Aprendizado Profundo

Frameworks de Desenvolvimento de Redes Neurais - PyTorch

Professor: Lucas Silveira Kupssinskü

Agenda

- *Hardware* (o mínimo do mínimo que precisamos saber)
- Frameworks de Desenvolvimento de Redes Neurais
- PyTorch
  - Tensores
  - Operações básicas
  - *Autograd*
  - Regressão Linear Simples
  - Camadas Lineares
  - Otimizadores
  - Funções de Custo
  - Laço de Treino e Laço de Teste

Hardware

- Quando treinamos redes neurais, alguns processamentos ocorrem na CPU enquanto outros ocorrem na GPU

Hardware

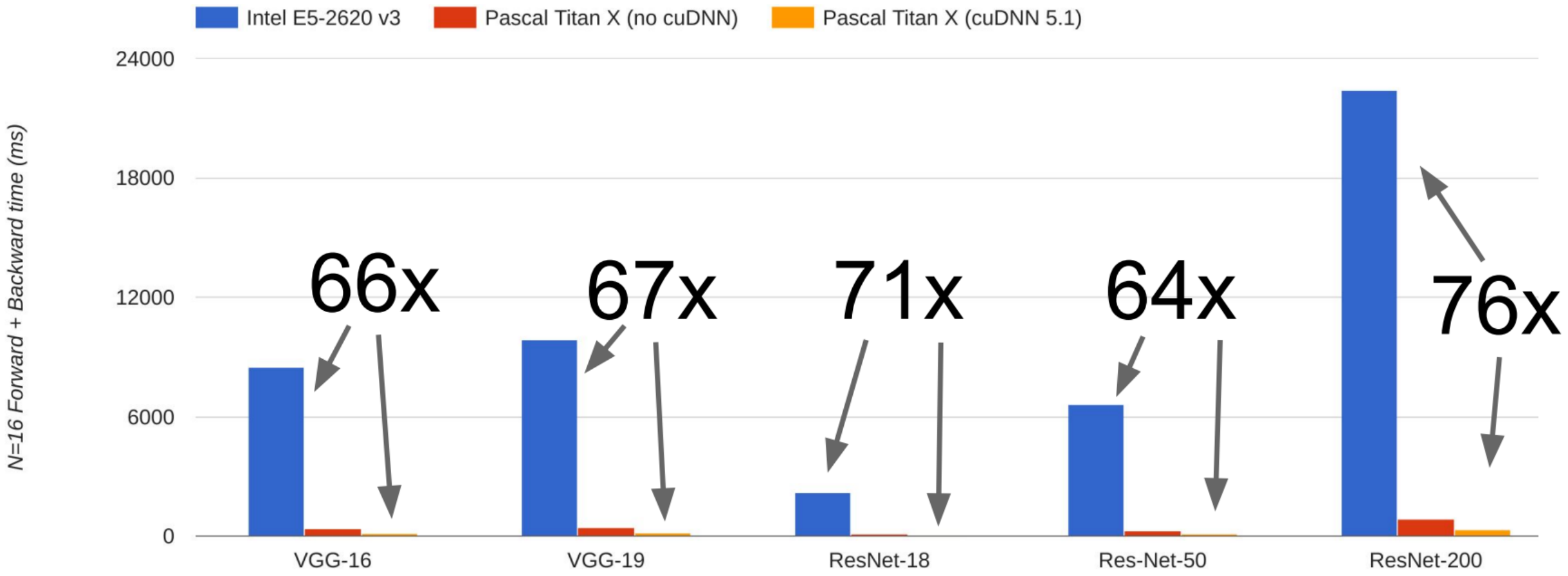
	Cores	Clock Speed	Memory	Preço	Velocidade
i9-14900K	24 (8 performance, 16 efficiency)	3.2GHz	System RAM	$590,00	1,95 TFLOPS
NVIDIA H100	18.432 cores	1,125GHz	80GB HBM3	$43000,00	~360 TFLOPs (FP32)

- CPUs tem menos *cores* porém cada um é muito rápido
  - Ótimo para tarefas sequenciais.
- GPUs tem muitos *cores* mas cada um deles é muito mais lento e limitado
  - Ótimo para executar diversas pequenas operações em paralelo

Para desenvolver para GPUs

- Opções para criar programas que rodam operações na GPU
- CUDA (NVIDIA)
  - Código C-like que roda diretamente na GPU
  - APIs: cuBLAS, cuFFT, cuDNN,...
- OpenCL
  - Similar ao CUDA mas é OpenSource
- HIP
  - https://github.com/ROCm/HIP

