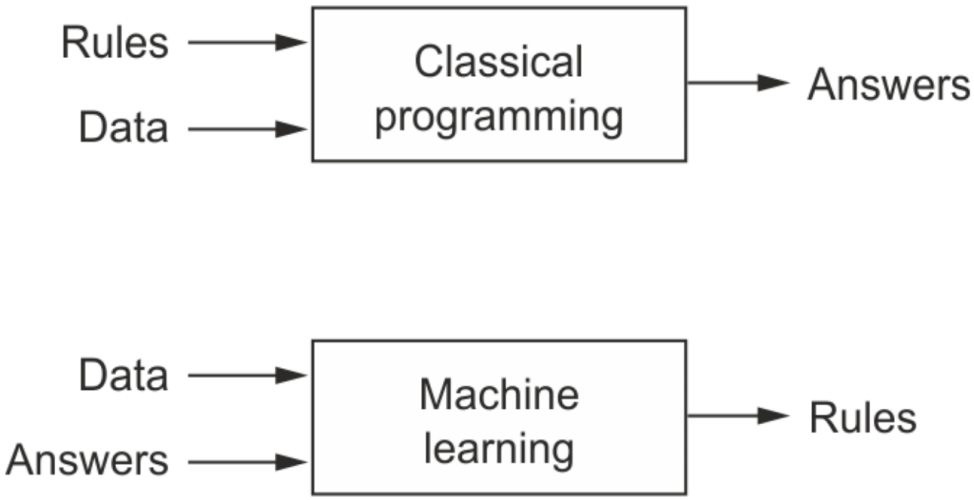
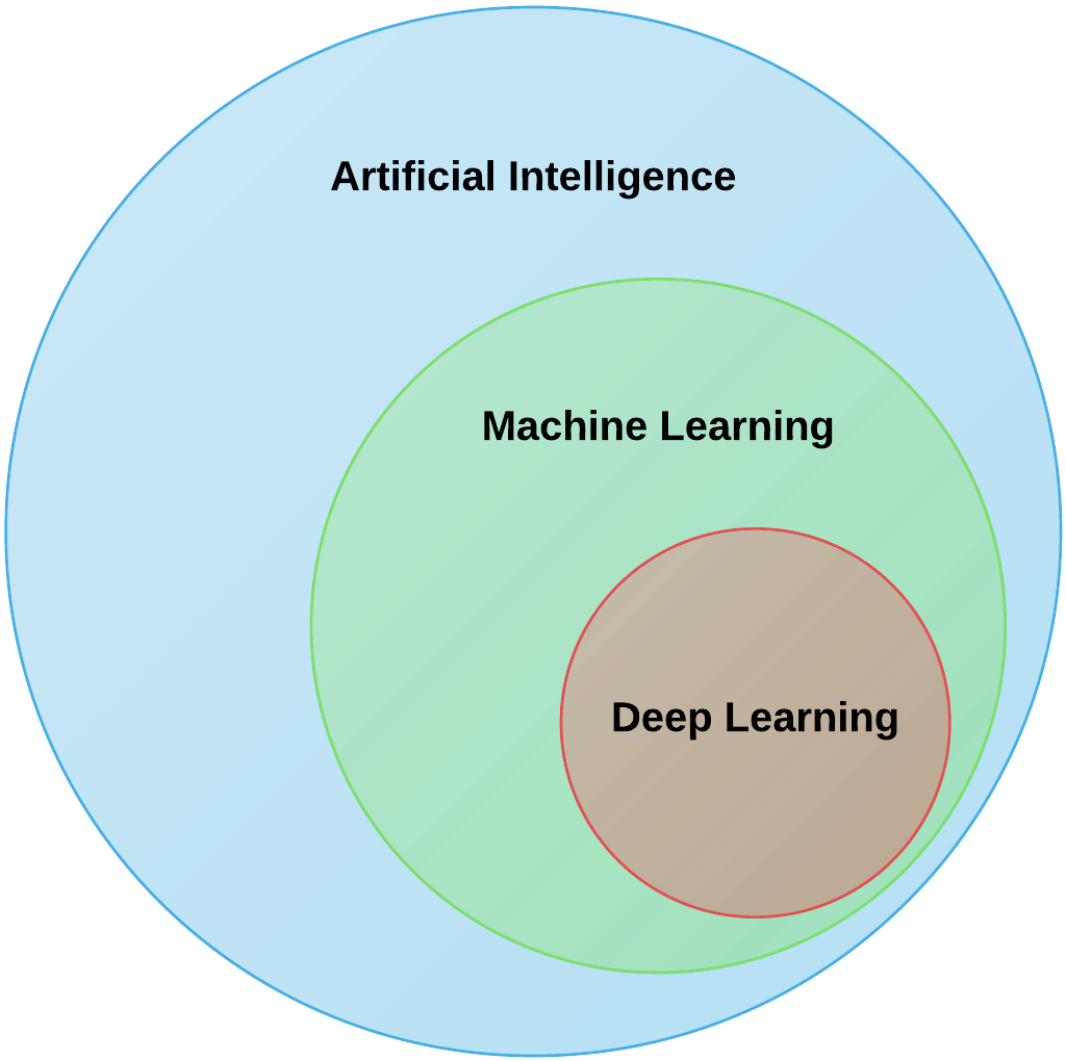
Deep Learning with PyTorch: Zero to GANs

