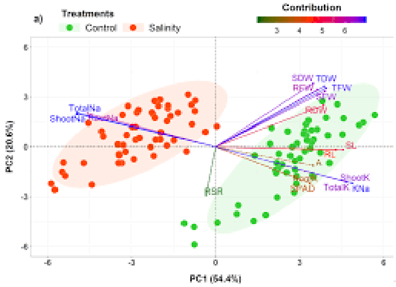
Random Forest



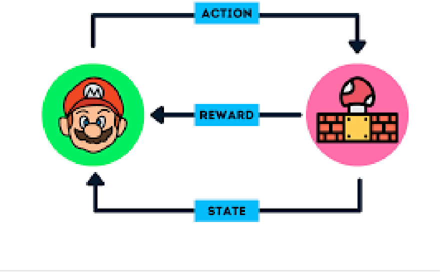
Perceptron



Linear regression



PCA

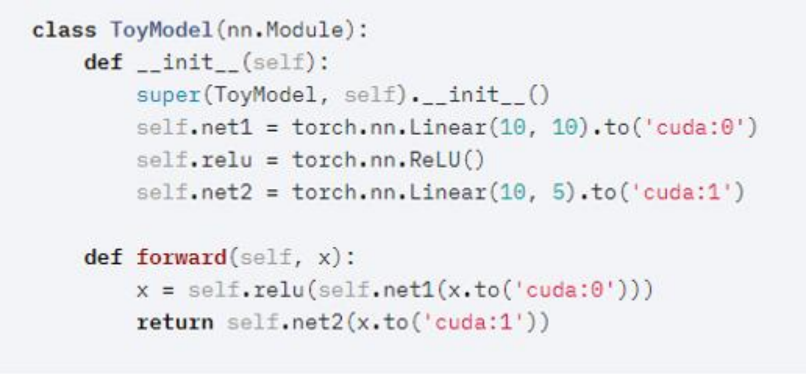


Reinforcement learning

- Models = algorithms?
- How to define, store & use models?

Model definitions

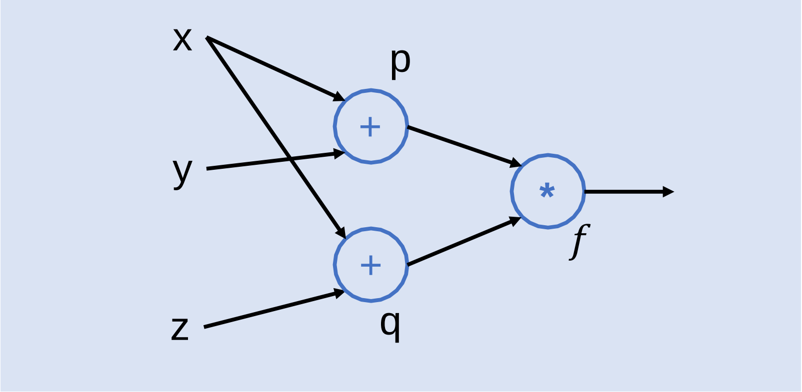
- PyTorch, Tensorflow, JAX etc. use functional declarations
  - Direct mapping to a compute graph, no ambiguity



```
class ToyModel(nn.Module):
    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'))
```

Model definition



(not matching code)

Algorithmic workflows

- Various ML systems exist for Boosting trees, Graph neural networks etc.

- This lecture focuses on Large Generative Models (LGMs)
  - Deep neural networks trained w/ Stochastic Gradient Descent (SGD)

