Deep Learning Fundamentals with PyTorch Series

- PyTorch is a `machine learning framework` based on the `Torch library`, used for applications such as `computer vision` and `natural language processing`, originally developed by Meta AI and now part of the Linux Foundation umbrella.
- A number of pieces of deep learning software are built on top of PyTorch, including Tesla Autopilot, Uber's Pyro, Hugging Face's Transformers, PyTorch Lightning, and Catalyst.

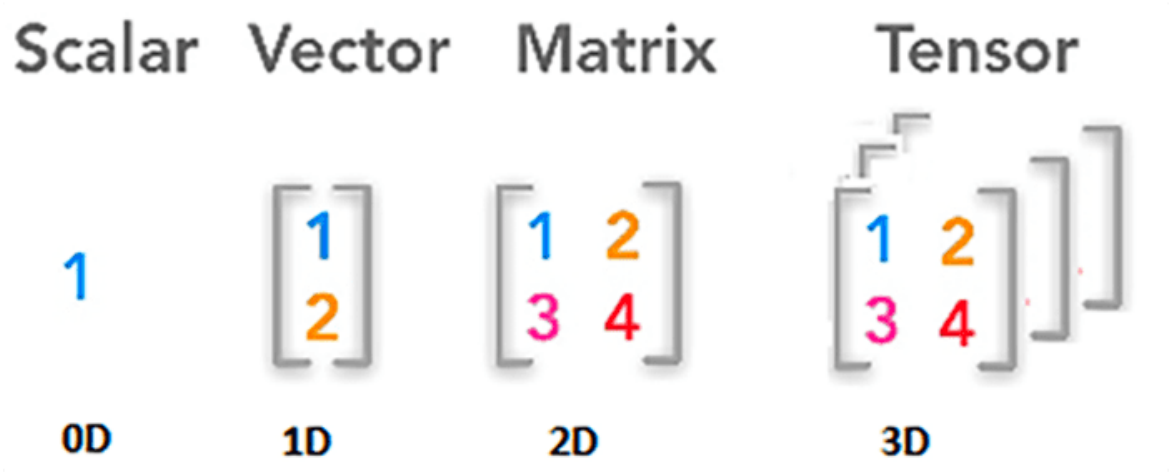
Pytorch Basics

- Saclars, Arrays and Matrix
- Rank, Axes, and Shape
- Creat Tensors
- Tensor Operations
- Gradients
- Back Propagation
- Gradient Descent(Autograd)
- Linear Regression
- Activation Functions
- Dataset and Dataloader
- Data Transform

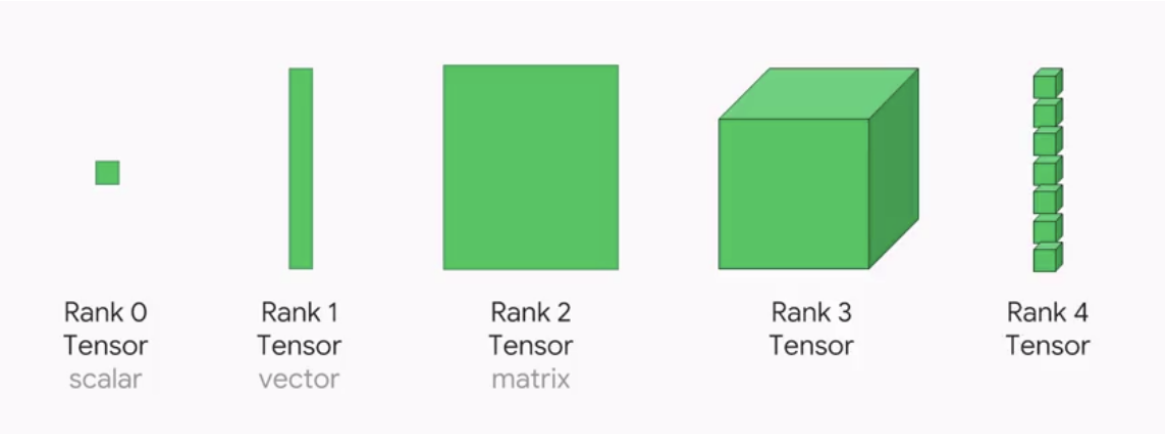
Why Tensor

- `Tensor` is the primary `Data Structure` used by `Neural Network`.
- Number, Arrya, and 2d-array -> `Computer Science`
- Scaler, Vector, and matrix -> `Mathematics`
- n(Index required)
- nd-array(Computer Science)
- nd-tensor(Mathematics)

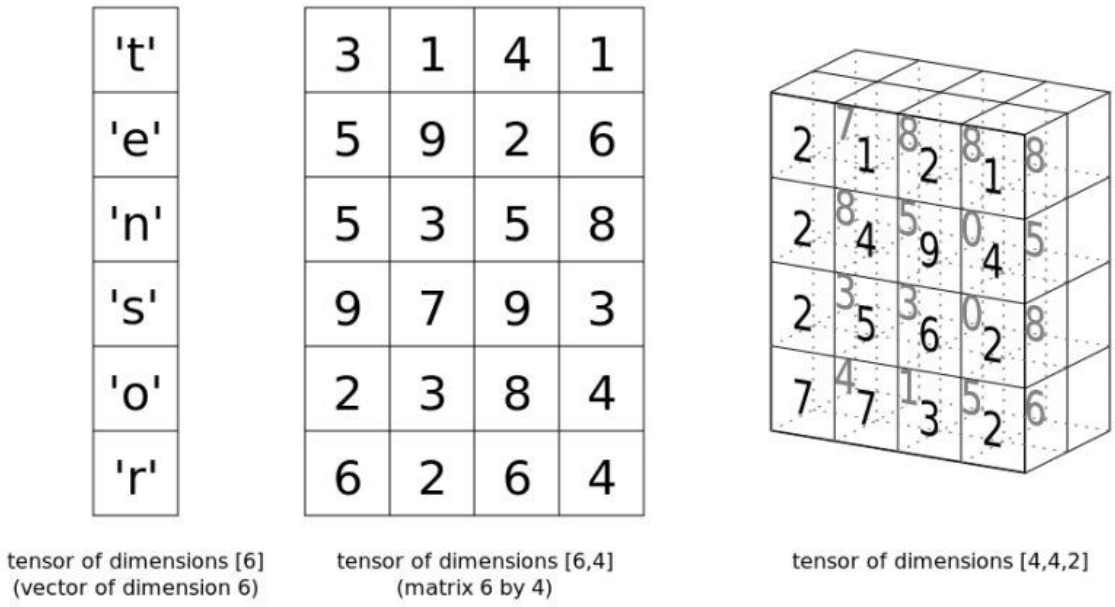
Saclars, Arrays and Matrix



Rank, Axes, and Shape



Shape




```
import torch
```

```
import numpy as np
```

```
torch.get_default_dtype()
```

```
torch.float32
```

```
data = np.array([1, 2, 3])
print(data)
print(type(data))
print(data.dtype)
```

```
[1 2 3]
<class 'numpy.ndarray'>
int64
```

type inference : dtype based on the incoming data

```
print(torch.Tensor(data))
print(torch.Tensor(data).dtype)
print(torch.tensor(data))
print(torch.tensor(data).dtype)
print(torch.as_tensor(data))
print(torch.as_tensor(data).dtype)
print(torch.from_numpy(data))
print(torch.from_numpy(data).dtype)
```

```
tensor([1., 2., 3.])
torch.float32
tensor([1, 2, 3])
torch.int64
tensor([1, 2, 3])
torch.int64
tensor([1, 2, 3])
torch.int64
```

Sharing Memory for Perfomance: Copy vs Share

```
data = np.array([1, 2, 3])

o1 = torch.Tensor(data)
o2 = torch.tensor(data)
o3 = torch.as_tensor(data)
o4 = torch.from_numpy(data)

data[0] = 0

print(o1)
print(o2)
print(o3)
print(o4)
```

```
tensor([1., 2., 3.])
tensor([1, 2, 3])
tensor([0, 2, 3])
tensor([0, 2, 3])
```

Converting Pytorch Tensor back to numpy array

```
print(o3.numpy())
print(type(o3.numpy()))

print(o4.numpy())
print(type(o4.numpy()))
```

```
[0 2 3]
<class 'numpy.ndarray'>
[0 2 3]
<class 'numpy.ndarray'>
```

Creating tensors without data

```
iden = torch.eye(2)
iden
```

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

```
iden = torch.eye(4)
iden
```

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

```
zero = torch.zeros([2, 2])
zero
```

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

```
one = torch.ones([2, 2])
one
```

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

```
rand_num = torch.rand([2, 2])
rand_num
```

```
tensor([[0.9335, 0.7939],
        [0.0817, 0.7658]])
```

tensor data type

```
torch.get_default_dtype()
```

```
torch.float32
```

```
data = np.array([1, 2, 3])
print(torch.tensor(data))
print(torch.tensor(data).dtype)
```

```
tensor([1, 2, 3])
torch.int64
```

```
torch.tensor(data, dtype=torch.float32)
```

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

```
data = np.array([1, 2, 3])
print(torch.as_tensor(data))
print(torch.as_tensor(data).dtype)
```

```
tensor([1, 2, 3])
torch.int64
```

```
torch.as_tensor(data, dtype=torch.float32)
```

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

High Level Tensor Operations

- Reshaping Operations
- Element-wise Operations
- Reduction Operations
- Access Operations

Reshaping Operations

```
t = torch.tensor([[1, 1, 1, 1],
                  [2, 2, 2, 2],
                  [3, 3, 3, 3]],
                 dtype=torch.float32)
```

```
t
```

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

```
t.size()
```

```
torch.Size([3, 4])
```

```
t.shape
```

```
torch.Size([3, 4])
```

```
len(t.shape)
```

```
2
```

```
# the number of elements
t.numel()
```

```
12
```

```
t.reshape([1, 12])
```

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

```
t.reshape([2,6])
```

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

```
# Reshaping does not change the underlying data
t
```

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

```
t.reshape(2, 2, 3)
```

```
tensor([[[1., 1., 1.],
         [1., 2., 2.]],

        [[2., 2., 3.],
         [3., 3., 3.]]])
```

Use `squeeze` and `unsqueeze` to reshape

```
t
```

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

```
print(t.reshape([1, 12]))
print(t.reshape([1, 12]).shape)
print(t.reshape([1, 12]).squeeze())
print(t.reshape([1, 12]).squeeze().shape)
print(t.reshape([1, 12]).squeeze().unsqueeze(dim=0))
print(t.reshape([1, 12]).squeeze().unsqueeze(dim=0).shape)
print(t.reshape([1, 12]).squeeze().unsqueeze(dim=1))
print(t.reshape([1, 12]).squeeze().unsqueeze(dim=1).shape)
```

```
tensor([[1., 1., 1., 1., 2., 2., 2., 2., 3., 3., 3., 3.]])
torch.Size([1, 12])
tensor([1., 1., 1., 1., 2., 2., 2., 2., 3., 3., 3., 3.])
torch.Size([12])
tensor([[1., 1., 1., 1., 2., 2., 2., 2., 3., 3., 3., 3.]])
torch.Size([1, 12])
tensor([[1.],
        [1.],
        [1.],
        [1.],
        [2.],
        [2.],
        [2.],
        [2.],
        [3.],
        [3.],
        [3.],
        [3.]])
torch.Size([12, 1])
```

Use `flatten` to reshape

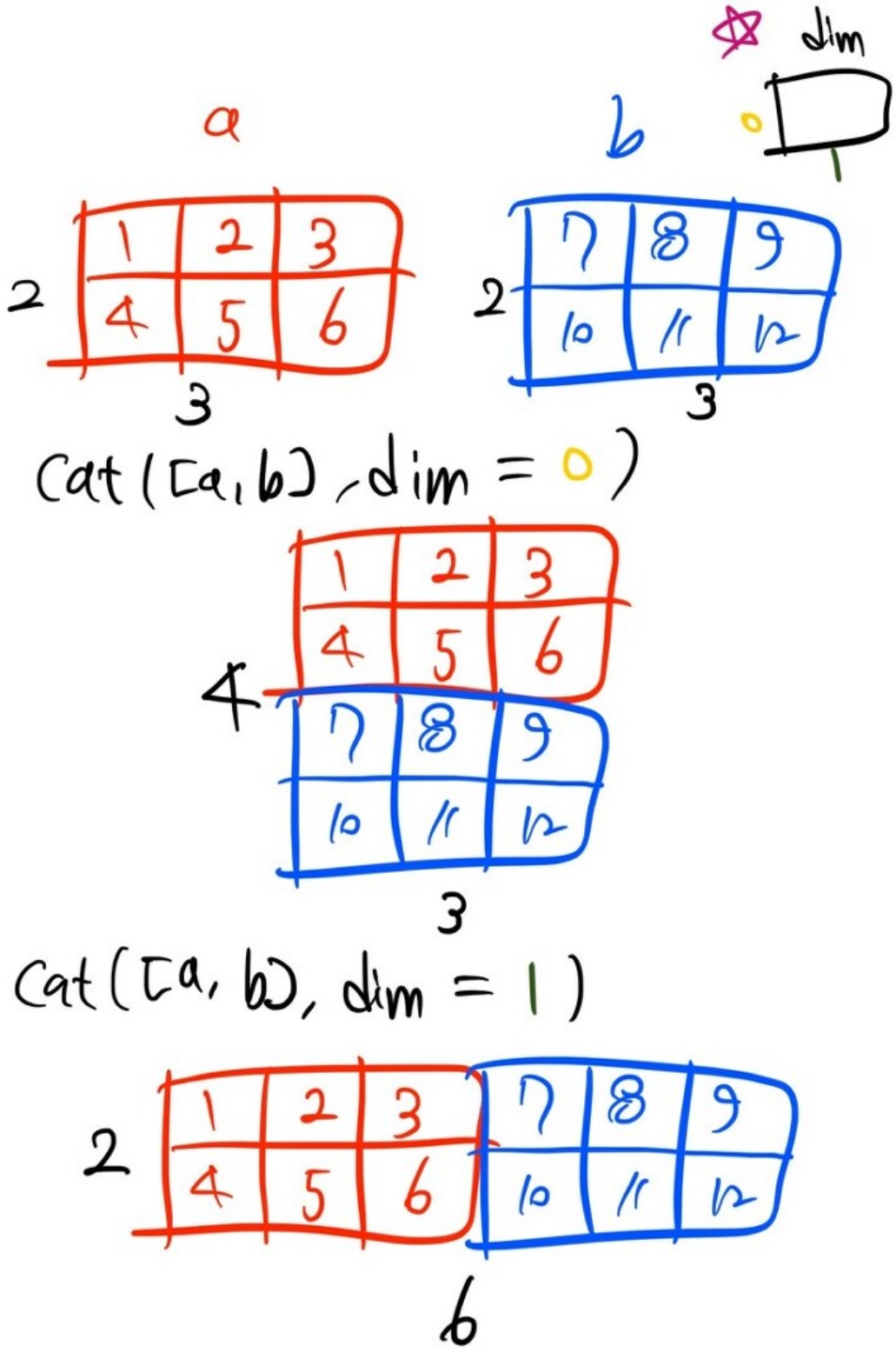
```
print(t)
print(t.numel())
print(t.flatten())
print(t.flatten().shape)
print(t.reshape(1, -1))
print(t.reshape(1, -1).shape)
print(t.reshape(1, -1).squeeze())
print(t.reshape(1, -1).squeeze().shape)
```

```
tensor([[1., 1., 1., 1.],
        [2., 2., 2., 2.],
        [3., 3., 3., 3.]])
12
tensor([1., 1., 1., 1., 2., 2., 2., 2., 3., 3., 3., 3.])
torch.Size([12])
tensor([[1., 1., 1., 1., 2., 2., 2., 2., 3., 3., 3., 3.]])
torch.Size([1, 12])
tensor([1., 1., 1., 1., 2., 2., 2., 2., 3., 3., 3., 3.])
torch.Size([12])
```

```
def flatten(x):
  x = x.reshape(1, -1)
  x = x.squeeze()
  return x

print(flatten(t))
print(flatten(t).shape)
```

```
tensor([1., 1., 1., 1., 2., 2., 2., 2., 3., 3., 3., 3.])
torch.Size([12])
```

torch.cat

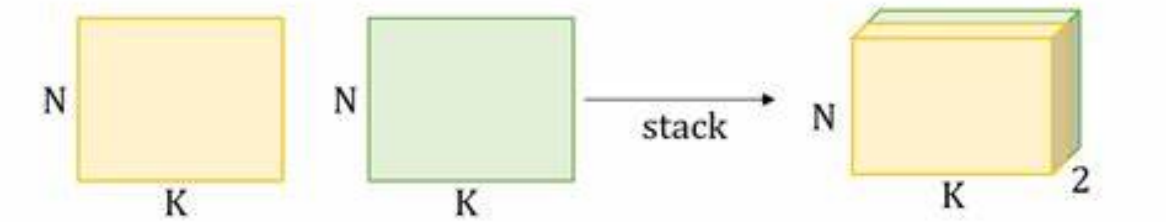



```
t1 = torch.tensor([[1, 2],
                   [3 , 4]])

t2 = torch.tensor([[5, 6],
                   [7 , 8]])

print(t1.shape)

print(torch.cat((t1, t2), dim=0))
print((torch.cat((t1, t2), dim=0)).shape)
```

```
print(torch.cat((t1, t2), dim=1))
print((torch.cat((t1, t2), dim=1)).shape)
```

```
torch.Size([2, 2])
tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
torch.Size([4, 2])
tensor([[1, 2, 5, 6],
        [3, 4, 7, 8]])
torch.Size([2, 4])
```

torch.stack




```
x = torch.tensor([2, 3, 4, 5])
y = torch.tensor([4, 10, 30, 40])
z = torch.tensor([8, 7, 16, 14])
```

```
x.shape
```

```
torch.Size([4])
```

```
stack_0 = torch.stack((x, y, z), dim=0)
print(stack_0)
print(stack_0.shape)
```

```
tensor([[ 2,  3,  4,  5],
        [ 4, 10, 30, 40],
        [ 8,  7, 16, 14]])
torch.Size([3, 4])
```

```
xyz = torch.cat((x.unsqueeze(0), y.unsqueeze(0), z.unsqueeze(0)), dim=0)
print(xyz)
print(xyz.shape)
```

```
tensor([[ 2,  3,  4,  5],
        [ 4, 10, 30, 40],
        [ 8,  7, 16, 14]])
torch.Size([3, 4])
```

```
stack_1 = torch.stack((x, y, z), dim=1)
print(stack_1)
```

```
print(stack_1.shape)
```

```
tensor([[ 2,  4,  8],
        [ 3, 10,  7],
        [ 4, 30, 16],
        [ 5, 40, 14]])
torch.Size([4, 3])
```

```
t1 = torch.tensor([[1, 1, 1, 1],
                   [2, 2, 2, 2],
                   [3, 3, 3 ,3],
                   [4, 4, 4, 4]])

t2 = torch.tensor([[1, 1, 1, 1],
                   [2, 2, 2, 2],
                   [3, 3, 3 ,3],
                   [4, 4, 4, 4]])

t3 = torch.tensor([[1, 1, 1, 1],
                   [2, 2, 2, 2],
                   [3, 3, 3 ,3],
                   [4, 4, 4, 4]])

t = torch.stack((t1, t2, t3))

print(t1.shape)
print(t)
print(t.shape)
```

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

        [[1, 1, 1, 1],
         [2, 2, 2, 2],
         [3, 3, 3, 3],
         [4, 4, 4, 4]],

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

```
t = t.reshape([3, 1, 4, 4])
print(t)
print(t.shape)
```

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

```
               [[[1, 1, 1, 1],
                 [2, 2, 2, 2],
                 [3, 3, 3, 3],
                 [4, 4, 4, 4]]],


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

```
print(t[0])
print(t[0].shape)
```

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

```
print(t[0][0])
print(t[0][0].shape)
```

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