> https://youtube.com/playlist?list=PLyMom0n-MBroupZiLfVSZqK5asX8KfoHL

Machine Learning vs. Classical Programming



Deep Learning



Pytorch Fundamention

> https://pytorch.org/docs/stable/index.html

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sn
```

Tensors

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

- At its core, PyTorch is a library for processing tensors. A tensor is a number, vector, matrix or any n-dimensional array.

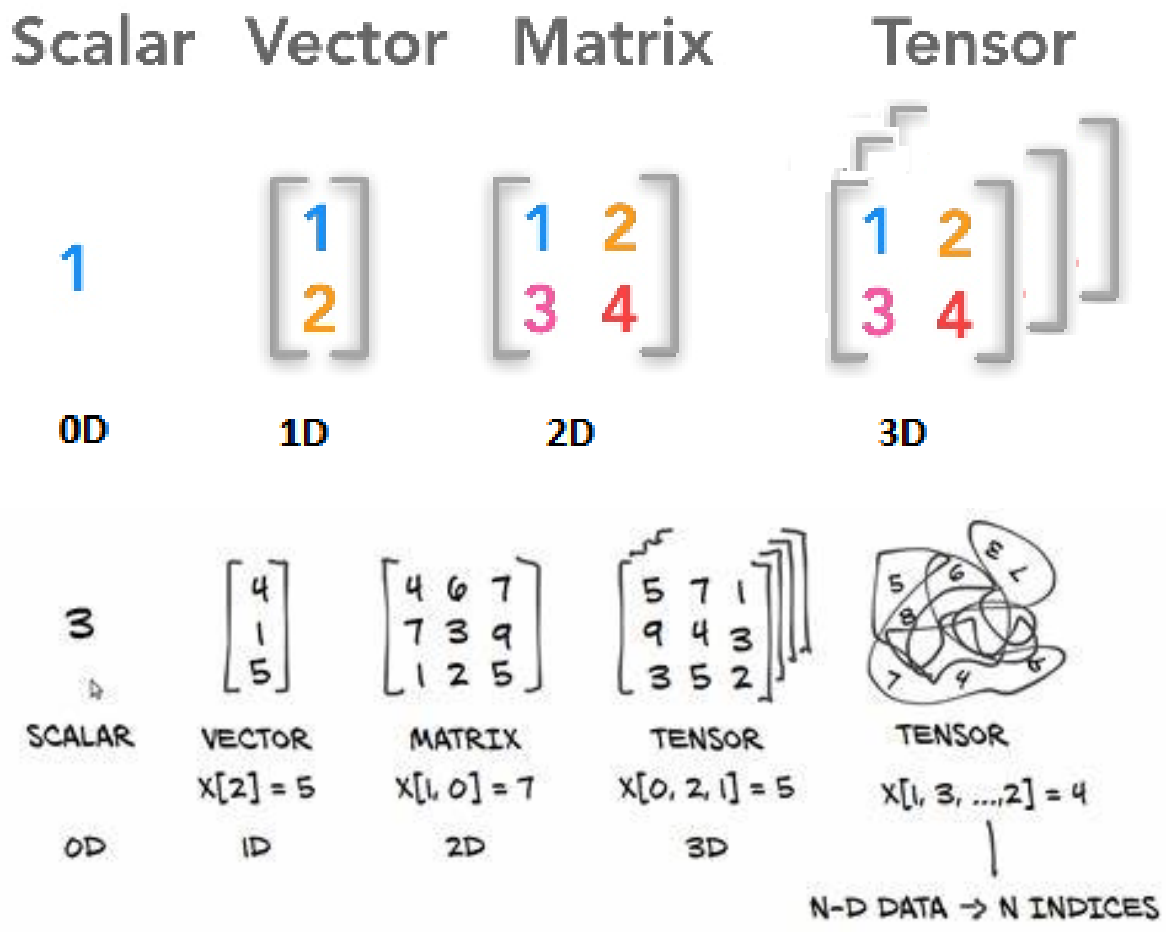