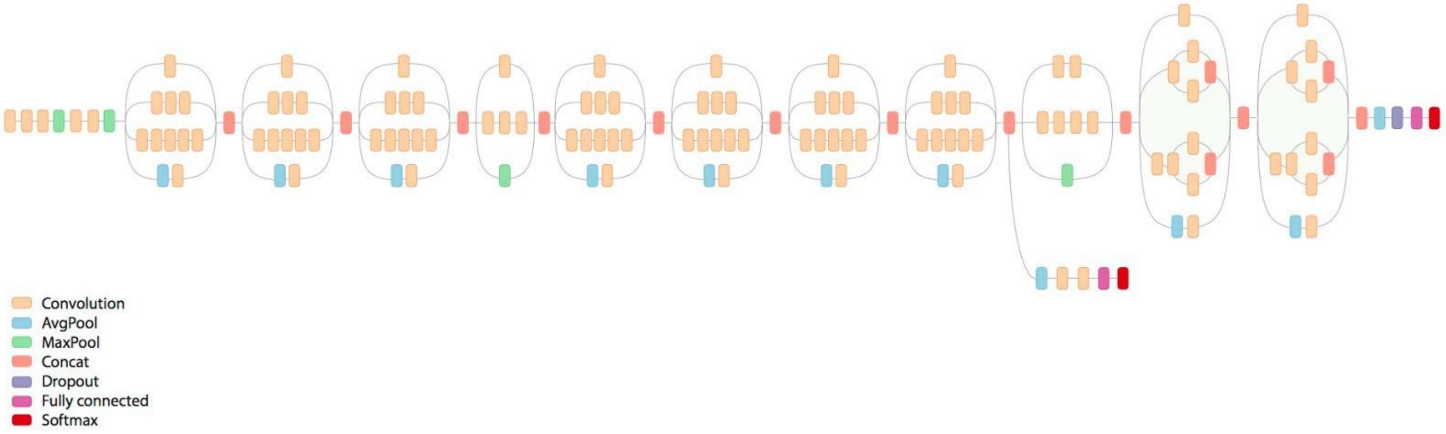
Algorithmic workflows: recap

Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights

Model inputs



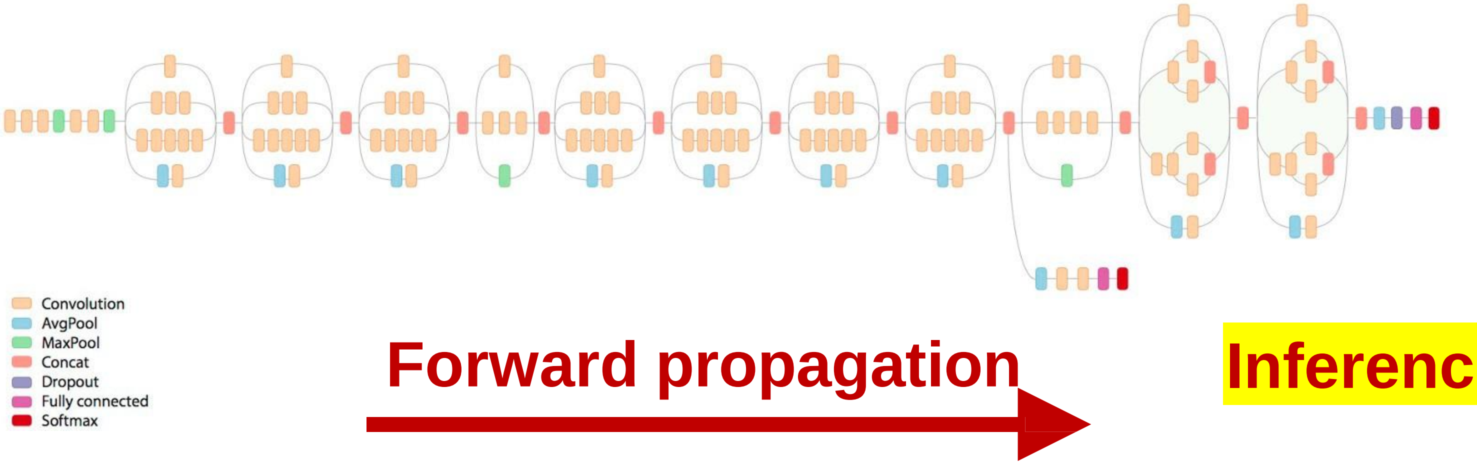
Model prediction

Algorithmic workflows: recap

Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights



Model inputs

Model prediction

Forward propagation

Inference stops here

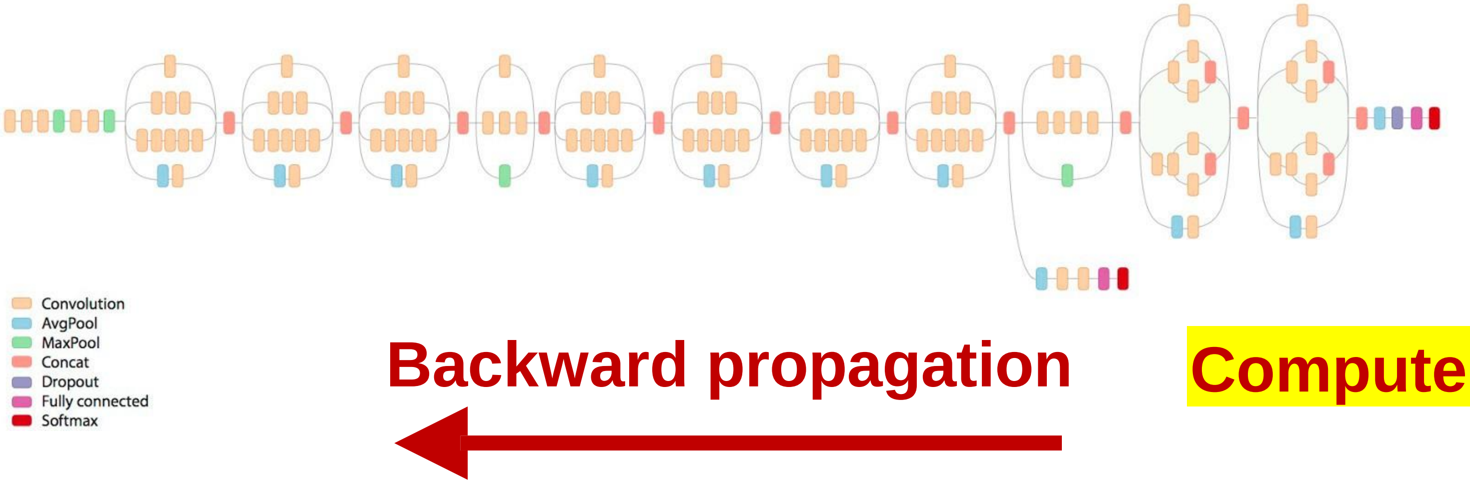
Algorithmic workflows: recap

Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights

Model inputs



Model prediction

Backward propagation

Compute loss/gradients

Algorithmic workflows: recap

Stochastic Gradient Descent (SGD)

