# PyTorch

## And Some Things About It

Viet Nhat Nguyen - 21520378

Summer Training
University of Information Technology

Jul. 12, 2023

# Summary

# Warm-up: numpy

# Warm-up: Numpy

We can implement the network using numpy, but

- it does not know anything about computation graphs, or deep learning, or gradients.
- start from scratch, or manually!

# Warm-up: Numpy

We can implement the network using numpy, but

- it does not know anything about computation graphs, or deep learning, or gradients.
- start from scratch, or manually!

# Warm-up: Numpy

We can implement the network using numpy, but

- it does not know anything about computation graphs, or deep learning, or gradients.
- start from scratch, or manually!

# PyTorch

PyTorch is a python package that provides two high-level features:

- An n-dimensional Tensor, similar to numpy
- Automatic differentiation for building and training neural networks

# PyTorch

PyTorch is a python package that provides two high-level features:

- An n-dimensional Tensor, similar to numpy but **can run on GPUs**
- Automatic differentiation for building and training neural networks

# Why PyTorch?

- More Pythonic (imperative)
  - Flexible
  - Intuitive and cleaner code
  - Easy to debug
- More Neural Networkic
  - Write code as the network works
  - forward/backward

# Install PyTorch

| | | | | |
|---|---|---|---|---|
| PyTorch Build | Stable (2.0.1) | | Preview (Nightly) | |
| Your OS | Linux | Mac | Windows | |
| Package | Conda | Pip | LibTorch | Source |
| Language | Python | | C++ / Java | |
| Compute Platform | CUDA 11.7 | CUDA 11.8 | ~~ROCm 5.4.2~~ | CPU |
| Run this Command: | `pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu117` | | | |

# Creating Tensors

# Creating Tensors

So many ways to create a Tensor:

- torch.empty()
- torch.zeros()
- torch.ones()
- torch.rand()
- torch.tensor()

```
import torch
my_tensor = torch.tensor([2, 10, 23])
```

# Tensor Data Types

# Tensor Data Types

`dtype` argument:

- `torch.bool`
- `torch.int8`
- `torch.uint8`
- `torch.int16`
- `torch.int32`
- `torch.int64`
- `torch.half`
- `torch.float`
- `torch.double`
- `torch.bfloat`

# Tensor Shapes

# Tensor Shapes

```
my_tensor.shape
```

# Tensor Shapes

Sometimes, tensors need to be of the same *shape* - that is, having the same number of dimensions and the same number of cells in each dimensions.

- We have `torch.*_like()` methods

# Tensor Shapes

Sometimes, tensors need to be of the same *shape* - that is, having the same number of dimensions and the same number of cells in each dimensions.

- We have `torch.*_like()` methods

# Math & Logic with PyTorch Tensors

# Math & Logic with PyTorch Tensors

- Basic arithmetic: $+ \; - \; * \; / \; \ldots$
  - Tensor with Scalar
  - Tensor with Tensor

As we can see, it is element-wise

# Math & Logic with PyTorch Tensors

- Basic arithmetic: $+ - * / \ldots$
  - Tensor with Scalar
  - Tensor with Tensor

As we can see, it is element-wise

# Math & Logic with PyTorch Tensors

- An **element-wise** operation operates on corresponding elements between tensors.

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{2,2} \end{bmatrix} \diamond \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1} \diamond b_{1,1} & a_{1,2} \diamond b_{1,2} \\ a_{2,1} \diamond b_{2,1} & a_{2,2} \diamond b_{2,2} \\ a_{3,1} \diamond b_{3,1} & a_{3,2} \diamond b_{3,2} \end{bmatrix}$$

- But, what happens when we try to perform operations on tensors with dissimilar shape?

# Math & Logic with PyTorch Tensors

- An **element-wise** operation operates on corresponding elements between tensors.