Hardware



Data from https://github.com/jcjohnson/cnn-benchmarks

Software melhor também ajuda

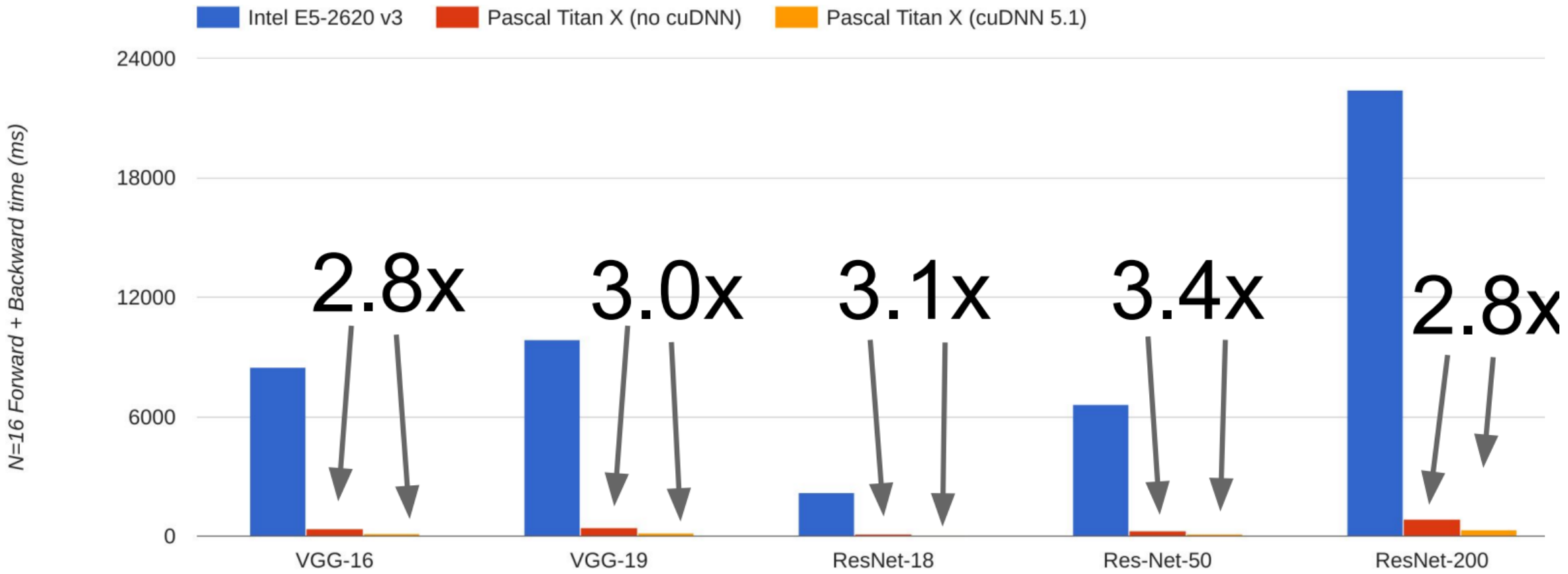
NVIDIA cuDNN

The NVIDIA CUDA® Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, attention, matmul, pooling, and normalization.

cuDNN Accelerated Frameworks

cuDNN accelerates widely used deep learning frameworks, including Caffe2, Chainer, Keras, MATLAB, MxNet, PaddlePaddle, PyTorch, and TensorFlow.


Hardware



Data from https://github.com/jcjohnson/cnn-benchmarks

Comparação usando cuDNN ou não

Hardware

- O modelo fica na GPU
- Os dados ficam no dispositivo de armazenamento
- As vezes o gargalo do treinamento pode ser na leitura dos dados e carga na GPU
- Nesses casos:
  - Carregar os dados na RAM
  - SSDs
  - Usar várias threads (*workers*) para carregar os dados

Frameworks

TensorFlow

Keras

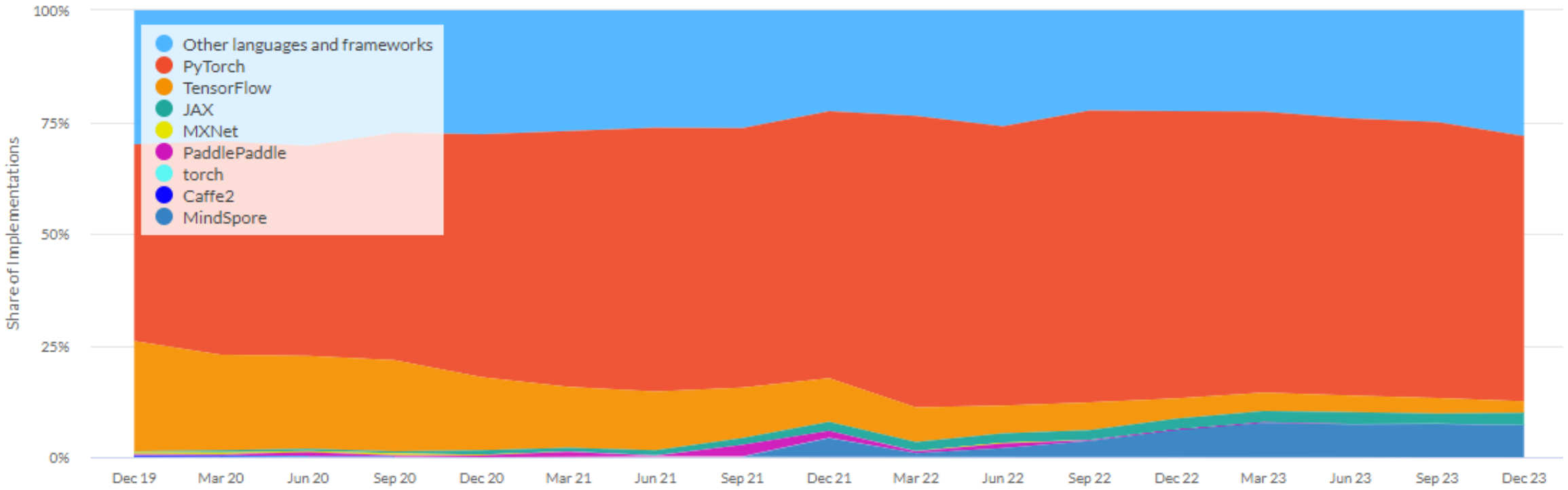
PyTorch

PyTorch Lightning

Por que PyTorch?

