



# TUTORIAL

NTHU Computer Vision Lab

Alex Lin

# Deep Learning Frameworks



Caffe



theano



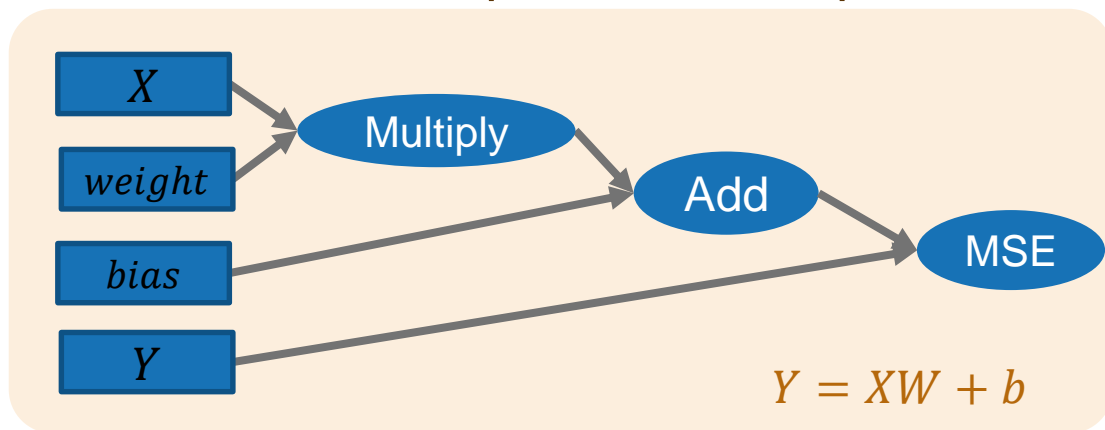
DEEPLARNING4J



# Why Using Deep Learning Frameworks?

1. Easily build big computational graphs
2. Easily compute gradients in computational graphs
3. Run on GPU to accelerate computation

Computational Graph

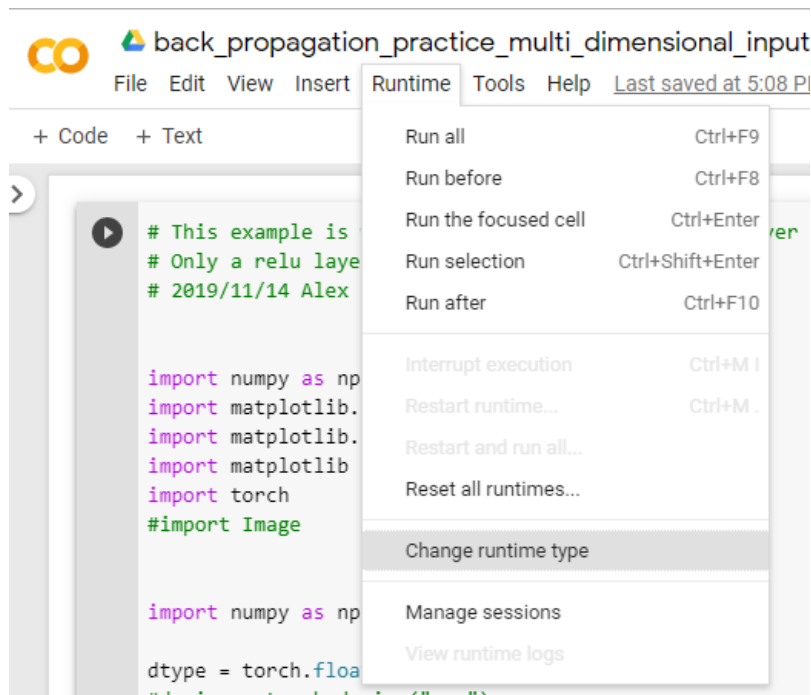


# Introduction



- PyTorch is an open source machine learning library for Python, based on Torch.
- It's developed by Facebook's artificial-intelligence research group