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{2,2} \end{bmatrix} \diamond \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1} \diamond b_{1,1} & a_{1,2} \diamond b_{1,2} \\ a_{2,1} \diamond b_{2,1} & a_{2,2} \diamond b_{2,2} \\ a_{3,1} \diamond b_{3,1} & a_{3,2} \diamond b_{3,2} \end{bmatrix}$$

- But, what happens when we try to perform operations on tensors with dissimilar shape?

# Math & Logic with PyTorch Tensors

- Basic arithmetic: $+ \; - \; * \; / \; ...$
  - *Tensor with Scalar*
  - *Tensor with Tensor (same-shape)*
  - **Tensor with Tensor (not same-shape)***

# Math & Logic with PyTorch Tensors

The exception to the same-shape rule is *tensor broadcasting*

# Math & Logic with PyTorch Tensors

- **Broadcasting** is a way to perform an operation between tensors that have similarities in their shapes.
  - multiplying a tensor of learning weights by a *batch* of input tensors
- Broadcasting's rules:
  - Each tensor must have at least one dimension - no empty tensors.
  - Comparing the dimension sizes of the two tensors, *backward*:
    - Each dimension must be equal, *or*
    - One of the dimensions must be of size 1, *or*
    - The dimension does not exist in one of the tensor.
  - Tensors of the identical shape.

# Math & Logic with PyTorch Tensors

- **Broadcasting** is a way to perform an operation between tensors that have similarities in their shapes.
  - multiplying a tensor of learning weights by a *batch* of input tensors

- Broadcasting's rules:
  - Each tensor must have at least one dimension - no empty tensors.
  - Comparing the dimension sizes of the two tensors, *backward*:
    - Each dimension must be equal, *or*
    - One of the dimensions must be of size 1, *or*
    - The dimension does not exist in one of the tensor.
  - Tensors of the identical shape.

# Math & Logic with PyTorch Tensors

- **Broadcasting** is a way to perform an operation between tensors that have similarities in their shapes.
  - multiplying a tensor of learning weights by a *batch* of input tensors

- Broadcasting's rules:
  - Each tensor must have at least one dimension - no empty tensors.
  - Comparing the dimension sizes of the two tensors, *backward*:
    - Each dimension must be equal, *or*
    - One of the dimensions must be of size 1, *or*
    - The dimension does not exist in one of the tensor.
  - Tensors of the identical shape.

# Math & Logic with PyTorch Tensors

- **Broadcasting** is a way to perform an operation between tensors that have similarities in their shapes.
  - multiplying a tensor of learning weights by a *batch* of input tensors

- Broadcasting's rules:
  - Each tensor must have at least one dimension - no empty tensors.
  - Comparing the dimension sizes of the two tensors, *backward*:
    - Each dimension must be equal, *or*
    - One of the dimensions must be of size 1, *or*
    - The dimension does not exist in one of the tensor.
  - Tensors of the identical shape.

# Math & Logic with PyTorch Tensors

- **Broadcasting** is a way to perform an operation between tensors that have similarities in their shapes.
  - multiplying a tensor of learning weights by a *batch* of input tensors

- Broadcasting's rules:
  - Each tensor must have at least one dimension - no empty tensors.
  - Comparing the dimension sizes of the two tensors, *backward*:
    - Each dimension must be equal, *or*
    - One of the dimensions must be of size 1, *or*
    - The dimension does not exist in one of the tensor.
  - Tensors of the identical shape.

# Math & Logic with PyTorch Tensors

- **Broadcasting** is a way to perform an operation between tensors that have similarities in their shapes.
  - multiplying a tensor of learning weights by a *batch* of input tensors

- Broadcasting's rules:
  - Each tensor must have at least one dimension - no empty tensors.
  - Comparing the dimension sizes of the two tensors, *backward*:
    - Each dimension must be equal, *or*
    - One of the dimensions must be of size 1, *or*
    - The dimension does not exist in one of the tensor.
  - Tensors of the identical shape.