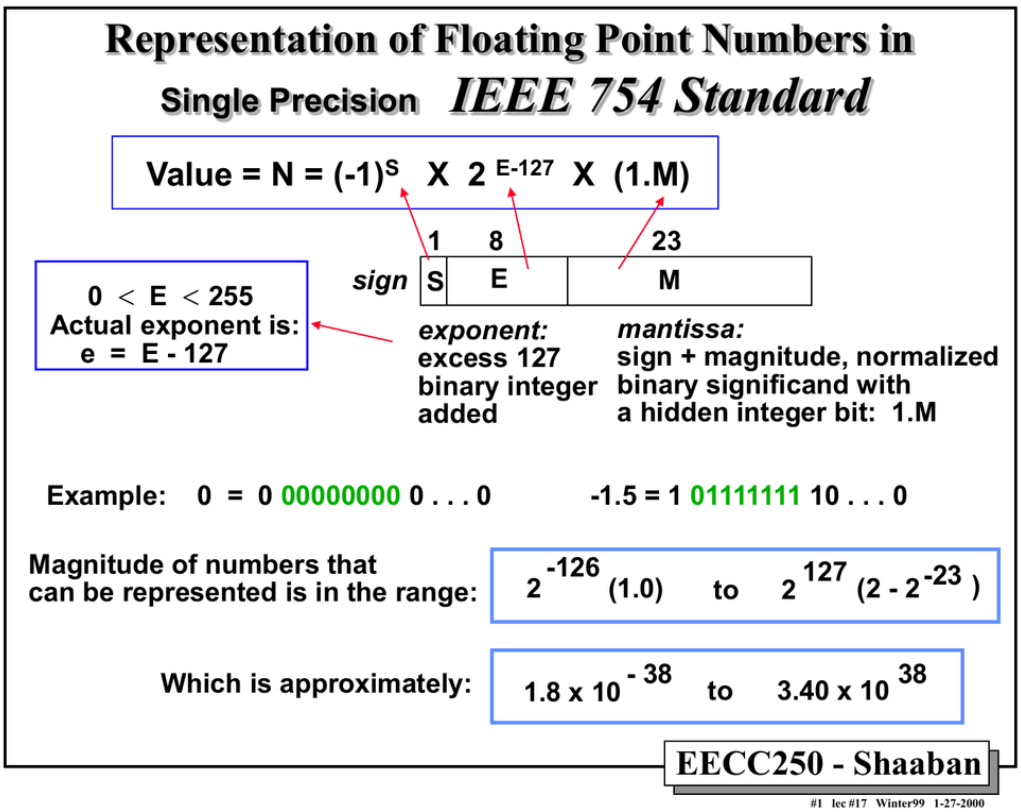
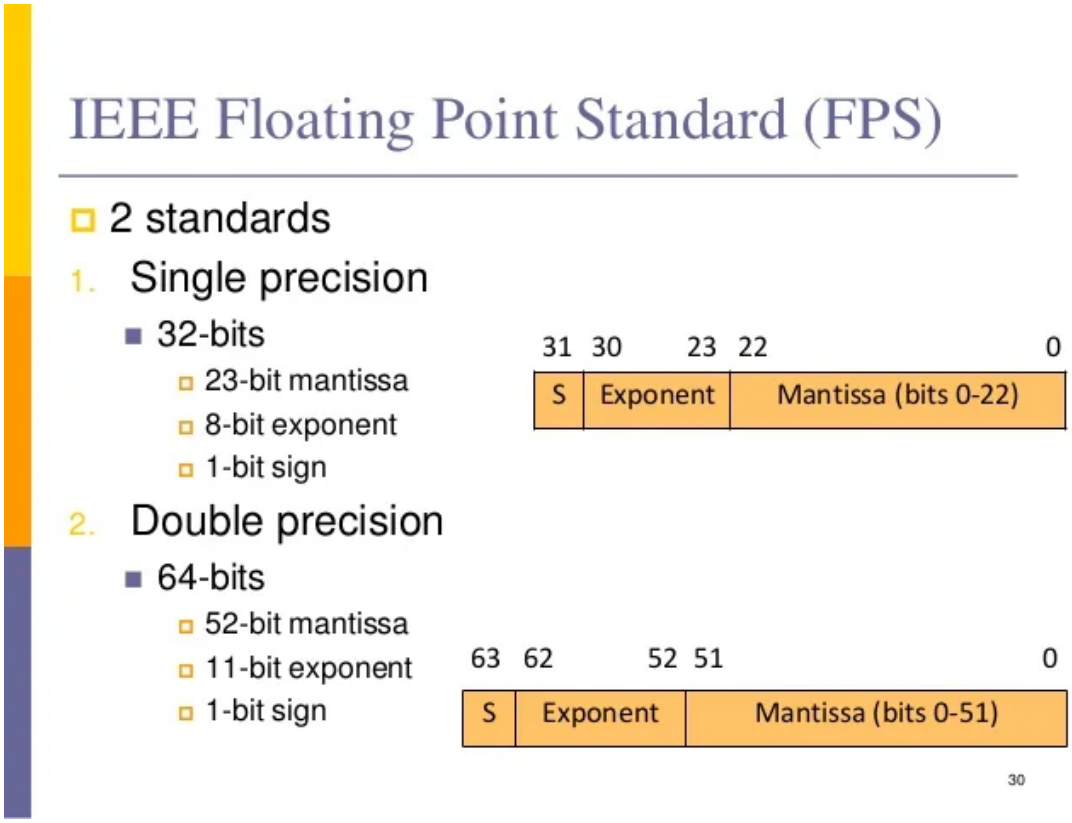


Figure 3.2 Tensors are the building blocks for representing data in PyTorch.

Tensor Data Type

- Single-precision floating-point format
  https://en.wikipedia.org/wiki/Single-precision_floating-point_format





PyTorch Basics: Tensors

https://jovian.ai/aakashns/01-pytorch-basics
https://jovian.com/learn/deep-learning-with-pytorch-zero-to-gans/lesson/lesson-1-pytorch-basics-and-linear-regression