# Using Colab with Free “GPU”



## Notebook settings

Runtime type

Python 3

Hardware accelerator

GPU

None



☐ Omit code cell output

GPU

TPU

aving this notebook

CANCEL

SAVE

# Tutorial Documents

The screenshot shows the PyTorch website's tutorial page. The top navigation bar includes links for 'Get Started', 'Ecosystem', 'Mobile', 'Blog', 'Tutorials' (highlighted with a red dot), 'Docs', 'Resources', and 'Github'. On the left sidebar, the version '1.3.0' is circled in red, and a search bar is present. The main content area displays the title 'DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ' by 'Soumith Chintala'. Below the title is a video player with the PyTorch logo and the text 'PyTorch 60-Minute Blitz A Quick Preview'. The video player has controls for '稍後觀看' (Watch later) and '分享' (Share). Below the video, the 'Goal of this tutorial:' section lists two bullet points: 'Understand PyTorch's Tensor library and neural networks at a high level.' and 'Train a small neural network to classify images'. A note at the bottom states 'This tutorial assumes that you have a basic familiarity of numpy'.

PyTorch

Get Started Ecosystem Mobile Blog **Tutorials** Docs Resources Github

1.3.0

Search Tutorials

Getting Started

Deep Learning with PyTorch: A 60 Minute Blitz  
Writing Custom Datasets, DataLoaders and Transforms  
Visualizing Models, Data, and Training with TensorBoard

Image

TorchVision Object Detection Finetuning Tutorial  
Transfer Learning for Computer Vision Tutorial  
Spatial Transformer Networks Tutorial  
Neural Transfer Using PyTorch  
Adversarial Example Generation  
DCGAN Tutorial

Audio

torchaudio Tutorial

Tutorials > Deep Learning with PyTorch: A 60 Minute Blitz

## DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ

Author: Soumith Chintala

PyTorch Tutorial: A Quick Preview

稍後觀看 分享

PyTorch 60-Minute Blitz  
A Quick Preview

Goal of this tutorial:

- Understand PyTorch's Tensor library and neural networks at a high level.
- Train a small neural network to classify images

*This tutorial assumes that you have a basic familiarity of numpy*

[https://pytorch.org/tutorials/beginner/deep\\_learning\\_60min\\_blitz.html](https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)

# PyTorch Tensors vs Numpy arrays

- Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

# Converting a Torch Tensor to a NumPy Array

Converting a Torch Tensor to a NumPy Array

```
a = torch.ones(5)
print(a)
```

Out:

```
tensor([1., 1., 1., 1., 1.])
```

```
b = a.numpy()
print(b)
```

Out:

```
[1. 1. 1. 1. 1.]
```

See how the numpy array changed in value.

```
a.add_(1)
print(a)
print(b)
```

Out:

```
tensor([2., 2., 2., 2., 2.])
[2. 2. 2. 2. 2.]
```

The Torch Tensor and NumPy array will share their underlying memory locations (if the Torch Tensor is on **CPU**), and changing one will change the other.