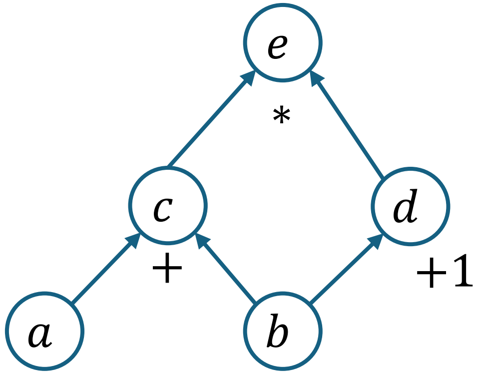
Train ML models through many iterations of 3 stages

1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

Back propagation by example

- $e = (a + b) \cdot (b + 1)$, compute the following:



$$\frac{\partial e}{\partial c} =$$

$$\frac{\partial e}{\partial d} =$$

$$\frac{\partial e}{\partial a} =$$

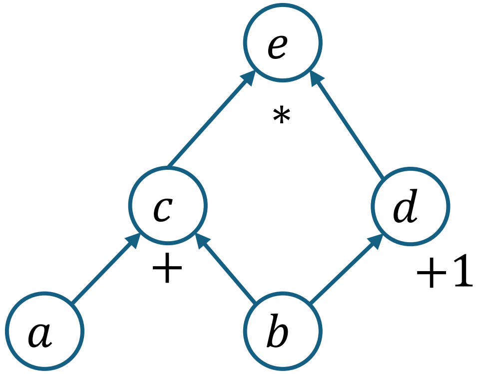
$$\frac{\partial e}{\partial b} =$$

Applying chain rule to compute gradient

- Back-tracking from the root to write down partial derivatives. + for branches, * for adjacent nodes

Back propagation by example

- $e = (a + b) \cdot (b + 1)$, compute the following :



$$\frac{\partial e}{\partial c} = \frac{\partial (c \cdot d)}{\partial c} = d$$

$$\frac{\partial e}{\partial d} = \frac{\partial (c \cdot d)}{\partial d} = c$$

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial a} = d \cdot 1 = d$$

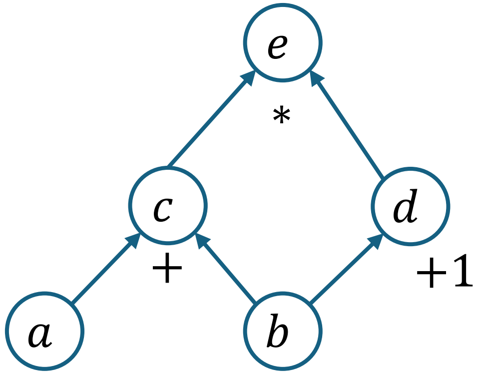
$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = c + d$$

Applying chain rule to compute gradient

- Back-tracking from the root to write down partial derivatives. + for branches, * for adjacent nodes

Back propagation by example

- $e = (a + b) \cdot (b + 1)$, compute the following :



$$\frac{\partial e}{\partial c} = \frac{\partial (c \cdot d)}{\partial c} = d$$

$$\frac{\partial e}{\partial d} = \frac{\partial (c \cdot d)}{\partial d} = c$$

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial a} = d \cdot 1 = d$$

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = c + d$$

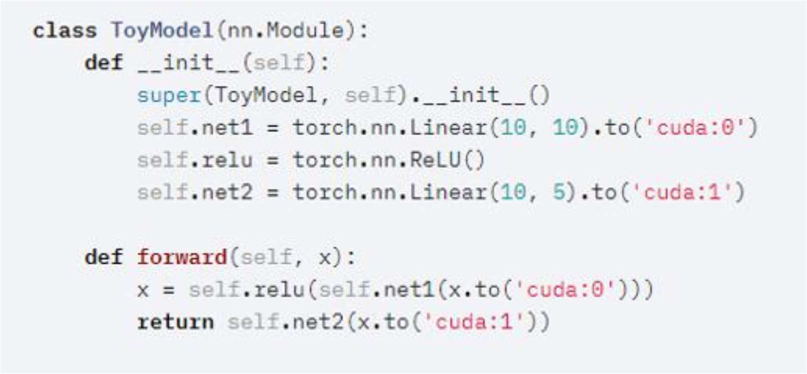
Applying chain rule to compute gradient

- Back-tracking from the root to write down partial derivatives. + for branches, * for adjacent nodes
- Given the actual Loss, compute gradient digits

