PYTORCH

TUTORIAL

NTHU Computer Vision Lab

Alex Lin

Deep Learning Frameworks

TensorFlow

Chainer

Keras

theano

PYTORCH

torch

Microsoft CNTK

Caffe2

dmlc mxnet

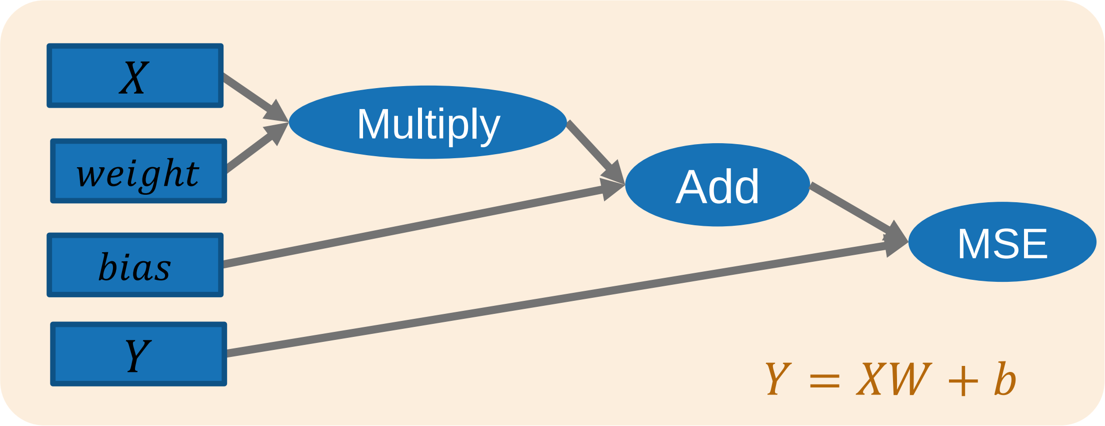
Caffe

DEEPLEARNING4J

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