# Converting NumPy Array to Torch Tensor

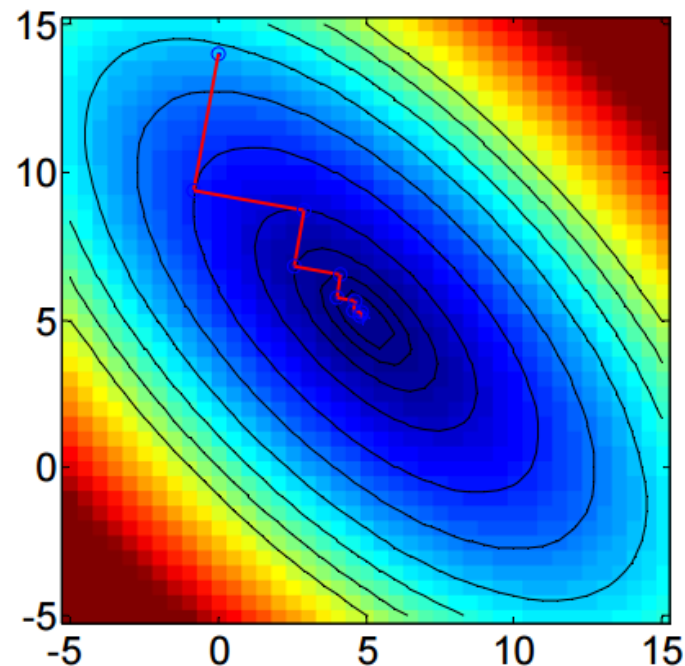
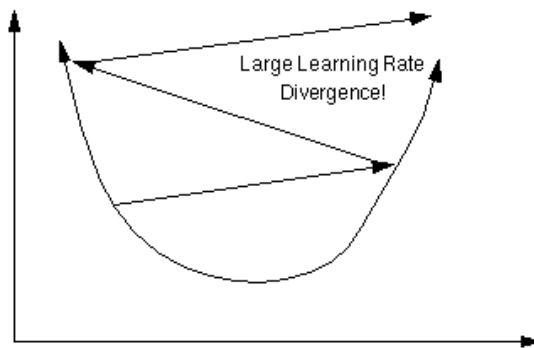
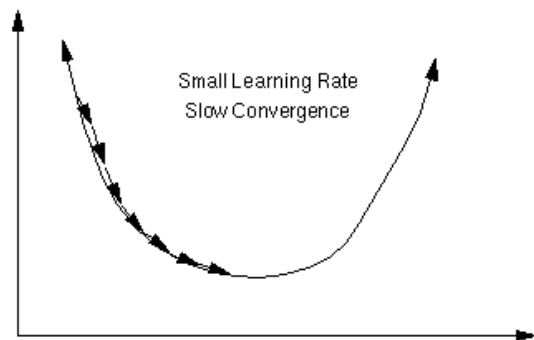
See how changing the np array changed the Torch Tensor automatically

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

Out:

```
[2.  2.  2.  2.  2.]
tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
```

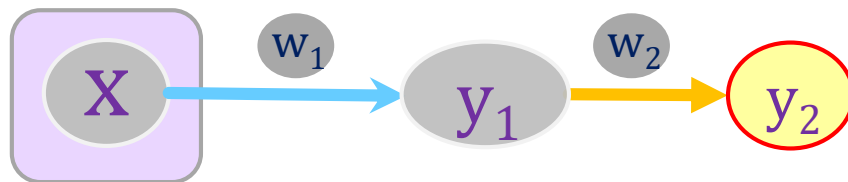
# How Gradient Descent actually works?



# Goal

- In the following examples, you will learn
  - How numpy array is helpful in doing forward/backward pass
  - Why Python is modular, high level....etc.,
  - How gradient flow is related to back-propagation
  - How Neural Nets actually work
  - How Relu works and when it is dead
  - How back-propagation is actually done with and without mini-batch.
  - Autograd provided by PyTorch is so convenient