```
print(t[0][0][0])
print(t[0][0][0].shape)
```

```
     tensor([1, 1, 1, 1])
     torch.Size([4])
```

```
print(t[0][0][0][0])
print(t[0][0][0][0].shape)
```

```
     tensor(1)
     torch.Size([])
```

```
print(t.reshape(1, -1))
print(t.reshape(1, -1).shape)
print(t.reshape(1, -1)[0])
print(t.reshape(1, -1)[0].shape)
```

```
     tensor([[1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2,
              3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4]])
     torch.Size([1, 48])
     tensor([1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2,
             3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4])
     torch.Size([48])
```

```
print(t.reshape(-1))
print(t.reshape(-1).shape)
```

```
tensor([1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2,
        3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4])
torch.Size([48])
```

```
print(t.view(t.numel()))
print(t.view(t.numel()).shape)
```

```
tensor([1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2,
        3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4])
torch.Size([48])
```

```
print(t.flatten())
print(t.flatten().shape)
```

```
tensor([1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2,
        3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4])
torch.Size([48])
```

Flatten specific axes `(Channels, Height, Width)`

- `(Batch size, Channels, Height, Width)`
- `Flattening An RGB Image`

```
print(t.flatten(start_dim=1))
print(t.flatten(start_dim=1).shape)
```

```
tensor([[1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4],
        [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4],
        [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4]])
torch.Size([3, 16])
```

Element-wise Operations

```
t1 = torch.tensor([[1, 2],
                   [3, 4]], dtype=torch.float32)

t2 = torch.tensor([[9, 8],
                   [7, 6]], dtype=torch.float32)

print(t1[0])
print(t1[0][0])
print(t1 + t2)
```

```
tensor([1., 2.])
tensor(1.)
```

```
tensor([[10., 10.],
        [10., 10.]])
```

```
print(t1+2)
print(t1.add(2))
print(t1-2)
print(t1.sub(2))
print(t1*2)
print(t1.mul(2))
print(t1/2)
print(t1.div(2))
```

```
tensor([[3., 4.],
        [5., 6.]])
tensor([[3., 4.],
        [5., 6.]])
tensor([[-1.,  0.],
        [ 1.,  2.]])
tensor([[-1.,  0.],
        [ 1.,  2.]])
tensor([[2., 4.],
        [6., 8.]])
tensor([[2., 4.],
        [6., 8.]])
tensor([[0.5000, 1.0000],
        [1.5000, 2.0000]])
tensor([[0.5000, 1.0000],
        [1.5000, 2.0000]])
```

tensor broadcasting

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

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

```
np.broadcast_to(2, x.shape)
```

```
array([[2, 2],
       [2, 2]])
```

```
x + torch.tensor(np.broadcast_to(2, t1.shape),
                 dtype=torch.float32)
```

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

```
y = torch.tensor([2, 2])
```

```
print(np.broadcast_to(y.numpy(), x.shape))

print(x + np.broadcast_to(y.numpy(), x.shape))
```

```
[[2 2]
 [2 2]]
tensor([[3., 4.],
        [5., 6.]], dtype=torch.float64)
```

```
t = torch.tensor([[0, 5, 0],
                  [6, 0, 7],
                  [0, 8, 0]], dtype=torch.float32)

print(t.eq(0))
print(t.ge(0))
print(t.gt(0))
print(t.lt(0))
print(t.le(7))
```

```
tensor([[ True, False,  True],
        [False,  True, False],
        [ True, False,  True]])
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
tensor([[False,  True, False],
        [ True, False,  True],
        [False,  True, False]])
tensor([[False, False, False],
        [False, False, False],
        [False, False, False]])
tensor([[ True,  True,  True],
        [ True,  True,  True],
        [ True, False,  True]])
```

```
t <= torch.tensor(np.broadcast_to(7, t.shape),
                  dtype=torch.float32)
```

```
tensor([[ True,  True,  True],
        [ True,  True,  True],
        [ True, False,  True]])
```

```
print(t.abs())
print(t.sqrt())
print(t.neg())
print(t.neg().abs())
```

```
tensor([[0., 5., 0.],
        [6., 0., 7.],
        [0., 8., 0.]])
tensor([[0.0000, 2.2361, 0.0000],
        [2.4495, 0.0000, 2.6458],
        [0.0000, 2.8284, 0.0000]])
tensor([[-0., -5., -0.],
        [-6., -0., -7.],
```

```
        [-0., -8., -0.]])
tensor([[0., 5., 0.],
        [6., 0., 7.],
        [0., 8., 0.]])
```

In place Operations

```
t1 = torch.tensor([[1, 2],
                   [3, 4]], dtype=torch.float32)

t2 = torch.tensor([[9, 8],
                   [7, 6]], dtype=torch.float32)

print(t1.add(t2))
print(t1)

print(t1.add_(t2))
print(t1)
```

```
tensor([[10., 10.],
        [10., 10.]])
tensor([[1., 2.],
        [3., 4.]])
tensor([[10., 10.],
        [10., 10.]])
tensor([[10., 10.],
        [10., 10.]])
```

```
t1 = torch.tensor([[1, 2],
                   [3, 4]], dtype=torch.float32)

t2 = torch.tensor([[9, 8],
                   [7, 6]], dtype=torch.float32)

print(t1.sub(t2))
print(t1)

print(t1.sub_(t2))
print(t1)
```

```
tensor([[-8., -6.],
        [-4., -2.]])
tensor([[1., 2.],
        [3., 4.]])
tensor([[-8., -6.],
        [-4., -2.]])
tensor([[-8., -6.],
        [-4., -2.]])
```

```
t1 = torch.tensor([[1, 2],
                   [3, 4]], dtype=torch.float32)
```

```
t2 = torch.tensor([[9, 8],
                   [7, 6]], dtype=torch.float32)

print(t1.mul(t2))
print(t1)

print(t1.mul_(t2))
print(t1)
```

```
tensor([[ 9., 16.],
        [21., 24.]])
tensor([[1., 2.],
        [3., 4.]])
tensor([[ 9., 16.],
        [21., 24.]])
tensor([[ 9., 16.],
        [21., 24.]])
```

Reduction Operations

```
t = torch.tensor([[0, 1, 0],
                  [2, 0, 2],
                  [0, 3, 0]], dtype=torch.float32)


print(t.sum())
print(t.mean())
print(t.std())
print(t.prod())
```

```
tensor(8.)
tensor(0.8889)
tensor(1.1667)
tensor(0.)
```

Understanging Reduction by Axes

```
t = torch.tensor([[1, 1, 1, 1],
                  [2, 2, 2, 2],
                  [3, 3, 3, 3]], dtype=torch.float32)

print(t.sum(dim=0))
print(t.sum(dim=1))

print(t[0])
print(t[1])
print(t[2])
print(t[0] + t[1] +t[2] )
```

```
tensor([6., 6., 6., 6.])
tensor([ 4.,  8., 12.])
tensor([1., 1., 1., 1.])
tensor([2., 2., 2., 2.])
tensor([3., 3., 3., 3.])
tensor([6., 6., 6., 6.])
```

argmax tensor Reduction Operation

- Argmax returns the `index` location of the `maximum` value inside a tensor

```
t = torch.tensor([[1, 0, 0, 2],
                  [0, 3, 3, 0],
                  [4, 0, 0, 5]], dtype=torch.float32)

print(t.max())
print(t.max(dim=0))
print(t.max(dim=1))
print(t.argmax())
print(t.argmax(dim=0))
print(t.argmax(dim=1))
print(t.flatten())
```

```
tensor(5.)
torch.return_types.max(
values=tensor([4., 3., 3., 5.]),
indices=tensor([2, 1, 1, 2]))
torch.return_types.max(
values=tensor([2., 3., 5.]),
indices=tensor([3, 1, 3]))
tensor(11)
tensor([2, 1, 1, 2])
tensor([3, 1, 3])
tensor([1., 0., 0., 2., 0., 3., 3., 0., 4., 0., 0., 5.])
```

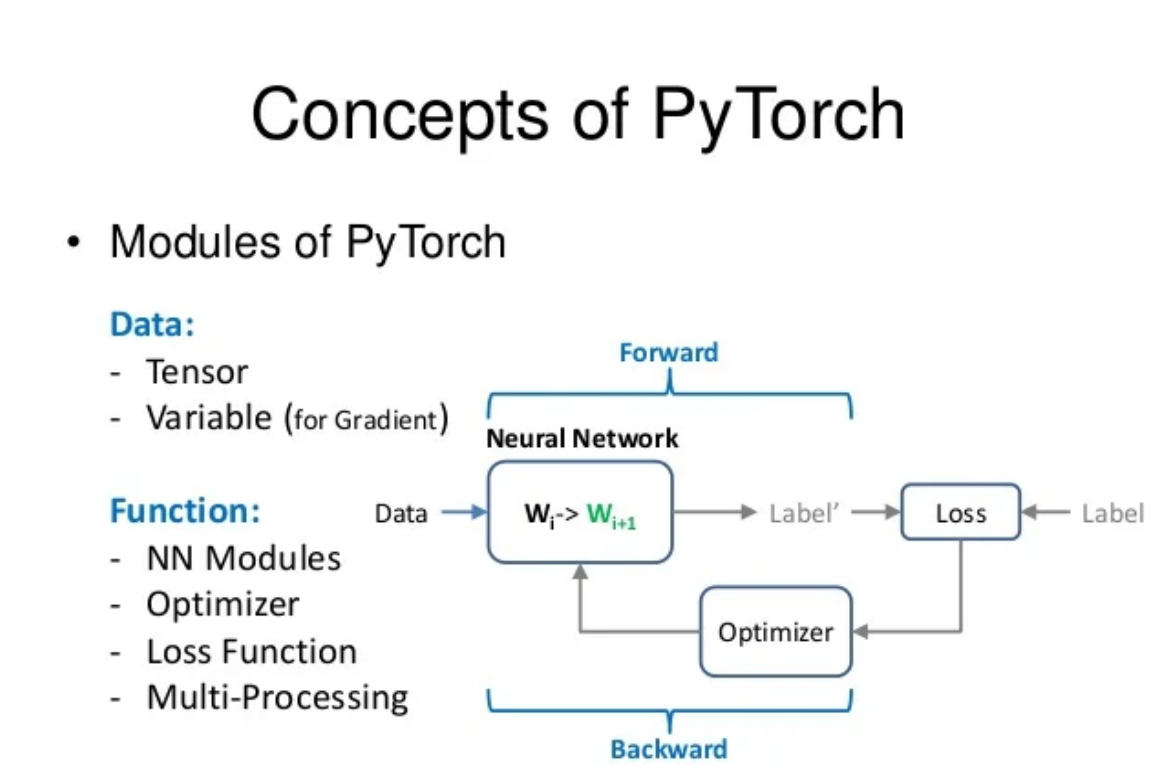
Access Operations

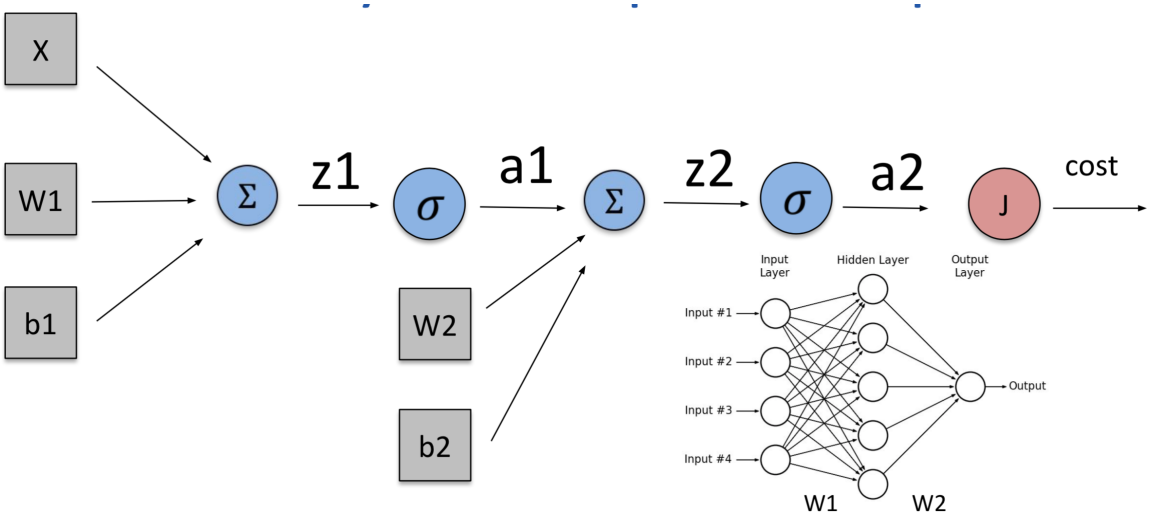
```
t = torch.tensor([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]], dtype=torch.float32)


print(t.mean())
print(t.mean().item())
```

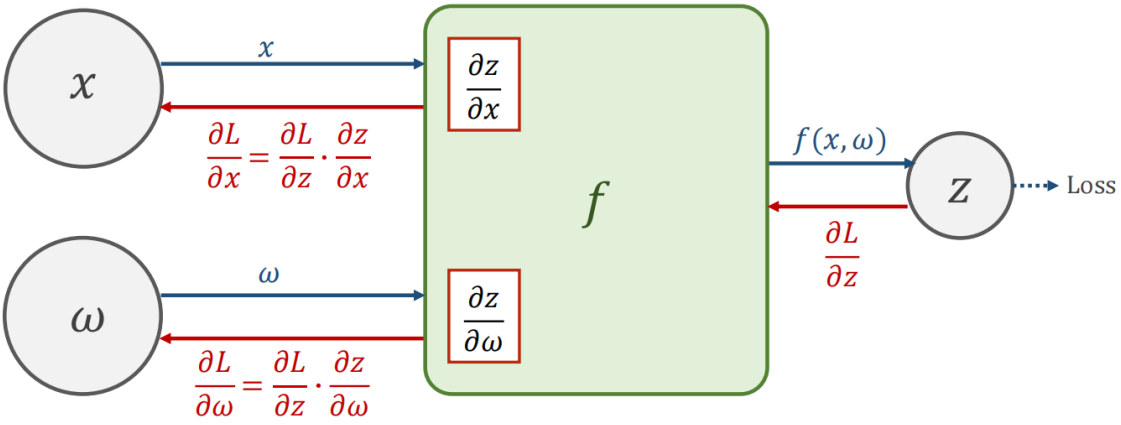
```
tensor(5.)
5.0
```

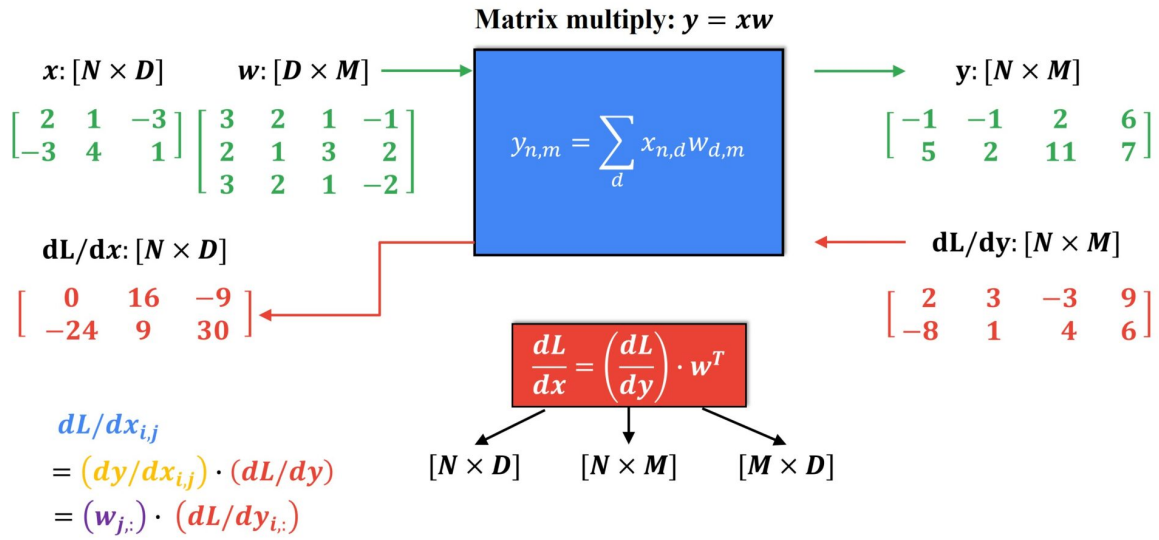
Gradients Theory

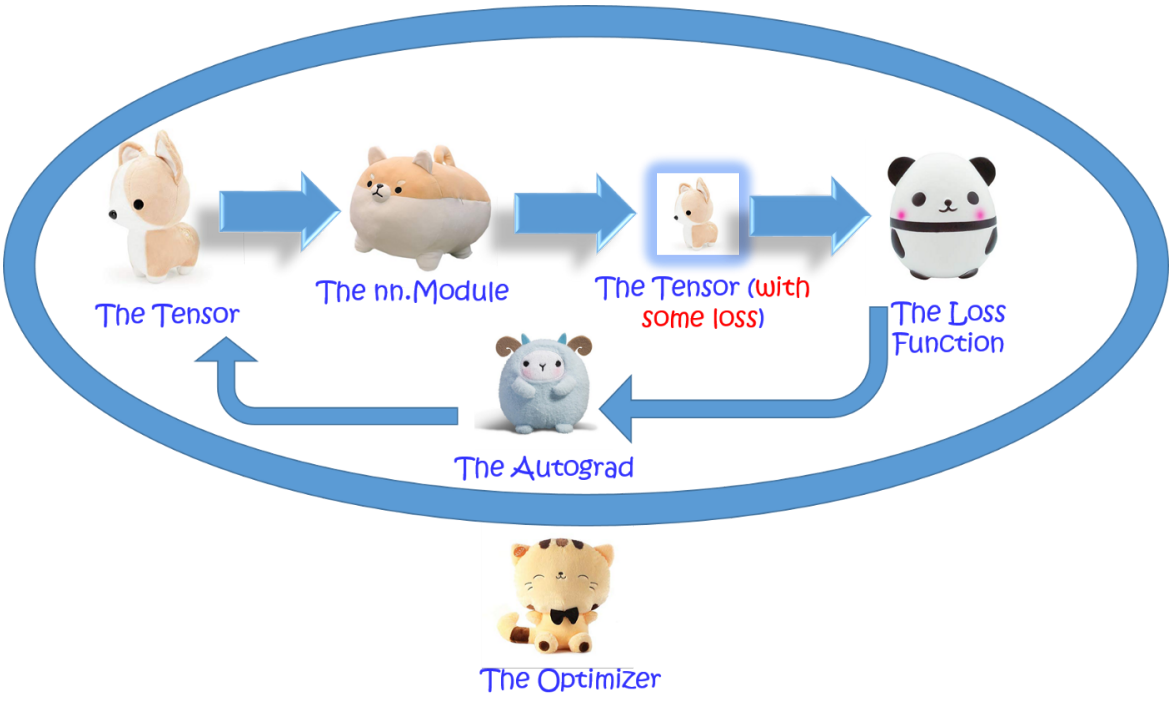
- `Forward Propagation` -> Loss Cost
- `Backward propagation` -> Calculate the gradient of Loss Cost
- `Chain Rule`
- `Jacobian Matrix`
- `Autograd` -> `Calculate gradients` and `optimize our parameters`
- `Turning Autograd Off temporarily` -> `torch.no_grad()`
- `detach()` -> Create a copy of the tensor that is detached from the computation history






x
ω
f
z
Loss
$f(x, \omega)$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial \omega}$
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x}$
$\frac{\partial L}{\partial \omega} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial \omega}$


Matrix multiply: $y = xw$
x: $[N \times D]$
w: $[D \times M]$
y: $[N \times M]$
$y_{n,m} = \sum_d x_{n,d} w_{d,m}$
$\begin{bmatrix} 2 & 1 & -3 \\ -3 & 4 & 1 \end{bmatrix}$
$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & 3 & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$
$\begin{bmatrix} -1 & -1 & 2 & 6 \\ 5 & 2 & 11 & 7 \end{bmatrix}$
dL/dx: $[N \times D]$
dL/dy: $[N \times M]$
$\begin{bmatrix} 0 & 16 & -9 \\ -24 & 9 & 30 \end{bmatrix}$
$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$
$\frac{dL}{dx} = \left(\frac{dL}{dy}\right) \cdot w^T$
$[N \times D]$
$[N \times M]$
$[M \times D]$
$dL/dx_{i,j}$
$= (dy/dx_{i,j}) \cdot (dL/dy)$
$= (w_{j,:}) \cdot (dL/dy_{i,:})$


The Tensor
The nn.Module
The Tensor (with some loss)
The Loss Function
The Autograd
The Optimizer

