Deep Learning for Science and Engineering Teaching Kit

A primer on TensorFlow and PyTorch

Contents

Objectives

- ❑ Brief introduction of tensors and algebraic operations on tensors using PyTorch and TensorFlow

- ❑ A brief introduction on preparing data for training processes

- ❑ An example of implementation of regression problem in python with and with out PyTorch and TensorFlow

- ❑Demonstration on implementation of feed-forward fully-connected network in PyTorch and TensorFlow

- ❑Demonstration on implementation of AD process in PyTorch and TensorFlow

Deep Learning frameworks

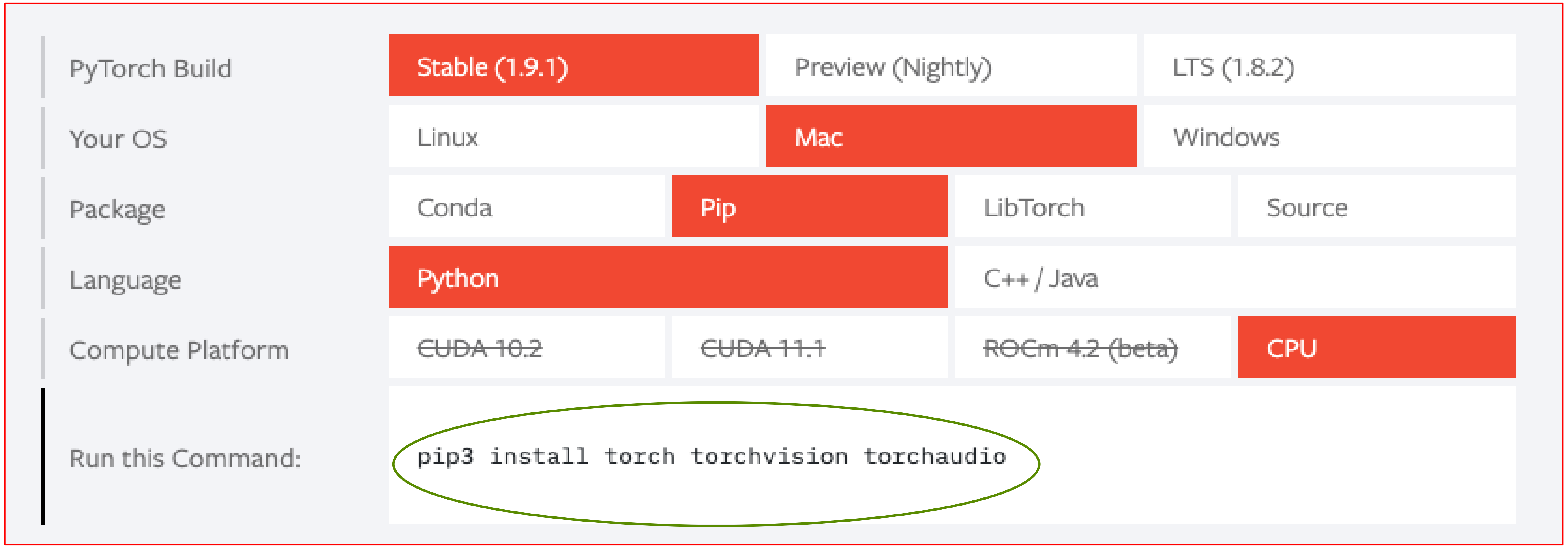
- In this course, we will demonstrate the implementation of machine learning algorithms in two frameworks.

- PyTorch: PyTorch is the product of Facebook: Feels more ``pythonic" with an object-oriented approach.

- TensorFlow: TensorFlow is developed and maintained by Google Brain. Has several options from which you may choose.

Installing PyTorch

❑ Go to PyTorch Website and select the environment and configuration; e.g, on Mac with CPU and using pip as package builder, we have



❑ Then run this in jupyter not book as follow

Note the ! (exclamation) before the pip

```
In [2]:  1  !pip3 install torch torchvision torchaudio
```

CRUNCH Group
Brown University

DEEP LEARNING INSTITUTE
BROWN

Installing Tensorflow

- Very simple: run this on ***jupyter*** note book

```
In [ ]:   1  !pip3 install tensorflow
```

- Sidenote: To know the list of packages in your current environment; do

```
In [3]:   1  !pip freeze
```

Demo: *Installing the PyTorch and TensorFlow*

Computational Model in PyTorch and TensorFlow

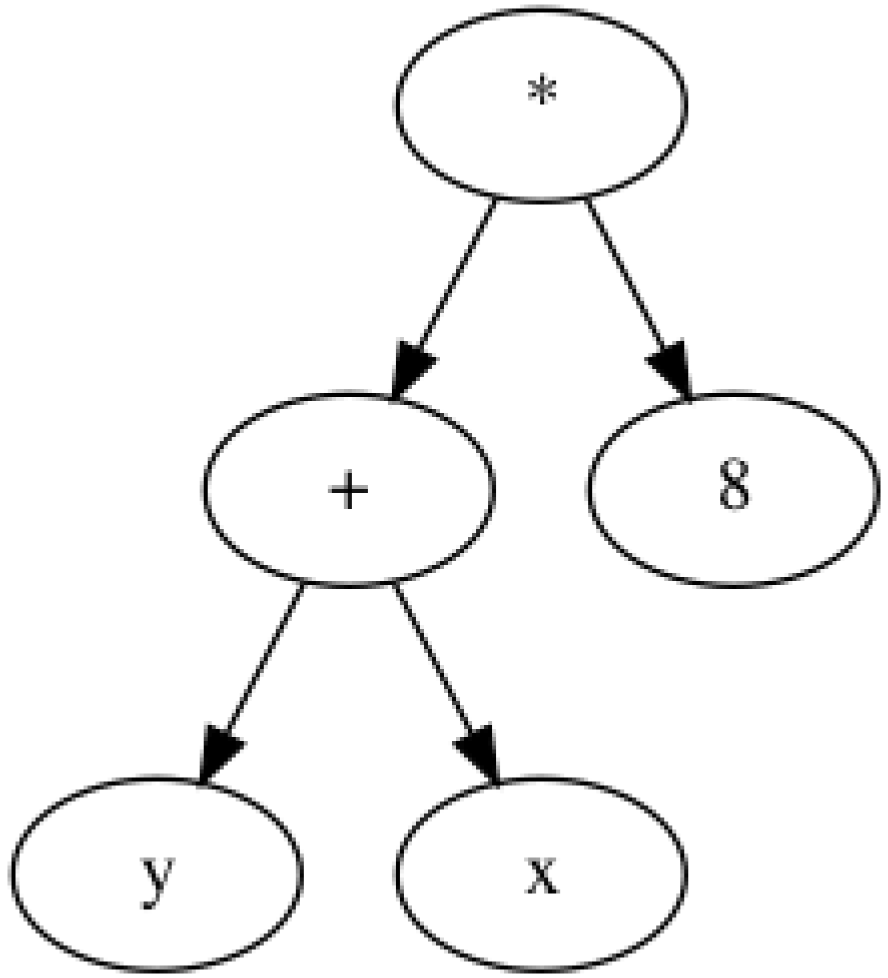
- Mathematical expressions are evaluated by first constructing the computational graph

- In graph operators are represented by nodes and edges as data or arrays or placeholders

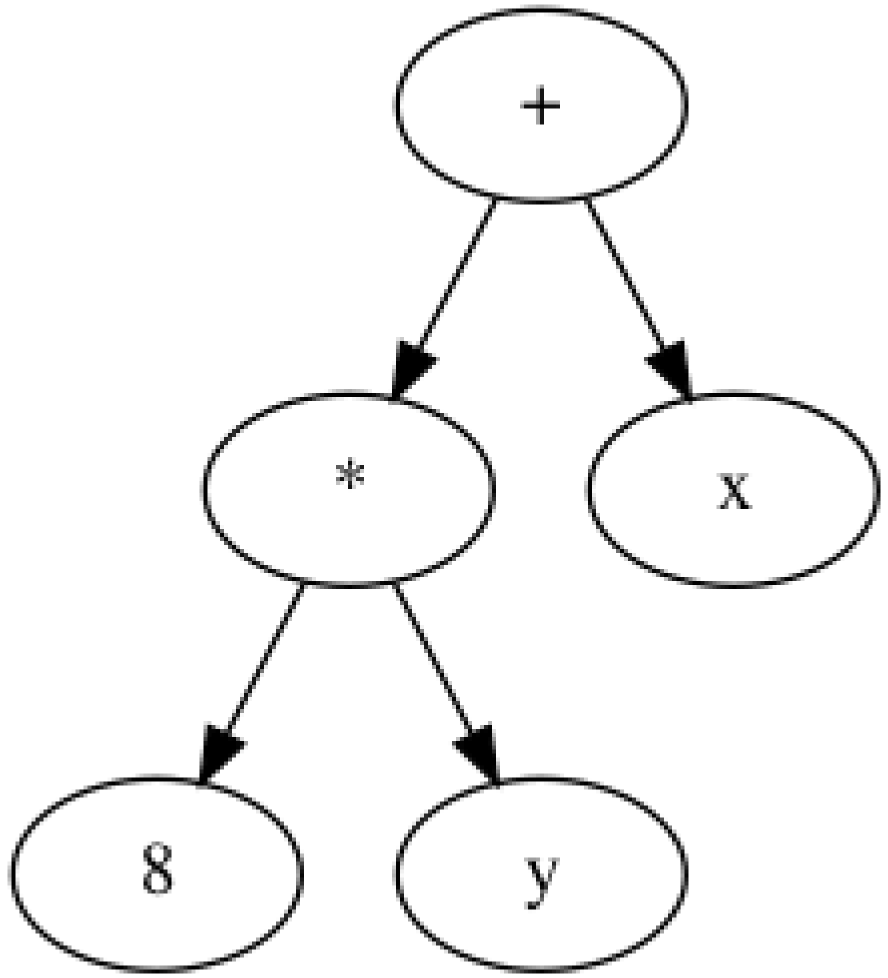
- Evaluate the graph by passing the data for variables