# 1<sup>st</sup> example: two layers with 1 input and 1 output



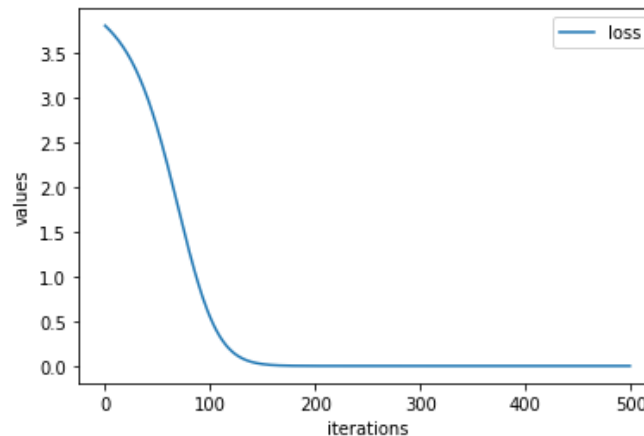
- There is a relu in  $y_1$
- $y_1 = w_1 x$  and  $y_2 = w_2 y_1$
- In the learning process, both  $w_1$  and  $w_2$  are adjusted in hope that  $y_2$  approaches its ground-truth  $\bar{y}_2$ .
- Here, we adopt 2<sup>nd</sup> norm for the loss function.
- Loss =  $(\bar{y}_2 - y_2)^2$ .
- By chain rule,  $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y_2} \frac{\partial y_2}{\partial w_2} = 2(\bar{y}_2 - y_2) y_1$ ,  $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_2} \frac{\partial y_2}{\partial y_1} \frac{\partial y_1}{\partial w_1} = 2(\bar{y}_2 - y_2) w_2 x$
- $w_2 = w_2 - \alpha \frac{\partial L}{\partial w_2}$ ,  $w_1 = w_1 - \alpha \frac{\partial L}{\partial w_1}$  where  $\alpha$  = learning rate