```
import torch

x = torch.randn(3)
print(x)
print(x.shape)
print(x.dtype)
```

```
tensor([-0.8977,  1.3379, -0.1186])
torch.Size([3])
torch.float32
```

```
# Enbale gradient
x  = torch.randn(3, requires_grad=True)
print(x)
print(x.dtype)
```

```
tensor([-0.8022,  1.0610,  1.1878], requires_grad=True)
torch.float32
```

```
y = x + 2
y
```

```
tensor([1.1978, 3.0610, 3.1878], grad_fn=<AddBackward0>)
```

```
z = y * y * 2
z
```

```
tensor([ 2.8696, 18.7398, 20.3246], grad_fn=<MulBackward0>)
```

```
z = z.mean()
z
```

```
tensor(13.9780, grad_fn=<MeanBackward0>)
```

```
z.backward() #dz/dx
print(x.grad)
```

```
tensor([1.5971, 4.0814, 4.2505])
```

```
c = y * y * 2
c
```

```
tensor([ 2.8696, 18.7398, 20.3246], grad_fn=<MulBackward0>)
```

```
c.backward()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-70-c64e6f25b1ab> in <cell line: 1>()
----> 1 c.backward()

                                 2 frames
/usr/local/lib/python3.10/dist-packages/torch/autograd/__init__.py in
_make_grads(outputs, grads, is_grads_batched)
     86             if out.requires_grad:
     87                 if out.numel() != 1:
---> 88                     raise RuntimeError("grad can be implicitly created only
```

```
v = torch.tensor([0.1, 1.0, 0.001], dtype=torch.float32)
c.backward(v)
print(x.grad)
```

```
tensor([ 2.0762, 16.3255,  4.2632])
```

Preventing pytorch from tracking gradients

- x.requirds_grad_(flase)
- x.detach()
- with torch.no_grad():

```
x = torch.randn(3, requires_grad=True)
print(x)
y = x + 2
print(y)
```

```
tensor([ 1.3164, -0.5736,  0.3020], requires_grad=True)
tensor([3.3164, 1.4264, 2.3020], grad_fn=<AddBackward0>)
```

```
with torch.no_grad():
  y = x + 2
  print(y)
```

```
tensor([3.3164, 1.4264, 2.3020])
```

Calling the backward function will cause the gradents to be accumuated

```
weights = torch.ones(4, requires_grad=True)
print(weights)
```

```
tensor([1., 1., 1., 1.], requires_grad=True)
```

```
weights = torch.ones(4, requires_grad=True)
```

```
for epoch in range(3):
  model_output = (weights * 3).sum()
  model_output.backward()
  print(weights.grad)
```

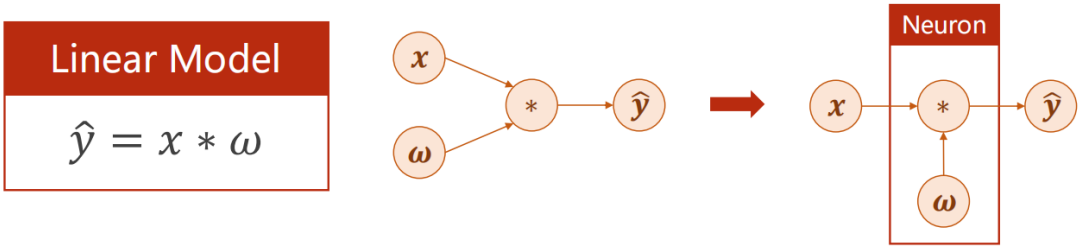
```
tensor([3., 3., 3., 3.])
tensor([6., 6., 6., 6.])
tensor([9., 9., 9., 9.])
```

```
weights = torch.ones(4, requires_grad=True)

for epoch in range(3):
  model_output = (weights * 3).sum()
  model_output.backward()
  print(weights.grad)
  weights.grad.zero_()
  print(weights.grad)
```

```
tensor([3., 3., 3., 3.])
tensor([0., 0., 0., 0.])
tensor([3., 3., 3., 3.])
tensor([0., 0., 0., 0.])
tensor([3., 3., 3., 3.])
tensor([0., 0., 0., 0.])
```

Back Propagation

- `Forward Pass` : Compute `Loss`
- Compute Local Gradients
- `Backward Pass` : Compute `dLoss/dWeiths` using the `Chain Rule`

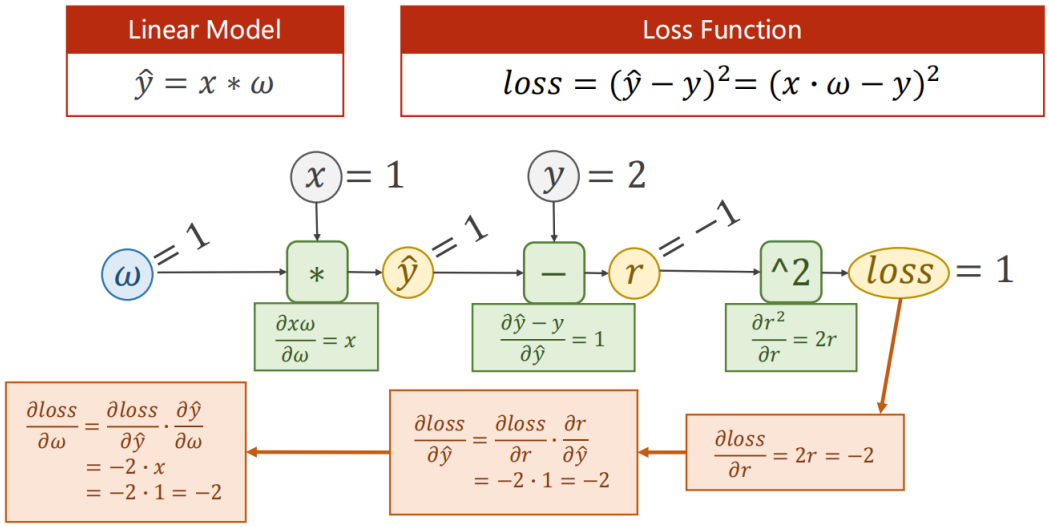


Stochastic Gradient Descent

$$\omega = \omega - \alpha \frac{\partial loss}{\partial \omega}$$

Derivative of Loss Function

$$\frac{\partial loss_n}{\partial \omega} = 2 \cdot x_n \cdot (x_n \cdot \omega - y_n)$$




```
# Forward Pass: Compute Loss
x = torch.tensor(1.0)

w = torch.tensor(1.0, requires_grad=True)

y = torch.tensor(2.0)

y_hat = w * x

loss = (y_hat - y) ** 2

print(loss)
```

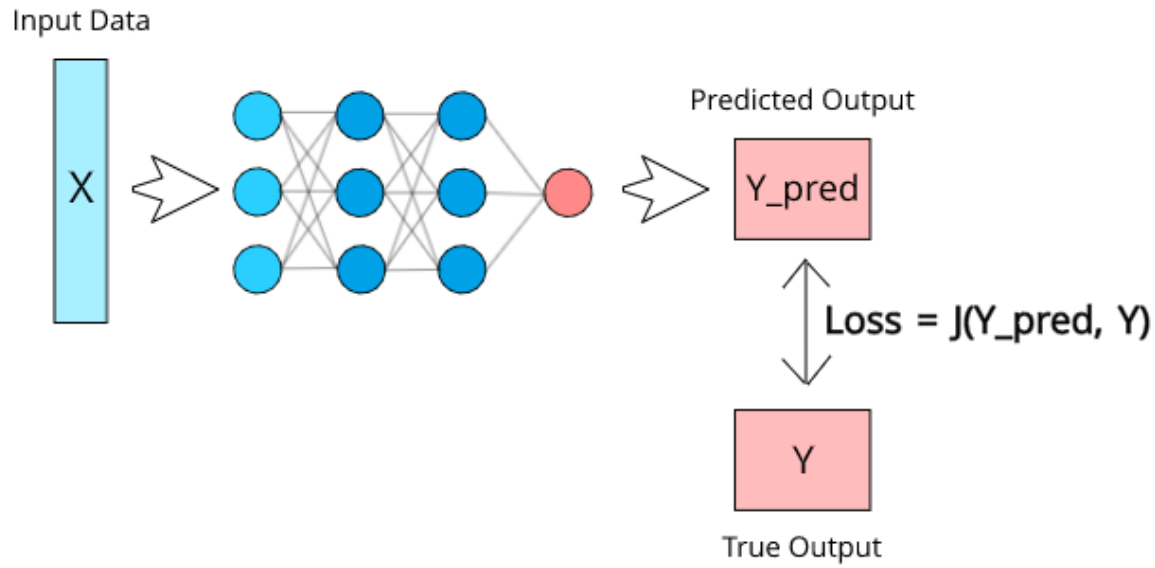
```
tensor(1., grad_fn=<PowBackward0>)
```

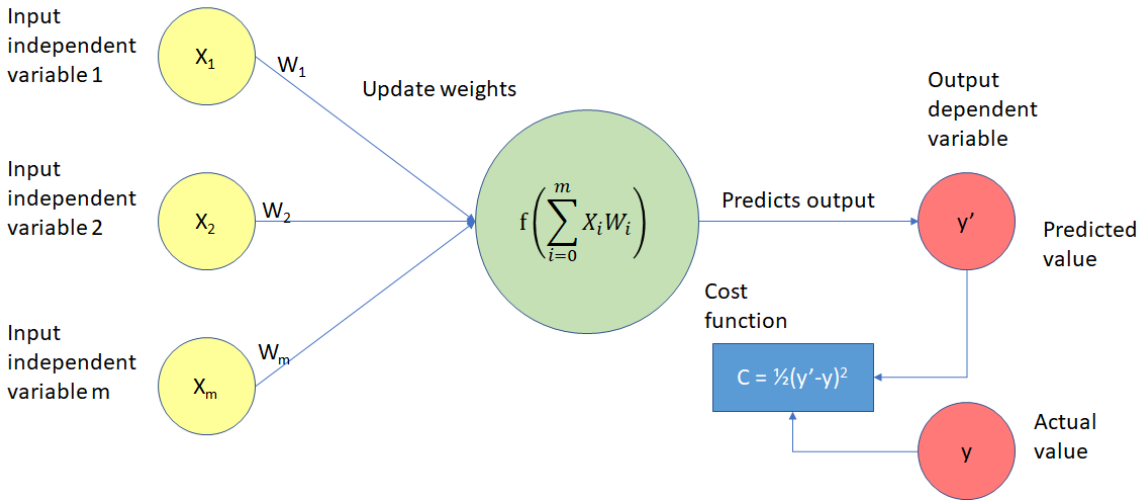
```
# Compute Local Gradients and Backward Pass: Compute dLoss/dWeiths using the Chain Rule
loss.backward()
print(w.grad)
```

```
tensor(-2.)
```

Loss Theory

- `Loss functions` measure how close a predicted value is to the actual value.
  It guides the model training process toward correct predictions.
- `Regression Loss Functions`
- `Classification Loss Functions`
- `Ranking Loss Functions`





Regresion Loss Functions

L1 Loss(Mean Absolute Error)

$$L1LossFunction = \sum_{i=1}^{n} |y_{true} - y_{predicted}|$$

$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

```
import numpy as np
import torch
import torch.nn as nn

torch.__version__
```

```
'2.0.0+cu118'
```

```
x = torch.randn(2, 3)
y = torch.randn(2, 3)

print(x)
print(y)
```

```
tensor([[ 0.1187,  0.3280, -1.1896],
        [ 0.7771, -0.0889,  0.8224]])
tensor([[-0.1714,  0.5595, -0.5074],
        [-0.3716, -0.4300, -1.1684]])
```

```
# Calculate L1 Loss |x - y| tensor
abs(x.numpy() - y.numpy())
```

```
array([[0.29014012, 0.2315253 , 0.6821835 ],
       [1.1487386 , 0.34104922, 1.9908128 ]], dtype=float32)
```

```
# MAE
abs(x.numpy() - y.numpy()).mean()
```

```
0.78074163
```

```
nn.L1Loss()(x, y)
```

```
tensor(0.7807)
```

```
nn.L1Loss(reduce=False)(x, y)
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/_reduction.py:42: UserWarning: size_
  warnings.warn(warning.format(ret))
tensor([[0.2901, 0.2315, 0.6822],
        [1.1487, 0.3410, 1.9908]])
```

Smooth L1 Loss

$$L_{\text{loc}}(t^u, v) = \sum_{i \in \{\text{x,y,w,h}\}} \text{smooth}_{L_1}(t_i^u - v_i), \qquad (2)$$

in which

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases} \qquad (3)$$

```
nn.SmoothL1Loss()(x, y)
```

```
tensor(0.4165)
```

```
nn.SmoothL1Loss(reduce=False)(x, y)
```

```
tensor([[0.0421, 0.0268, 0.2327],
        [0.6487, 0.0582, 1.4908]])
```

```
def smoothL1Loss(x, y):
  z = abs(x - y)
  if (z < 1):
    return 0.5 * (x - y) ** 2
  else:
    return z - 0.5


lst = []

for i in range(len(x)):
  lsti = []
  for j in range(len(x[i])):
    lsti.append(smoothL1Loss(x[i][j], y[i][j]))

  lst.append(lsti)

print(np.array(lst))
print(np.mean(lst))
```

```
[[0.04209065 0.02680198 0.23268716]
 [0.6487386  0.05815729 1.4908128 ]]
0.4165481
```

L2 Loss Function(Mean Squared Error)

$$\text{L2 loss} = |f(x) - Y|^2$$

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \hat{y}\right)}^{2}$$

The square of the difference between actual and predicted

```
(x.numpy() - y.numpy()) ** 2
```

```
array([[0.08418129, 0.05360397, 0.46537432],
       [1.3196005 , 0.11631458, 3.9633355 ]], dtype=float32)
```