Computational graph: An example

$$\mathrm{ex}_1 : \ 8 * (y + x)$$



$$\mathrm{ex}_2 : \ 8 * y + x$$



Demo: *Construct Graph in Python*

Lets code up: Graph based computation

“Constant” sub-class: e.g., 8

```python
# Super class for Expressions:

class Expr:
    pass

### Subclass of Expr for Constant e.g., 3

class Const(Expr):
    def __init__(self, val):
        self.val = val

    def getVal(self):
        return self.val

    def __str__(self):
        return str(self.getVal())

    def eval(self, env):
        return self.getVal()
```

“Variable” sub-class: e.g., 8

```python
### Subclass of Expr for Variables e.g., x, y
class Var(Expr):
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def __str__(self):
        return self.getName()

    def eval(self, env):
        return env[self.name]
```

CRUNCH Group
Brown University

NVIDIA. DEEP LEARNING INSTITUTE | BROWN

Lets code up: Graph based computation:

Binary Operation: “Plus” sub-class

```
class Plus(Expr):
    def __init__(self, l, r):
        self.l = l
        self.r = r
    def __str__(self):
        return "(" + str(self.l) + "+" + str(self.r) + ")"

    def getLeft(self):
        return self.l

    def getRight(self):
        return self.r

    def eval(self, env):
        return self.getLeft().eval(env) + self.getRight().eval(env)
```

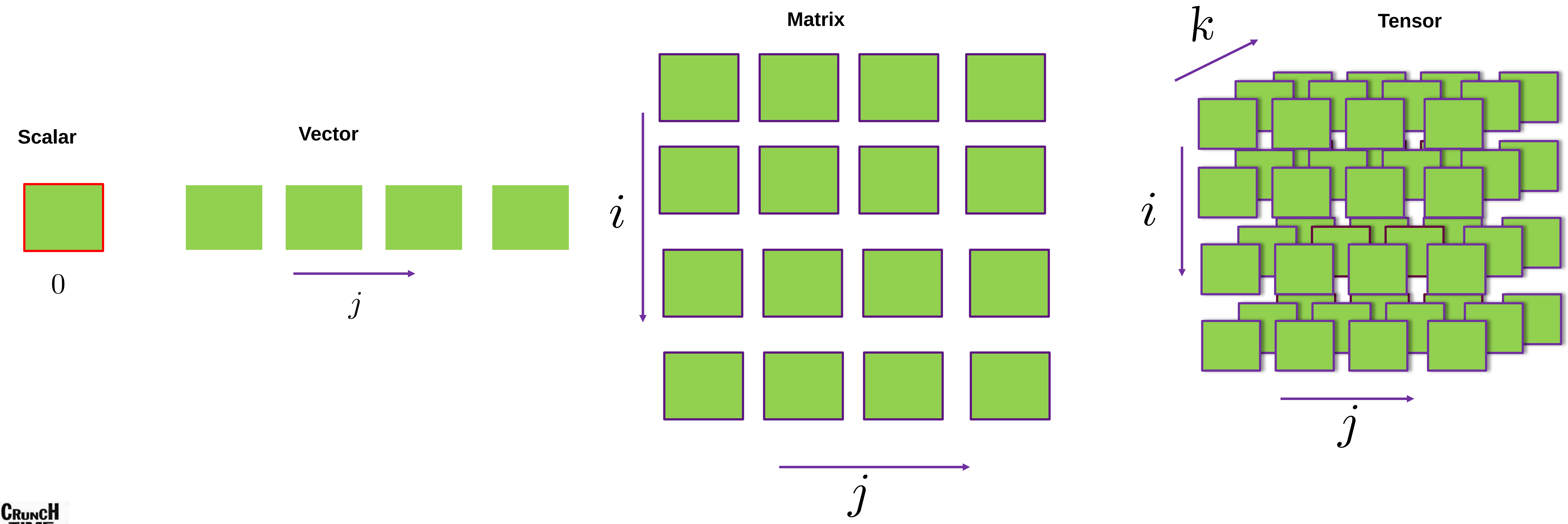
Binary Operation: “Times” sub-class

```
### Subclass of Expr for Binary Operations: e.g., x, y
class Times(Expr):
    def __init__(self, l, r):
        self.l = l
        self.r = r

    def getLeft(self):
        return self.l

    def getRight(self):
        return self.r

    def __str__(self):
        return "(" + str(self.getLeft()) + "*" + str(self.getRight()) + ")"
```

Demo: *Graph Based Computation*

jupyter

Scalars, Vectors, Matrices and Tensors



Data Structures: NumPy and PyTorch

NumPy

```python
import numpy as np
```

```python
# Set seed for reproducibility
np.random.seed(0)


# Predefine Matrix of shape=(2,3)
np.array([[4, 5, 6], [1, 7, 8]])

# Zero Matrix of shape=(3,3)
np.zeros((3,3))

# Identity Matrix of shape=(2,2)
np.eye(2)

# Random Matrix of shape=(3,3)
np.random.rand(3,3)
```

```python
In [30]: import torch
```

```python
# Set seed for reproducibility
torch.manual_seed(0)


# Predefine Matrix of shape=(2,3)
torch.tensor([[4, 5, 6], [1, 7, 8]])

# Zero Matrix of shape=(3,3)
torch.zeros((3,3))

# Identity Matrix of shape=(2,2)
torch.eye(2)

# Random Matrix of shape=(3,3)
torch.rand(3,3)
```

Properties of Tensor

```
# Scaler
s = torch.tensor(1.)
print(f"Sclaer x: {s}")

# Check dimension of Scaler: which is Rank in Linear Algebra Term
d = s.dim()
print(f"Dimension of vector is: {d}")

# Vectors
v = torch.tensor([1., 2., 3.])
print(f"Vector v: {v}")
#Check dimension of Vectors
d = v.dim()
print(f"Dimension of vector is: {d}")


# Matrix
m = torch.tensor([[1., 2., 3.],[4., 5., 6.]])
d = m.dim()
print(f"Dimension of matrix is: {d}")


# Tensor
# Matrix
m = torch.tensor([[[1., 2., 3.],[4., 5., 6.], [1., 2., 3.],[4., 5., 6.]]])
d = m.dim()
print(f"Dimension of Tensor is: {d}")

```

Demo: *Tensors in PyTorch*

Methods on Tensors: dimensions

```python
# Scaler
s = torch.tensor(1.)
print(f"Sclaer x: {s}")

# Check dimension of Scaler: which is Rank in Linear Algebra Term
d = s.dim()
print(f"Dimension of vector is: {d}")

# Vectors
v = torch.tensor([1., 2., 3.])
print(f"Vector v: {v}")
#Check dimension of Vectors
d = v.dim()
print(f"Dimension of vector is: {d}")


# Matrix
m = torch.tensor([[1., 2., 3.],[4., 5., 6.]])
d = m.dim()
print(f"Dimension of matrix is: {d}")


# Tensor
# Matrix
m = torch.tensor([[[1., 2., 3.],[4., 5., 6.], [1., 2., 3.],[4., 5., 6.]]])
d = m.dim()
print(f"Dimension of Tensor is: {d}")

```

Demo: dim *`Methods on Tensors`*

Methods on Tensors : sum and reshape

Sum

```python
# Set seed for reproducibility
torch.manual_seed(0)


# Random Matrix of shape=(3,3)
x = torch.rand(3,2)
print(f"x: {x}")

xsum = torch.sum(x, dim=1)
print(f"xsum using mthod1: {xsum}")

x.sum(dim=1)
print(f"xsum using mthod2: {xsum}")
```

Reshape : `view` and `reshape` methods

In [62]:

```python
#### Inplace Reshaping

# A vector of length N=10
x = torch.tensor([1,2,3,4,5,6,7,8,9,10, 11, 12])
# Reshape in amatrix of shape= (2,5)
x.view(3,4)

# Reshape with unspecified number of rows and 4 column
x.view(-1, 4)

#### Reshaping via copying

# A vector of length N=10
x = torch.tensor([1,2,3,4,5,6,7,8,9,10,11,12])

# Reshape in amatrix of shape= (2,5)
y3 = x.reshape(3,4)

# Reshape with unspecified number of rows and 4 column
y4 = x.reshape(-1,4)
```

Demo: *Methods sum and reshape Tensors in PyTorch*

Methods on Tensors : computing norms

$$L_p \text{ norm} : ||\mathbf{x}||_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}}$$

$$L^1 \text{ norm} : ||\mathbf{x}||_1 = |x_1| + |x_2| + \ldots + |x_n|$$

$$L^2 \text{ norm} : ||\mathbf{x}||_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$$

Using inbuild PyTorch method

```
torch.manual_seed(0)
x = torch.rand(3)
x.norm(p=1)
x.norm(p=2)
print(f"L1 Norm of x is:{x.norm(p=1)}")
print(f"L2 Norm of x is:{x.norm(p=2)}")
```

Using PyTorch Primitives

```
n1 = torch.sum(torch.abs(x))
print(f"L1 norm: is: {n1}")
n2 = torch.sqrt(torch.sum(x**2))
print(f"L2 norm: is: {n2}")

## Or Calling method directly on the data structures
n1 = x.abs().sum()
print(f"L1 norm: is: {n1}")
n2 = (x**2).sum().sqrt()
print(f"L2 norm: is: {n2}")
```

Demo: *Norms*

jupyter

Tensors on GPUs

Mapping tensors to GPU

```
dev_cpu = torch.device("cpu")
dev_gpu = torch.device("cuda:0")

# Send Tensor to GPU
x.to(dev_cpu)