```
import torch
torch.__version__
```

```
'2.0.0+cu118'
```

```
# scalar / number
t1 = torch.tensor(4.)
t1
```

```
tensor(4.)
```

```
t1.type()
```

```
'torch.FloatTensor'
```

```
t1.dtype
```

```
torch.float32
```

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

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

```
t2.dtype
```

```
torch.float32
```

```
# matrix
t3 = torch.tensor([[5., 6],
                   [7, 8],
                   [9, 10]])
t3
```

```
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.]])
```

```
t3.dtype
```

```
torch.float32
```

```
t4 = torch.tensor([[[11., 12, 13],
                    [13, 14, 15]],
                   [[15, 16, 17],
                    [17, 18, 19]]])
t4
```

```
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
```

```
t4.dtype
```

```
torch.float32
```

```
print(t1)
t1.shape, t1.ndim
```

```
tensor(4.)
(torch.Size([]), 0)
```

```
print(t2)
t2.shape, t2.ndim
```

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

```
print(t3)
t3.shape, t3.ndim
```

```
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.]])
(torch.Size([3, 2]), 2)
```

```
print(t4)
t4.shape, t4.ndim
```

```
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
(torch.Size([2, 2, 3]), 3)
```

```
t5 = torch.tensor([[[5., 6, 11],
                    [7, 8, 9]]])
```

```
t5 + t4
```

```
tensor([[[16., 18., 24.],
         [20., 22., 24.]],

        [[20., 22., 28.],
         [24., 26., 28.]]])
```

```
t5 * t4
```

```
tensor([[[ 55.,  72., 143.],
         [ 91., 112., 135.]],

        [[ 75.,  96., 187.],
         [119., 144., 171.]]])
```

Tensor Functions

```
t6 = torch.full((3, 2), 42)
t6, t6.shape
```

```
(tensor([[42, 42],
         [42, 42],
         [42, 42]]),
 torch.Size([3, 2]))
```

```
t7 = torch.cat((t3, t6))
t7, t7.shape
```

```
(tensor([[ 5.,  6.],
         [ 7.,  8.],
         [ 9., 10.],
         [42., 42.],
         [42., 42.],
         [42., 42.]]),
 torch.Size([6, 2]))
```