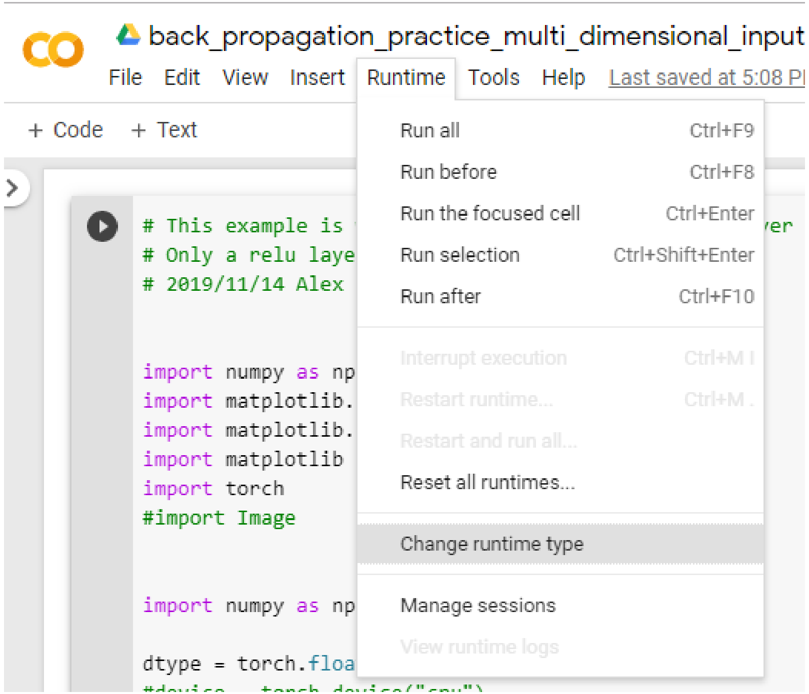


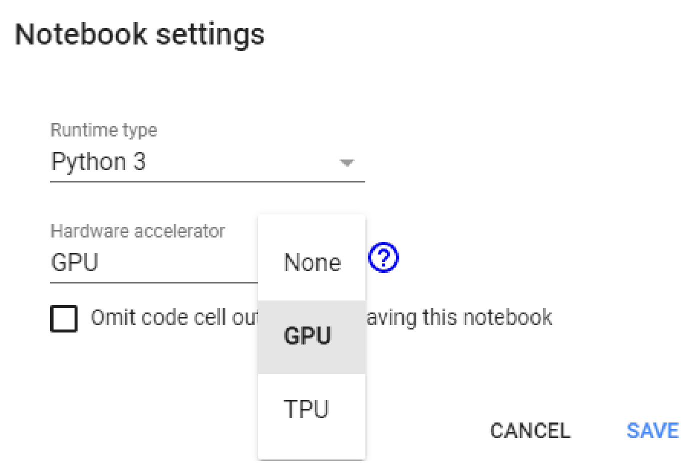
Introduction

- PyTorch is an open source machine learning library for Python, based on Torch.

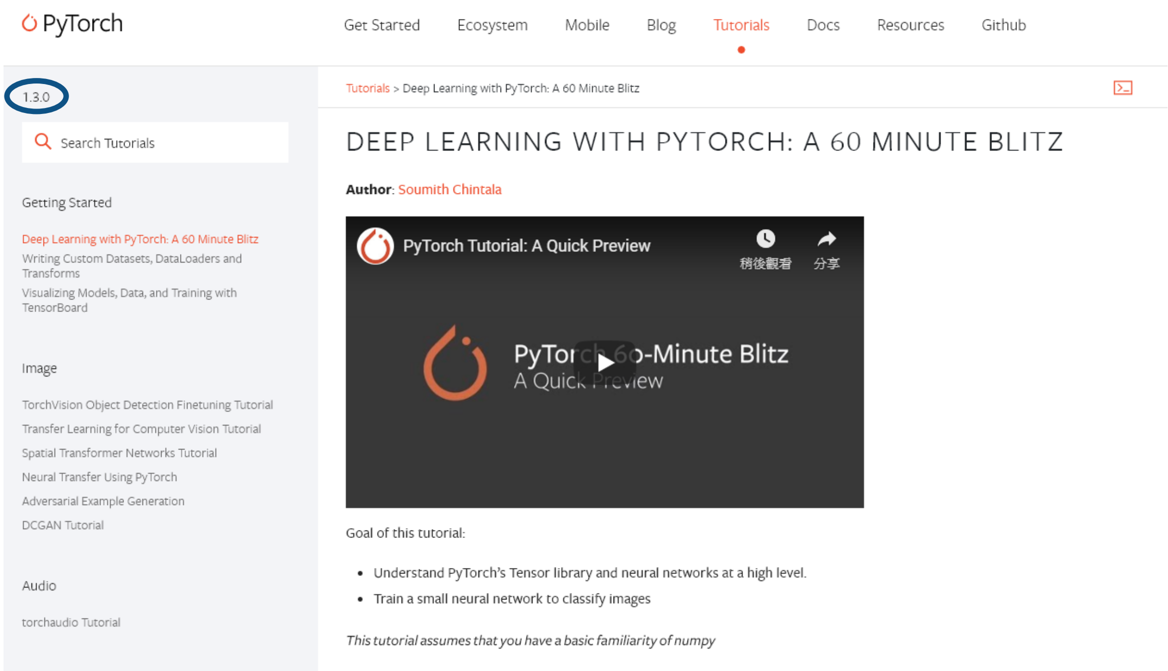
- It’s developed by Facebook's artificial-intelligence research group

Using Colab with Free “GPU”