```

```
tensor([[4., 5., 8.],
        [3., 8., 9.]])
```

```
# At the start of your code
device = torch.device("cpu" if not torch.cuda.is_available() else "cuda")

# For later dispatch
x.to(device)
```

```
tensor([[4., 5., 8.],
        [3., 8., 9.]])
```

NumPy ----> PyTorch ----> NumPy

```python
import numpy as np

X = np.random.random((4,4))
#print(X)
```

```python
# NumPy to PyTorch
Y = torch.from_numpy(X)
#print(Y)
```

```python
# PyTorch ---> NumPy
```

```python
X = Y.numpy()
#print(X)
```

Timing GPU Operations

```python
A = torch.rand(100, 400, 400)
#B = A.cuda()
A.size()
```

```
torch.Size([100, 400, 400])
```

```python
%timeit -n 3 torch.bmm(A, A)
%timeit -n 3 torch.bmm(B, B)
```

CRUNCH TIME

CRUNCH Group

Brown University

DEEP LEARNING INSTITUTE

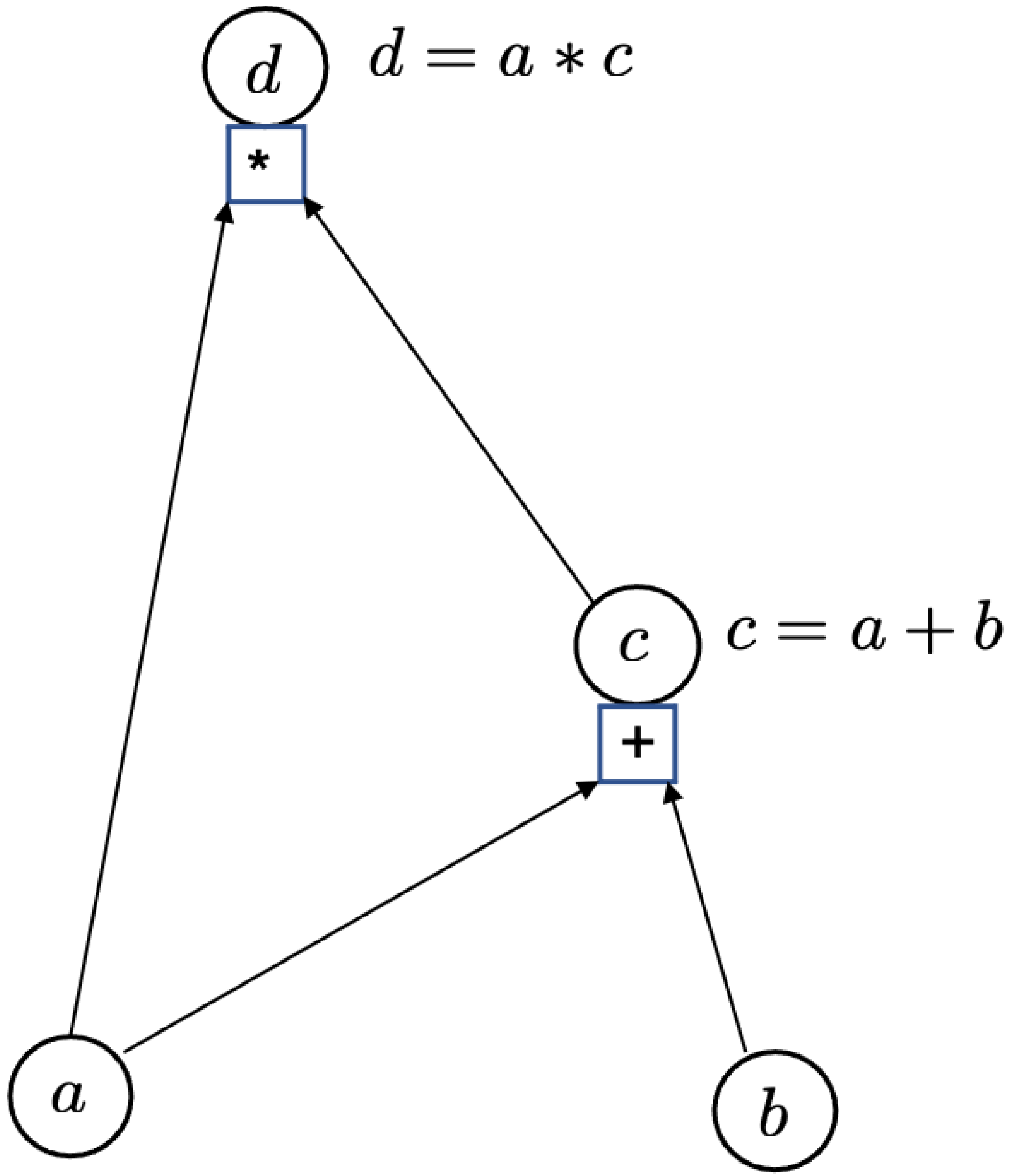
BROWN

Demo: *Tensors on GPU, NumPy<->PyTorch, and,Timing*

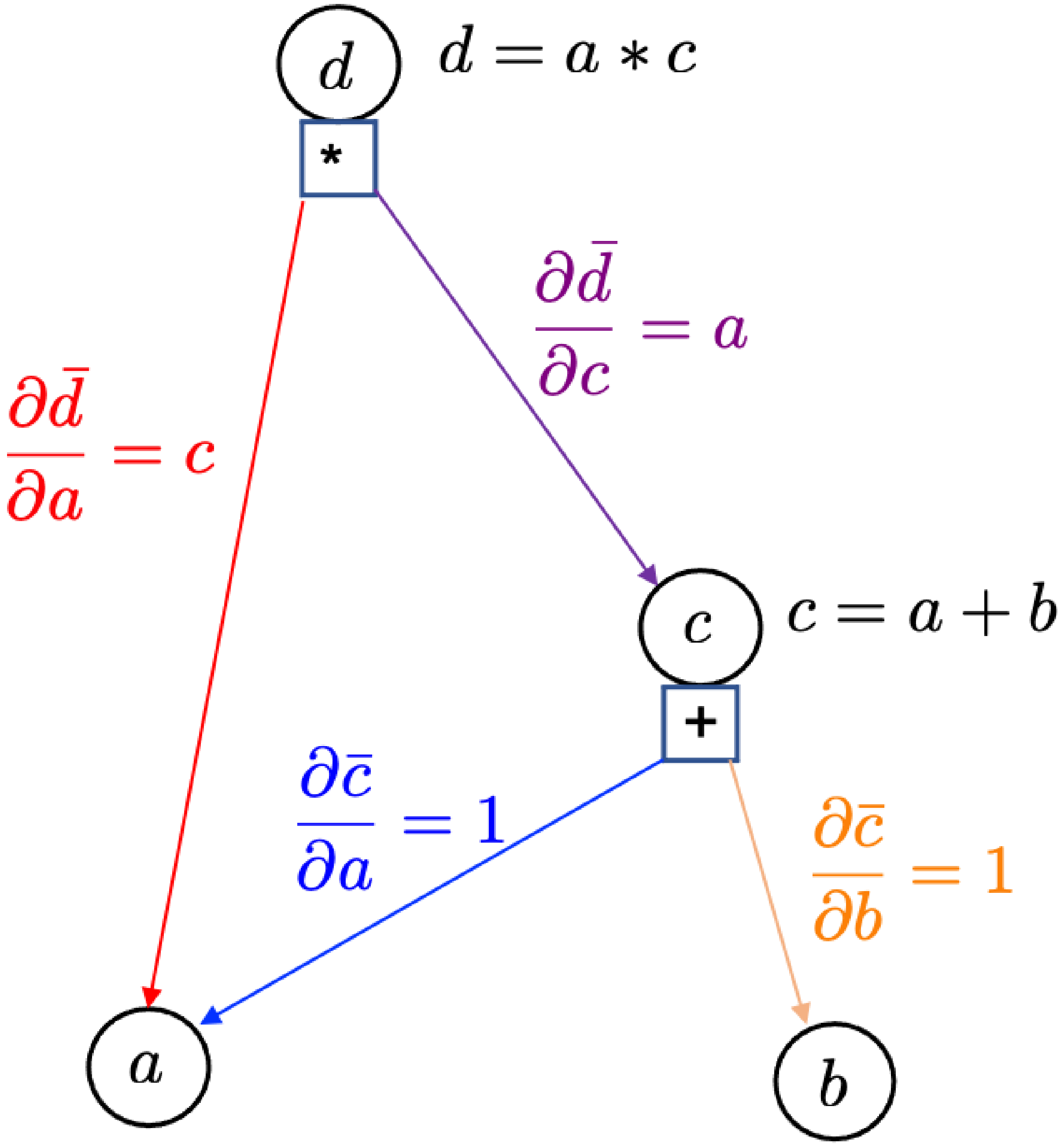
Automatic Differentiation in Python

$$d = a * (a + b)$$

Forward pass



Backward pass



By chain rule:

$$\frac{\partial d}{\partial a} = \frac{\partial \bar{d}}{\partial a} + \frac{\partial \bar{d}}{\partial c} * \frac{\partial \bar{c}}{\partial a}$$