```
t8 = torch.sin(t7)
t8, t8.shape
```

```
(tensor([[-0.9589, -0.2794],
         [ 0.6570,  0.9894],
         [ 0.4121, -0.5440],
         [-0.9165, -0.9165],
         [-0.9165, -0.9165],
         [-0.9165, -0.9165]]),
 torch.Size([6, 2]))
```

```
t9 = t8.reshape((3, 2, 2))
print(t9, t9.shape)
print(t8, t8.shape)
```

```
tensor([[[-0.9589, -0.2794],
         [ 0.6570,  0.9894]],

        [[ 0.4121, -0.5440],
         [-0.9165, -0.9165]],

        [[-0.9165, -0.9165],
         [-0.9165, -0.9165]]]) torch.Size([3, 2, 2])
tensor([[-0.9589, -0.2794],
```

```
[ 0.6570,  0.9894],
[ 0.4121, -0.5440],
[-0.9165, -0.9165],
[-0.9165, -0.9165],
[-0.9165, -0.9165]]) torch.Size([6, 2])
```

Interoperability with Numpy

```
import numpy as np
```

```
x = np.array([[1., 2],
              [3, 4]])
x.dtype, x
```

```
(dtype('float64'),
 array([[1., 2.],
        [3., 4.]]))
```

```
# Convert the numpy array to torch tensor
y = torch.from_numpy(x)
y.dtype, y
```

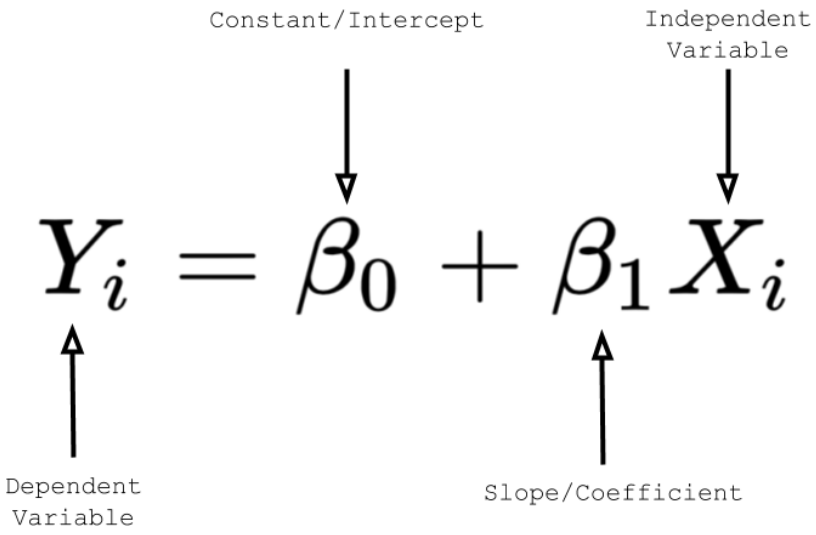
```
(torch.float64,
 tensor([[1., 2.],
         [3., 4.]], dtype=torch.float64))
```

```
# Convert torch tensor to numpy array
z = y.numpy()
z.dtype, z
```

```
(dtype('float64'),
 array([[1., 2.],
        [3., 4.]]))
```

PyTorch Basics: Tensor operations

Linear Regression




```
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
x, w, b
```

```
(tensor(3.), tensor(4., requires_grad=True), tensor(5., requires_grad=True))
```

```
# Arithmetic Operations
y = w * x + b
y
```

```
tensor(17., grad_fn=<AddBackward0>)
```

```
y.type()
```

```
'torch.FloatTensor'
```

```
y.dtype
```

```
torch.float32
```

PyTorch Basics: Gradients

- The derivates of y w.r.t the input tensors are stored in the .grad property of the respective tensors.

```
# Compute Derivatives
y.backward()
```

```
# Display gradients
print("dy/dx: ", x.grad)
print("dy/dw: ", w.grad)
print("dy/db: ", b.grad)
```