A lot of repetitive compute

- Proper caching & reusing in the graph nodes

Building forward & backward compute graph

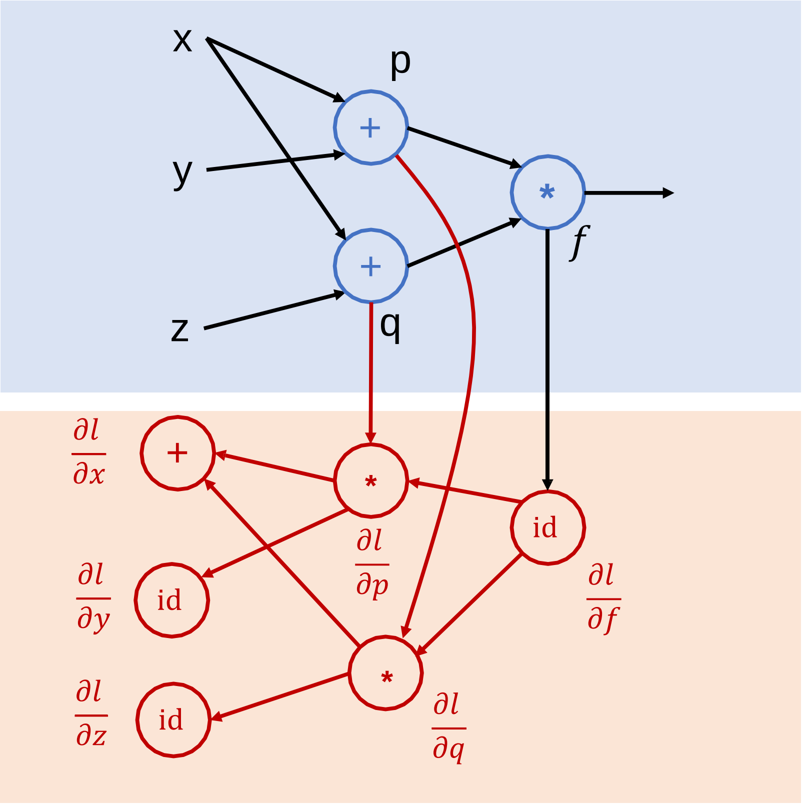


```
class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = torch.nn.Linear(10, 10).to('cuda:0')
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 5).to('cuda:1')

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

Model definition

Compute graph builder



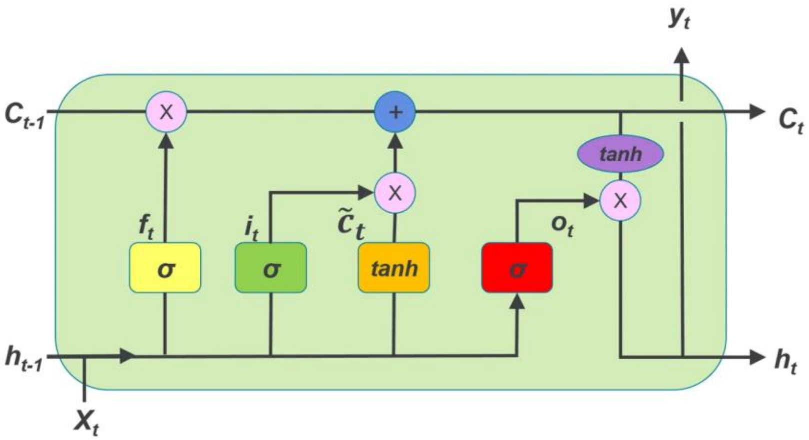
Forward computation graph

Backward computation graph

(not matching code)

Back propagation for LSTM

- Long-short term memory (LSTM)



$$f_t = \sigma_g\left(W_f \times x_t + U_f \times h_{t-1} + b_f\right)$$

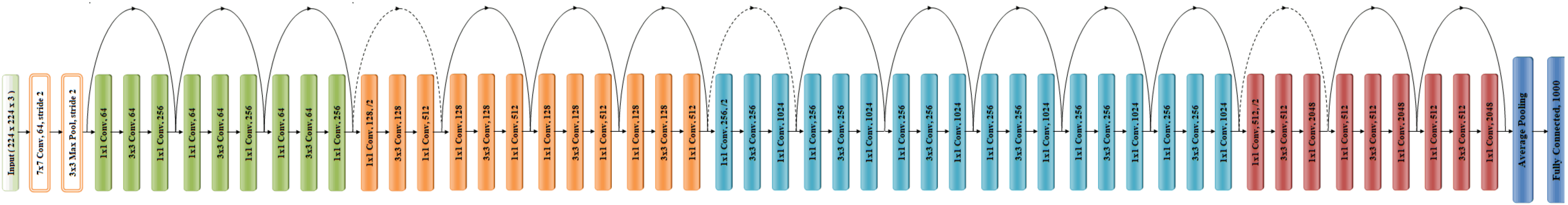
$$i_t = \sigma_g\left(W_i \times x_t + U_i \times h_{t-1} + b_i\right)$$
$$o_t = \sigma_g\left(W_o \times x_t + U_o \times h_{t-1} + b_o\right)$$
$$c'_t = \sigma_c\left(W_c \times x_t + U_c \times h_{t-1} + b_c\right)$$
$$c_t = f_t \cdot c_{t-1} + i_t \cdot c'_t$$
$$h_t = o_t \cdot \sigma_c(c_t)$$

- Derive the back-prop formulations for all parameters