# Python code

```
for t in range(iterations):
    # 1st layer inference
    y1 = x.dot(w1)
    # doing relu for the output of 1st layer
    y1_relu = np.maximum(y1, 0)
    # 2nd layer inference
    y2_pred = y1_relu.dot(w2)
    # Compute the loss
    loss = np.square(y2_pred - y2_GT).sum()

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y2_pred = 2.0 * (y2_pred - y2_GT) # d_loss/d_y2
    grad_w2 = y1_relu.dot(grad_y2_pred) # d_loss/d_w2 = (d_loss/d_y2)*(d_y2/d_w2)
    grad_y1_relu = grad_y2_pred.dot(w2) # d_loss/d_y1 = (d_loss/d_y2)*(d_y2/d_y1)
    grad_y1 = grad_y1_relu.copy()
    grad_y1[y1 < 0] = 0 # only weightings through relu would be conducted back pass
    grad_w1 = x.dot(grad_y1) # d_loss/d_w1=(d_loss/d_y2)*(d_y2/d_y1)*(d_y1/d_w1)=(d_loss/d_y1)*(d_y1/d_w1)
    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

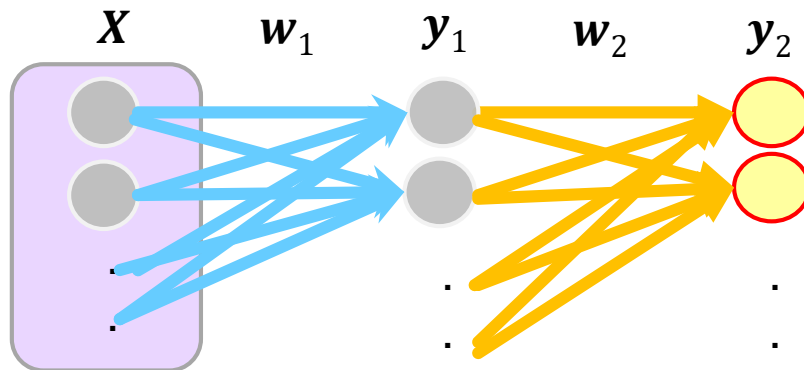
initial input=0.500000  
goal of learning=2.000000



# Discussion

- In what situation would this neural net fail?

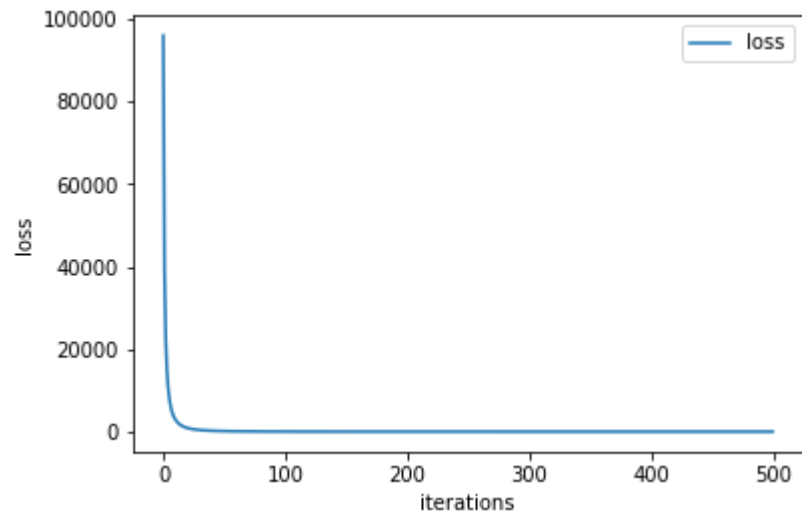
## 2<sup>nd</sup> example: Two layers with multiple-dimensional input and output



- Every neuron in  $y_1$  and  $y_2$  accompanies a relu
- $\mathbf{x}$  (1-by-k),  $\mathbf{y}_1$  (1-by-n) and  $\mathbf{y}_2$  (1-by-m) are vectors;  $\mathbf{w}_1$  (k-by-n) &  $\mathbf{w}_2$  (n-by-m) are matrices.
- $\mathbf{y}_1 = \mathbf{x}\mathbf{w}_1$  and  $\mathbf{y}_2 = \mathbf{y}_1\mathbf{w}_2$
- In the learning process, both  $\mathbf{w}_1$  and  $\mathbf{w}_2$  are adjusted in hope that  $\mathbf{y}_2$  approaches its ground-truth  $\bar{\mathbf{y}}_2$ .
- Here, we adopt 2<sup>nd</sup> norm for the loss function.
- Loss =  $(\bar{\mathbf{y}}_2 - \mathbf{y}_2)^2$ .
- By chain rule,  $\frac{\partial L}{\partial \mathbf{w}_2} = \frac{\partial \mathbf{y}_2}{\partial \mathbf{w}_2} \frac{\partial L}{\partial \mathbf{y}_2} = 2\mathbf{y}_1^t (\bar{\mathbf{y}}_2 - \mathbf{y}_2)$ ,  $\frac{\partial L}{\partial \mathbf{w}_1} = \frac{\partial \mathbf{y}_1}{\partial \mathbf{w}_1} \frac{\partial L}{\partial \mathbf{y}_2} \frac{\partial \mathbf{y}_2}{\partial \mathbf{y}_1} = 2\mathbf{x}^t (\bar{\mathbf{y}}_2 - \mathbf{y}_2) \mathbf{w}_2^t$
- $\mathbf{w}_2 = \mathbf{w}_2 - \alpha \frac{\partial L}{\partial \mathbf{w}_2}$ ,  $\mathbf{w}_1 = \mathbf{w}_1 - \alpha \frac{\partial L}{\partial \mathbf{w}_1}$  where  $\alpha$  = learning rate

# Python code

```
for t in range(iterations):
    # 1st layer inference
    y1 = x.dot(w1)
    # doing relu for the output of 1st layer
    y1_relu = np.maximum(y1, 0) # result is a row vector
    # store the output of the 1st layer
    y1_history[t]= np.mean(y1_relu)
    # performing 2nd layer computation
    y2_pred = y1_relu.dot(w2) # result is a row vector
    # Compute and print loss
    loss = np.square(y2_pred - y).sum()
    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y2_pred = 2.0 * (y2_pred - y) # d_loss/d_y2
    grad_w2 = y1_relu.T.dot(grad_y2_pred) # (d_y2/d_w2)*(d_loss/d_y2)=d_loss/d_w2
    grad_y1_relu = grad_y2_pred.dot(w2.T) # (d_loss/d_y2)*(d_y2/d_y1)=d_loss/d_y1
    grad_y1 = grad_y1_relu.copy()
    grad_y1[y1 < 0] = 0 # only numbers through relu would be conducted backward pass
    grad_w1 = x.T.dot(grad_y1) # (d_y1/d_w1)*(d_loss/d_y2)*(d_y2/d_y1)=(d_y1/d_w1)*(d_loss/d_y1)=d_loss/d_w1
    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