Tutorial Documents



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

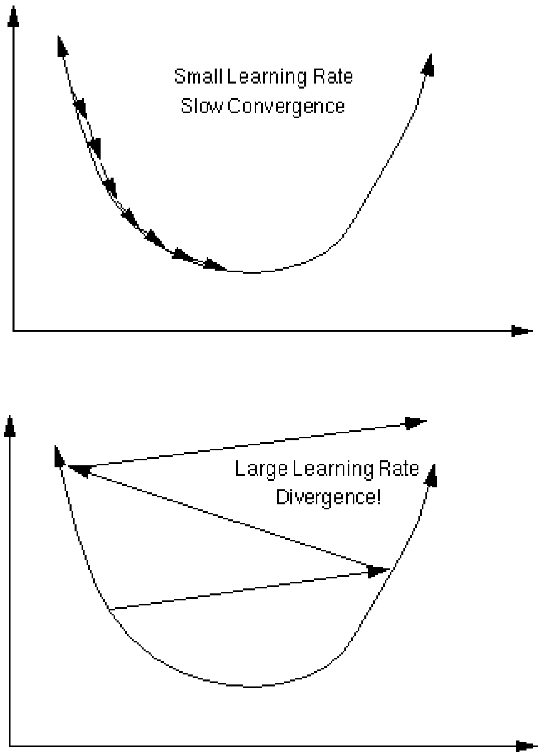
See how changing the np array changed the Torch Tensor automatically

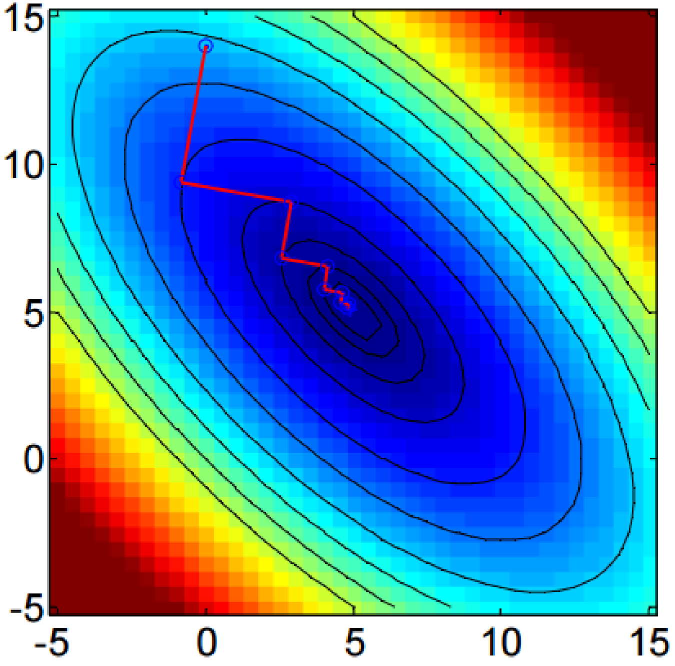
```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

Out:

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

How Gradient Descent actually works?

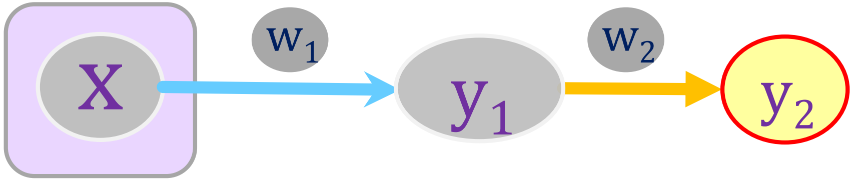




Goal

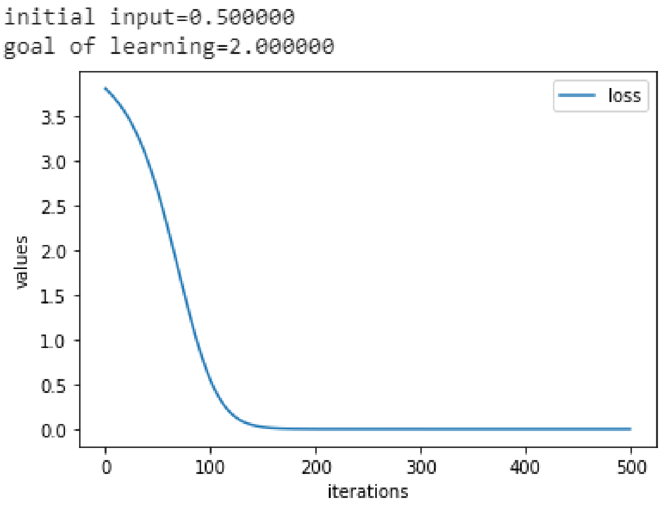
- In the following examples, you will learn
    - How numpy array is helpful in doing forward/backward pass
    - Why Python is modular, high level….etc.,
    - How gradient flow is related to back-propagation
    - How Neural Nets actually work