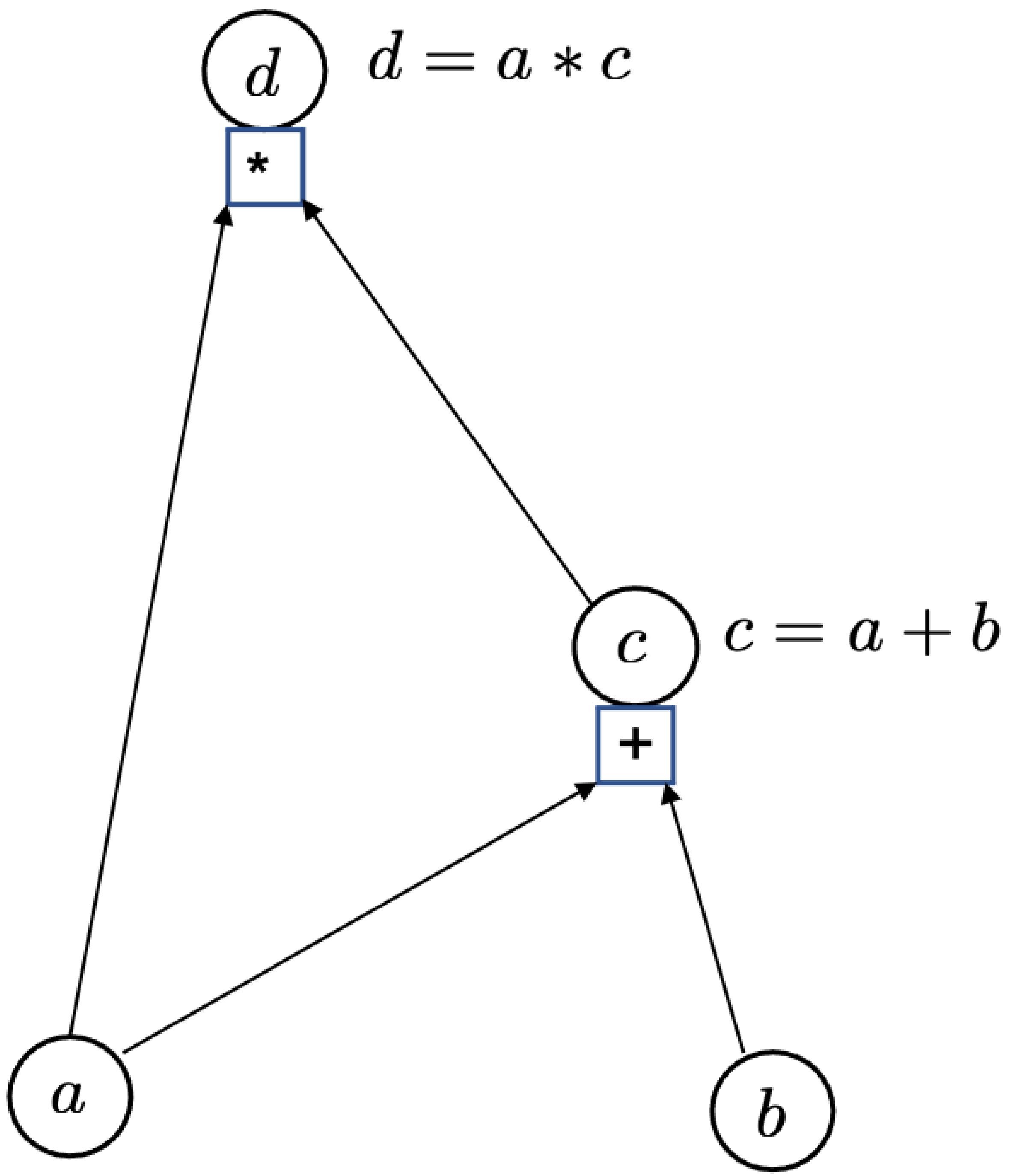
$$= c + a$$

$$\frac{\partial d}{\partial b} = \frac{\partial d}{\partial c} * \frac{\partial c}{\partial b} = a * 1 = a$$

AD in Python from Scratch

```python
from collections import defaultdict

class Var:
    def __init__(self, val, local_grad=()):
        self.val = val
        self.local_grad = local_grad

    def __add__(self, other):
        y = self.val + other.val
        local_grad = ((self, 1), (other, 1))
        return Var(y, local_grad)

    def __mul__(self, other):
        y = self.val*other.val
        local_grad = ((self, other.val), (other, self.val))
        return Var(y, local_grad)

    def __sub__(self, other):
        y = self.val - other.val
        local_grad = ((self, 1), (other, -1))
        return Var(y, local_grad)
```

Forward pass



AD in Python from Scratch

```python

def get_grads(var):
    grad = defaultdict(lambda:0)

    def compute_grad(var, path):
        for child_var, loc_grad in var.local_grad:
            val_path_child = path * loc_grad
            grad[child_var] += val_path_child
            compute_grad(child_var,val_path_child)

    compute_grad(var, path=1)

    return grad


```

$$\frac{\partial d}{\partial a} = \frac{\partial \bar{d}}{\partial a} + \frac{\partial \bar{d}}{\partial c} * \frac{\partial \bar{c}}{\partial a}$$

$$= c + a$$

$$\frac{\partial d}{\partial b} = \frac{\partial d}{\partial c} * \frac{\partial c}{\partial b} = a * 1 = a$$

AD in Python from Scratch

Evaluation of derivatives

```python
a = Var(8)
b = Var(6)

## AD for Addition

c = a + b
d = a*c

grad = get_grads(d)

print(f"AD of addition: {grad[a]}")

## AD for Subtraction

c = a - b
d = a*c

grad = get_grads(d)

print(f"AD of subtraction: {grad[a]}")


```

```
AD of addition: 22
AD of subtraction: 10
```

CRUNCH Group
Brown University

NVIDIA. DEEP LEARNING INSTITUTE BROWN

Demo: *Reverse AD in Python*

Einstein Summation-2D

$$\begin{bmatrix} 8 & 6 & 8 \\ 6 & 7 & 9 \\ 8 & 4 & 8 \\ 4 & 8 & 5 \end{bmatrix} \times \begin{bmatrix} 5 & 9 & 7 & 4 \\ 5 & 5 & 7 & 3 \\ 8 & 7 & 3 & 7 \end{bmatrix} = \begin{bmatrix} 126 & 174 & 111 \\ 98 & 132 & 90 \\ 135 & 158 & 128 \end{bmatrix}$$

$$A_{ij} \qquad B_{jk} \qquad C_{ik}$$

In Einstein Notation: Summation over repeated indices

$$C_{ik} = A_{ij} B_{jk}$$

$$i, j : \text{fixed indices}$$

$$k : \text{free index}$$

```
In [13]:  1 torch.matmul(A, B)

Out[13]: tensor([[126, 174, 111],
                 [ 98, 132,  90],
                 [135, 158, 128]])

In [15]:  1 C= torch.einsum("ij, jk -> ik", A, B)
          2 C

Out[15]: tensor([[126, 174, 111],
                 [ 98, 132,  90],
                 [135, 158, 128]])
```

CRUNCH TIME
CRUNCH Group
Brown University

Einstein Summation-ND

$$
\begin{bmatrix} 9 & 7 & 3 \\ 8 & 3 & 4 \\ 9 & 7 & 3 \\ 5 & 5 & 5 \end{bmatrix} \times \begin{bmatrix} 7 & 5 & 4 & 4 \\ 7 & 9 & 8 & 4 \\ 8 & 4 & 8 & 5 \end{bmatrix} = \begin{bmatrix} 107 & 108 & 73 & 102 \\ 83 & 86 & 56 & 82 \\ 107 & 108 & 73 & 102 \\ 78 & 97 & 58 & 98 \end{bmatrix}
$$

$$
\begin{bmatrix} 4 & 8 & 3 \\ 6 & 7 & 4 \\ 9 & 8 & 6 \\ 4 & 5 & 8 \end{bmatrix} \times \begin{bmatrix} 8 & 6 & 8 & 6 \\ 7 & 9 & 8 & 4 \\ 8 & 4 & 8 & 5 \end{bmatrix} = \begin{bmatrix} 112 & 108 & 120 & 71 \\ 129 & 115 & 136 & 84 \\ 176 & 150 & 184 & 116 \\ 131 & 101 & 136 & 84 \end{bmatrix}
$$

$$
\begin{bmatrix} 7 & 8 & 8 \\ 9 & 6 & 4 \\ 7 & 4 & 5 \\ 7 & 7 & 6 \end{bmatrix} \times \begin{bmatrix} 5 & 9 & 7 & 4 \\ 5 & 5 & 7 & 3 \\ 8 & 7 & 3 & 7 \end{bmatrix} = \begin{bmatrix} 139 & 159 & 129 & 108 \\ 107 & 139 & 117 & 82 \\ 95 & 118 & 92 & 75 \\ 118 & 140 & 116 & 91 \end{bmatrix}
$$

$(b, i, j) = (3, 4, 3)$ $(b, j, k) = (3, 3, 4)$ $(b, i, k) = (3, 4, 4)$

$b = \text{Batch Size}$

```python
C = torch.einsum("bij, bjk->bik", A,B)
C
```

```
tensor([[[107, 108,  73, 102],
         [ 83,  86,  56,  82],
         [107, 108,  73, 102],
         [ 78,  97,  58,  98]],

        [[112, 108, 120,  71],
         [129, 115, 136,  84],
         [176, 150, 184, 116],
         [131, 101, 136,  84]],

        [[139, 159, 129, 108],
         [107, 139, 117,  82],
         [ 95, 118,  92,  75],
         [118, 140, 116,  91]]])
```

```python
Ct = torch.matmul(A, B)
```

```python
Ct
```

```
tensor([[[107, 108,  73, 102],
         [ 83,  86,  56,  82],
         [107, 108,  73, 102],
         [ 78,  97,  58,  98]],

        [[112, 108, 120,  71],
         [129, 115, 136,  84],
         [176, 150, 184, 116],
         [131, 101, 136,  84]],

        [[139, 159, 129, 108],
         [107, 139, 117,  82],
         [ 95, 118,  92,  75],
         [118, 140, 116,  91]]])
```