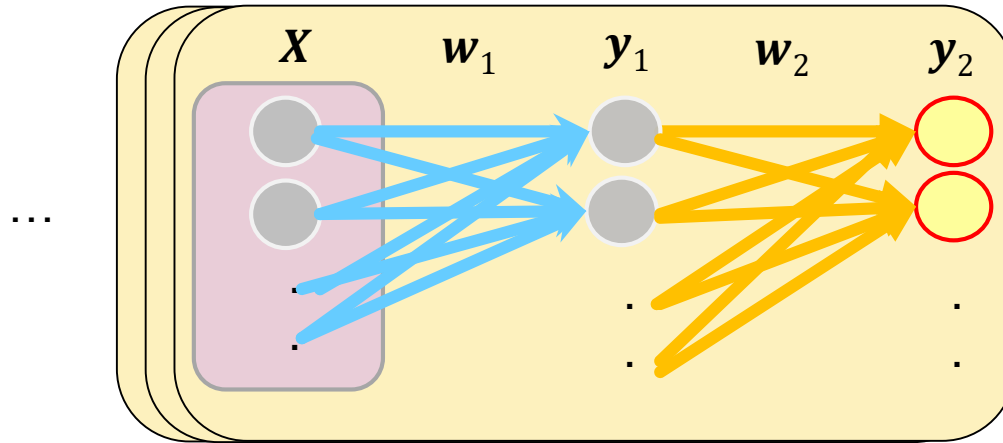




# Discussion

- What are the outputs if dead Relu happens?

### 3<sup>rd</sup> example: Two layers with multiple-dimensional inputs and outputs (mini-batch)



- Every neuron in  $y_1$  and  $y_2$  accompanies a relu
- $\mathbf{x}$  (N-by-k),  $\mathbf{y}_1$  (N-by-n) and  $\mathbf{y}_2$  (N-by-m) are vectors;  $\mathbf{w}_1$  (k-by-n) &  $\mathbf{w}_2$  (n-by-m) are matrices.
- $\mathbf{y}_1 = \mathbf{x}\mathbf{w}_1$  and  $\mathbf{y}_2 = \mathbf{y}_1\mathbf{w}_2$
- In the learning process, both  $\mathbf{w}_1$  and  $\mathbf{w}_2$  are adjusted in hope that  $\mathbf{y}_2$  approaches its ground-truth  $\bar{\mathbf{y}}_2$ .
- Here, we adopt 2<sup>nd</sup> norm for the loss function.
- Loss =  $(\bar{\mathbf{y}}_2 - \mathbf{y}_2)^2$ .
- By chain rule,  $\frac{\partial L}{\partial \mathbf{w}_2} = \frac{\partial \mathbf{y}_2}{\partial \mathbf{w}_2} \frac{\partial L}{\partial \mathbf{y}_2} = 2\mathbf{y}_1^t (\bar{\mathbf{y}}_2 - \mathbf{y}_2)$ ,  $\frac{\partial L}{\partial \mathbf{w}_1} = \frac{\partial \mathbf{y}_1}{\partial \mathbf{w}_1} \frac{\partial L}{\partial \mathbf{y}_2} \frac{\partial \mathbf{y}_2}{\partial \mathbf{y}_1} = 2\mathbf{x}^t (\bar{\mathbf{y}}_2 - \mathbf{y}_2) \mathbf{w}_2^t$
- $\mathbf{w}_2 = \mathbf{w}_2 - \alpha \frac{\partial L}{\partial \mathbf{w}_2}$ ,  $\mathbf{w}_1 = \mathbf{w}_1 - \alpha \frac{\partial L}{\partial \mathbf{w}_1}$  where  $\alpha$  = learning rate