Frameworks

Paper Implementations grouped by framework



PyTorch

- Por que não fazemos tudo em numpy?
  - + Código é limpo e fácil de escrever
  - - Derivadas tem que ser controladas na mão
  - - Não roda em GPU

```
import numpy as np
np.random.seed(0)

N, D = 3, 4
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x*y
b = a+z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c*np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a*y

print(grad_x)
```

PyTorch

- PyTorch é muito semelhante ☺

```python
import torch

N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D, requires_grad=True)
z = torch.randn(N, D, requires_grad=True)

a = x*y
b = a+z
c = torch.sum(b)

c.backward()
print(x.grad)
```

```python
import numpy as np

N, D = 3, 4
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x*y
b = a+z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c*np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a*y

print(grad_x)
```

PyTorch

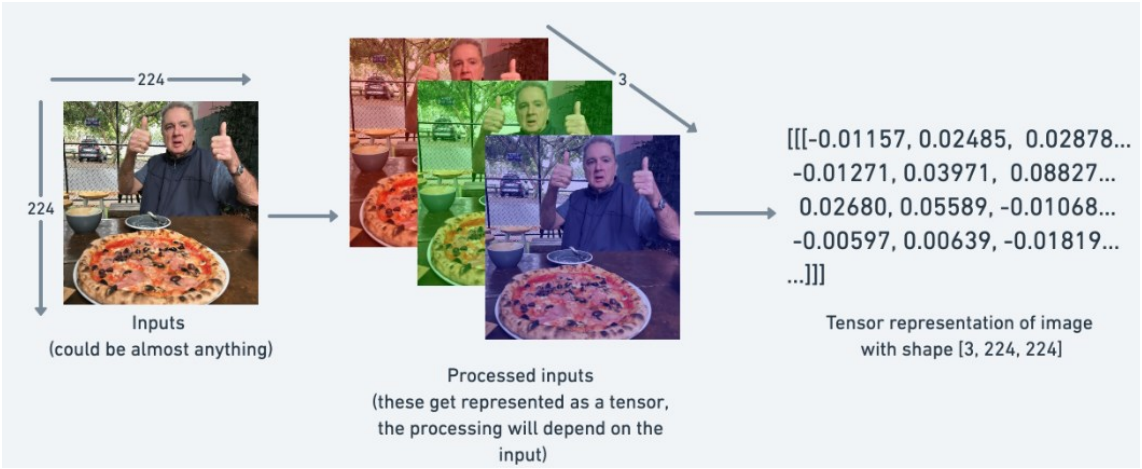
- E para rodar em GPU basta especificar o *device*

```
import torch

device='cuda:0'
N, D = 3, 4
x = torch.randn(N, D, device=device, requires_grad=True)
y = torch.randn(N, D, device=device, requires_grad=True)
z = torch.randn(N, D, device=device, requires_grad=True)

a = x*y
b = a+z
c = torch.sum(b)

c.backward()
print(x.grad)
```

```
import numpy as np

N, D = 3, 4
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x*y
b = a+z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c*np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a*y

print(grad_x)
```

PyTorch

- Um `torch.Tensor` é uma matriz multi-dimensional que contém elementos de um único tipo
- Uma imagem RGB pode ser representada por um tensor `[channels, height, width]`



https://pytorch.org/docs/stable/tensors.html

PyTorch

- Vamos criar um Tensor que representa um escalar

```
scalar = torch.tensor(7)
scalar
```

```
>>tensor(7)
```

```
scalar.ndim
```

```
>>0
```

```
# Funciona apenas para escalares
scalar.item()
```

```
>>7
```

https://pytorch.org/docs/stable/tensors.html

PyTorch

- Vamos criar um Tensor que representa um vetor

```
vector = torch.tensor([7, 7])
vector
```

```
>>tensor([7, 7])
```

```
vector.ndim
```

```
>>1
```

```
vector.shape
```

```
>>torch.Size([2])
```

https://pytorch.org/docs/stable/tensors.html

PyTorch

- Vamos criar um Tensor que representa uma matriz

```
matrix = torch.tensor([[7, 8],
                       [9, 10]])
matrix
>> tensor([[ 7, 8],
           [ 9, 10]])

matrix.ndim
>>2

matrix.shape
>>torch.Size([2,2])
```

https://pytorch.org/docs/stable/tensors.html

PyTorch

- Vamos criar um Tensor de três dimensões

```
tensor = torch.tensor([[[1, 2, 3],
                        [3, 6, 9],
                        [2, 4, 5]]])
tensor
```

```
>> tensor([[[1, 2, 3],
            [3, 6, 9],
            [2, 4, 5]]])
```