# Math & Logic with PyTorch Tensors

- *Basic arithmetic:* $+ \; - \; * \; / \; \ldots$
  - *Tensor with Scalar*
  - *Tensor with Tensor (same-shape)*
  - *Tensor with Tensor (not same-shape)**
- More Math with Tensors

# Moving to GPU

# Autograd

# Automatic Differentiation

# Automatic Differentiation

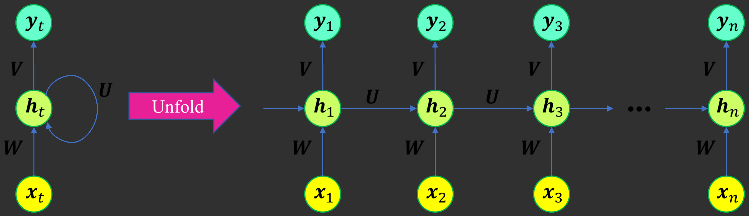



Figure: Training a RNN

# Automatic Differentiation

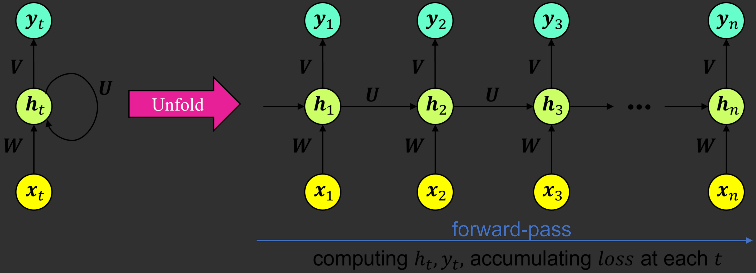



Figure: Training a RNN

# Automatic Differentiation

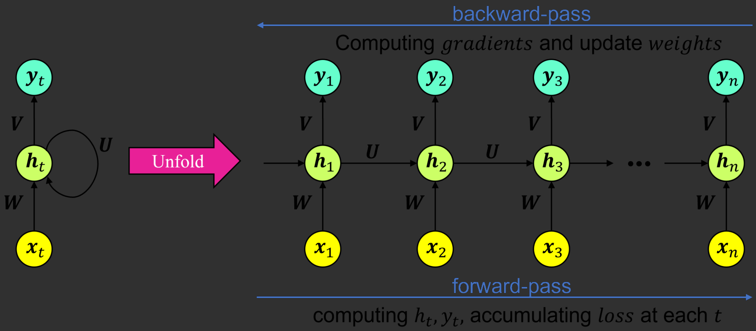



Figure: Training a RNN

# Automatic Differentation

- Machine learning model: $y = M(x)$
- The *loss* function: $L(y) = L(M(x))$
  - Our goal is to *minimize* the loss

# Automatic Differentation

- Machine learning model: $y = M(x)$
- The *loss* function: $L(y) = L(M(x))$
    - Our goal is to *minimize* the loss
    - Nudge the weights until a tolerable loss for wide variety of inputs

# Automatic Differentation

- Machine learning model: $y = M(x)$
- The *loss* function: $L(y) = L(M(x))$
  - Our goal is to *minimize* the loss
  - Nudge the weights until a tolerable loss for wide variety of inputs
  - $\frac{\partial L}{\partial x} = \frac{\partial L(y)}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$

# Automatic Differentation

- Machine learning model: $y = M(x)$
- The *loss* function: $L(y) = L(M(x))$
    - Our goal is to *minimize* the loss
    - Nudge the weights until a tolerable loss for wide variety of inputs
    - $\frac{dL}{dx} = \frac{dL(y)}{\partial x} = \frac{dL}{dy}\frac{\partial y}{\partial x} = \frac{dL}{dy}\frac{\partial M(x)}{\partial x}$