    - How Relu works and when it is dead
    - How back-propagation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

1st 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 $\overline{y}_2$.
- Here, we adopt 2nd norm for the loss function.
- Loss= $(\overline{y}_2 - y_2)^2$.
- By chain rule, $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial w_2} = 2(\overline{y}_2 - y_2)\ y_1$, $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial y_1}\frac{\partial y_1}{\partial w_1} = 2(\overline{y}_2 - y_2)\ w_2 x$
- $w_2 = w_2 - \alpha\frac{\partial L}{\partial w_2}$, $w_1 = w_1 - \alpha\frac{\partial L}{\partial w_1}$ where α = learning rate

Python code



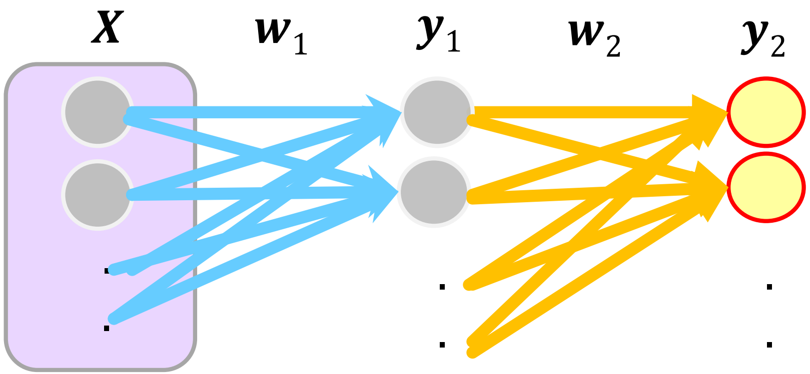