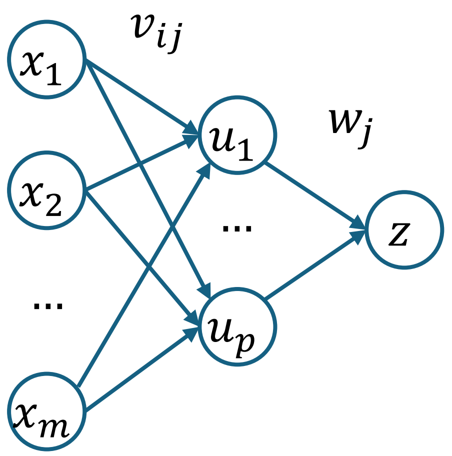
- Instructor’s experience 10 years back:
  - 1 full page of equations, 30~40 steps
  - Implementing on GPU, extremely hard to debug

How about very large neural networks?

- We need
  - Automatic computation of gradients
  - Optimization with proper caching and compute node reuse



Quiz : back propagation for MLP



- MLP is a simple DNN, where a single perceptron is defined as:

$$y = \sigma(W \cdot x + b)$$

- A 2-layer perceptron for univariate regression with l_2 loss:

$$z = \sigma(W \cdot u + b)$$
$$u = \sigma(V \cdot x + b)$$
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

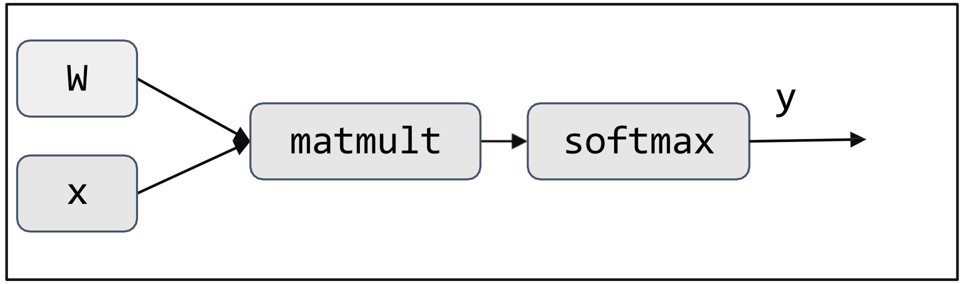
hint: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

derive gradients for W and V.

Computational graph construction by step

Construct the compute graph for $y = \text{softmax}(W \cdot x)$ with cross entropy loss

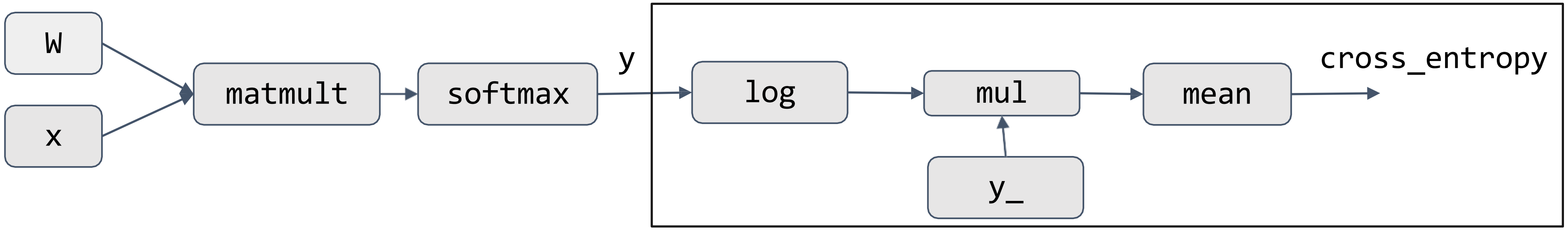
1. Construct forward graph



Computational graph construction by step

Construct the compute graph for $y = \text{softmax}(W \cdot x)$ with cross entropy loss