# Automatic Differentation

- Machine learning model: $y = M(x)$
- The *loss* function: $L(y) = L(M(x))$
  - Our goal is to *minimize* the loss
  - Nudge the weights until a tolerable loss for wide variety of inputs
  - $\frac{\partial L}{\partial x} = \frac{\partial L(y)}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} \qquad \frac{\partial M(x)}{\partial x}$

# Automatic Differentation

- Machine learning model: $y = M(x)$
- The *loss* function: $L(y) = L(M(x))$
  - Our goal is to *minimize* the loss
  - Nudge the weights until a tolerable loss for wide variety of inputs
  - $\dfrac{\partial L}{\partial x} = \dfrac{\partial L(y)}{\partial x} = \dfrac{\partial L}{\partial y}\dfrac{\partial y}{\partial x} = \dfrac{\partial L}{\partial y}\dfrac{\partial M(x)}{\partial x}$

# Automatic Differentation

- Machine learning model: $y = M(x)$
- The *loss* function: $L(y) = L(M(x))$
  - Our goal is to *minimize* the loss
  - Nudge the weights until a tolerable loss for wide variety of inputs
  - $\dfrac{\partial L}{\partial x} = \dfrac{\partial L(y)}{\partial x} = \dfrac{\partial L}{\partial y}\dfrac{\partial y}{\partial x} = \dfrac{\partial L}{\partial y}\dfrac{\partial M(x)}{\partial x}$

# Automatic Differentation

- Machine learning model: $y = M(x)$
- The *loss* function: $L(y) = L(M(x))$
  - Our goal is to *minimize* the loss
  - Nudge the weights until a tolerable loss for wide variety of inputs
  - $\dfrac{\partial L}{\partial x} = \dfrac{\partial L(y)}{\partial x} = \dfrac{\partial L}{\partial y}\dfrac{\partial y}{\partial x} = \dfrac{\partial L}{\partial y}\dfrac{\partial M(x)}{\partial x}$

# Automatic Differentation

- Machine learning model: $y = M(x)$.
- The *loss* function: $L(y) = L(M(x))$
  - Our goal is to *minimize* the loss
  - Nudge the weights until a tolerable loss for wide variety of inputs
  - $\frac{\partial L}{\partial x} = \frac{\partial L(y)}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial M(x)}{\partial x}$

# Simple Autograd

Practice

# Vector Calculus using `autograd`

# Vector Calculus using `autograd`

$$\vec{y} = f(\vec{x})$$

The Jacobian matrix $J$:

$$J = \left( \frac{\partial y}{\partial x_1} \cdots \frac{\partial y}{\partial x_n} \right) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

The gradient of a scalar function $l = g(\vec{y})$:

$$v = \left( \frac{\partial l}{\partial y_1} \cdots \frac{\partial l}{\partial y_m} \right)^T$$

# Vector Calculus using `autograd`

$$\vec{y} = f(\vec{x})$$

The Jacobian matrix $J$:

$$J = \left(\frac{\partial y}{\partial x_1} \cdots \frac{\partial y}{\partial x_n}\right) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

The gradient of a scalar function $l = g(\vec{y})$:

$$v = \left(\frac{\partial l}{\partial y_1} \cdots \frac{\partial l}{\partial y_m}\right)^T$$

# Vector Calculus using `autograd`

$$\vec{y} = f(\vec{x})$$

The Jacobian matrix $J$:

$$J = \left( \frac{\partial y}{\partial x_1} \cdots \frac{\partial y}{\partial x_n} \right) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

The gradient of a scalar function $l = g(\vec{y})$:

$$v = \left( \frac{\partial l}{\partial y_1} \cdots \frac{\partial l}{\partial y_m} \right)^T$$