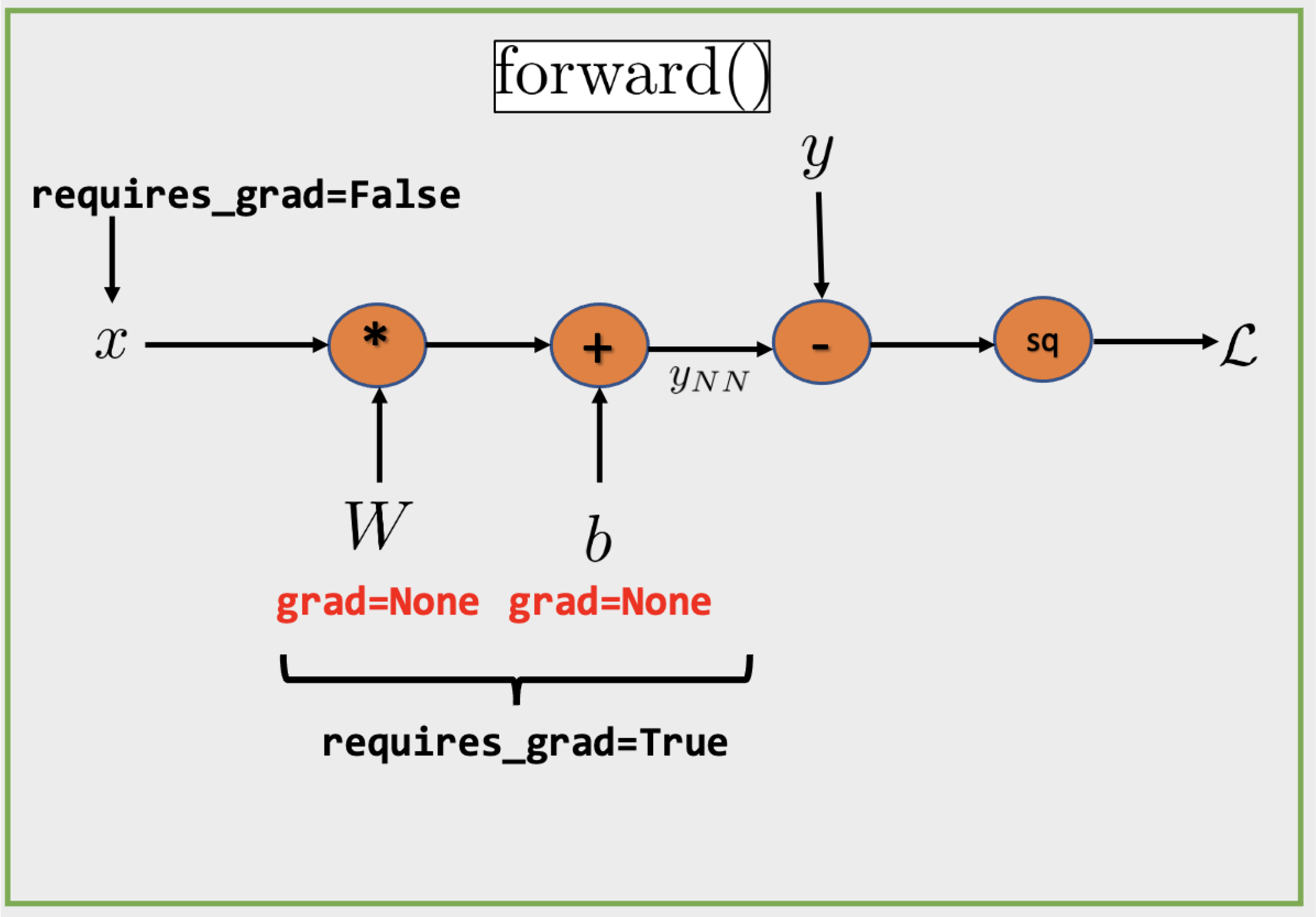
Demo: *Einstein Summation*

NN + Function Approximation

$$y_{NN} = f(x) = \Phi\left(\mathbf{Wx} + \mathbf{b}\right)$$

$$y = f(x)$$

$$\mathcal{L} = \frac{1}{N}\sum_i \left(y_{NN} - y\right)^2$$



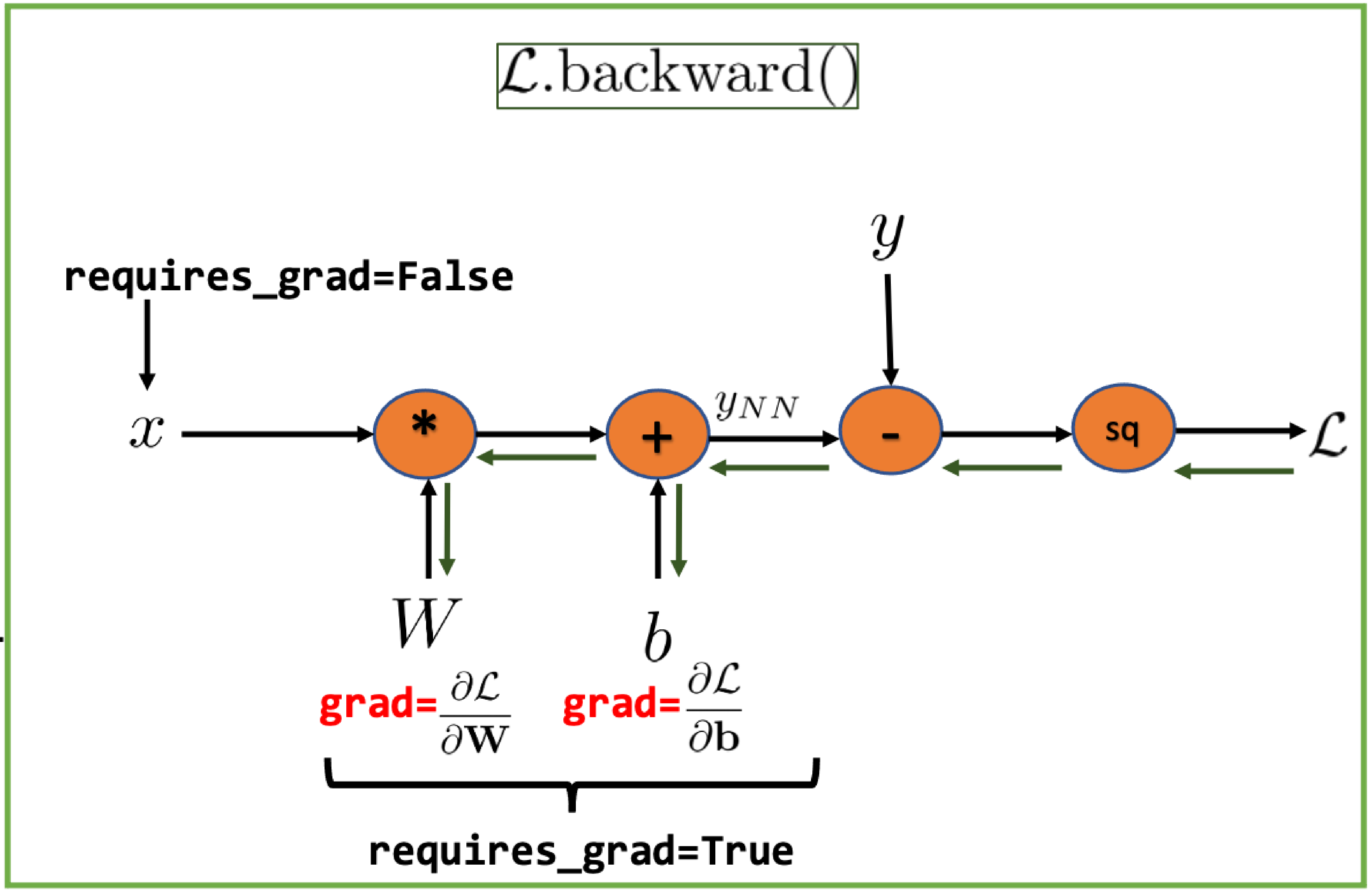
NN + Function Approximation

$$y_{NN} = f(x) = \Phi\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)$$

$$y = f(x)$$

$$\mathcal{L} = \frac{1}{N}\sum_i (y_{NN} - y)^2$$

To Find $\mathbf{W}$ and $\mathbf{b}$, we need to compute $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$ using backpropagation.