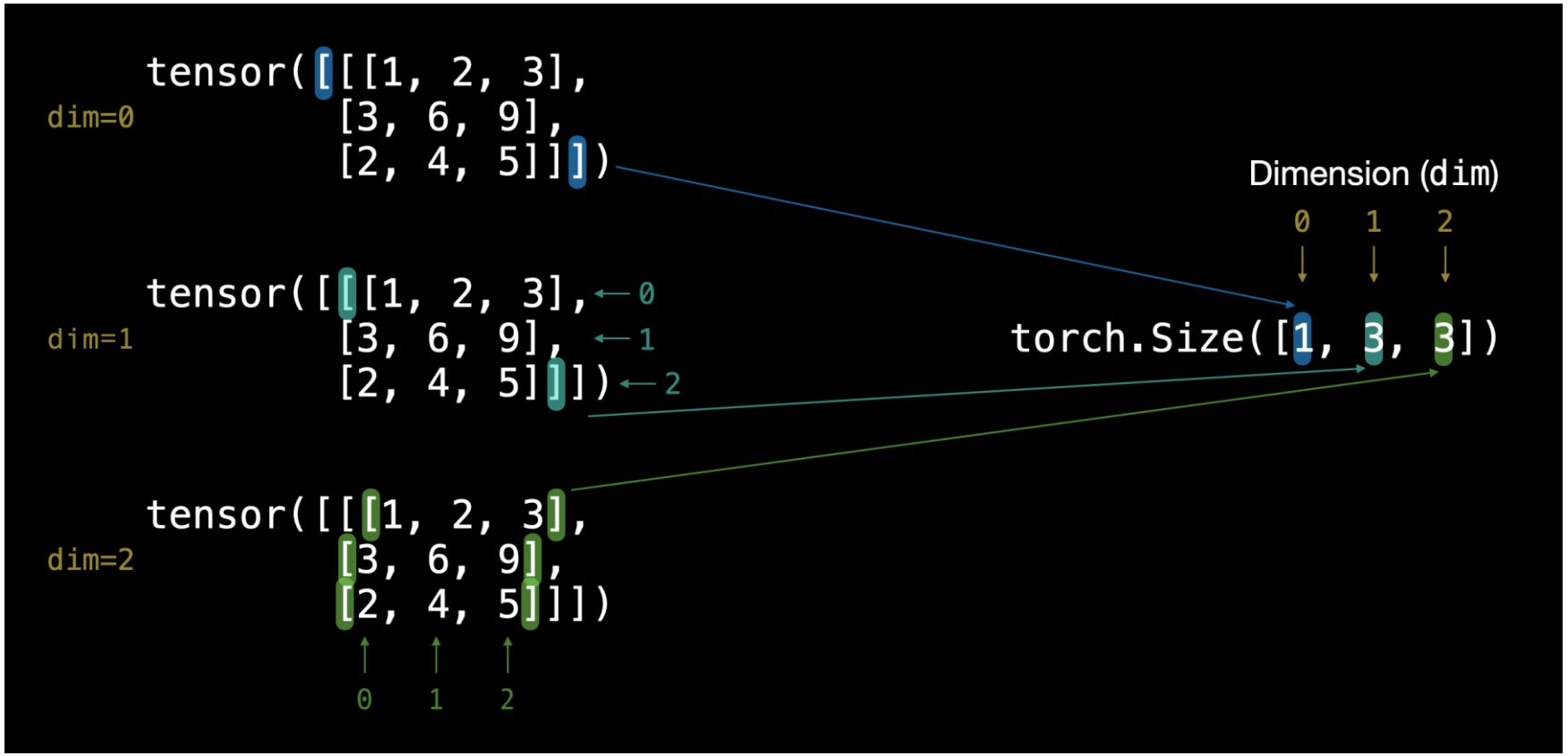
```
print(f'{tensor.ndim=}')
print(f'{tensor.shape=}')
```

```
>> tensor.ndim=3
>> tensor.shape=torch.Size([1, 3, 3])
```

https://pytorch.org/docs/stable/tensors.html

PyTorch

- Vamos criar um Tensor de três dimensões



https://pytorch.org/docs/stable/tensors.html

PyTorch

- Podemos criar Tensores com 0s, 1s, com valores pré-determinados ou com valores aleatórios

```
torch.zeros(2, 2)
```

```
tensor([[0., 0.],
        [0., 0.]])
```

```
torch.ones(2, 2)
```

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

```
torch.tensor([[1, 2],
              [3, 4]])
```

```
tensor([[1, 2],
        [3, 4]])
```

```
torch.rand(2, 2)
```

```
tensor([[0.0500, 0.9815],
        [0.6279, 0.3406]])
```

```
torch.eye(2)
```

```
tensor([[1., 0.],
        [0., 1.]])
```

https://pytorch.org/docs/stable/tensors.html

PyTorch

```
torch.matmul(input, other, *, out=None) → Tensor
```

- A principal operação em redes neurais é a multiplicação de matrizes
  - Implementada pela função `torch.matmul()`
- O comportamento é dependente da dimensionalidade dos tensores passados como argumento

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

```
torch.matmul(input, other, *, out=None) → Tensor
```

- Se ambos tensores tiverem apenas 1 dimensão e tiverem o mesmo tamanho então retorna o produto escalar

```
tensor1 = torch.rand(3)
tensor2 = torch.rand(3)
print(f'{tensor1=}', f'{tensor2=}')
print(f'{torch.matmul(tensor1, tensor2)=}')
```

```
>> tensor1=tensor([0.5186, 0.4053, 0.5219])
>> tensor2=tensor([0.6939, 0.6930, 0.9816])
>> torch.matmul(tensor1, tensor2)=tensor(1.1531)
```

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

`torch.matmul(input, other, *, out=None) → Tensor`

- Se ambos tensores tiverem apenas 2 dimensões e o número de colunas de *input* for igual as linhas de *other* então retorna o produto matriz-matriz

```
tensor1 = torch.rand(2,2)
tensor2 = torch.rand(2,3)
print(f'{tensor1=}', f'\n{tensor2=}')
print(f'{torch.matmul(tensor1, tensor2)=}')
```

```
>> tensor1=tensor([[0.8927, 0.9574],
                   [0.4500, 0.6236]])

>> tensor2=tensor([[0.2400, 0.3922, 0.6944],
                   [0.1999, 0.5105, 0.9782]])

>> torch.matmul(tensor1, tensor2)=tensor([[0.4056, 0.8389, 1.5563],
                                          [0.2326, 0.4948, 0.9224]])
```

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

```
torch.matmul(input, other, *, out=None) → Tensor
```