# Vector Calculus using `autograd`

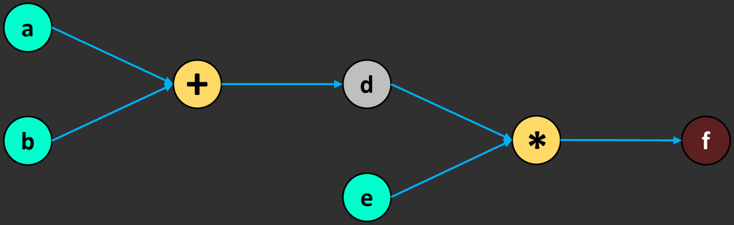
By the chain rule, gradient of $l$ w.r.t $\vec{x}$:

$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

*Note: We can also use the equivalent operation* $v^T \cdot J$

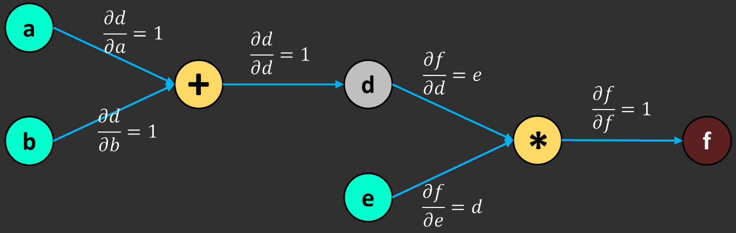
# Computational Graphs

# Computational Graphs



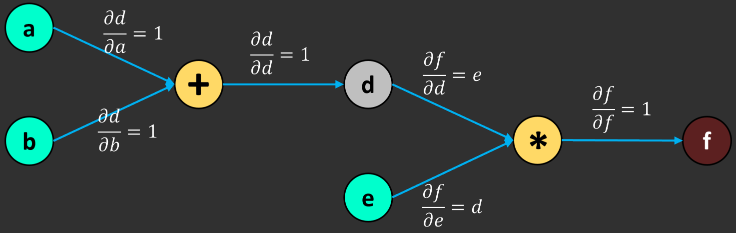

**Chain rule:** $\frac{\partial f}{\partial a} = \frac{\partial f}{\partial d} \times \frac{\partial d}{\partial a} = e \times 1 = e$

# Computational Graphs



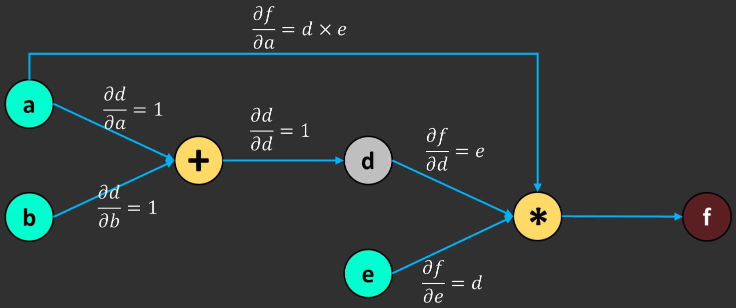

**Chain rule:** $\frac{\partial f}{\partial a} = \frac{\partial f}{\partial d} \times \frac{\partial d}{\partial a} = e \times 1 = e$

# Computational Graphs




**Chain rule:** $\frac{\partial f}{\partial a} = \frac{\partial f}{\partial d} \times \frac{\partial d}{\partial a} = e \times 1 = e$

# Computational Graphs




**Chain rule:** $\frac{\partial f}{\partial a} = \frac{\partial f}{\partial d} \times \frac{\partial d}{\partial a} + \frac{\partial f}{\partial a} = e \times 1 + d \times e = (e+1) \times d$

# Computational Graphs

In the forward pass:

- run the requested operation to compute a resulting tensor, and
- maintain the operation's *gradient function* in the DAG.

The backward pass kicks off when `.backward()`:

- computes the gradients from each `.grad_fn`,
- accumulates them in the repestive tensor's `.grad` attribute, and
- using the chain rule, propagates all the way to the leaf tensors.

# Building a Backwards Graph

# Hooks

# Other Basics

# Dataset and DataLoaders

## Save and Load the Model

# The End