# Python code

```
for t in range(iterations):
    # 1st layer inference
    y1 = x.dot(w1)
    # doing relu for the output of 1st layer
    y1_relu = np.maximum(y1, 0) # result is a matrix
    # store the output of the 1st layer
    y1_history[t] = np.mean(y1_relu)
    # performing 2nd layer computation
    y2_pred = y1_relu.dot(w2) # result is a row vector
    # Compute and print loss
    loss = np.square(y2_pred - y2_GT).sum()
    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y2_pred = 2.0 * (y2_pred - y2_GT) # d_loss/d_y2
    grad_w2 = y1_relu.T.dot(grad_y2_pred) # (d_y2/d_w2)*(d_loss/d_y2)=d_loss/d_w2
    grad_y1_relu = grad_y2_pred.dot(w2.T) # (d_loss/d_y2)*(d_y2/d_y1)=d_loss/d_y1
    grad_y1 = grad_y1_relu.copy()
    grad_y1[y1 < 0] = 0 # only numbers through relu would be conducted backward pass
    grad_w1 = x.T.dot(grad_y1) # (d_y1/d_w1)*(d_loss/d_y2)*(d_y2/d_y1)=(d_y1/d_w1)*(d_loss/d_y1)=d_loss/d_w1
    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

# Autograd is powerful

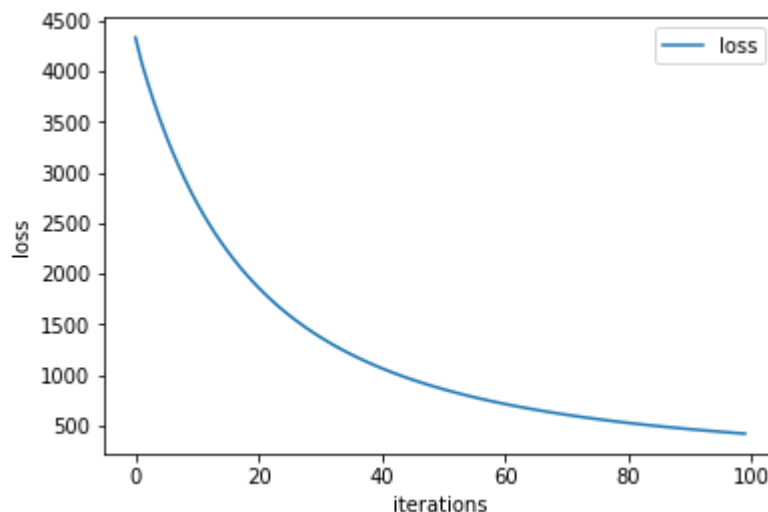
- Derivation of back-propagation is prone to fail
- PyTorch provides Autograd for Tensors!

# 4<sup>th</sup> example: using Autograd in the 3<sup>rd</sup> example

```
dtype = torch.float
device = torch.device("cuda") # device = torch.device("cpu")
x = torch.ones(10,10, device=device, dtype=dtype)
y2_GT = torch.randn(10, 10, device=device, dtype=dtype)
```

```
for t in range(iterations):
    # 1st layer inference
    y1 = x.mm(w1)
    # doing relu for the output of 1st layer
    y1_relu = y1.clamp(min=0) # result is a matrix
    # performing 2nd layer computation
    y2_pred = y1_relu.mm(w2)
    # Compute and print loss
    loss = (y2_pred - y2_GT).pow(2).sum()
    loss.backward()
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

    # Manually zero the gradients after updating weights
    w1.grad.zero_()
    w2.grad.zero_()
```



# Discussion

- Is lower learning rate always better?
- How can you manually produce dead Relu or gradient explosion in terms of hyperparameters?
- Could initial weightings be 0?
- How could we determine “appropriate” initial weightings?

Thank you!