```
((x.numpy() - y.numpy()) ** 2).mean()
```

```
1.0004016
```

```
nn.MSELoss(reduce=False)(x, y)
```

```
tensor([[0.0842, 0.0536, 0.4654],
        [1.3196, 0.1163, 3.9633]])
```

```
nn.MSELoss()(x, y)
```

```
tensor(1.0004)
```

Classification Loss Functions

Negative Log-Likehood

Negative Log-Likelihood

Negative Log-Likelihood for Gaussian:

$$
\begin{aligned}
& -\log p_{\boldsymbol{X}|Y}(\boldsymbol{x}|k) \\
&= -\log\left(\frac{1}{\sqrt{(2\pi)^d|\boldsymbol{\Sigma}_k|}}\exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^T\boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k)\right\}\right) \\
&= \underbrace{\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^T\boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k)}_{\text{contains } \boldsymbol{x}} \underbrace{-\frac{n}{2}\log 2\pi - \frac{1}{2}\log|\boldsymbol{\Sigma}_k|}_{\text{no } \boldsymbol{x}}.
\end{aligned}
$$

- $(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}) \geq 0$, always.
- $\sqrt{(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}$ is called **Mahalanobis distance**.

Binary Cross-Entroy Loss

$$BCE = -\frac{1}{N}\sum_{i=0}^{N} y_i \cdot log(\hat{y}_i) + (1-y_i)\cdot log(1-\hat{y}_i)$$

$$\mathcal{L}(\hat{p}^{(1)}, \hat{p}^{(2)}, \cdots, \hat{p}^{(m)} \mid (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})) = \prod_{i=1}^{m} \hat{p}^{(i)^{y^{(i)}}}(1-\hat{p}^{(i)})^{1-y^{(i)}}$$

This is the ***likelihood(ℒ)*** function, where we are trying to maximize the *each* probability, $\hat{p}$, given the examples $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})$.

$\prod_{i=1}^{m}$ represents product over examples 1 to m

The above equation can be broken down as follows:

when $y = 1$

$$\begin{aligned} L(y=1,\hat{p}) &= -y\log(\hat{p}) - (1-y)\log(1-\hat{p}) \\ &= -1 * \log(\hat{p}) - (1-1) * \log(1-\hat{p}) \\ &= -\log(\hat{p}) - 0 * \log(1-\hat{p}) \\ &= -\log(\hat{p}) \end{aligned}$$

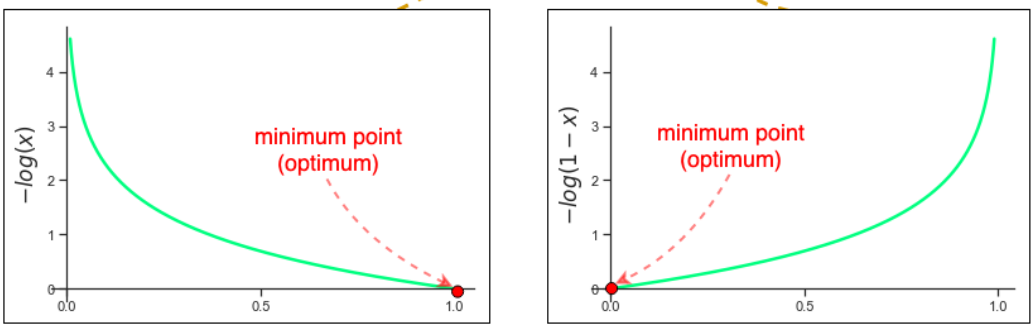
simlarly, when $y = 0$

$$\begin{aligned} L(y=0,\hat{p}) &= -y\log(\hat{p}) - (1-y)\log(1-\hat{p}) \\ &= -0 * \log(\hat{p}) - (1-0) * \log(1-\hat{p}) \\ &= -1 * \log(1-\hat{p}) \\ &= -\log(1-\hat{p}) \end{aligned}$$

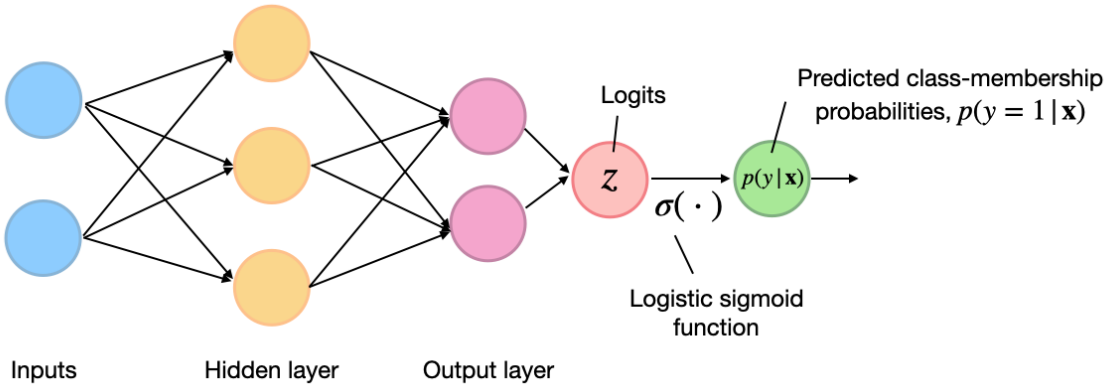
resulting in the following ***piecewise equation:***

$$\therefore\ L(y,\hat{p}) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1-\hat{p}) & \text{if } y = 0 \end{cases}$$

$$\begin{aligned} \textit{Binary Cross Entropy Loss}(y,\hat{p}) &= -\log(P(y \mid x) \\ &= -(y\log(\hat{p}) + (1-y)\log(1-\hat{p})) \\ &= \mathbf{-y\log(\hat{p}) - (1-y)\log(1-\hat{p})} \end{aligned}$$



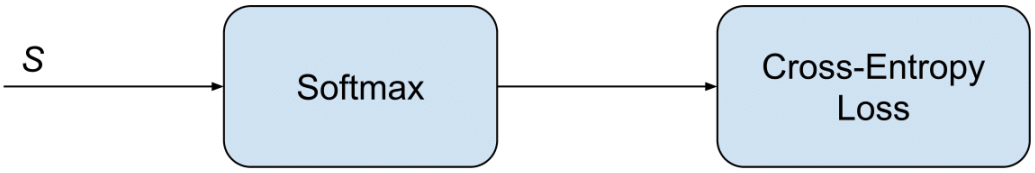
BCE with Sigmoid Layer



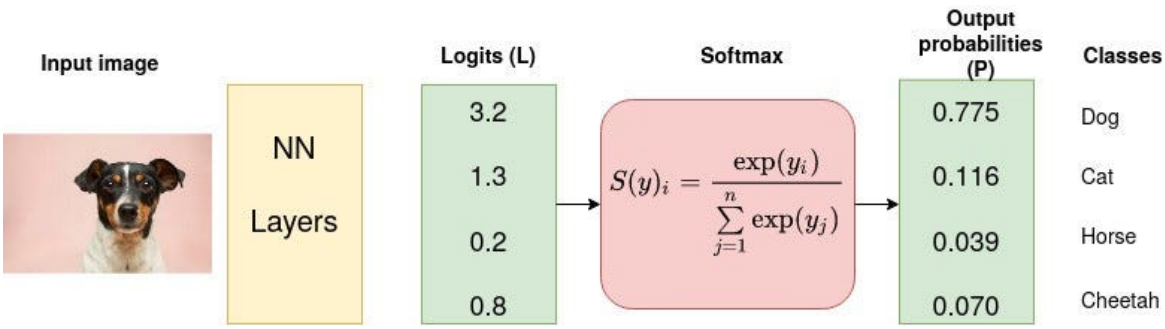
$$y = labels \qquad p_{ij} = sigmoid(logits_{ij}) = \frac{1}{1+e^{-logits_{ij}}}$$

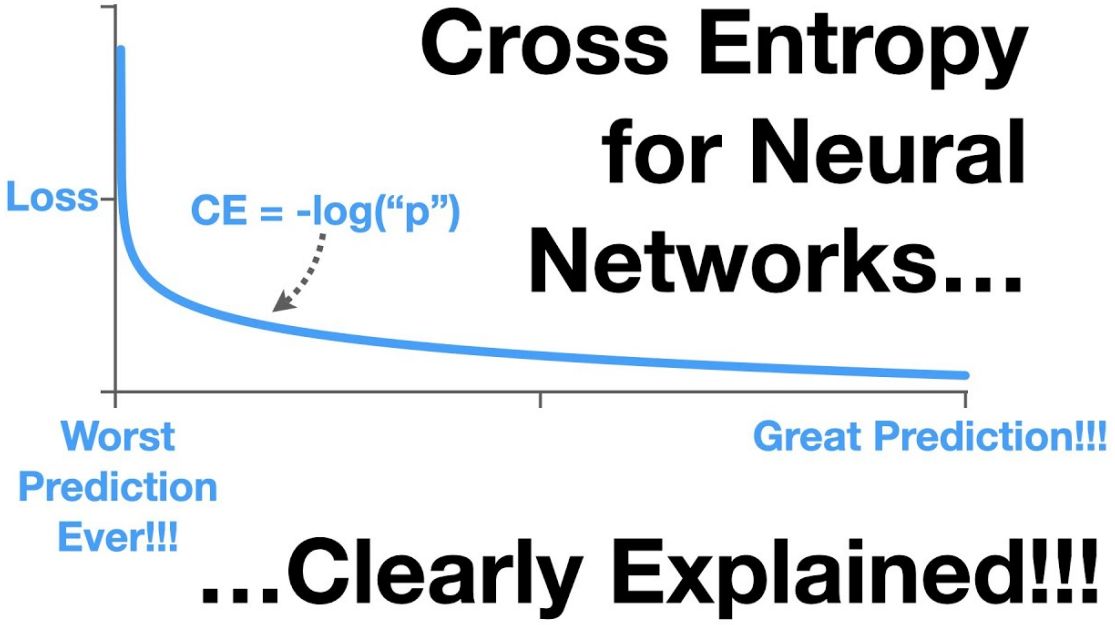
$$loss_{ij} = -[y_{ij} * ln\, p_{ij} + (1 - y_{ij})\, ln(1 - p_{ij})]$$

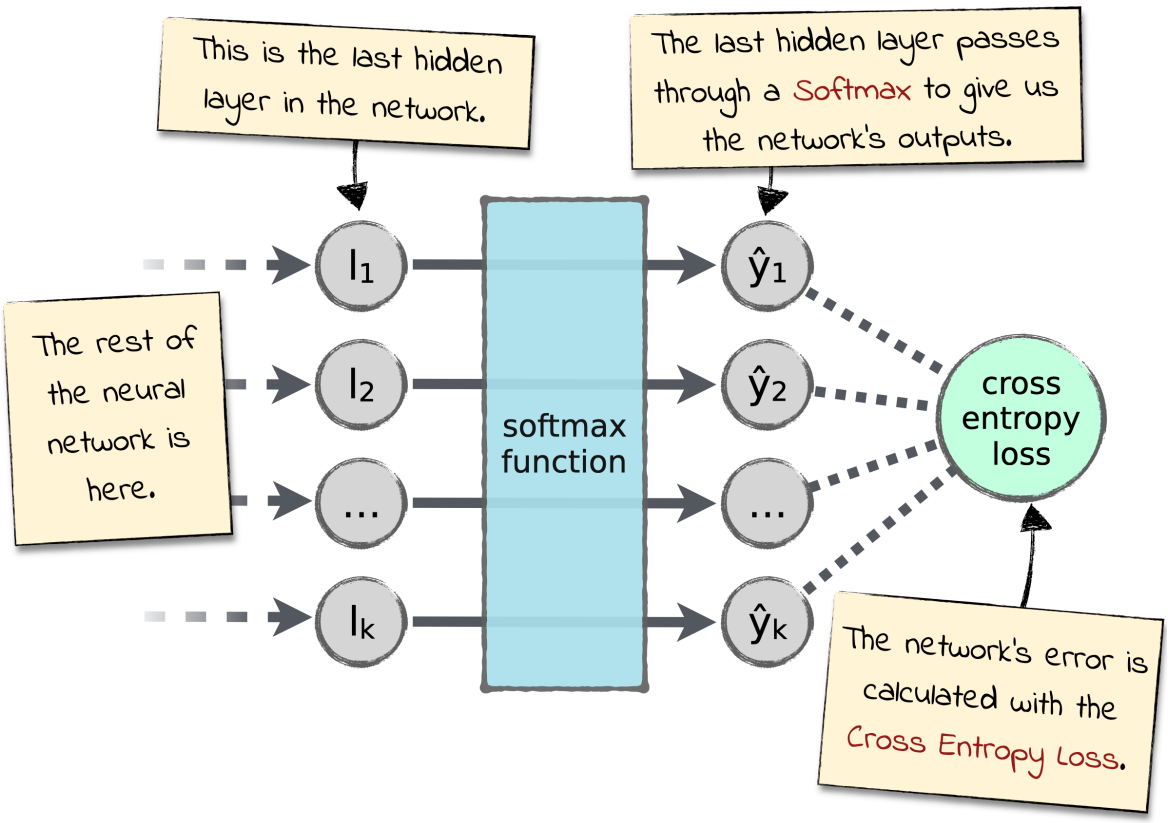
Cross-Entroy Loss



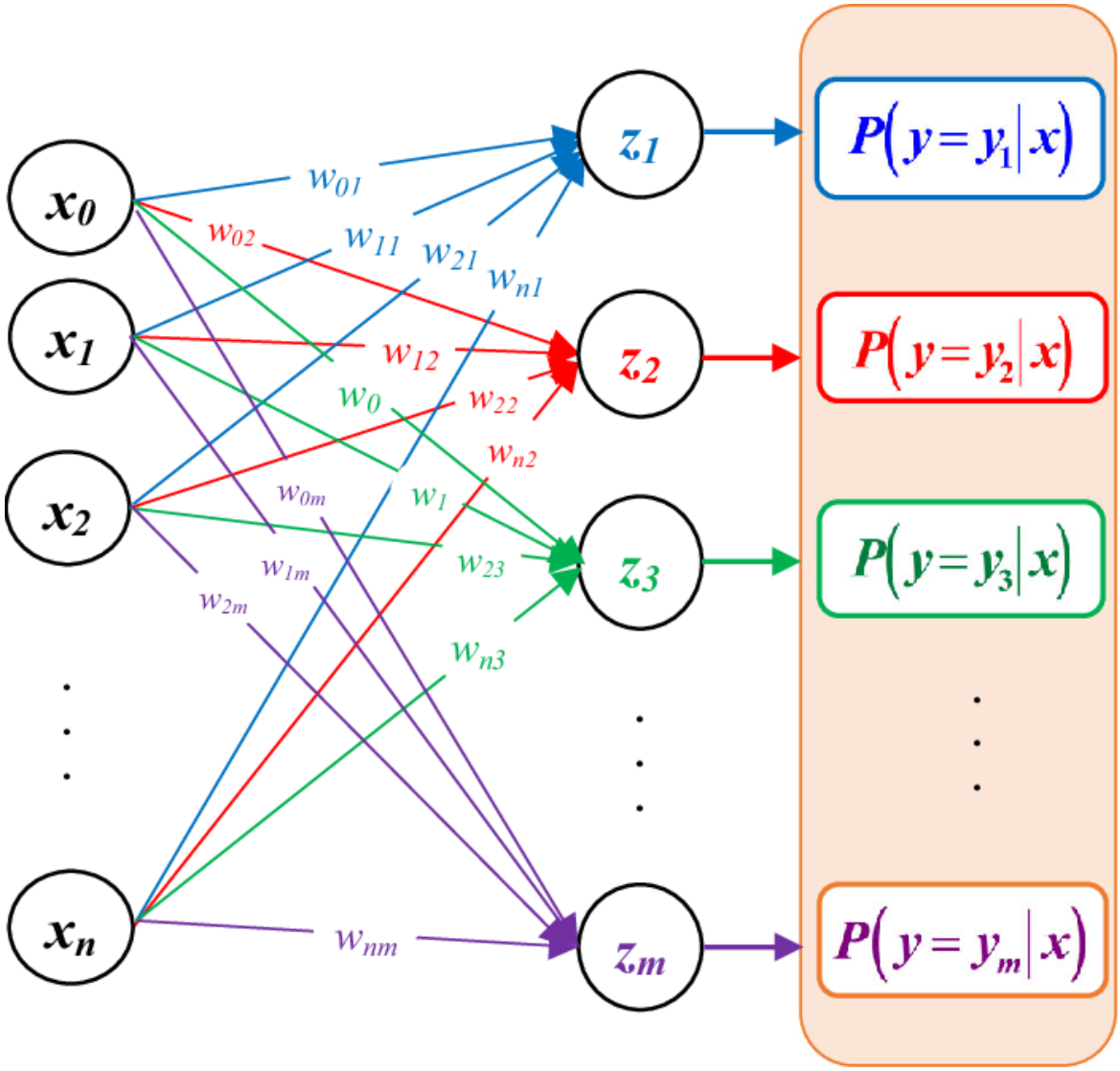
$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$







Multi-class Classification



Multi-class Classification (3 classes as example)

[Bishop, P209-210]

C_1: w^1, b_1 $\quad z_1 = w^1 \cdot x + b_1$

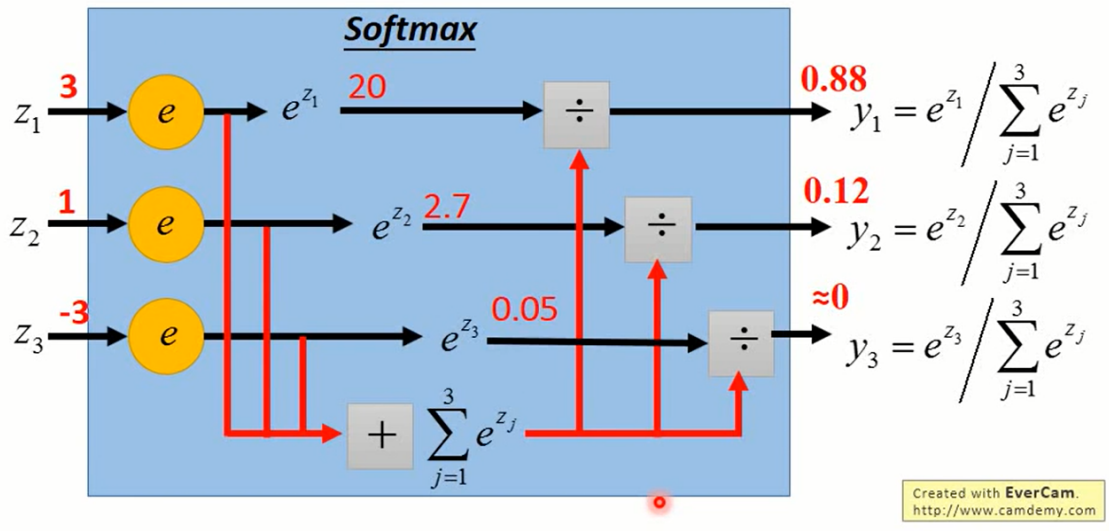
C_2: w^2, b_2 $\quad z_2 = w^2 \cdot x + b_2$

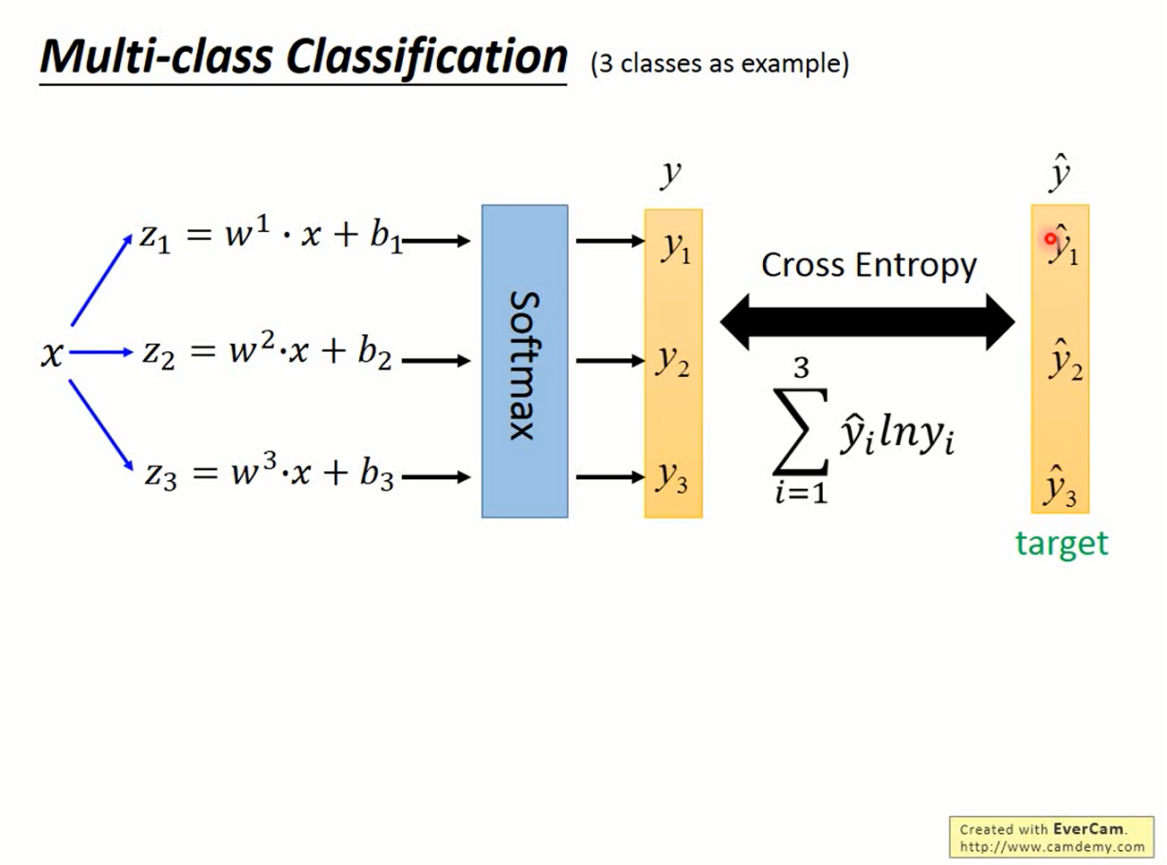
C_3: w^3, b_3 $\quad z_3 = w^3 \cdot x + b_3$

Probability:

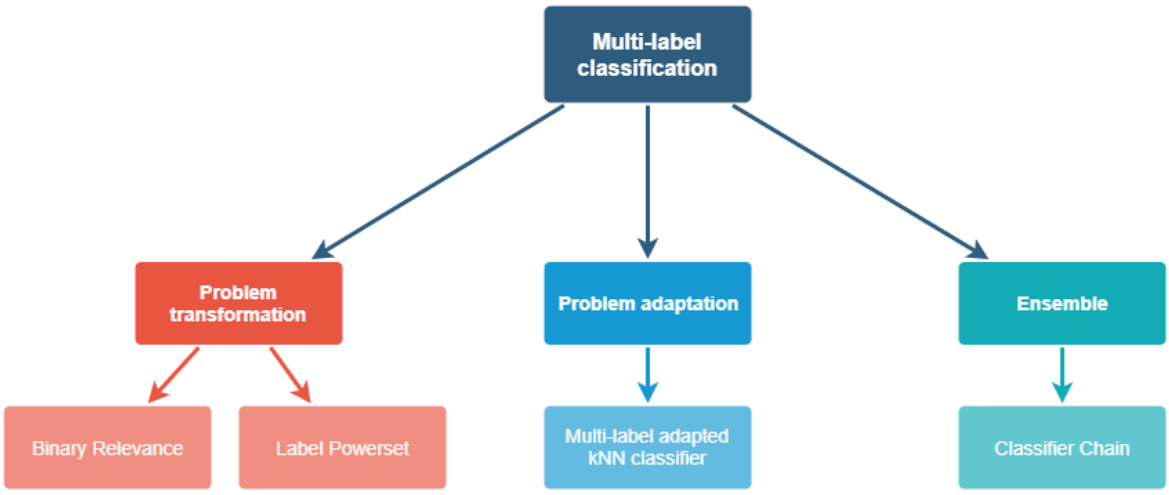
- $1 > y_i > 0$
- $\sum_i y_i = 1$

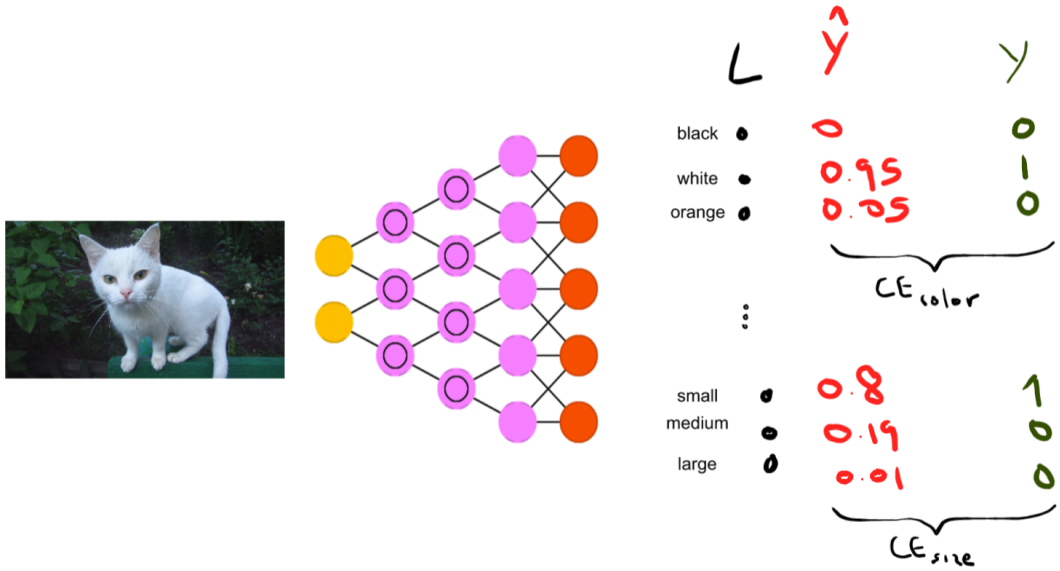
$$y_i = P(C_i \mid x)$$

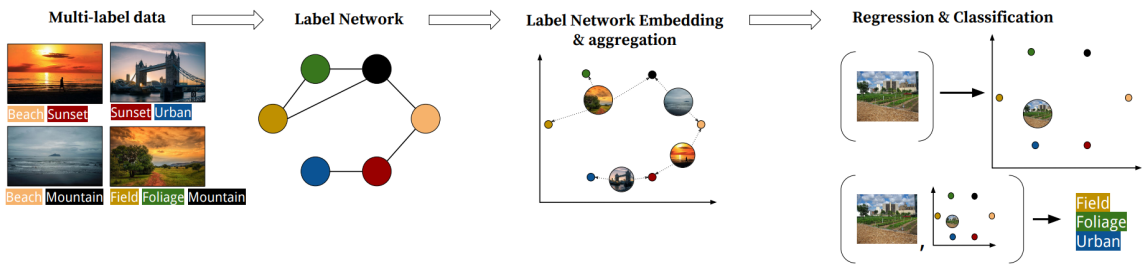


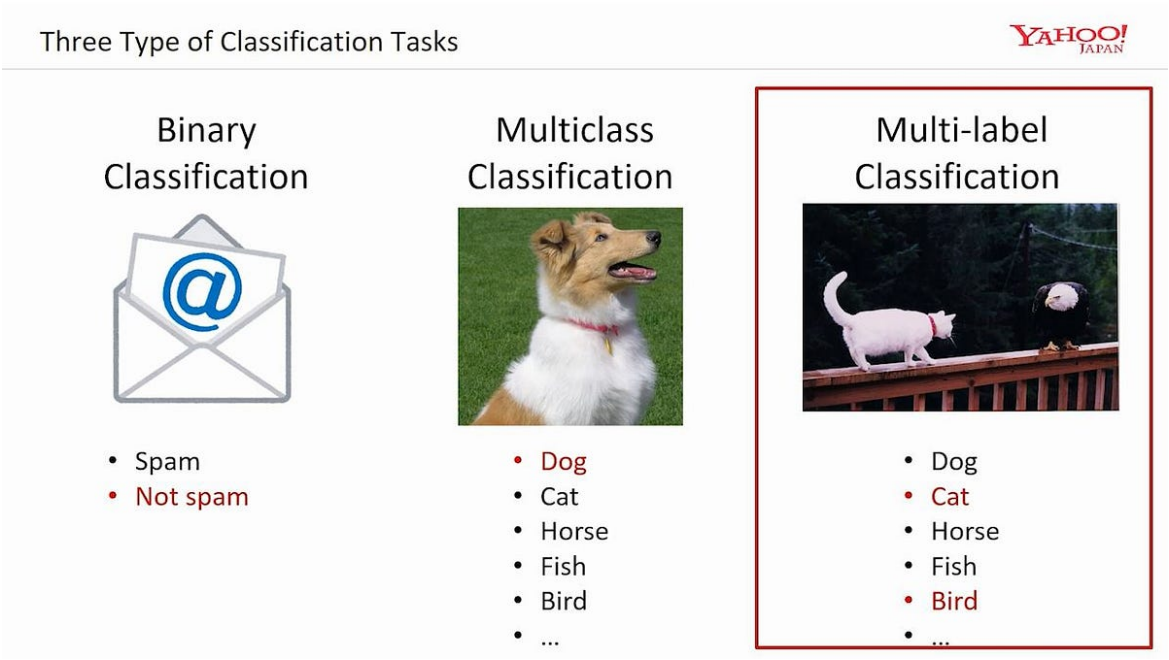


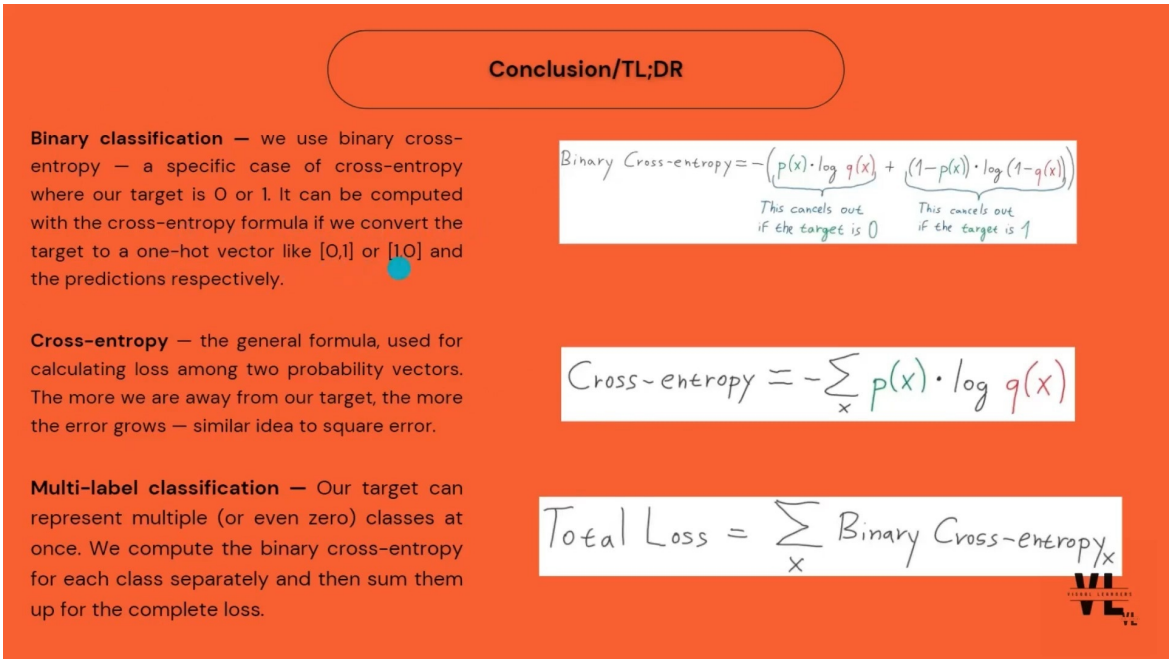
Multi-label Classification




L
ŷ
y
black
white
orange
small
medium
large
CE color
CE size


Multi-label data
Label Network
Label Network Embedding & aggregation
Regression & Classification
Beach Sunset
Sunset Urban
Beach Mountain
Field Foliage Mountain
Field
Foliage
Urban

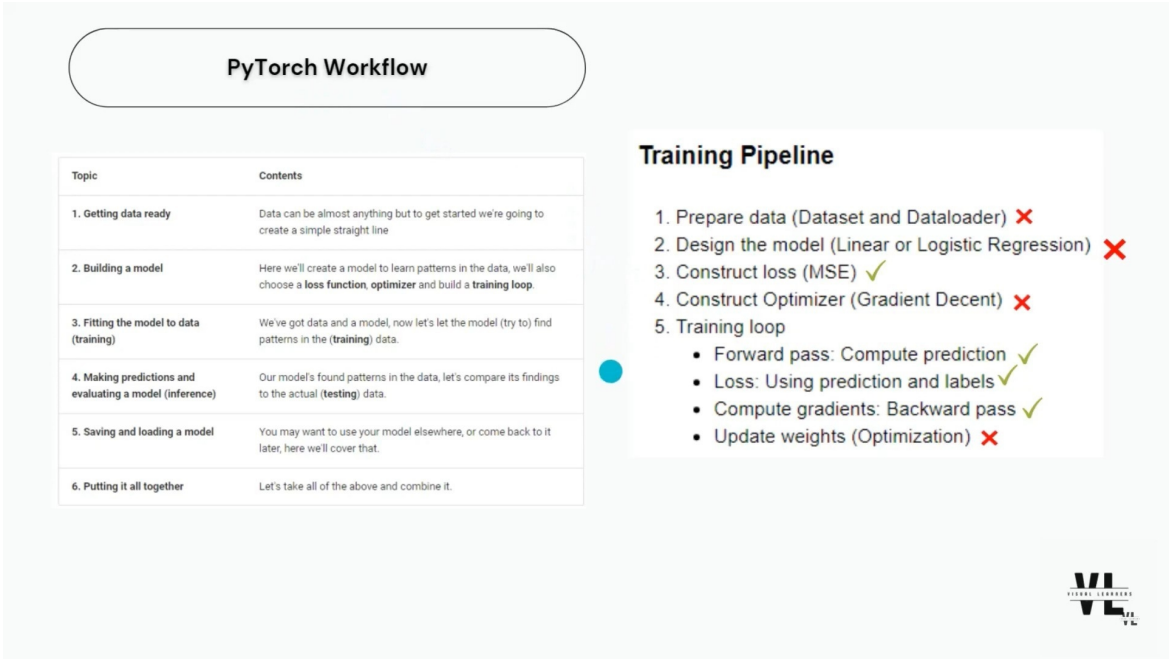

Three Type of Classification Tasks
YAHOO! JAPAN
Binary Classification
Spam
Not spam
Multiclass Classification
Dog
Cat
Horse
Fish
Bird
...
Multi-label Classification
Dog
Cat
Horse
Fish
Bird
...



Optimizers Theory

- `Optimizers` are methods that change the value of parameters so that losses reach the minimum.

PyTorch Workflow



Parameters vs Hyperparameters

Parameters

- Variables whose value we learn from training any machine learning model. e.g., **weights and biases** in Neural Networks

Hyperparameters

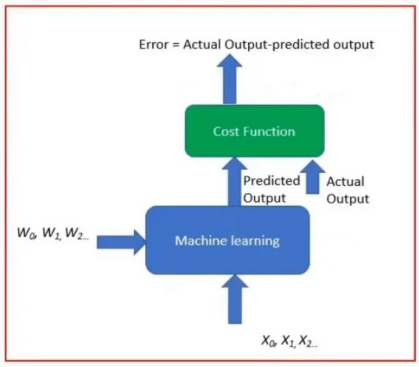
- A parameter whose value is used to control the learning process. E.g., The number of hidden layers in a Neural Network.
- They are parameters whose value is set before the model start training. They cannot be learned by fitting the model to the data.

Model hyperparameters in different models:

- Learning rate in gradient descent
- Number of iterations in gradient descent
- Number of layers in a Neural Network

PARAMETERS	HYPERPARAMETERS
They are required for making predictions	They are required for estimating the model parameters
They are estimated by optimization algorithms(Gradient Descent, Adam, Adagrad)	They are estimated by hyperparameter tuning
They are not set manually	They are set manually
The final parameters found after training will decide how the model will perform on unseen data	The choice of hyperparameters decide how efficient the training is. In gradient descent the learning rate decide how efficient and accurate the optimization process is in estimating the parameters

The objective of Machine Learning algorithm