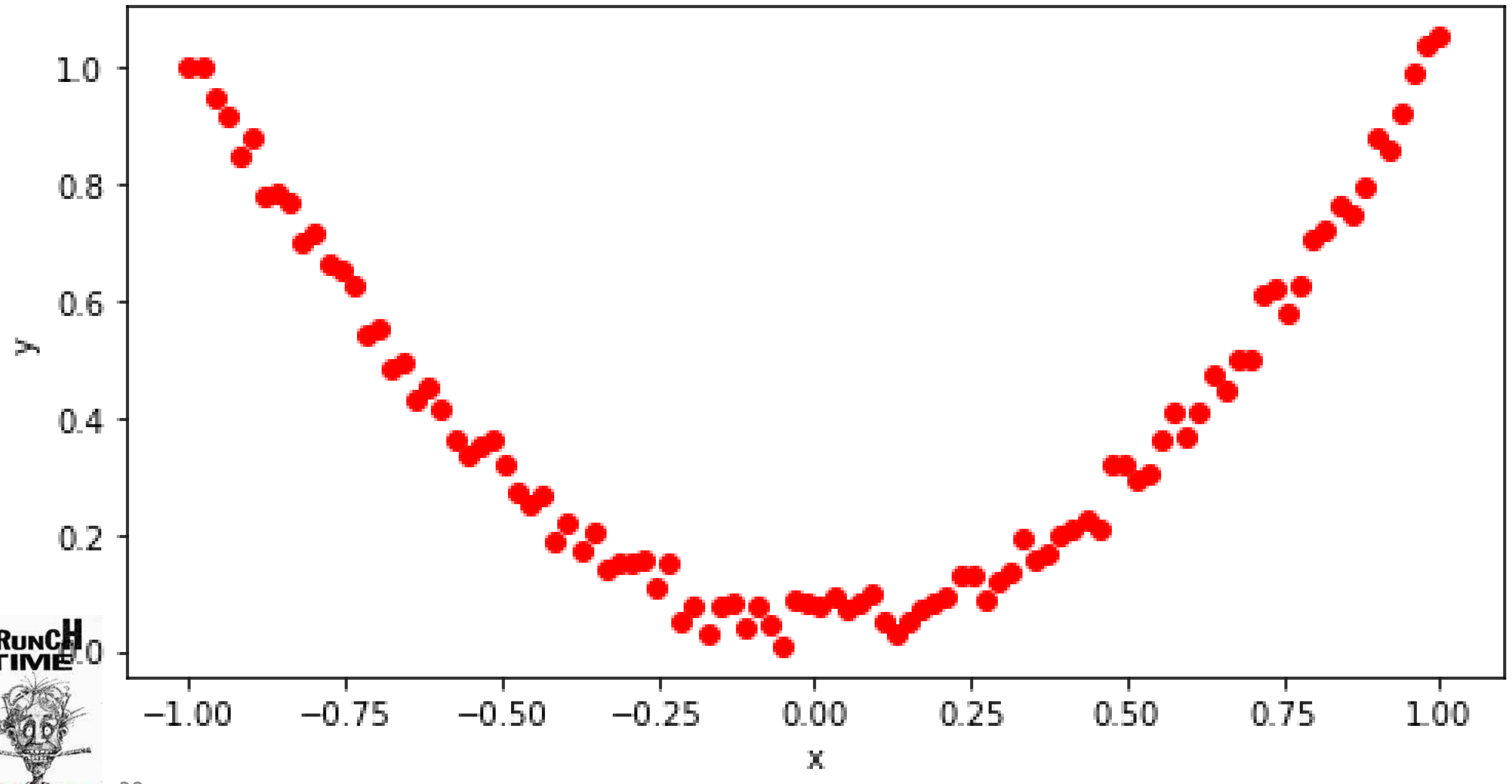


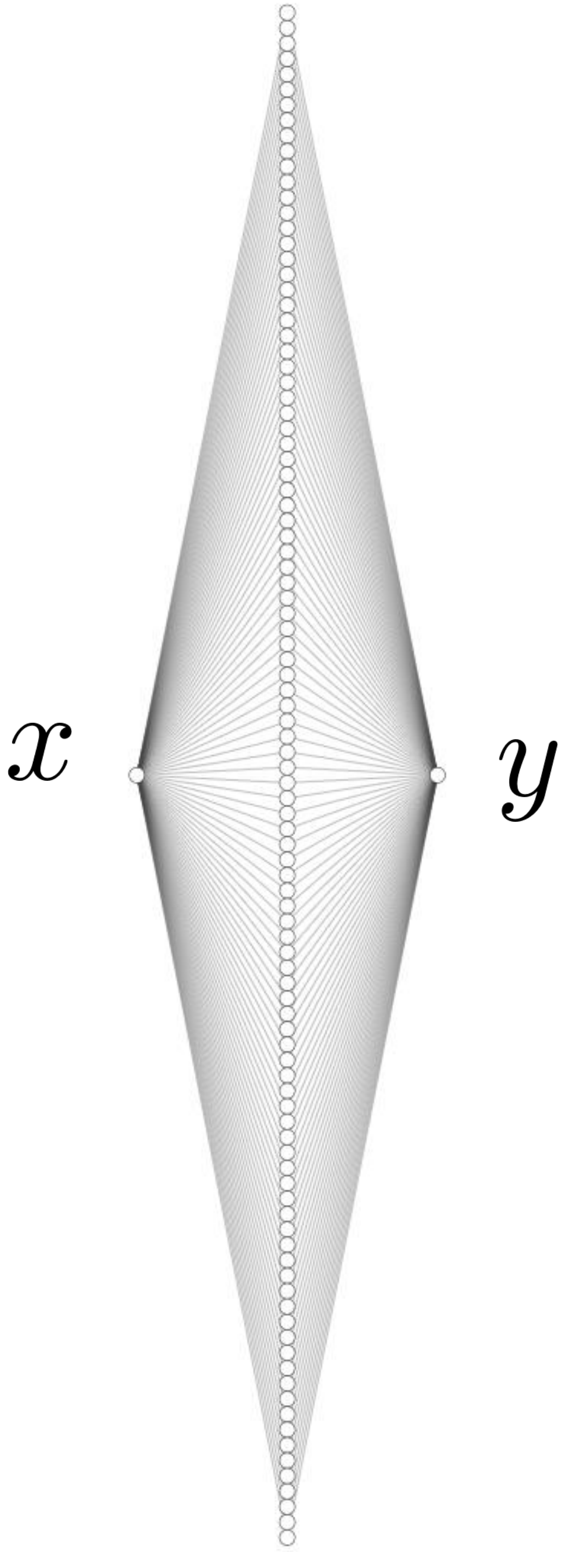
NN + Function Approximation

$$y = x^2 + \epsilon$$

$$\epsilon \sim U[0, 1), \text{and } x \in [-1, 1]$$

$$\text{Noise} = 10\%$$





NN + Function Approximations + PyTorch

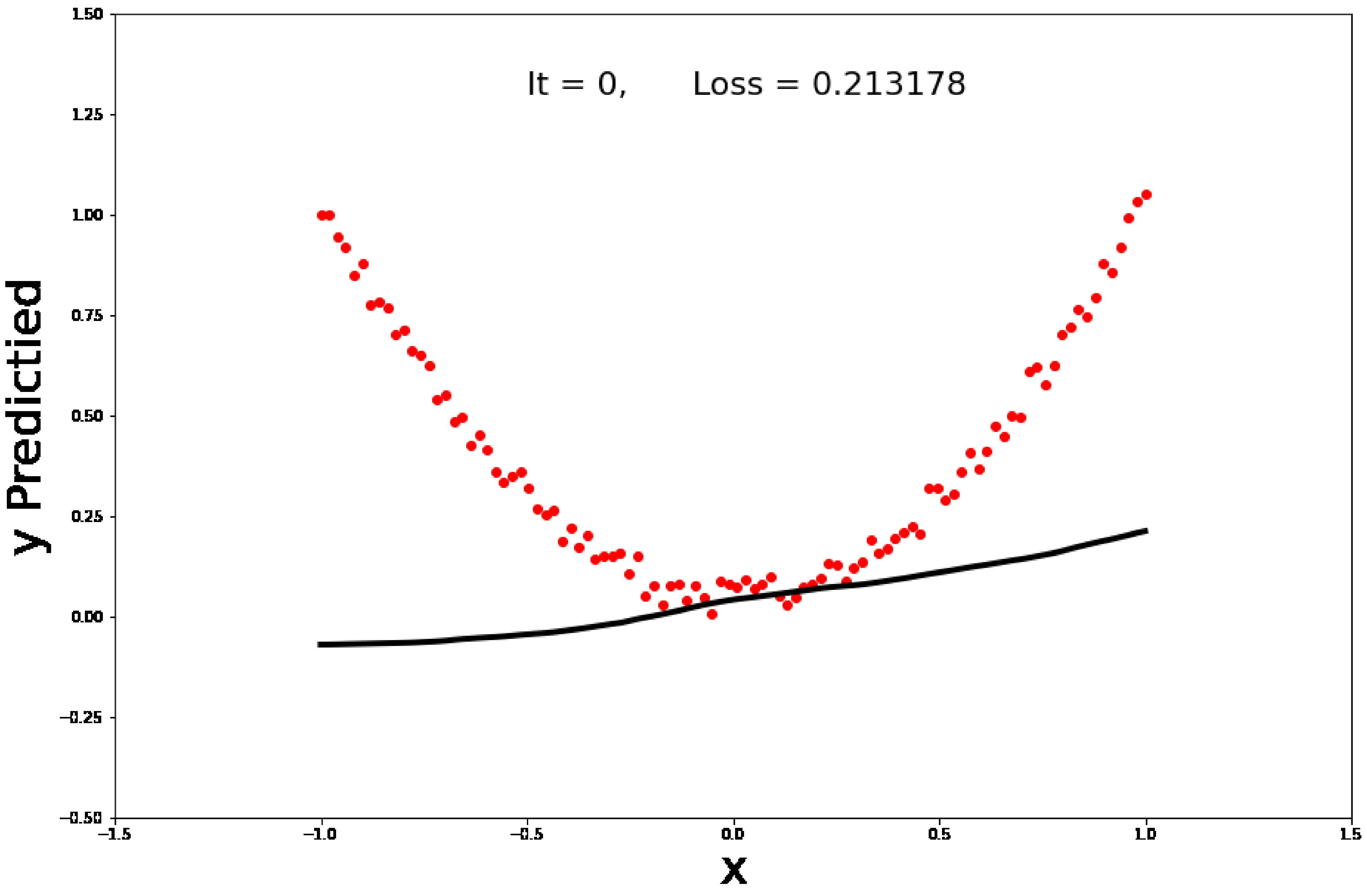
Data Preparation

```python
import numpy as np
import imageio
import torch
import torch.nn.functional as F
import torch.utils.data as Data
from torch.autograd import Variable
import matplotlib.pyplot as plt
%matplotlib inline
torch.manual_seed(1234)

### Input data
x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)
# torch.unsqueeze: Returns a new tensor
# with a dimension of size one inserted at the specified position.
y = torch.square(x)
# Add Random Noise
y = y + 0.1*torch.rand(y.size())

# Plot the data
plt.figure(figsize=(8,4))

x_plot, y_plot = x.numpy(), y.numpy()
plt.scatter(x_plot, y_plot, color="red")
plt.xlabel("x")
plt.ylabel("y")
plt.show("Data for Regression Analysis")
plt.show()
```