- Caso o *input* tenha 1 dimensão e o *other* tenha duas, então adiciona uma dimensão em *input* para possibilitar o produto matriz-matriz

```
tensor1 = torch.rand(2)
tensor2 = torch.rand(2,3)
print(f'{tensor1=}', f'\n{tensor2=}')
print(f'{torch.matmul(tensor1, tensor2)=}')
```

```
>> tensor1=tensor([0.7479, 0.4668])

>> tensor2=tensor([[0.2346, 0.0941, 0.9526],
                   [0.8524, 0.9171, 0.3563]])

>> torch.matmul(tensor1, tensor2)=tensor([0.5734, 0.4985, 0.8787])
```

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

`torch.matmul(input, other, *, out=None) → Tensor`

- Caso o *input* tenha 2 dimensões e o *other* tenha 1, então retorna produto matriz-vetor
  - Basicamente o produto escalar de *other* com cada linha de *input*

```
tensor1 = torch.rand(2)
tensor2 = torch.rand(2,3)
print(f'{tensor1=}', f'\n{tensor2=}')
print(f'{torch.matmul(tensor1, tensor2)=}')
```

```
>> tensor1=tensor([[0.5327, 0.9980],
                   [0.8382, 0.6340]])

>> tensor2=tensor([0.4838, 0.8075])

>> torch.matmul(tensor1, tensor2)=tensor([1.0636, 0.9175])
```

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

`torch.matmul(input, other, *, out=None) → Tensor`

- Caso *input e Other* tenham pelo menos 1 dimensão, sendo um deles com mais do que 2 dimensões então é realizado broadcast de operações

```
tensor1 = torch.Tensor([[[1, 1],
                         [1, 1]],
                        [[2, 2],
                         [2, 2]]])
tensor2 = torch.Tensor([1,2])
print(f'{tensor1.shape=}', f'{tensor2.shape=}')
print(f'{torch.matmul(tensor1, tensor2)=}')
```

```
>> tensor1.shape=torch.Size([2, 2, 2]) tensor2.shape=torch.Size([2])

>> ???
```

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

`torch.matmul(input, other, *, out=None) → Tensor`

- Caso *input e Other* tenham pelo menos 1 dimensão, sendo um deles com mais do que 2 dimensões então é realizado broadcast de operações

```
tensor1 = torch.Tensor([[[1, 1],
                         [1, 1]],
                        [[2, 2],
                         [2, 2]]])
tensor2 = torch.Tensor([1,2])
print(f'{tensor1.shape=}', f'{tensor2.shape=}')
print(f'{torch.matmul(tensor1, tensor2)=}')
```

```
>> tensor1.shape=torch.Size([2, 2, 2]) tensor2.shape=torch.Size([2])

>> torch.matmul(tensor1, tensor2)=tensor([[3., 3.], [6., 6.]])
```

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

```
torch.matmul(input, other, *, out=None) → Tensor
```

- Outro exemplo com broadcast de operações

```
tensor1 = torch.Tensor([[[1, 1],
                         [1, 1]],
                        [[2, 2],
                         [2, 2]],
                        [[3, 3],
                         [3, 3]]])
tensor2 = torch.Tensor([[1,2,3],
                        [4,5,6]])
print(f'{tensor1.shape=}', f'{tensor2.shape=}')
print(f'{torch.matmul(tensor1, tensor2)=}')
```

```
>> tensor1.shape=torch.Size([3, 2, 2])
   tensor2.shape=torch.Size([2, 3])
>> ???
```

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

`torch.matmul(input, other, *, out=None) → Tensor`

- Outro exemplo com broadcast de operações

```
tensor1 = torch.Tensor([[[1, 1],
                         [1, 1]],
                        [[2, 2],
                         [2, 2]],
                        [[3, 3],
                         [3, 3]]])
tensor2 = torch.Tensor([[1,2,3],
                        [4,5,6]])
print(f'{tensor1.shape=}', f'{tensor2.shape=}')
print(f'{torch.matmul(tensor1, tensor2)=}')
```

```
>> tensor1.shape=torch.Size([3, 2, 2])
   tensor2.shape=torch.Size([2, 3])
>> torch.matmul(tensor1, tensor2)=
  tensor([[[ 5., 7., 9.],
           [ 5., 7., 9.]],

          [[10., 14., 18.],
           [10., 14., 18.]],

          [[15., 21., 27.],
           [15., 21., 27.]]])
```

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

```
torch.matmul(input, other, *, out=None) → Tensor
```

- Em alguns trechos de código você pode encontrar um @ ou `torch.mm` ao invés de `torch.matmul`
  - É exatamente a mesma coisa ☺

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

- É comum precisar transpor matrizes durante as operações que realizamos
  - Principalmente para poder multiplicar matrizes

```
x1 = torch.tensor([[1, 2, 3], [4, 5, 6]])
x2 = torch.tensor([[1, 2, 3], [4, 5, 6]])
torch.matmul(x1, x2.mT)
```

```
>> tensor([[14, 32], [32, 77]])
```

https://pytorch.org/docs/stable/generated/torch.matmul.html

PyTorch

- Muito cuidado na hora de transpor matrizes de mais de duas dimensões
  - Normalmente o que queremos é `Tensor.mT`

> Tensor.mT
>
> Returns a view of this tensor with the last two dimensions transposed.
>
> `x.mT` is equivalent to `x.transpose(-2, -1)`.