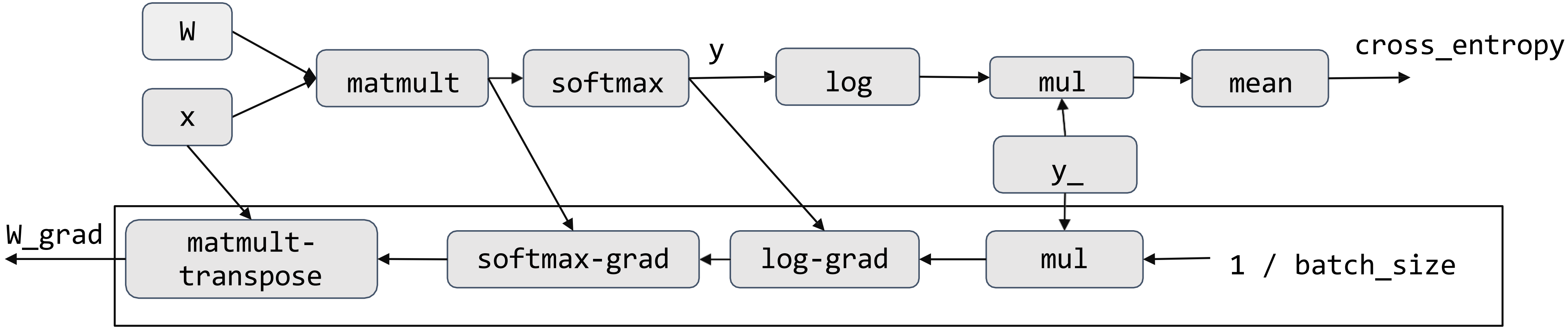
1. Construct forward graph
2. Add loss compute nodes



Computational graph construction by step

Construct the compute graph for $y = \mathrm{softmax}(W \cdot x)$ with cross entropy loss

1. Construct forward graph
2. Add loss compute nodes

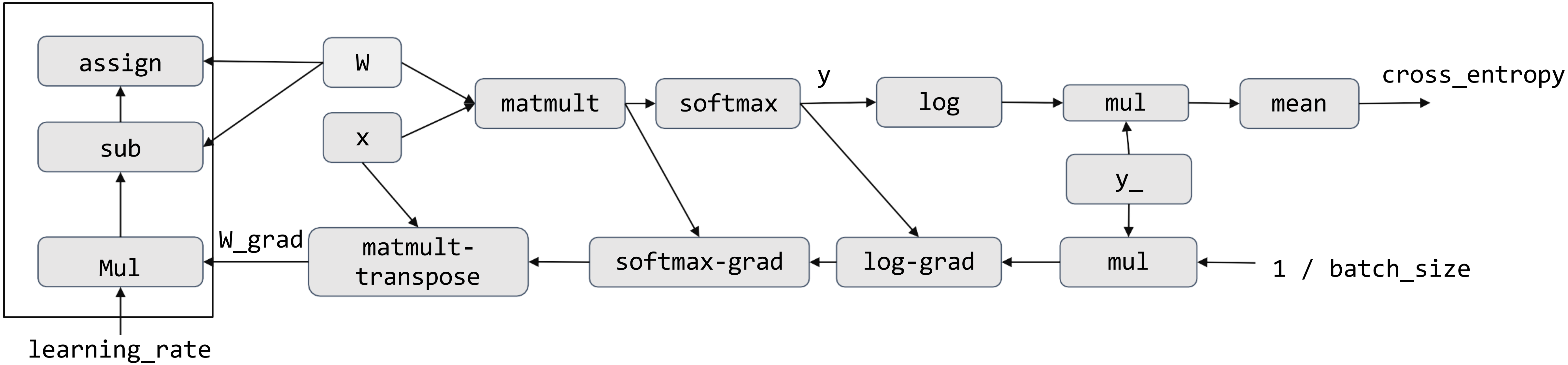


3. Construct backward graph by automatic differentiation More details in the next lecture

Computational graph construction by step

Construct the compute graph for $y = \mathrm{softmax}(W \cdot x)$ with cross entropy loss

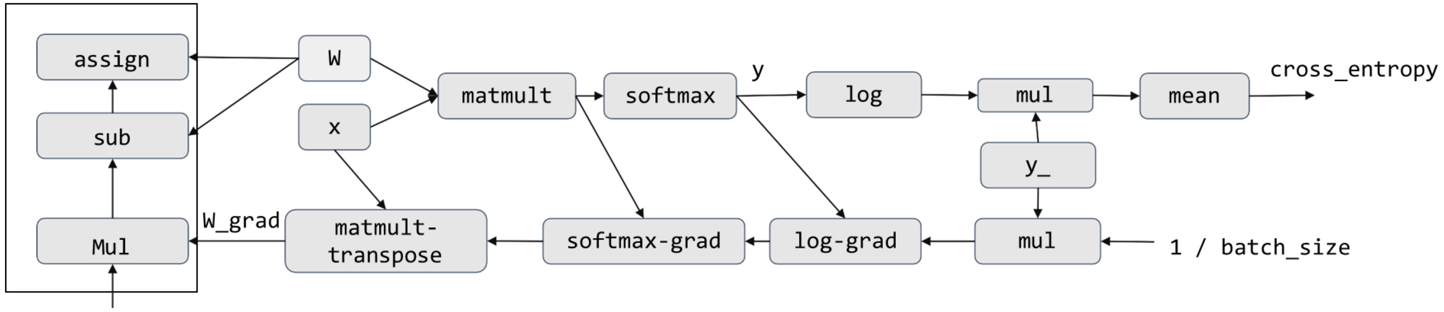
1. Construct forward graph
2. Add loss compute nodes



3. Construct backward graph by automatic differentiation
4. Update model weights

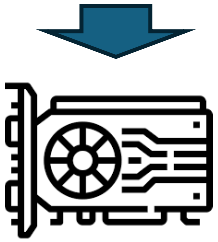
Mapping compute graph to actual runtime

- Key factors to consider:
  - Graph dependency
  - Parallelism & batching
  - Driver & API



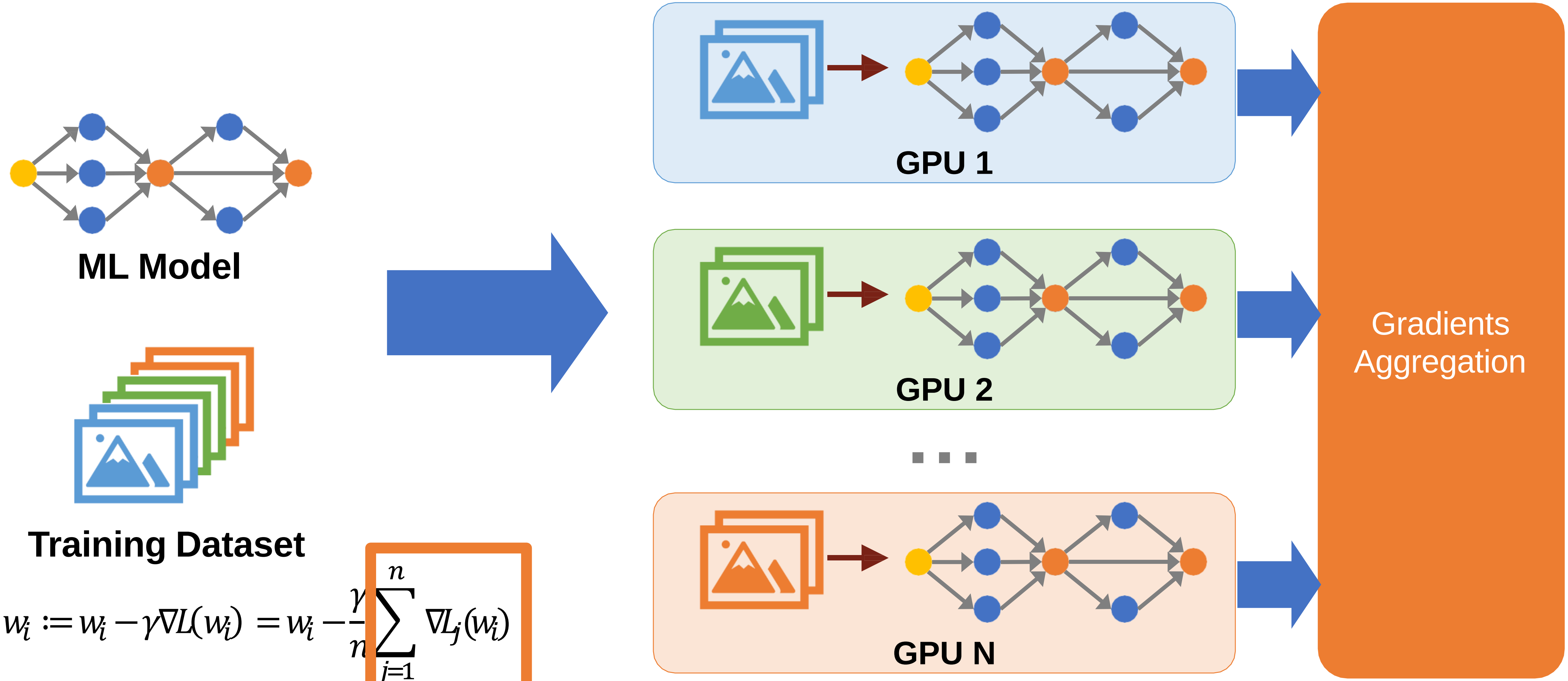
assign
sub
Mul
W
x
W_grad
matmult
softmax
y
log
mul
mean
cross_entropy
y_
mul
1 / batch_size