Training

```python
# Convert x and y to tracked variables
x = Variable(x)
y = Variable(y)

Net = torch.nn.Sequential(
        torch.nn.Linear(1, 100),
        torch.nn.LeakyReLU(),
        torch.nn.Linear(100, 1))

optimizer = torch.optim.Adam(Net.parameters(), lr = 0.01)
loss_function = torch.nn.MSELoss()

image_list = []
Niter = 300 + 1

fig, ax = plt.subplots(figsize=(15,10))

for it in range(150):
    y_pred = Net(x)
    loss = loss_function(y_pred, y)  # Notice the order: NN
    optimizer.zero_grad()    # Zero Out the gradient
    loss.backward()
    optimizer.step()
```

NN + Function Approximations + PyTorch

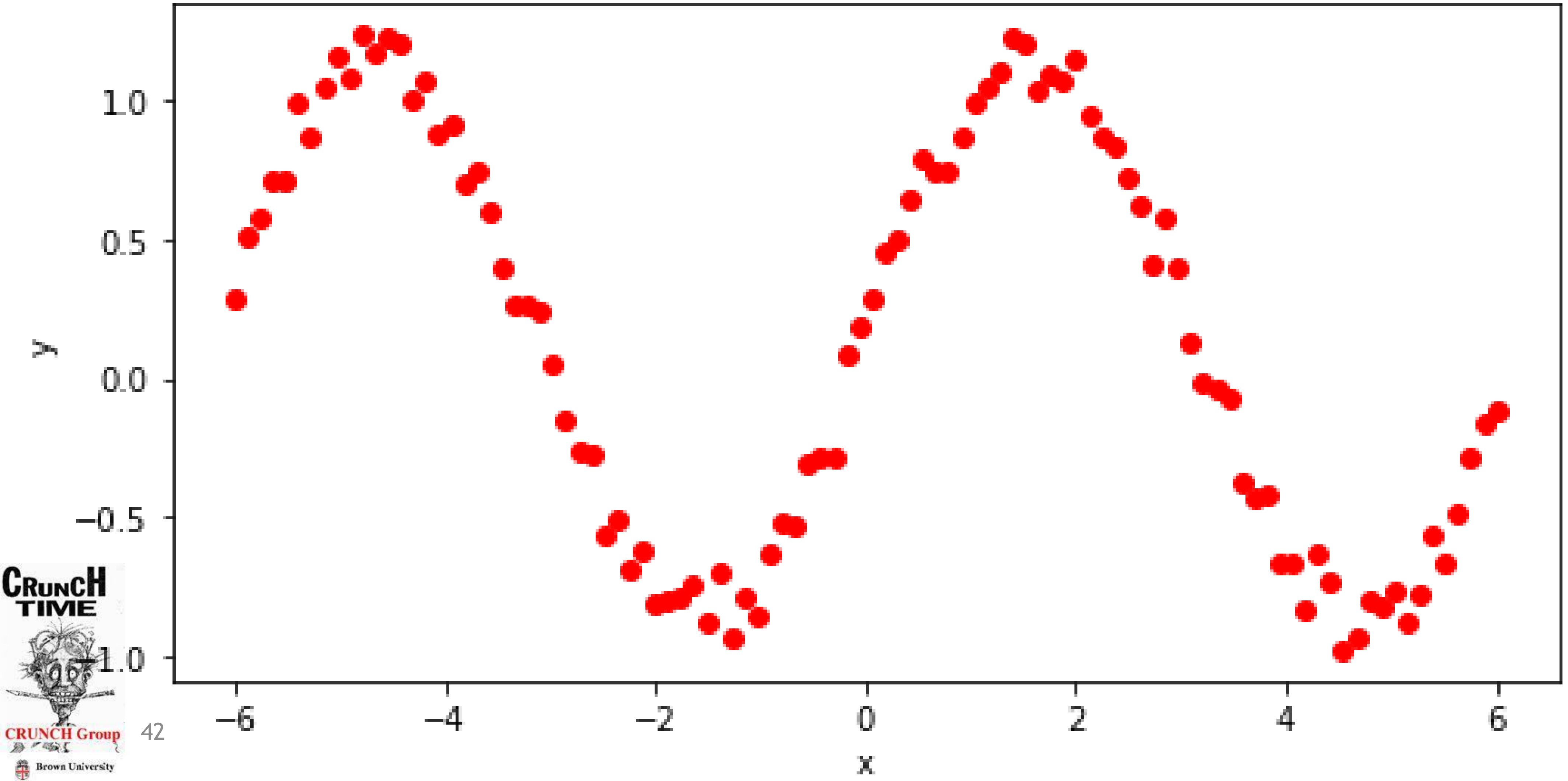


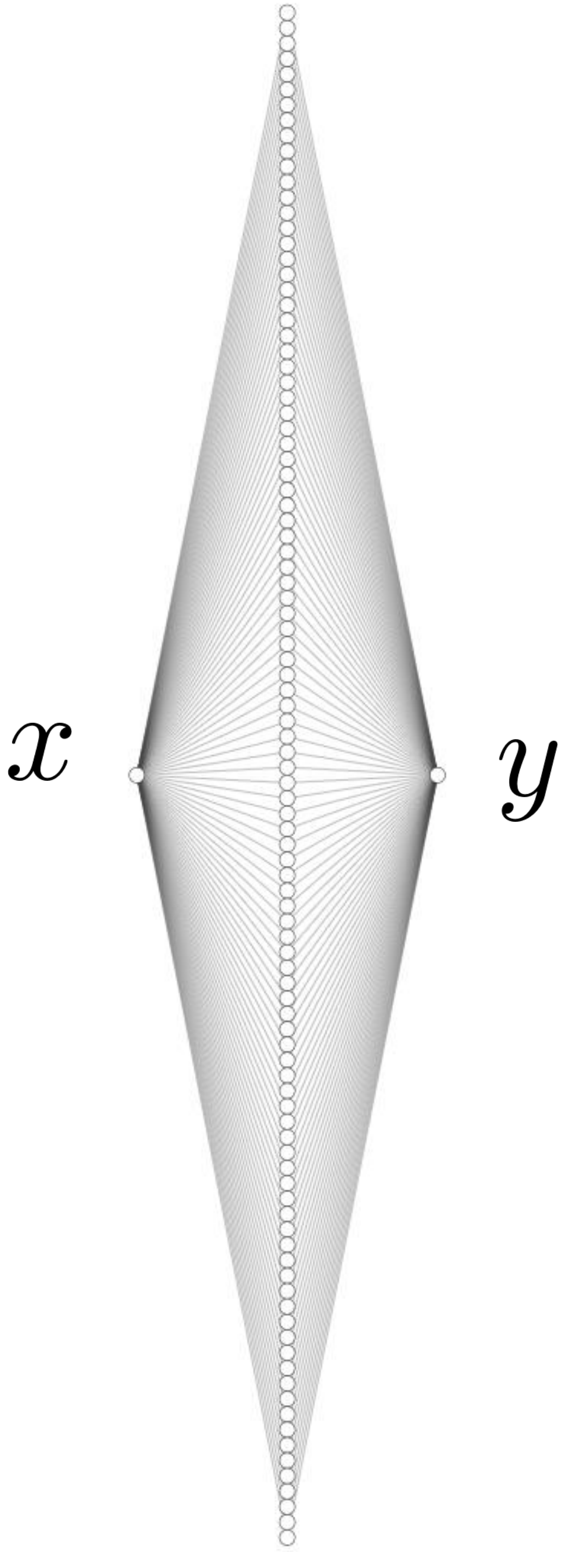
NN + Function Approximations + PyTorch

$$y = \sin(x) + \epsilon$$

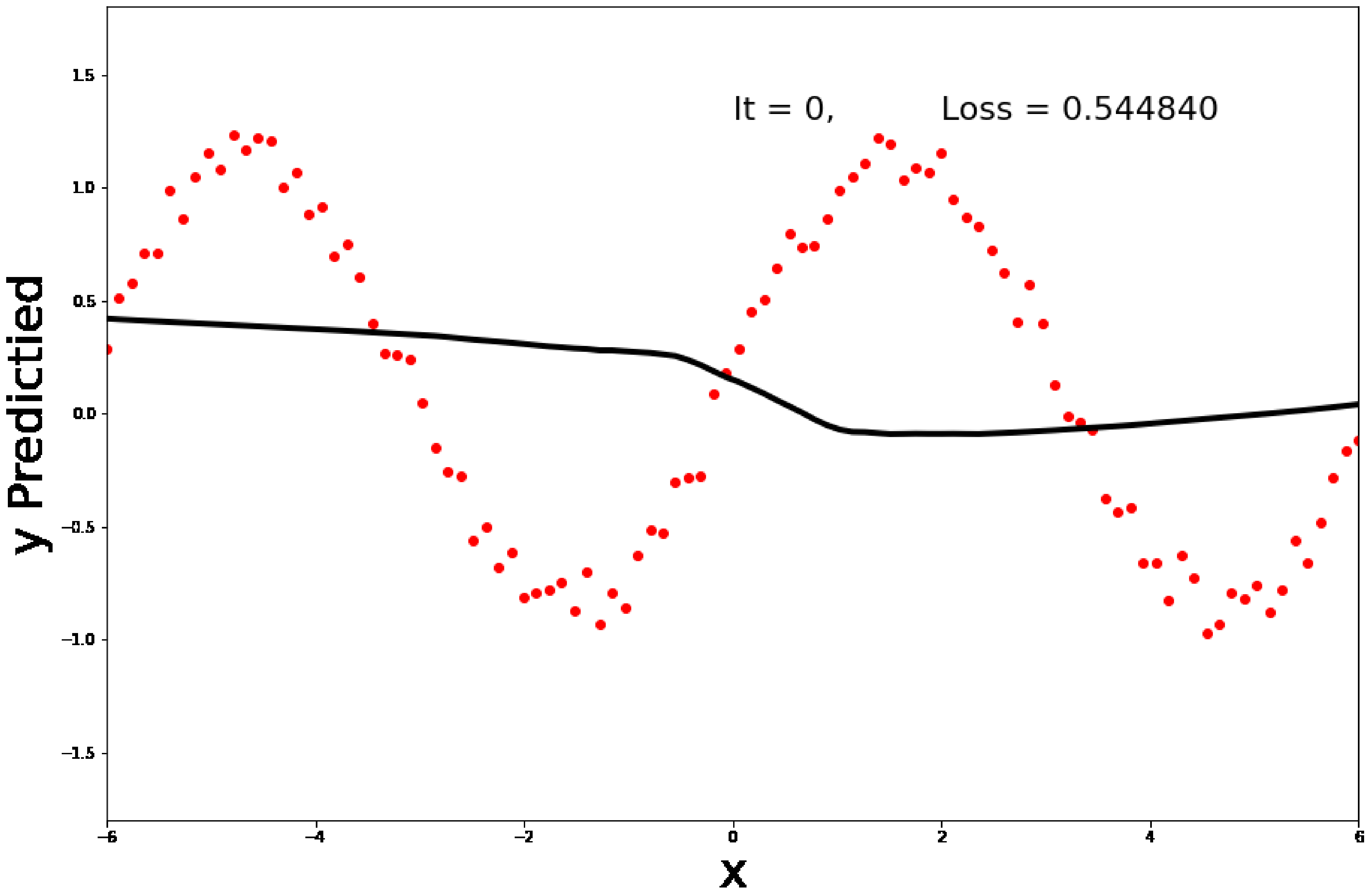
$$\epsilon \sim U[0, 1), \text{and } x \in [-6, 6]$$