https://pytorch.org/docs/stable/generated/torch.matmul.html

torch.nn.Linear

CLASS `torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Variables

- **~Linear.weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear

torch.nn.Linear

```
linear = torch.nn.Linear(in_features=2, out_features=1, bias=True)
print(linear.weight)
print(linear.bias)
```

```
>> Parameter containing: tensor([[-0.5916, -0.2051]], requires_grad=True)
>> Parameter containing: tensor([-0.0648], requires_grad=True)
```

https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear

torch.nn.Linear

```
linear = torch.nn.Linear(in_features=2, out_features=1, bias=True)
print(linear.weight)
print(linear.bias)
```

```
>> Parameter containing: tensor([[-0.5916, -0.2051]], requires_grad=True)
>> Parameter containing: tensor([-0.0648], requires_grad=True)
```

```
x = torch.tensor([[1, 2],
                  [3, 4],
                  [5, 6]], dtype=torch.float32)
linear(x)
```

```
>> tensor([[2.4147], [4.7381], [7.0615]], grad_fn=<AddmmBackward0>)
```

https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear

torch.nn.Linear

```
linear = torch.nn.Linear(in_features=2, out_features=3, bias=True)
print(linear.weight)
print(linear.bias)
```

```
>> Parameter containing: tensor([[ 0.4785, -0.2612],
                                  [-0.4352, 0.5959],
                                  [-0.0262, -0.4478]], requires_grad=True)
>> Parameter containing: tensor([0.3515, 0.5941, 0.1180], requires_grad=True)
```

https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear

torch.nn.Linear

```
linear = torch.nn.Linear(in_features=2, out_features=3, bias=True)
print(linear.weight)
print(linear.bias)
```

```
>> Parameter containing: tensor([[ 0.4785, -0.2612],
                                  [-0.4352, 0.5959],
                                  [-0.0262, -0.4478]], requires_grad=True)
>> Parameter containing: tensor([0.3515, 0.5941, 0.1180], requires_grad=True)
```

```
x = torch.tensor([[1, 2],
                  [3, 4],
                  [6, 2]], dtype=torch.float32)
y = linear(x)
print(f'{y=}')
class_ = y.argmax(dim=1)
print(f'{class_=}')
```

```
>> y=tensor([[ 0.3077, 1.3508, -0.8037],
             [ 0.7425, 1.6724, -1.7516],
             [ 2.7003, -0.8249, -0.9346]], grad_fn=<AddmmBackward0>)

>> class_=tensor([1, 1, 0])
```

https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear

Reshape

- Em alguns casos é necessário modificar as dimensões de um vetor
  - Várias funções cumprem esse papel, dependendo do caso de uso

Method	One-line description
torch.reshape(input, shape)	Reshapes input to shape (if compatible), can also use torch.Tensor.reshape().
Tensor.view(shape)	Returns a view of the original tensor in a different shape but shares the same data as the original tensor.
torch.stack(tensors, dim=0)	Concatenates a sequence of tensors along a new dimension (dim), all tensors must be same size.
torch.squeeze(input)	Squeezes input to remove all the dimenions with value 1.
torch.unsqueeze(input, dim)	Returns input with a dimension value of 1 added at dim.
torch.permute(input, dims)	Returns a view of the original input with its dimensions permuted (rearranged) to dims.

Reshape

```
x = torch.arange(1., 8.)
print(x, x.shape)
```

```
>> tensor([1., 2., 3., 4., 5., 6., 7.]) torch.Size([7])
```

```
x_reshaped = x.reshape(1, 7)
print(x_reshaped, x_reshaped.shape)
```

```
>> tensor([[1., 2., 3., 4., 5., 6., 7.]]) torch.Size([1, 7])
```

```
z = x.view(1, 7)
print(z, z.shape)
```

```
>>  tensor([[1., 2., 3., 4., 5., 6., 7.]]) torch.Size([1, 7])
```

```
z_squeeze = z.squeeze()
print(z_squeeze, z_squeeze.shape)
```

```
>> tensor([1., 2., 3., 4., 5., 6., 7.]) torch.Size([7])
```

```
z[:, 0] = 5
print(z, x)
```

```
>> tensor([[5., 2., 3., 4., 5., 6., 7.]]) tensor([5., 2., 3., 4., 5., 6., 7.])
```

Reshape

```
x_stacked = torch.stack([x, x, x, x], dim=0)
print(x_stacked, x_stacked.shape)
```

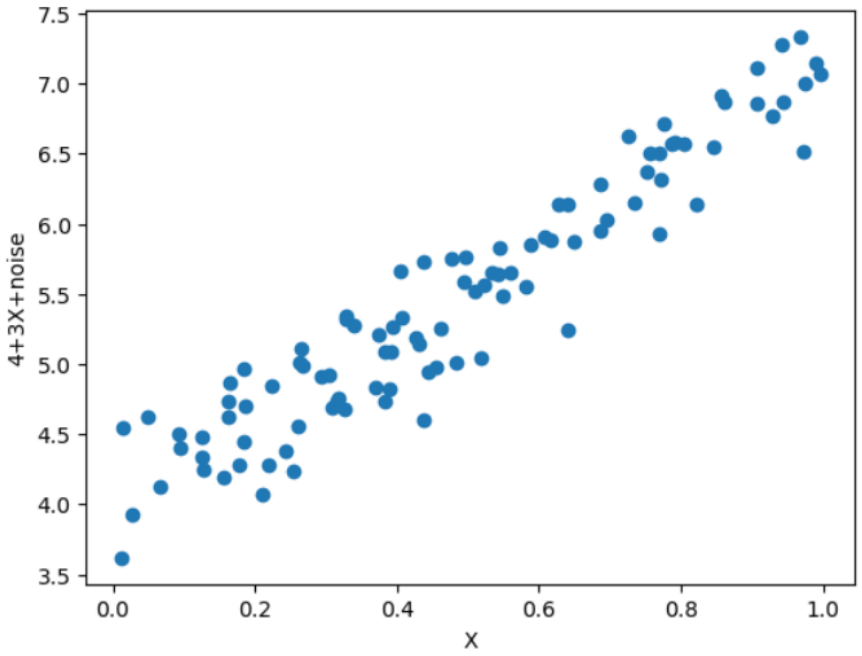
```
>>  tensor([[5., 2., 3., 4., 5., 6., 7.],
            [5., 2., 3., 4., 5., 6., 7.],
            [5., 2., 3., 4., 5., 6., 7.],
            [5., 2., 3., 4., 5., 6., 7.]]) torch.Size([4, 7])
```