- The goal of machine learning and deep learning is to reduce the difference between **the predicted output** and **the actual output**.
- This is also called as a **Cost function(C)** or **Loss function**. **Cost functions** are **convex functions**.
- As our goal is to minimize the cost function by finding the optimized value of weights. This can be done by updating the weights using **optimization algorithms.**
- To achieve this we run multiple iterations **with different weights**. This helps to find the **minimum cost**. This is **Gradient descent**.

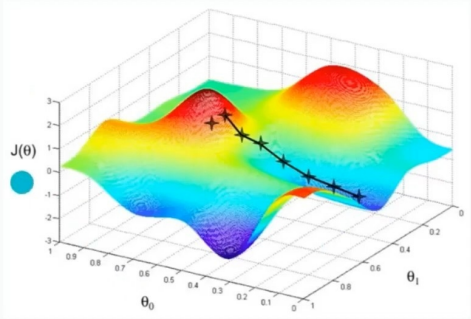
Optimizers

Optimizers

Methods that change the value of parameters so that losses reach the minimum. They are used to solve optimization problems by minimizing the cost function. **E.g., Gradient Descent**

Role of an optimizer

Optimizers update the **weight parameters** to **minimize the loss function**. Loss function acts as guides to the terrain telling optimizer **if it is moving in the right direction to reach the bottom of the valley, the global minimum**.



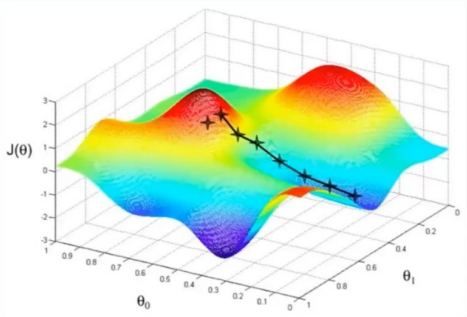
Gradient Descent

What is Gradient Descent?

Gradient descent is an iterative machine learning optimization algorithm to reduce the cost function so that we have models that makes accurate predictions.

Why do we need Gradient Descent?

In neural network our goal is to train the model to have optimized weights(w) to make better prediction. We get optimized weights using gradient descent.



Gradient Descent

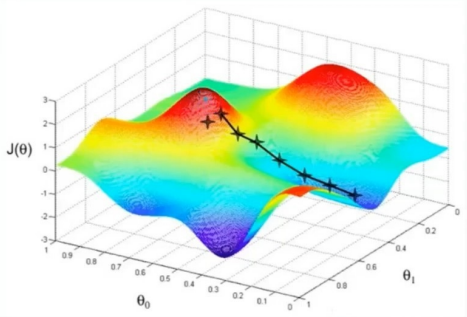
Mathematically

We randomly **initialize all the weights** for a neural network to a value **close to zero but not zero**.

We calculate **the gradient, ∂c/∂ω** which is a **partial derivative of cost with respect to weight**.

α is learning rate, helps adjust the weights with respect to gradient descent

Gradient indicates the direction of **increase (Steepest Ascent)**. As we want to find **the minimum point** in the valley we need to go **in the opposite direction** of the **gradient**. We **update parameters** in the **negative gradient direction to minimize the loss**



$$w := w - \alpha \frac{\partial c}{\partial \omega}$$

Gradient Descent

$$w := w - \alpha \frac{\partial c}{\partial \omega}$$

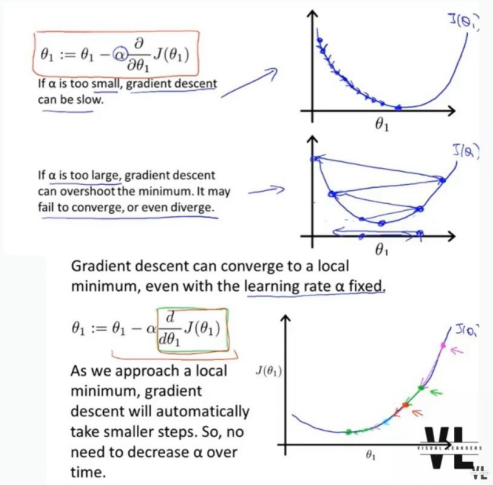
Learning Rate

Learning rate controls **how much we should adjust the weights with respect to the loss gradient**. Learning rates are **randomly initialized.**

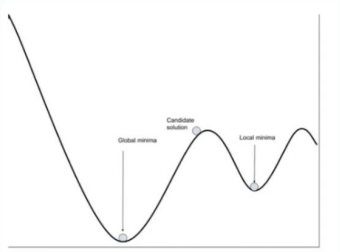
α is learning rate, helps adjust the weights with respect to gradient descent.

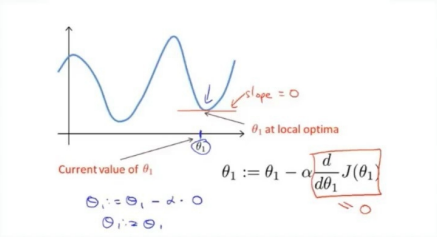
The lower the value of the learning rate, **the slower** will be the **convergence to global minima**.

A higher value for learning rate will not allow **the gradient descent to converge**



Local Vs Global Minima





Global Minima

- It is the smallest overall value of a function over its entire range. The loss function value reached a **minimum globally** over **the entire loss function domain**.

Local Minima

- The value of the loss function becomes minimum at that point in a **local region**. It is a point where the function value is smaller than nearby but possibly greater than at the lowest point.
- Some times we might get stuck in the local minima while trying to converge to the global minima.

Gradient Descent

1. **Batch Gradient Descent**

- In batch gradient we use the entire dataset to compute the gradient of the cost function for each iteration of the gradient descent and then update the weights.
- Since we use the entire dataset to compute the gradient convergence is slow.
- If the dataset is huge and contains millions or billions of data points then it is memory as well as computationally intensive.

Update w_i

Temp.	Visibility	Actual o/p	Predicted o/p
73	10	1	0
50	2	1	1
32	1	0	1
80	5	1	1
23	.1	0	1

Batch Gradient descent

Advantages of Batch Gradient Descent

- Theoretical analysis of weights and convergence rates are easy to understand

Disadvantages of Batch Gradient Descent

- Perform redundant computation for the same training example for large datasets
- Can be very slow and intractable as large datasets may not fit in the memory
- As we take the entire dataset for computation we can update the weights of the model for the new data

Gradient Descent

2. Mini Batch Gradient descent

- Mini-batch gradient is a variation of stochastic gradient descent where instead of single training example, mini-batch of samples is used.
- Mini batch gradient descent is widely used and converges faster and is more stable.
- Batch size can vary depending on the dataset.
- As we take a batch with different samples, it reduces the noise which is variance of the weight updates and that helps to have a more stable converge faster.

Update w_i

Update w_i

Temp.	Visibility	Actual o/p	Predicted o/p
73	10	1	0
50	2	1	1
32	1	0	1
80	5	1	1
23	.1	0	1
32	1	1	0

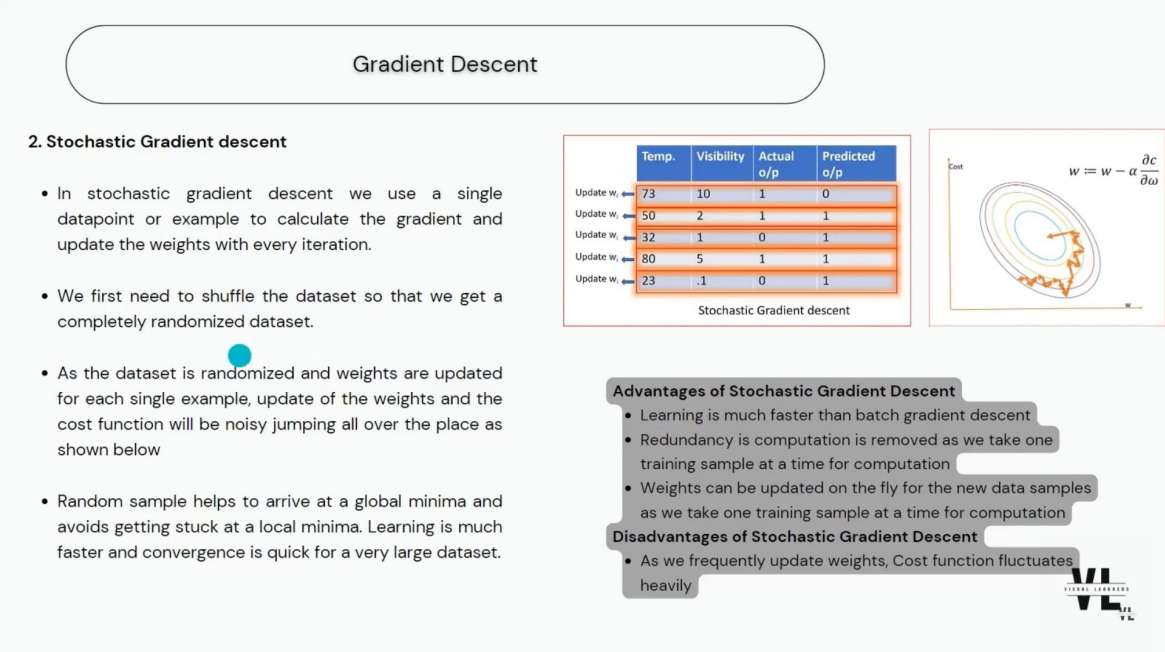
Mini batch Gradient descent

Advantages of Min Batch Gradient Descent

- Reduces variance of the parameter update and hence lead to stable convergence
- Speeds the learning
- Helpful to estimate the approximate location of the actual minimum

Disadvantages of Mini Batch Gradient Descent

- Loss is computed for each mini batch and hence total loss needs to be accumulated across all mini batches



Training Pipeline

1. Prepare Data(Dataset and Dataloader)
2. Design the model(Linear or Logistic Regression)
3. Construct Loss Function
4. Construct Optimizer
5. Training Loop

- Forward Pass: Compute prediction
- Loss: Using Prediction and Labels
- Backward Pass: Compute Gradients
- Update weights(Optimization)