$$\text{Noise} = 30\%$$



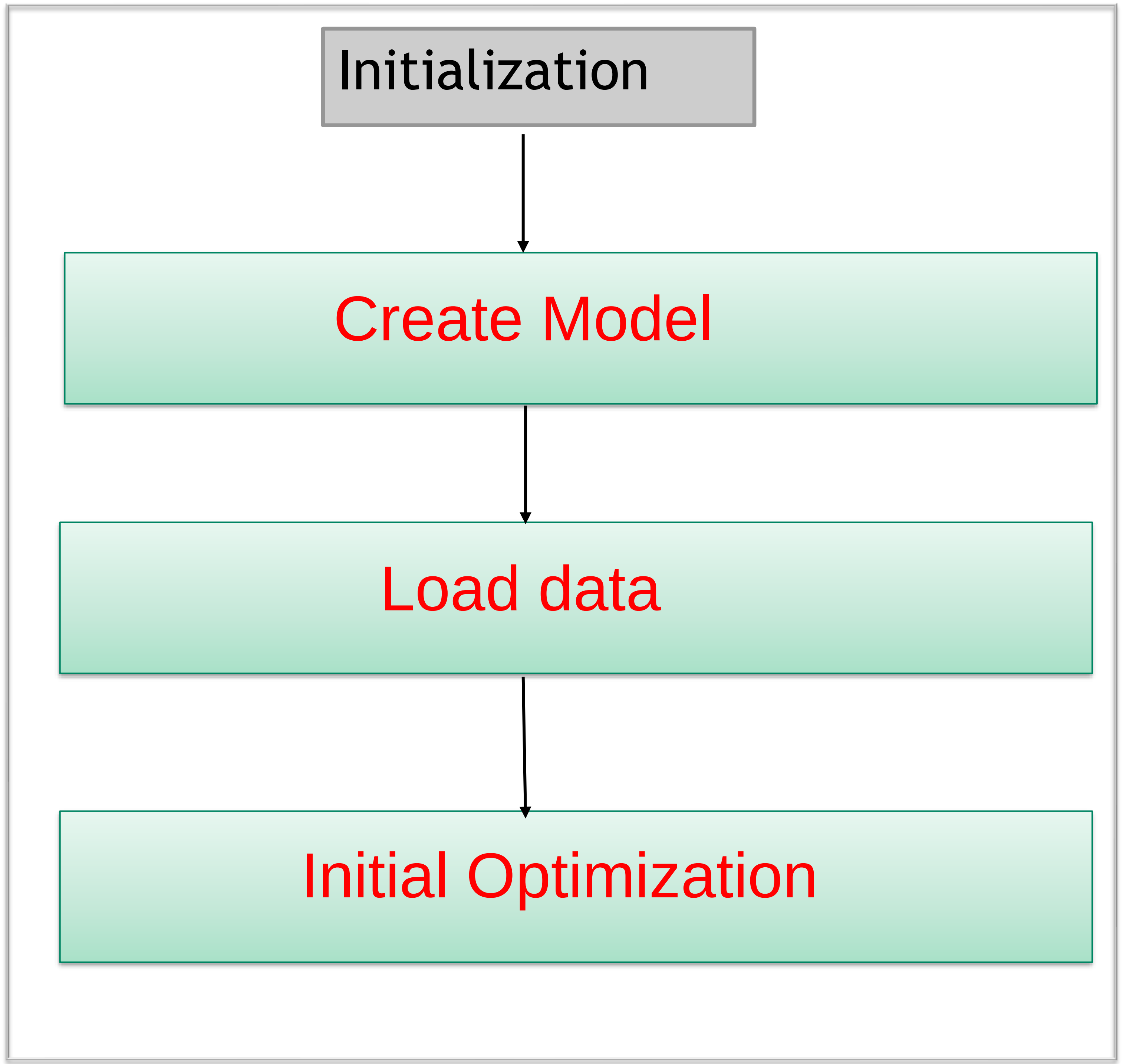


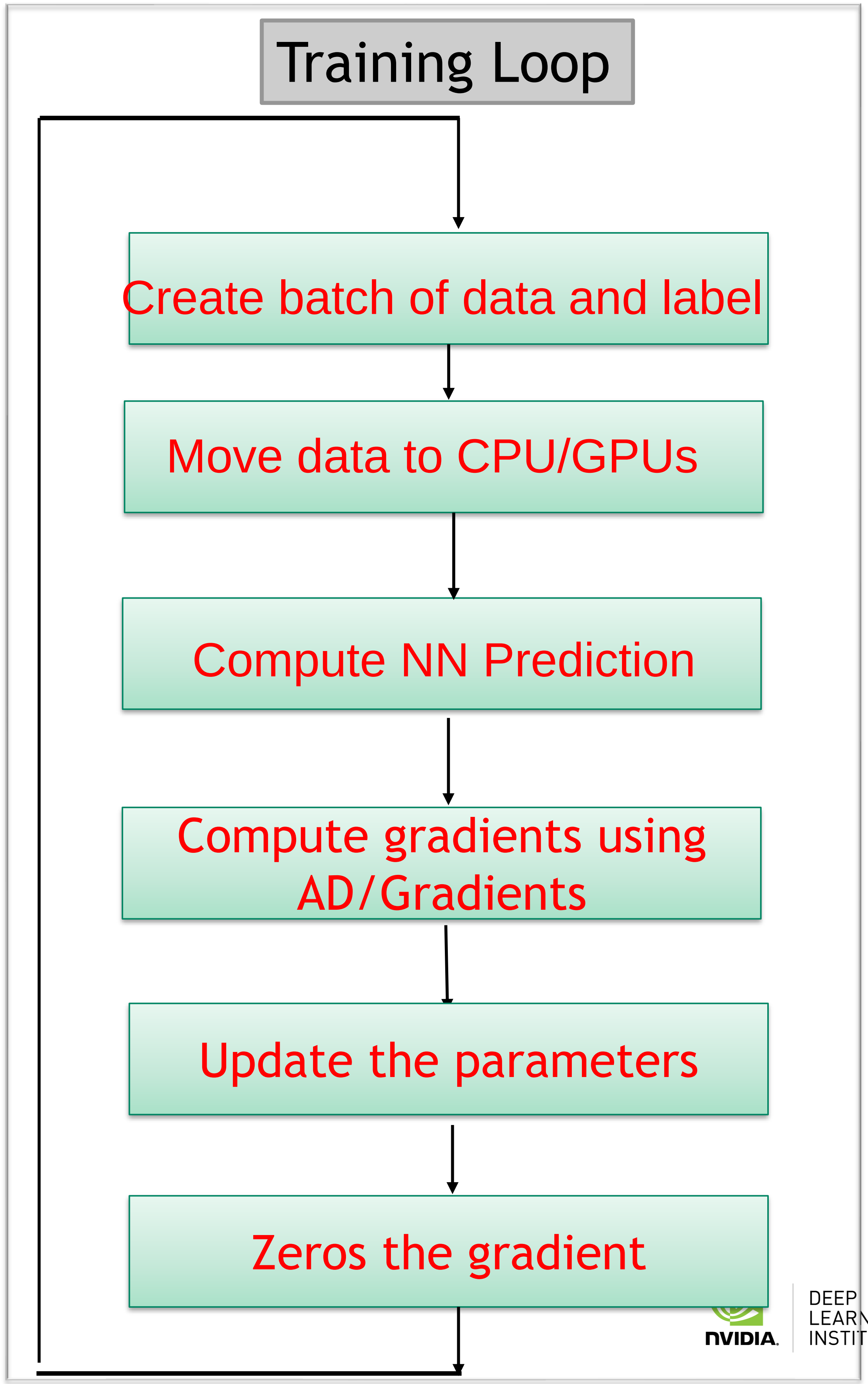
NN + Function Approximations + PyTorch



Demo: *Demonstration of Function Approximations*

Generic Code Template for Training in PyTorch





Lecture 2: Summary

- ❑ Getting familiar with programming environment of the course
- ❑Introduction of ***jupyter*** notebook and setting it up on your machine.
- ❑Basics of data structure and operation in NumpPy and SciPy
- ❑Installation of deep learning frameworks TensorFlow and PyTorch
- ❑Introduction to Nvidia's deep learning container and installation

Deep Learning for Science and Engineering Teaching Kit

Thank You