Regressão Linear em PyTorch

- Vamos fazer alguns dados toy para fazer o treinamento da regressão linear

```
X = torch.rand(100, 1)
y = 4 + 3 * X + 0.3*torch.randn(100, 1)
y[:5]
```

```
tensor([[5.1790],
        [7.6076],
        [3.5988],
        [3.6588],
        [7.3775]])
```



CLASS torch.nn.Module(*args, **kwargs) [SOURCE]

Regressão Linear

- Vamos definir uma classe para a Regressão Linear

```
class LinearRegression(torch.nn.Module):
   def __init__(self):
      super().__init__()
      self.linear = torch.nn.Linear(1, 1)
   def forward(self, X):
      return self.linear(X)
```

Todos os modelos devem ser subclasses de `torch.nn.Module`. Um Module pode conter outros Modules

Adicionamos uma camada `torch.nn.Linear` onde ficam os pesos da regressão linear

Toda classe que extende torch.nn.Module precisa implementar um método `forward`. Esse método descreve o *forward pass* do modelo

CLASS torch.nn.Module(*args, **kwargs) [SOURCE]

Regressão Linear

- Vamos definir uma classe para a Regressão Linear

```
list(linear.parameters())
```

```
[Parameter containing: tensor([[-0.3805]], requires_grad=True),
 Parameter containing: tensor([-0.9030],requires_grad=True)]
```

CLASS torch.nn.Module(*args, **kwargs) [SOURCE]

Regressão Linear

- Vamos definir uma classe para a Regressão Linear

```
optimizer = torch.optim.SGD(linear.parameters(), lr=0.01)
linear.train()
for i in range(500):
   y_hat = linear(X)
   loss = (y_hat - y).pow(2).mean()
   optimizer.zero_grad()
   loss.backward()
   optimizer.step()
   print(f'{loss.item()=}')
```

CLASS torch.nn.Module(*args, **kwargs) [SOURCE]

Regressão Linear

- Vamos definir uma classe para a Regressão Linear

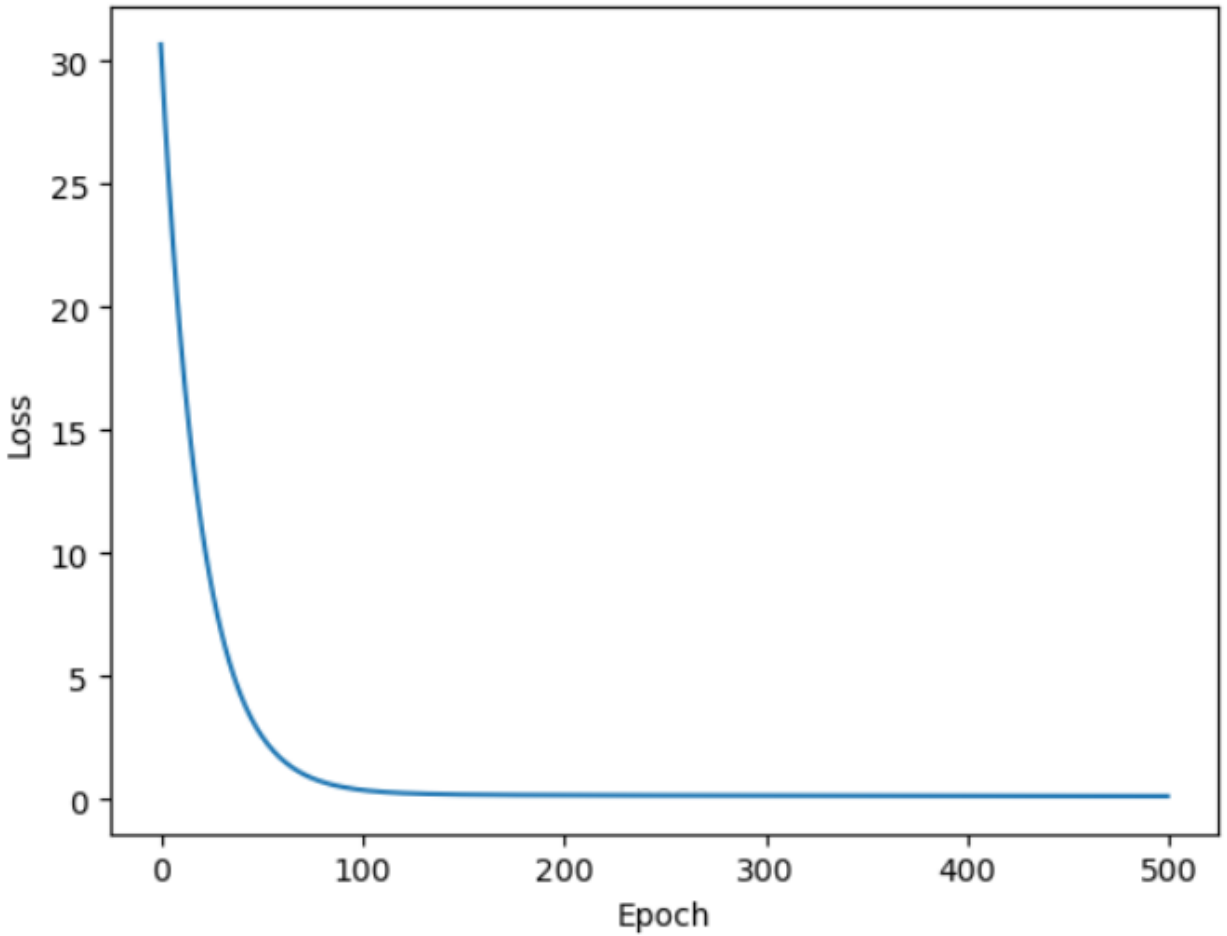
```
optimizer = torch.optim.SGD(linear.parameters(), lr=0.01)
linear.train()
for i in range(500):
    y_hat = linear(X)
    loss = (y_hat - y).pow(2).mean()
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    print(f'{loss.item()=}')
```