```
import numpy as np
import torch
```

Forward Pass and Loss

```
# f = w * x = 2 * x

x = np.array([1, 2, 3, 4], dtype=np.float32)
y = np.array([2, 4, 6, 8], dtype=np.float32)

# initialize weight
w = 0.0

# model prediction
def forward(x):
```

```
  return w * x

# MSE Loss
def loss(y, y_pred):
  return ((y_pred - y)**2).mean()
```

Backward Pass - Backpropagation - Gradient

- MSE = 1/N * (w*x - y) * 2
- dl/dw = 1/N * 2x(w*x - y)

```
def gradient(x, y, y_pred):
  return np.dot(2*x, (y_pred - y)).mean()


print(f'Prediction before training: f(5) = {forward(5):.3f}')
```

```
Prediction before training: f(5) = 0.000
```

Training

```
learning_rate = 0.01

num_epoch = 10

for epoch in range(num_epoch):
  # Forward Pass - Prediction
  y_pred = forward(x)

  # Loss
  ls = loss(y, y_pred)

  # Gradient
  dw = gradient(x, y, y_pred)

  # Update Weights
  w -= learning_rate * dw

  if epoch % 1 == 0:
    print(f'Epoch {epoch}: Weight = {w:.3f}, Loss = {ls:.6f}, dw: {dw:.4f}')
```

```
Epoch 0: Weight = 1.200, Loss = 30.000000, dw: -120.0000
Epoch 1: Weight = 1.680, Loss = 4.799999, dw: -48.0000
Epoch 2: Weight = 1.872, Loss = 0.768000, dw: -19.2000
Epoch 3: Weight = 1.949, Loss = 0.122880, dw: -7.6800
Epoch 4: Weight = 1.980, Loss = 0.019661, dw: -3.0720
```

```
Epoch 5: Weight = 1.992, Loss = 0.003146, dw: -1.2288
Epoch 6: Weight = 1.997, Loss = 0.000503, dw: -0.4915
Epoch 7: Weight = 1.999, Loss = 0.000081, dw: -0.1966
Epoch 8: Weight = 1.999, Loss = 0.000013, dw: -0.0786
Epoch 9: Weight = 2.000, Loss = 0.000002, dw: -0.0315
```

```
print(f'Prediction after training: f(5) = {forward(5):.3f}')
```

```
Prediction after training: f(5) = 9.999
```

Training Pipeline

Preparing Data

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

y = torch.tensor([[2],
                  [4],
                  [6],
                  [8]], dtype=torch.float32)

x_test = torch.tensor([5], dtype=torch.float32)

n_samples, n_features = x.shape
print(x.shape)
```

```
torch.Size([4, 1])
```

Pipeline

```
input_size = n_features
output_size = n_features


class LinearRegression(nn.Module):
  def __init__(self, input_size, output_size):
    super().__init__()
    # Define Layers
    self.linear = nn.Linear(in_features=input_size, out_features=output_size)

  def forward(self, x):
    return self.linear(x)

# model prediction
```

```
model = nn.Linear(in_features=input_size, out_features=output_size)
print(f'Prediction before training: f(5) = {model(x_test).item():.3f}')

# MSE Loss
loss_fn = nn.MSELoss()

# optimizer
learning_rate = 0.01
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)

# Training
num_epoch = 100

for epoch in range(num_epoch):
  # Forward Pass - Prediction
  y_pred = model(x)

  # Loss
  ls = loss_fn(y, y_pred)

  # Backward - Backpropagation - Gradient
  ls.backward()

  # Update weight
  optimizer.step()

  # Clear Gradients
  optimizer.zero_grad()

  if (epoch+1) % 10 == 0:
    [weight, bias] = model.parameters()
    print(f'epoch {epoch+1}: loss = {ls:.8f}, weight = {weight[0][0].item():.3f}, bias = {

print(f'Prediction after training: f(5) = {model(x_test).item():.3f}')
```

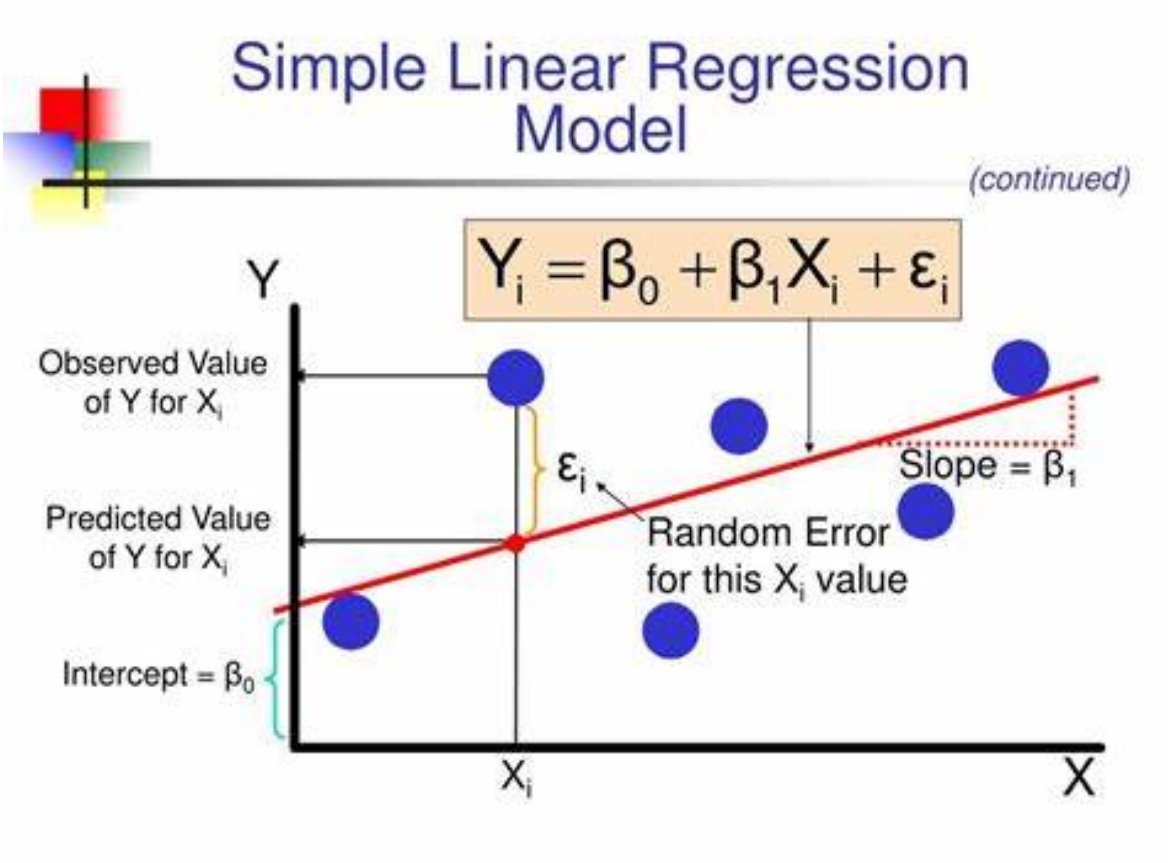
```
Prediction before training: f(5) = -3.357
epoch 10: loss = 2.09418464, weight = 1.377, bias = 0.562
epoch 20: loss = 0.12816277, weight = 1.709, bias = 0.652
epoch 30: loss = 0.07299092, weight = 1.768, bias = 0.650
epoch 40: loss = 0.06750808, weight = 1.783, bias = 0.633
epoch 50: loss = 0.06354676, weight = 1.791, bias = 0.615
epoch 60: loss = 0.05984720, weight = 1.797, bias = 0.597
epoch 70: loss = 0.05636378, weight = 1.803, bias = 0.579
epoch 80: loss = 0.05308308, weight = 1.809, bias = 0.562
epoch 90: loss = 0.04999341, weight = 1.814, bias = 0.545
epoch 100: loss = 0.04708350, weight = 1.820, bias = 0.529
Prediction after training: f(5) = 9.629
```

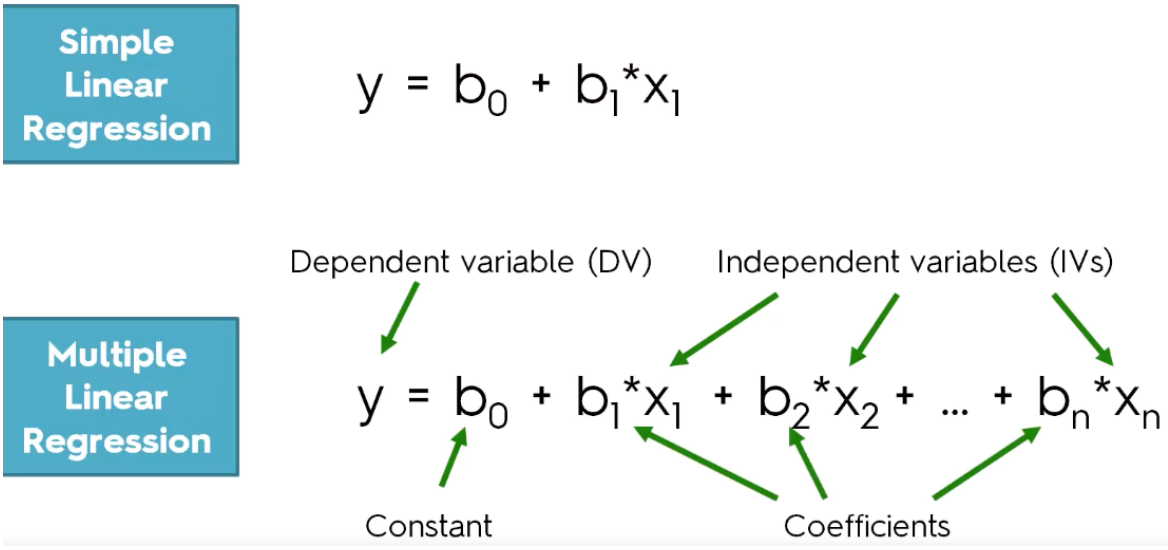
```
list(model.parameters())
```

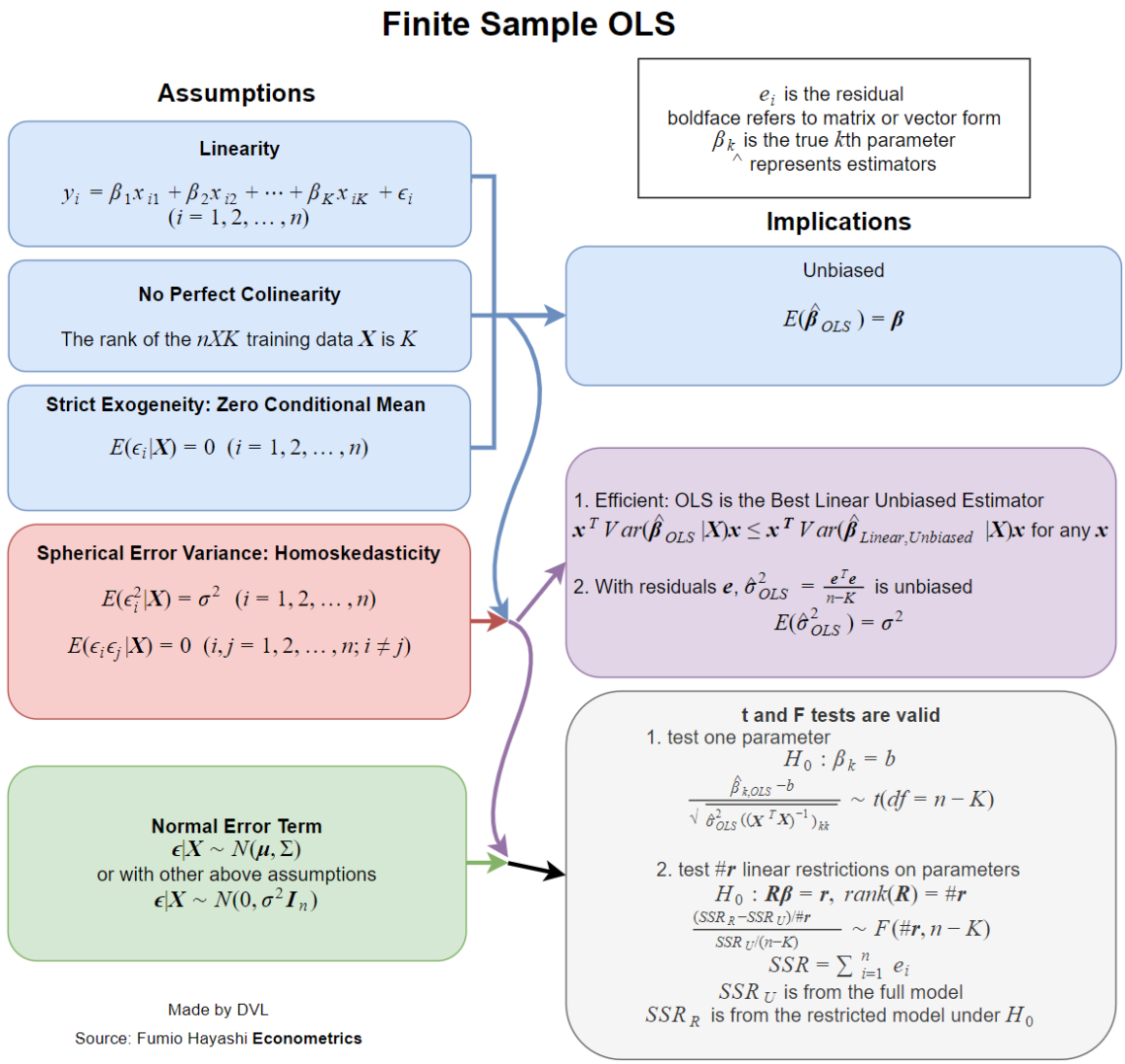
```
[Parameter containing:
 tensor([[1.8200]], requires_grad=True),
```

```
Parameter containing:
tensor([0.5294], requires_grad=True)]
```

Linear Regression





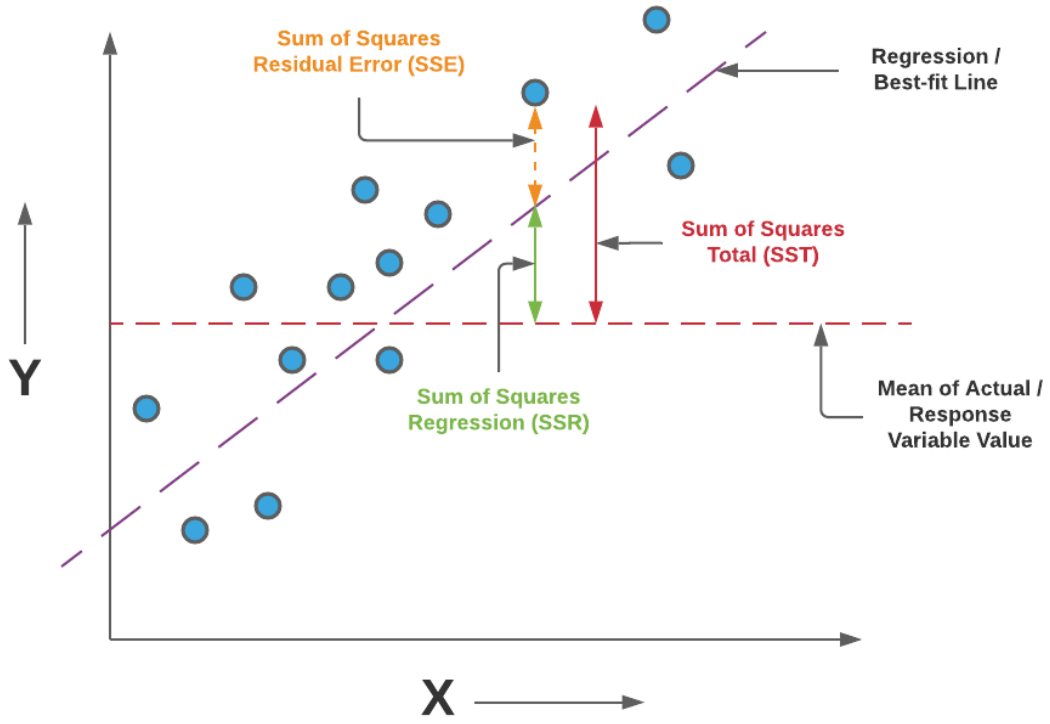


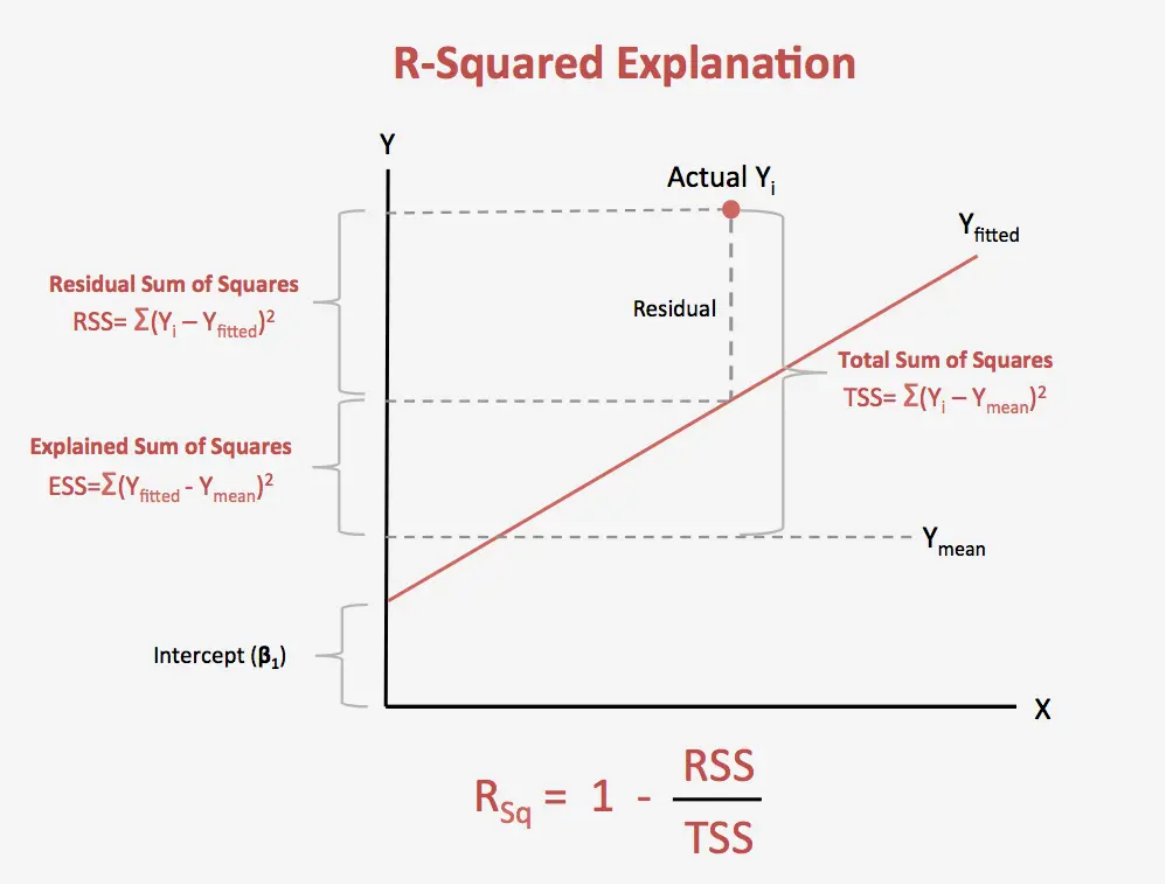
MULTIPLE REGRESSION ASSUMPTIONS

Multiple linear regression has the same underlying assumptions as single independent variable linear regression and some additional ones.