```
for t in range(iterations):
    # 1st layer inference
    y1 = x.dot(w1)
    # doing relu for the output of 1st layer
    y1_relu = np.maximum(y1, 0)
    # 2nd layer inference
    y2_pred = y1_relu.dot(w2)
    # Compute the loss
    loss = np.square(y2_pred - y2_GT).sum()

    # 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
```

Discussion

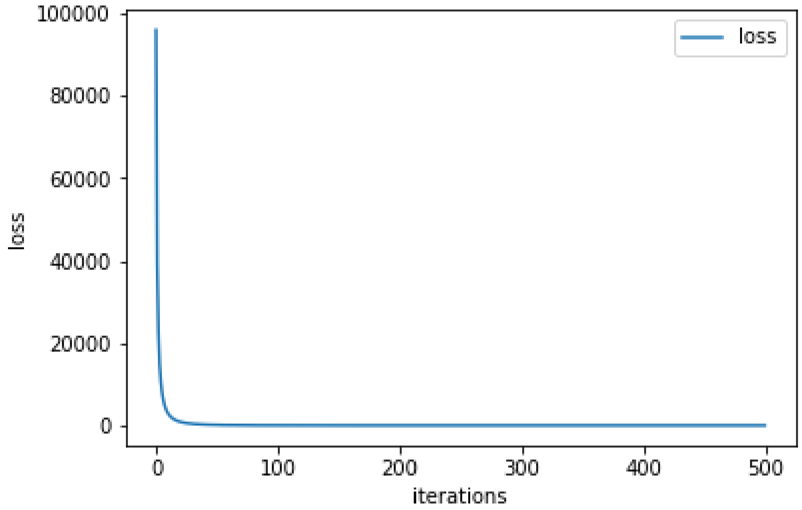
- In what situation would this neural net fail?

2^{nd} example: Two layers with multiple-dimensional input and output



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

Python code

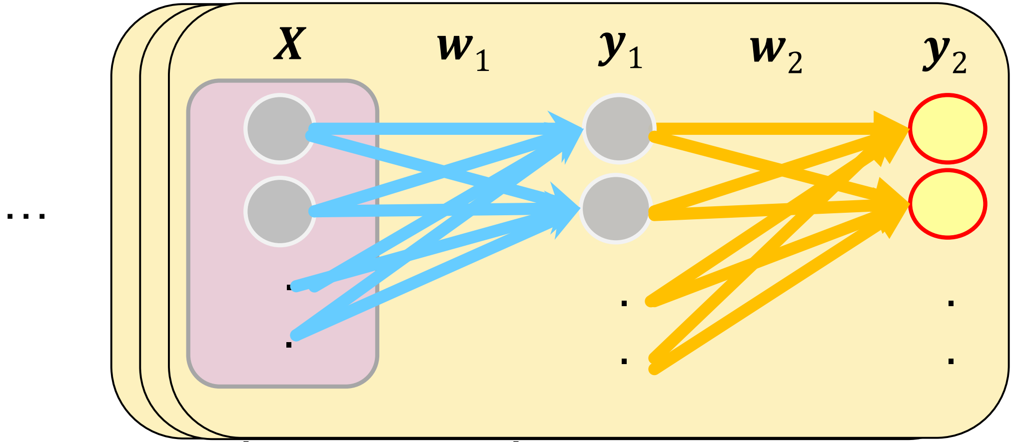



```
for t in range(iterations):
    # 1st layer inference
    y1 = x.dot(w1)
        # doing relu for the output of 1st layer
    y1_relu = np.maximum(y1, 0) # result is a row vector
    # store the output of the 1st layer
    y1_history[t]= np.mean(y1_relu)
    # performing 2nd layer computation
    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?

3rd example: Two layers with multiple-dimensional inputs and outputs (mini-batch)



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

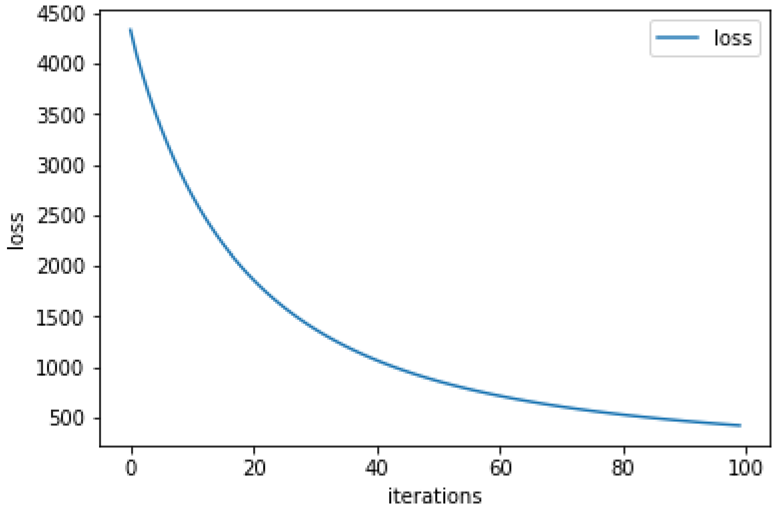
- Derivation of back-propagation is prone to fail
- PyTorch provides Autograd for Tensors!

4th example: using Autograd in the 3rd 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!