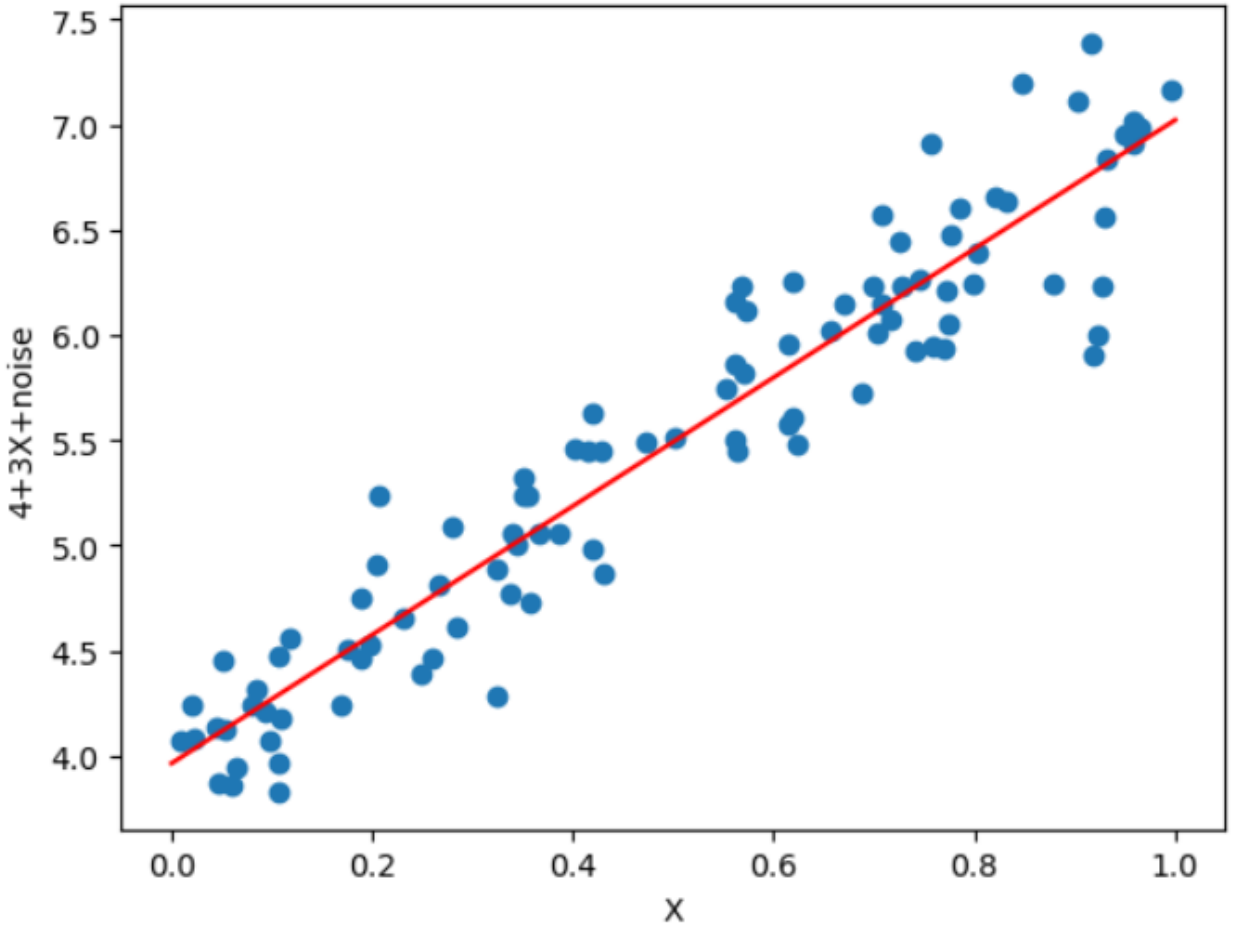
Precisamos definer um otimizador que conheça os parâmetros do nosso modelo. Cada otimizador vai receber um conjunto de parâmetros diferente

Depois calculamos a função de custo

Os gradientes por padrão são acumulados, então é importante zerar esses valores antes de realizar os passos de otimização

Fazemos o *backward pass* e realizamos um passo de otimização


Loss
Epoch
0
100
200
300
400
500
0
5
10
15
20
25
30


4+3X+noise
X
0.0
0.2
0.4
0.6
0.8
1.0
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5

Referências

- Documentação Padrão do PyTorch
  - https://pytorch.org/docs/stable/generated/
- PyTorch Fundamentals
  - https://www.learnpytorch.io/00_pytorch_fundamentals/