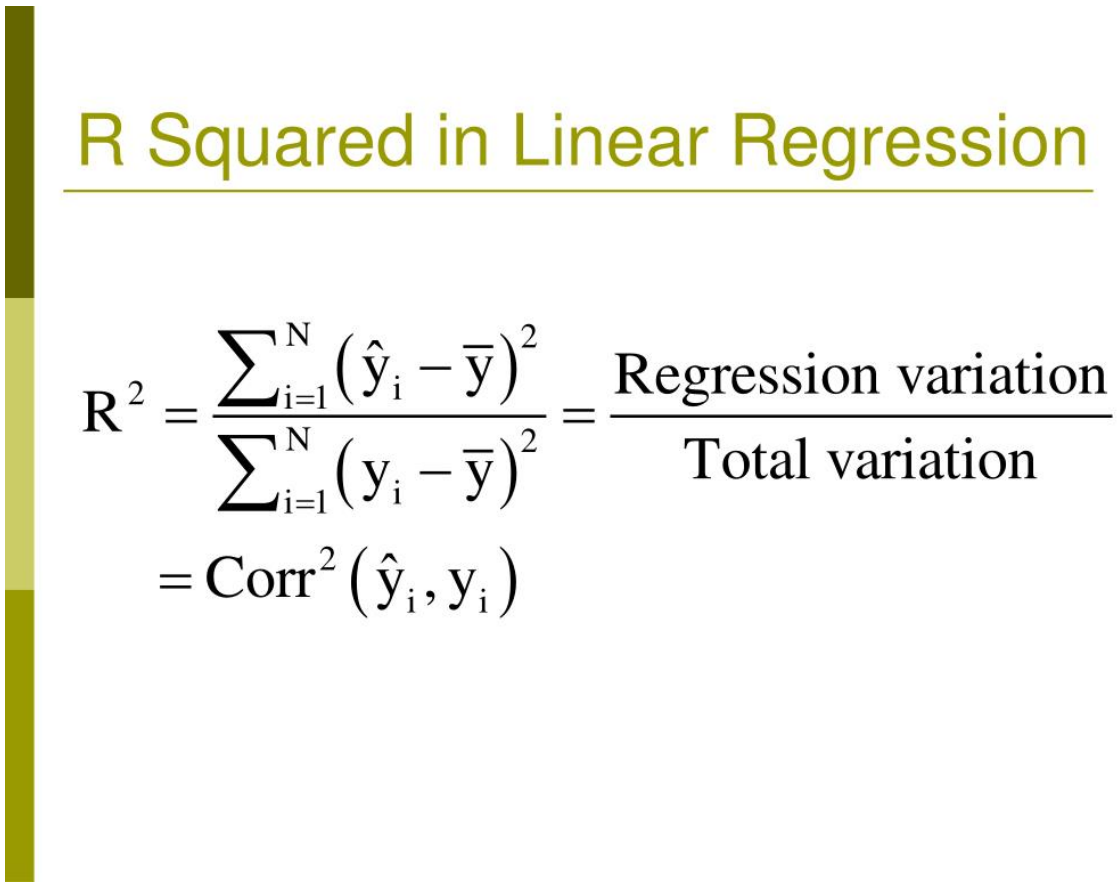
1. The relationship between the dependent variable, *Y*, and the independent variables (X_1, X_2, . . . , X_k) is linear.
2. The independent variables (X_1, X_2, . . . , X_k) are not random. Also, **no exact linear relation exists between two or more of the independent variables**.
3. The expected value of the error term, conditioned on the independent variables, is zero.
4. The variance of the error term is the same for all observations.
5. The error term is uncorrelated across observations: $E(\epsilon_i\epsilon_j) = 0$, $j \neq i$.
6. The error term is normally distributed.

R-Squared(Coefficient of Determination)





$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \overline{y})^2}$$




```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import torch
import torch.nn as nn
```

1. Get Data: Using sklearn Regression Dataset

- from sklearn import datasets

```
from sklearn import datasets

x, y = datasets.make_regression(n_samples=100,
                                n_features=1,
                                noise=20,
                                random_state=1)
```

```
x.shape, y.shape
```

```
((100, 1), (100,))
```

```
x[:5]
```

```
array([[-0.61175641],
       [-0.24937038],
       [ 0.48851815],
       [ 0.76201118],
       [ 1.51981682]])
```

```
y[:5]
```

```
array([-55.5385928 , -10.66198475,  22.7574081 , 101.09612881,
       144.3375575 ])
```

```
x.dtype, y.dtype
```

```
(dtype('float64'), dtype('float64'))
```

```
y.dtype
```

```
dtype('float64')
```

```
# Converting np.float64 to torch.float32
x_train = torch.from_numpy(x.astype(np.float32))
y_train = torch.from_numpy(y.astype(np.float32))
```

```
x[:5]
```

```
array([[-0.61175641],
       [-0.24937038],
       [ 0.48851815],
       [ 0.76201118],
       [ 1.51981682]])
```

```
y[:5]
```

```
array([-55.5385928 , -10.66198475,  22.7574081 , 101.09612881,
       144.3375575 ])
```

```
y_train.shape
```

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

```
print(y_train.shape)
print(y_train.view(y_train.shape[0] ,1).shape)
```

```
torch.Size([100])
torch.Size([100, 1])
```

```
y_train = y_train.view(y_train.shape[0] ,1)
```

2. Build Model

```
n_samples, n_features = x_train.shape
n_samples, n_features

input_size = n_features
output_size = 1

model = nn.Linear(in_features=input_size, out_features=output_size)
```

3. Loss and Optimizer

```
learning_rate = 0.01
loss_fn = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
```

4. Training Loop

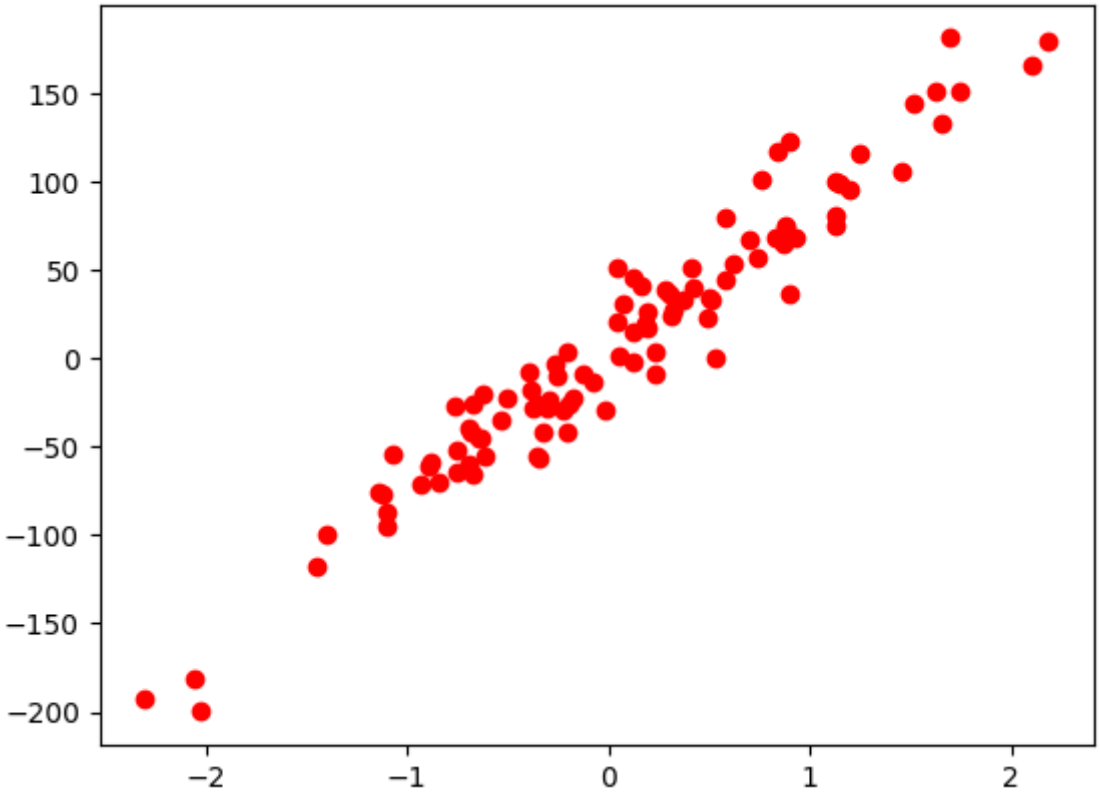
```
num_epoch = 100

for epoch in range(num_epoch):
  # Forward Pass
  y_pred = model(x_train)

  # Loss
  loss = loss_fn(y_pred, y_train)

  # Backward Pass
  loss.backward()

  # Update Parameters
  optimizer.step()

  # Clear Gradients
  optimizer.zero_grad()

  if (epoch+1) % 10 == 0:
    print(f'Epoch: {epoch+1}, Loss: {loss.item():.6f}')
```

```
Epoch: 10, Loss: 4306.095703
Epoch: 20, Loss: 3213.908203
Epoch: 30, Loss: 2423.815430
Epoch: 40, Loss: 1851.640991
Epoch: 50, Loss: 1436.864014
Epoch: 60, Loss: 1135.907715
Epoch: 70, Loss: 917.350952
Epoch: 80, Loss: 758.508362
Epoch: 90, Loss: 642.980835
Epoch: 100, Loss: 558.900818
```

5. Visualize

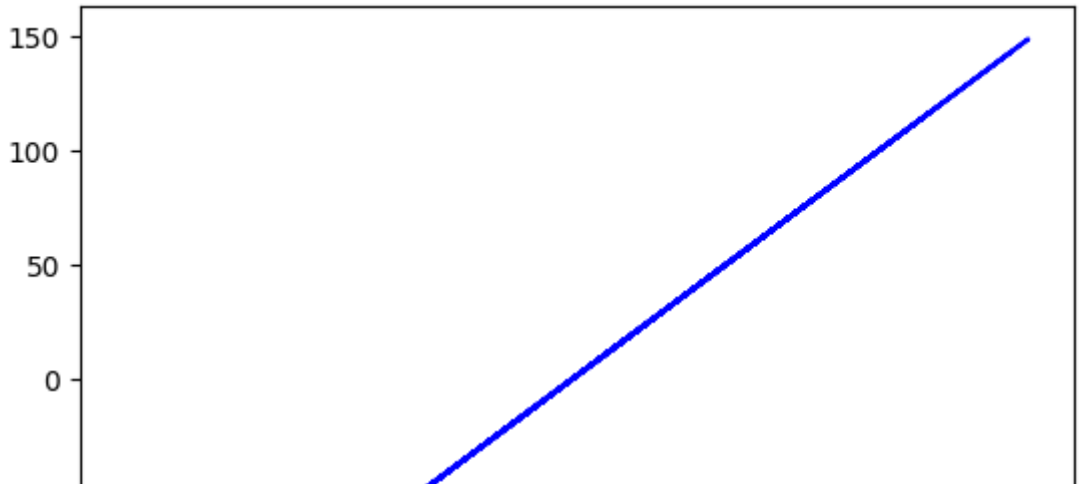
```
plt.plot(x, y, 'ro')
```

```
[<matplotlib.lines.Line2D at 0x7fcd6438cc10>]
```



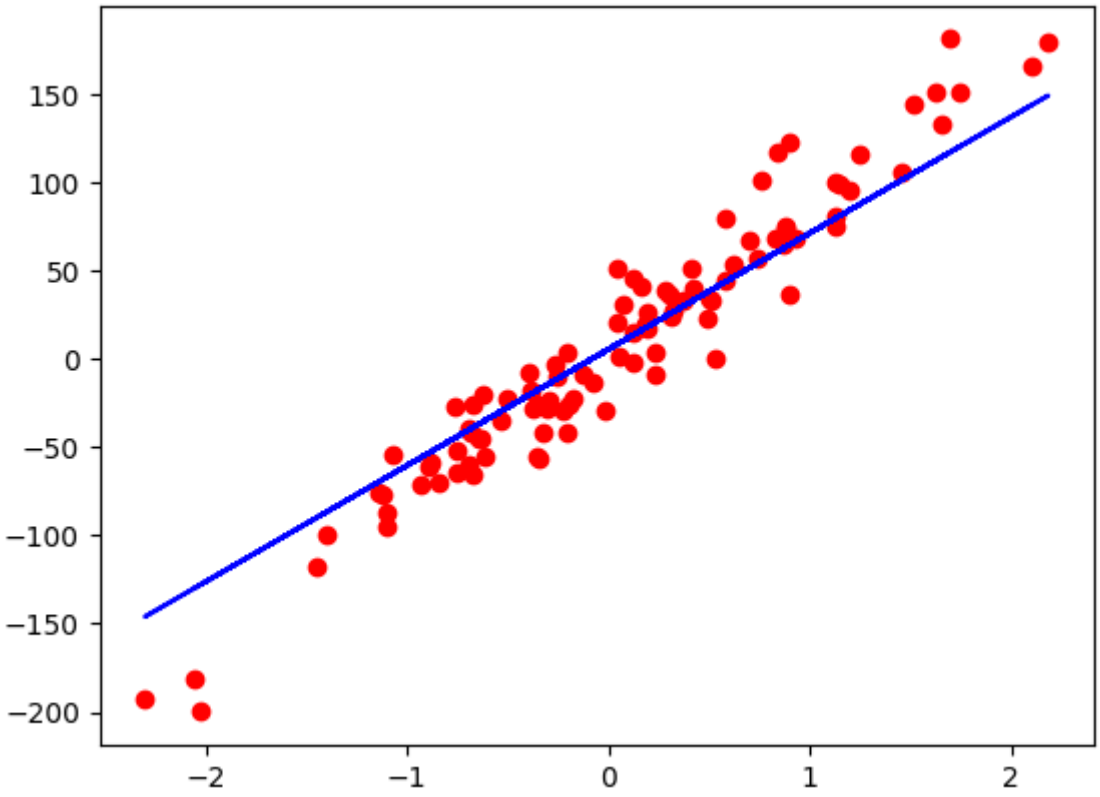

```
y_preds = model(x_train).detach().numpy()
print(y_preds.dtype)
plt.plot(x_train, y_preds, 'b')
```

```
float32
[<matplotlib.lines.Line2D at 0x7fcd6438f460>]
```

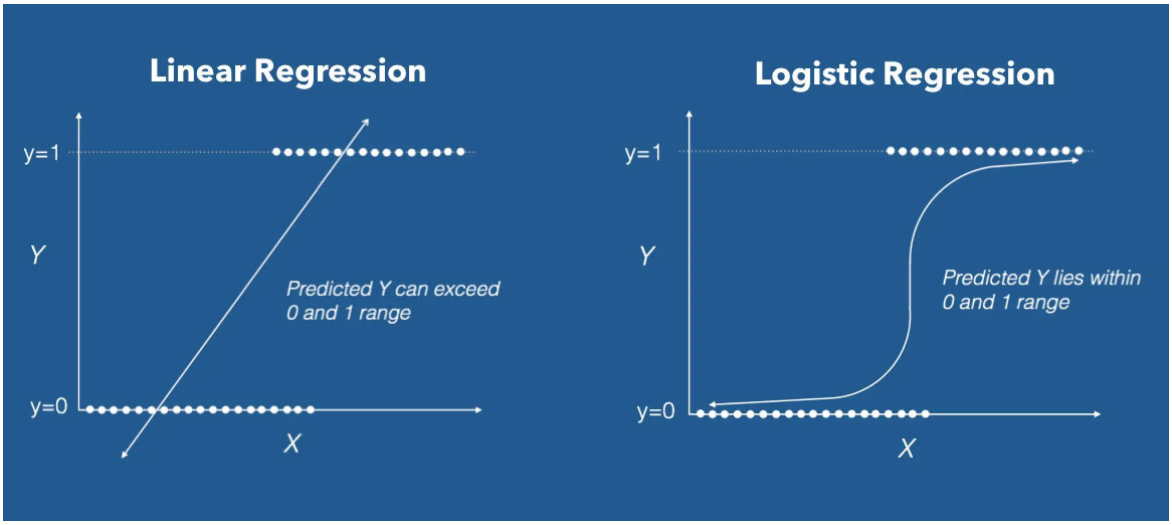



```
plt.plot(x, y, 'ro')
plt.plot(x_train, y_preds, 'b')
```

```
[<matplotlib.lines.Line2D at 0x7fcd6210af50>]
```