matmult-transpose
softmax-grad
log-grad

- CPU, GPU, TPU, FPGA, etc.
  - Each architecture has corresponding libraries and APIs



- Optimizations:
  - Operator code-gen and fusion
  - Graph-level optimizations

Execution of the compute graph: data parallelism



$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \boxed{\sum_{j=1}^{n} \nabla L_j(w_i)}$$

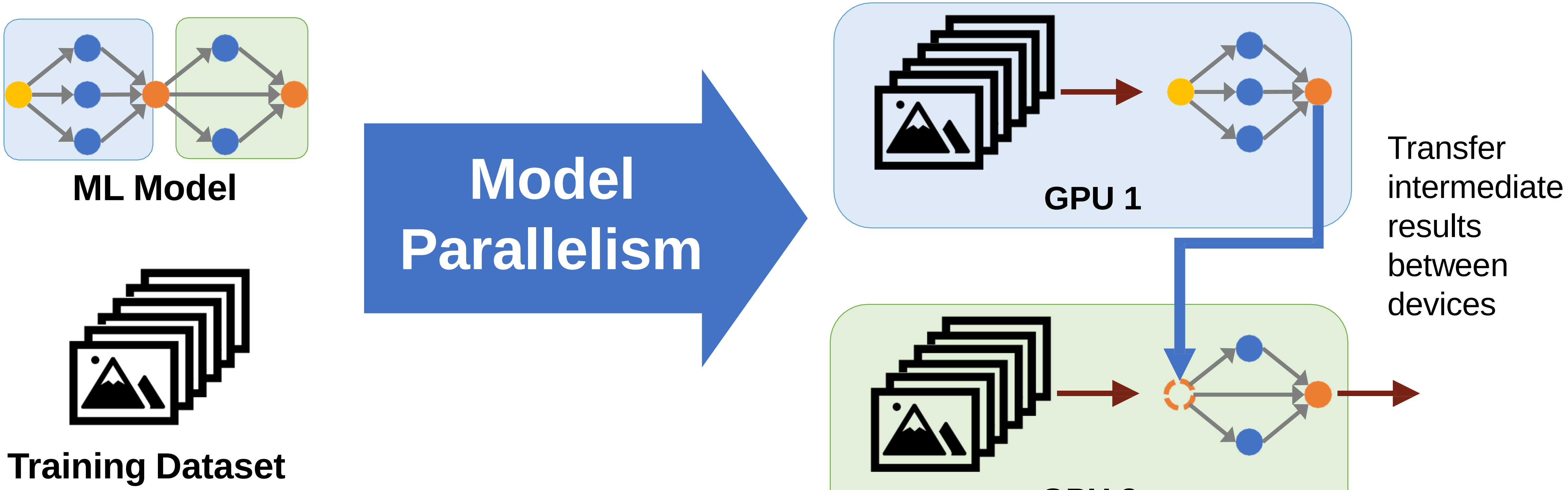
1. Partition training data into batches

2. Compute the gradients of each batch on a GPU

3. Aggregate gradients across GPUs

Execution of the compute graph: model parallelism

- Split a model into multiple subgraphs and assign them to different devices



ML Model

Model Parallelism

GPU 1

Transfer intermediate results between devices

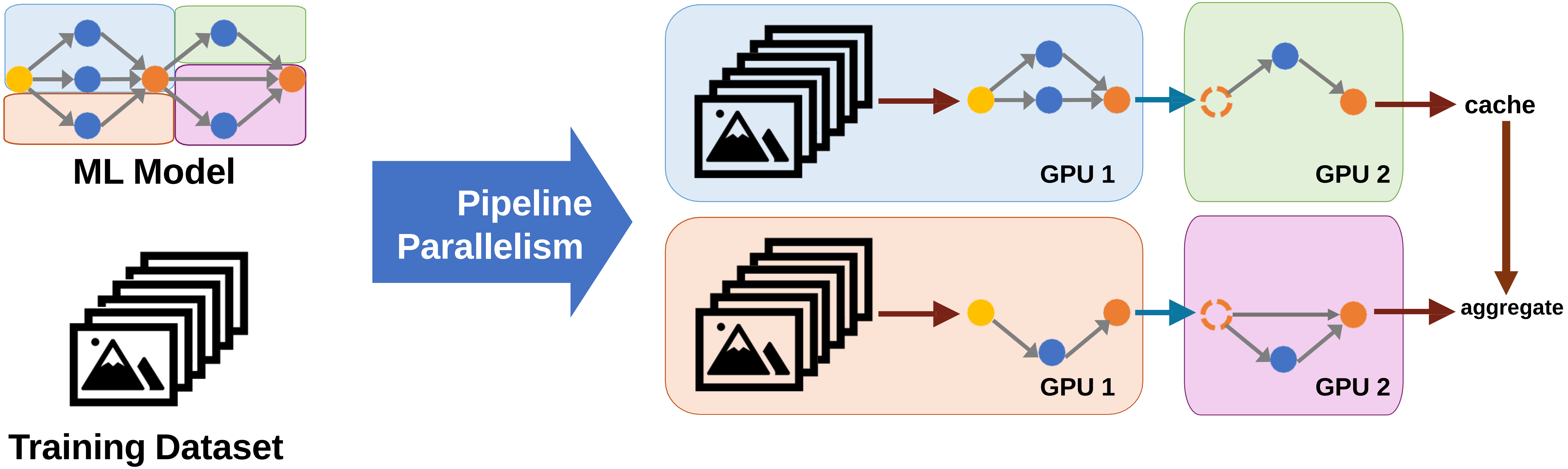
GPU 2

Training Dataset

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n}\sum_{j=1}^{n} \nabla L_j(w_i)$$

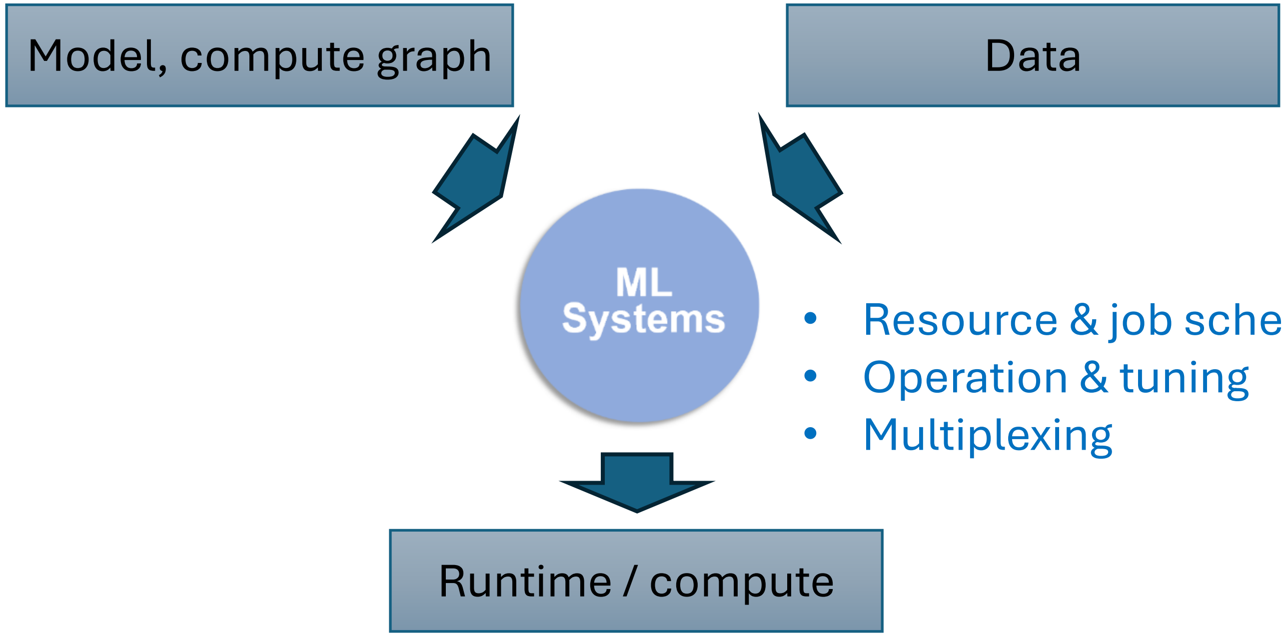
Execution of the compute graph: pipeline parallelism

- Split a model into multiple subgraphs and assign them to different devices. Run them by proper scheduling.



$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n}\sum_{j=1}^{n} \nabla L_j(w_i)$$

Summary: core modules in MLsys



- Graph optimization
- Model specific technologies

Model, compute graph

- Storage & caching
- Data preparation & quality

Data

ML Systems

- Resource & job scheduling
- Operation & tuning
- Multiplexing

Runtime / compute

- Kernel optimization
- Code generation
- New hardware

- R&D optimizes for
  - Training / Tuning:

    efficiency & scalability

  - Inference / Serving:

    latency & throughput

  - Cloud efficiency

Outlook of the course content

- Upcoming lectures
  - Task/Model specific technologies
  - LLM serving
  - Scale up and out
  - AI for systems

- Overview of Homework 2-4
  - HW2: back propagation and autograd
  - HW3: framework & LLM inference
  - HW4: LLM serving

Reading list for the next lecture

- How to read a paper

- TensorFlow: A System for Large-Scale Machine Learning
  OSDI 2016

- Questions will be posted on Canvas

No in-person lecture next week

- Lecture recordings will be provided.