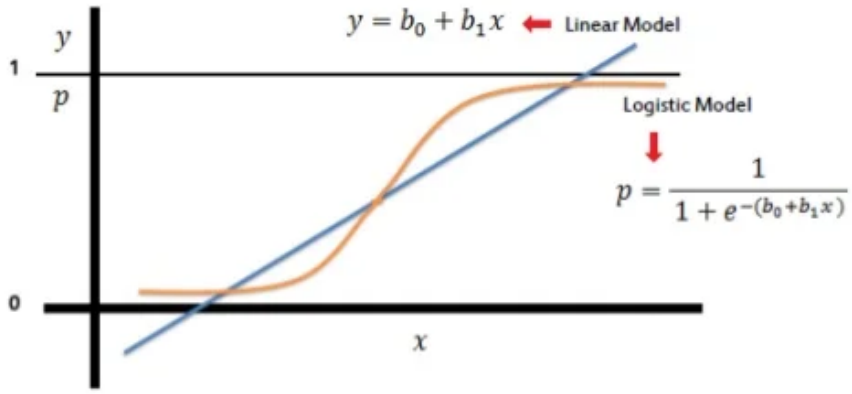


Logistic Regression

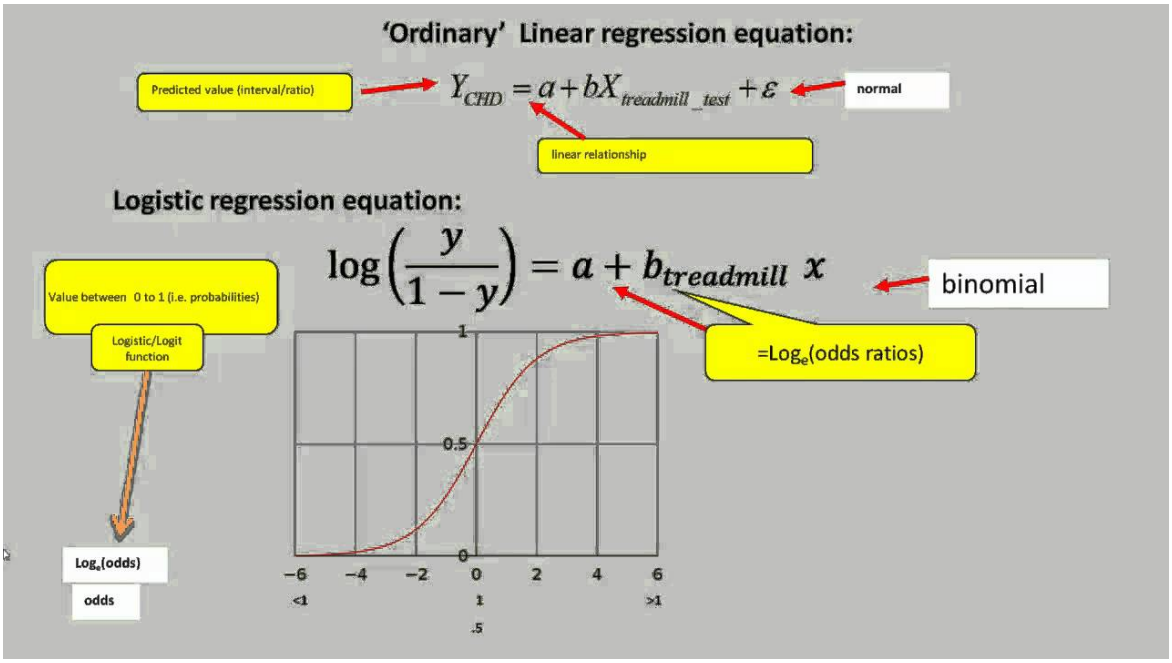


Logistic Regression

Logistic Regression from scratch



Prashant Mudgal



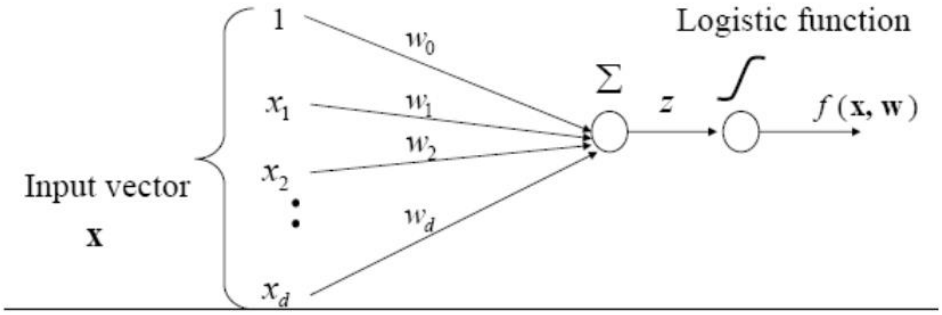
Logistic regression model

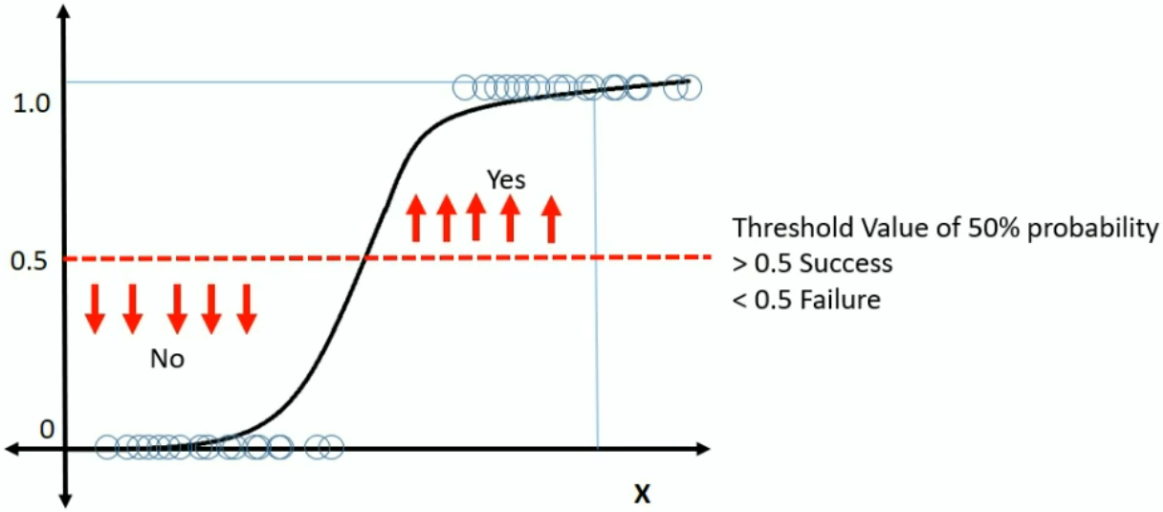
- **Defines a linear decision boundary**
- **Discriminant functions:**

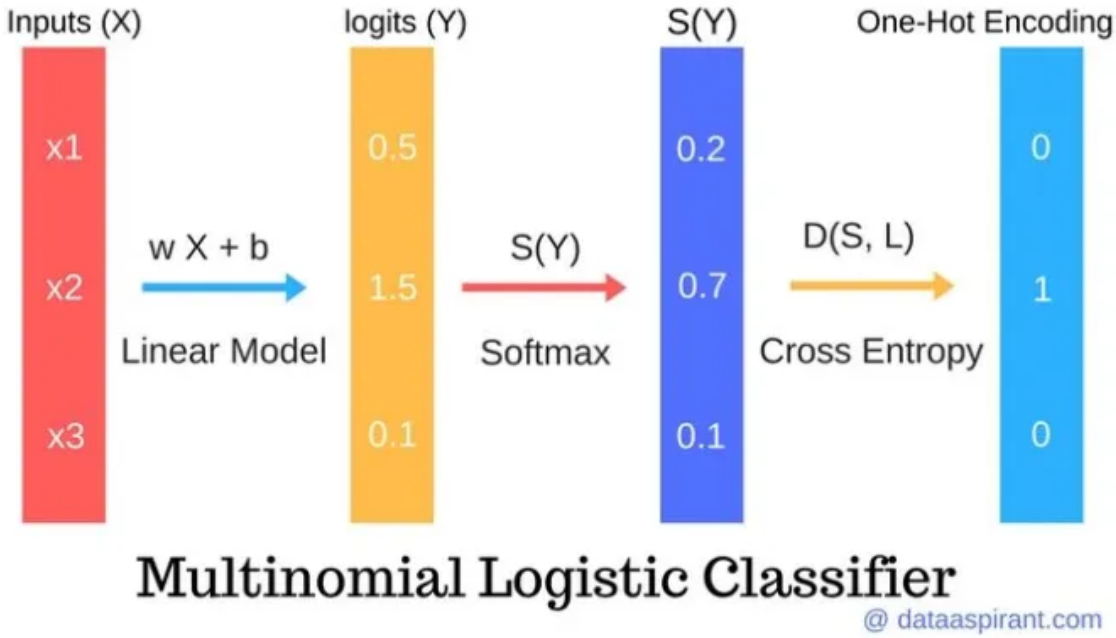
$$g_1(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x}) \qquad g_0(\mathbf{x}) = 1 - g(\mathbf{w}^T\mathbf{x})$$

- where $g(z) = 1/(1+e^{-z})$ - is a logistic function

$$f(\mathbf{x}, \mathbf{w}) = g_1(\mathbf{w}^T\mathbf{x}) = g(\mathbf{w}^T\mathbf{x})$$







Implementing Softmax using Numpy

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import torch
import torch.nn as nn


# sum of all probability is 1
def softmax(x):
  return np.exp(x) / np.sum(np.exp(x), axis=0)

x = np.array([2.0, 1.0, 0.1])
print('softmax numpy: ', softmax(x))
```

```
softmax numpy:  [0.65900114 0.24243297 0.09856589]
```

```
x = torch.tensor([2.0, 1.0, 0.1])

print('torch.softmax: ', torch.softmax(x, dim=0))
```

```
     torch.softmax:  tensor([0.6590, 0.2424, 0.0986])
```

Implementing Cross Entropy using Numpy

```
def cross_entroy(y_pred, y_actual):
  EPS = 1e-15
  y_pred = np.clip(y_pred, EPS, 1-EPS)
  loss = -np.sum(y_actual * np.log(y_pred))
  return loss


y_actual = np.array([1, 0, 0])
y_pred_good = np.array([0.7, 0.2, 0.1])
y_pred_bad = np.array([0.1, 0.3, 0.6])

loss_1 = cross_entroy(y_pred_good, y_actual)
loss_2 = cross_entroy( y_pred_bad, y_actual)

print(f'Loss_1 numpy: {loss_1:.6f}')
print(f'Loss_2 numpy: {loss_2:.6f}')
```

```
     Loss_1 numpy: 0.356675
     Loss_2 numpy: 2.302585
```

Implementing Cross Entroy using Pytorch

```
y_actual = torch.tensor([0])
y_pred_good = torch.tensor([[2.0, 1.0, 0.1]], dtype=torch.float32)
y_pred_bad = torch.tensor([[0.5, 2.0, 0.3]], dtype=torch.float32)

loss_fn = nn.CrossEntropyLoss()

loss_1 = loss_fn(y_pred_good, y_actual)
loss_2 = loss_fn(y_pred_bad, y_actual)

print(f'nn.CrossEntropyLoss good: {loss_1:.6f}')
print(f'nn.CrossEntropyLoss bad: {loss_2:.6f}')
print(f'nn.CrossEntropyLoss good: {loss_1.item()}')
print(f'nn.CrossEntropyLoss bad: {loss_2.item()}')
```

```
     nn.CrossEntropyLoss good: 0.417030
     nn.CrossEntropyLoss bad: 1.840616
     nn.CrossEntropyLoss good: 0.4170299470424652
     nn.CrossEntropyLoss bad: 1.840616226196289
```

```
torch.max(y_pred_good, 1)
```

```
torch.return_types.max(
values=tensor([2.]),
indices=tensor([0]))
```

```
# get predictions
values_1, indices_1 = torch.max(y_pred_good, 1)
values_2, indices_2 = torch.max(y_pred_bad, 1)
print(f'Actual Class: {y_actual}, y_pred_good: {indices_1}, y_pred_bad: {indices_2}')
```

```
Actual Class: tensor([0]), y_pred_good: tensor([0]), y_pred_bad: tensor([1])
```

```
# get predictions
values_1, indices_1 = torch.max(y_pred_good, 1)
values_2, indices_2 = torch.max(y_pred_bad, 1)
print(f'Actual Class: {y_actual.item()}, y_pred_good: {indices_1.item()}, y_pred_bad: {ind
```

```
Actual Class: 0, y_pred_good: 0, y_pred_bad: 1
```

```
y_actual = torch.tensor([2, 0, 1])
y_pred_good = torch.tensor([[0.1, 0.2, 3.9],
                            [1.2, 0.1, 0.3],
                            [0.3, 2.2, 0.2]], dtype=torch.float32)
y_pred_bad = torch.tensor([[0.9, 0.2, 0.1],
                           [0.1, 0.3, 1.5],
                           [1.2, 0.2, 0.5]], dtype=torch.float32)

loss_fn = nn.CrossEntropyLoss()

loss_1 = loss_fn(y_pred_good, y_actual)
loss_2 = loss_fn(y_pred_bad, y_actual)

print(f'nn.CrossEntropyLoss good: {loss_1:.6f}')
print(f'nn.CrossEntropyLoss bad: {loss_2:.6f}')
print(f'nn.CrossEntropyLoss good: {loss_1.item()}')
print(f'nn.CrossEntropyLoss bad: {loss_2.item()}')

# get predictions
print("\nMake Prediction...")
values_1, indices_1 = torch.max(y_pred_good, 1)
values_2, indices_2 = torch.max(y_pred_bad, 1)
print(f'Actual Class: {y_actual}, y_pred_good: {indices_1}, y_pred_bad: {indices_2}')
```

```
nn.CrossEntropyLoss good: 0.283422
nn.CrossEntropyLoss bad: 1.641845
nn.CrossEntropyLoss good: 0.28342217206954956
nn.CrossEntropyLoss bad: 1.6418448686599731

Make Prediction...
Actual Class: tensor([2, 0, 1]), y_pred_good: tensor([2, 0, 1]), y_pred_bad: tensor([
```

Binary Classification - Sigmoid

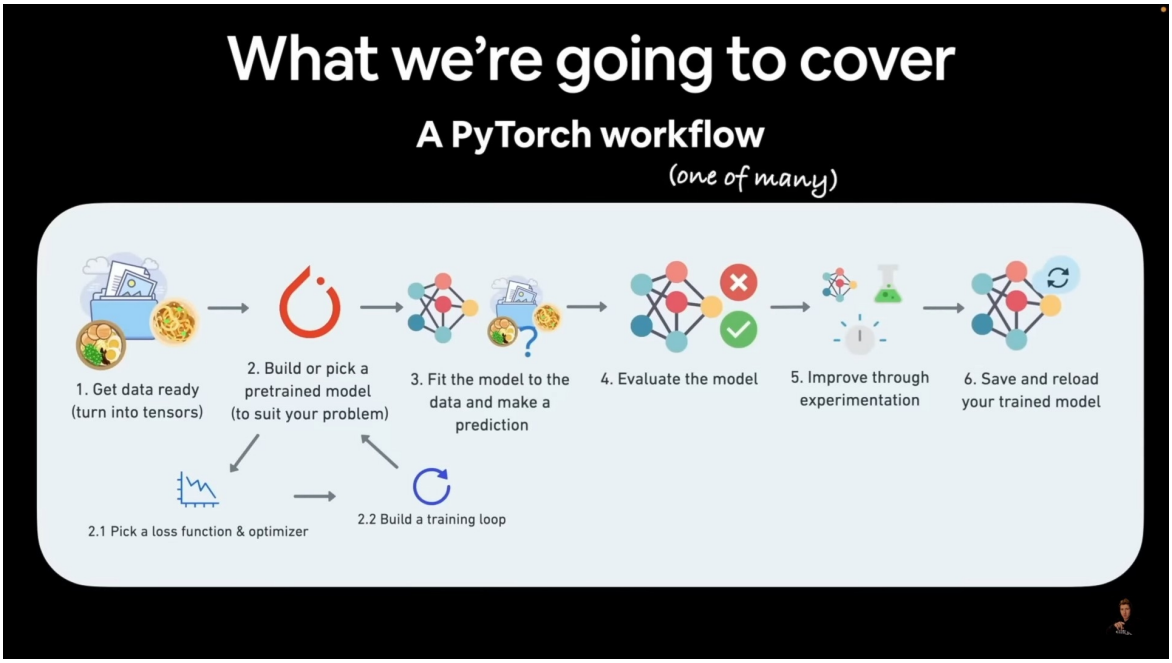
```
class NeuralNet_1(nn.Module):
  def __init__(self, input_size, hidden_size, output_size):
    super().__init__()
    self.linear_1 = nn.Linear(input_size, hidden_size)
    self.relu = nn.ReLU()
    self.linear_2 = nn.Linear(hidden_size, output_size)

  def forward(self, x):
    output = self.linear_1(x)
    output = self.relu(output)
    output = self.linear_2(output)
    # sigmoid at the end
    y_pred = torch.sigmoid(output)
    return y_pred


model = NeuralNet_1(input_size=28*28, hidden_size=5, output_size=1)

criterion = nn.BCELoss()
```

Binary Classification - Softmax

```
class NeuralNet_2(nn.Module):
  def __init__(self, input_size, hidden_size, output_size):
    super().__init__()
    self.linear_1 = nn.Linear(input_size, hidden_size)
    self.relu = nn.ReLU()
    self.linear_2 = nn.Linear(hidden_size, output_size)

  def forward(self, x):
    output = self.linear_1(x)
    output = self.relu(output)
    output = self.linear_2(output)

    return output


model = NeuralNet_2(input_size=28*28, hidden_size=5, output_size=3)

criterion = nn.CrossEntropyLoss()
```

Logistis Regression Project

- Dataset and Dataloader




```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

1. Get Data and Explore Data

1.1 Load Data

```
binaryclass = datasets.load_breast_cancer()
```

1.2 Explore Data

```
X, y = binaryclass.data, binaryclass.target

print('X.shape: ', X.shape)
print('y.shape: ', y.shape)
print('X: ', X[0])
print('y: ', binaryclass.target[0])
```

```
X.shape:  (569, 30)
y.shape:  (569,)
X:  [1.799e+01 1.038e+01 1.228e+02 1.001e+03 1.184e-01 2.776e-01 3.001e-01
 1.471e-01 2.419e-01 7.871e-02 1.095e+00 9.053e-01 8.589e+00 1.534e+02
 6.399e-03 4.904e-02 5.373e-02 1.587e-02 3.003e-02 6.193e-03 2.538e+01
 1.733e+01 1.846e+02 2.019e+03 1.622e-01 6.656e-01 7.119e-01 2.654e-01
 4.601e-01 1.189e-01]
y:  0
```

1.3 train-test split

```
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)

print('X_train shape:',X_train.shape)
print('X_test shape:',X_test.shape)
print('y_train shape:',y_train.shape)
print('y_test shape:',y_test.shape)
```

```
X_train shape: (455, 30)
X_test shape: (114, 30)
y_train shape: (455,)
y_test shape: (114,)
```

1.4 Normalizing the training data

```
print('Before normalization x:', X_train[0],'y:',y_train[0])
print('X_train max: ', X_train.max())
print('X_train min: ', X_train.min())
```

```
Before normalization x: [9.029e+00 1.733e+01 5.879e+01 2.505e+02 1.066e-01 1.413e-01
 4.375e-02 2.111e-01 8.046e-02 3.274e-01 1.194e+00 1.885e+00 1.767e+01
 9.549e-03 8.606e-02 3.038e-01 3.322e-02 4.197e-02 9.559e-03 1.031e+01
 2.265e+01 6.550e+01 3.247e+02 1.482e-01 4.365e-01 1.252e+00 1.750e-01
 4.228e-01 1.175e-01] y: 1
X_train max:  4254.0
X_train min:  0.0
```

```
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

print('After Normalization x:', X_train[0],'y:',y_train[0])
print('X_train max: ', X_train.max())
print('X_train min: ', X_train.min())
```

```
After Normalization x: [-1.44075296 -0.43531947 -1.36208497 -1.1391179   0.78057331
  2.82313451 -0.11914956  1.09266219  2.45817261 -0.26380039 -0.01605246
 -0.47041357 -0.47476088  0.83836493  3.25102691  8.43893667  3.39198733
  2.62116574  2.06120787 -1.23286131 -0.47630949 -1.24792009 -0.97396758
  0.72289445  1.18673232  4.67282796  0.9320124   2.09724217  1.88645014] y: 1
X_train max:  11.31029435566468
X_train min:  -3.100011345678773
```

1.5 Converting from numpy array to a tensor

```
X_train = torch.from_numpy(X_train.astype(np.float32))
X_test = torch.from_numpy(X_test.astype(np.float32))
y_train = torch.from_numpy(y_train.astype(np.float32))
y_test = torch.from_numpy(y_test.astype(np.float32))

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
torch.Size([455, 30])
torch.Size([114, 30])
torch.Size([455])
torch.Size([114])
```

1.6 Reshaping the label

```
y_train = y_train.view(y_train.shape[0], 1)
y_test = y_test.view(y_test.shape[0], 1)
print(y_train.shape)
print(y_test.shape)
```

```
torch.Size([455, 1])
torch.Size([114, 1])
```

2. Build Model

2.1 Modeling

```
class LogisticRegression(nn.Module):
  def __init__(self, n_features):
    super().__init__()
    self.linear = nn.Linear(n_features, 1)

  def forward(self, x):
    output = self.linear(x)
    y_pred = torch.sigmoid(output)
    return y_pred


n_samples,·n_features·=·X_train.shape

model·=·LogisticRegression(n_features)
```

```
model(X_train).shape
```

```
torch.Size([455, 1])
```

2.2 Loss and Optimeizer

```
learning_rate = 0.01

loss_fn = nn.BCELoss()

optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
```

2.3 Training Loop

```
num_epoch = 100

for epoch in range(num_epoch):
  # 1. Forward Pass - Make Prediction
  y_pred = model(X_train)

  # 2. Loss
  loss = loss_fn(y_pred, y_train)

  # 3. Backward Pass - Backpropagation - Gradient
  loss.backward()

  # 4. Update Weights
  optimizer.step()

  # 5. Clear Gradients
  optimizer.zero_grad()

  # 6. Show the outcome
  if (epoch + 1) % 10 == 0:
    print(f'Epoch: {epoch + 1}, loss = {loss.item():.6f}')
```

```
Epoch: 10, loss = 0.553902
Epoch: 20, loss = 0.468216
Epoch: 30, loss = 0.411383
Epoch: 40, loss = 0.371005
Epoch: 50, loss = 0.340780
Epoch: 60, loss = 0.317221
Epoch: 70, loss = 0.298268
Epoch: 80, loss = 0.282630
Epoch: 90, loss = 0.269461
Epoch: 100, loss = 0.258182
```

3. Make a prediction

```
print(model(X_test[0]))
print(y_test[0])
```

```
tensor([0.6669], grad_fn=<SigmoidBackward0>)
tensor([1.])
```

✓ 0s completed at 5:09 PM