```
dy/dx:  None
dy/dw:  tensor(3.)
dy/db:  tensor(1.)
```

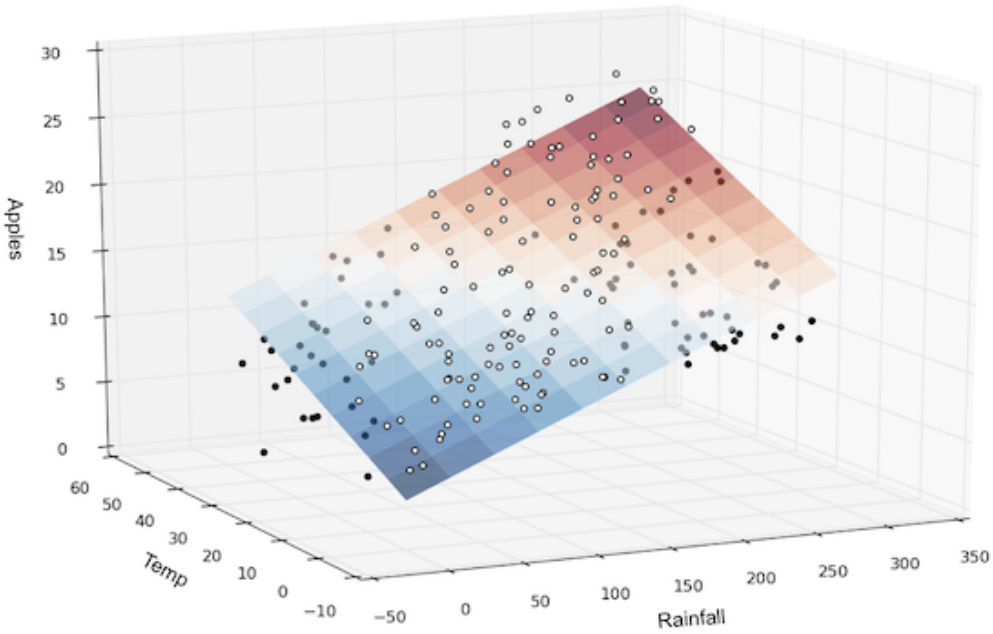
- As expected, dy/dw has the same value as x i.e. 3, and dy/db has the value 1. Note that x.grad is None, because x doesn't have requires_grad set to True.

Linear Regression - Multiple Variable

A linear regression model, each target variable is estimated to be a weighted sum of the input variables, offset by some constant, known as a bias :

yield_apple = w11 * temp + w12 * rainfall + w13 * humidity + b1
yield_orange = w21 * temp + w22 * rainfall + w23 * humidity + b2

$$\underset{X}{\begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix}} \times \underset{W^T}{\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix}} + \underset{b}{\begin{bmatrix} b_1 & b_2 \\ b_1 & b_2 \\ \vdots & \vdots \\ b_1 & b_2 \end{bmatrix}}$$



- floating in numpy is 64, in tenor is 32 usually

```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70]], dtype='float32')


# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119]], dtype='float32')


# Convert input and targets to tensors
inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)
print(inputs.dtype, inputs)
print(targets.dtype, targets)
```

```
     torch.float32 tensor([[ 73.,  67.,  43.],
             [ 91.,  88.,  64.],
             [ 87., 134.,  58.],
             [102.,  43.,  37.],
             [ 69.,  96.,  70.]])
     torch.float32 tensor([[ 56.,  70.],
             [ 81., 101.],
             [119., 133.],
```

```
             [ 22.,  37.],
             [103., 119.]])
```

```
w = torch.randn(2, 3, requires_grad=True)
b = torch.randn(2, requires_grad=True)
print(w.shape, w)
print(b.shape, b)
```

```
    torch.Size([2, 3]) tensor([[ 1.2713,  0.0073, -0.4182],
            [ 0.9873, -0.4139,  1.3412]], requires_grad=True)
    torch.Size([2]) tensor([0.1724, 0.6888], requires_grad=True)
```

```
# `@` represents matrix multiplication in PyTorch
# `.t` method returns the transpose of a tensor.
inputs @ w.t() + b
```

```
    tensor([[ 75.4836, 102.7057],
            [ 89.7381, 139.9518],
            [ 87.4983, 108.9181],
            [114.6849, 133.2224],
            [ 59.3189, 122.9679]], grad_fn=<AddBackward0>)
```

```
def model(x):
    return x @ w.t() + b
```

```
# make predictions
preds = model(inputs)

print(inputs.shape)
print(w.shape)
print(w.t().shape)
print(b.shape)
print(preds.shape)
print(preds)
```

```
    torch.Size([5, 3])
    torch.Size([2, 3])
    torch.Size([3, 2])
    torch.Size([2])
    torch.Size([5, 2])
    tensor([[ 75.4836, 102.7057],
            [ 89.7381, 139.9518],
            [ 87.4983, 108.9181],
            [114.6849, 133.2224],
            [ 59.3189, 122.9679]], grad_fn=<AddBackward0>)
```

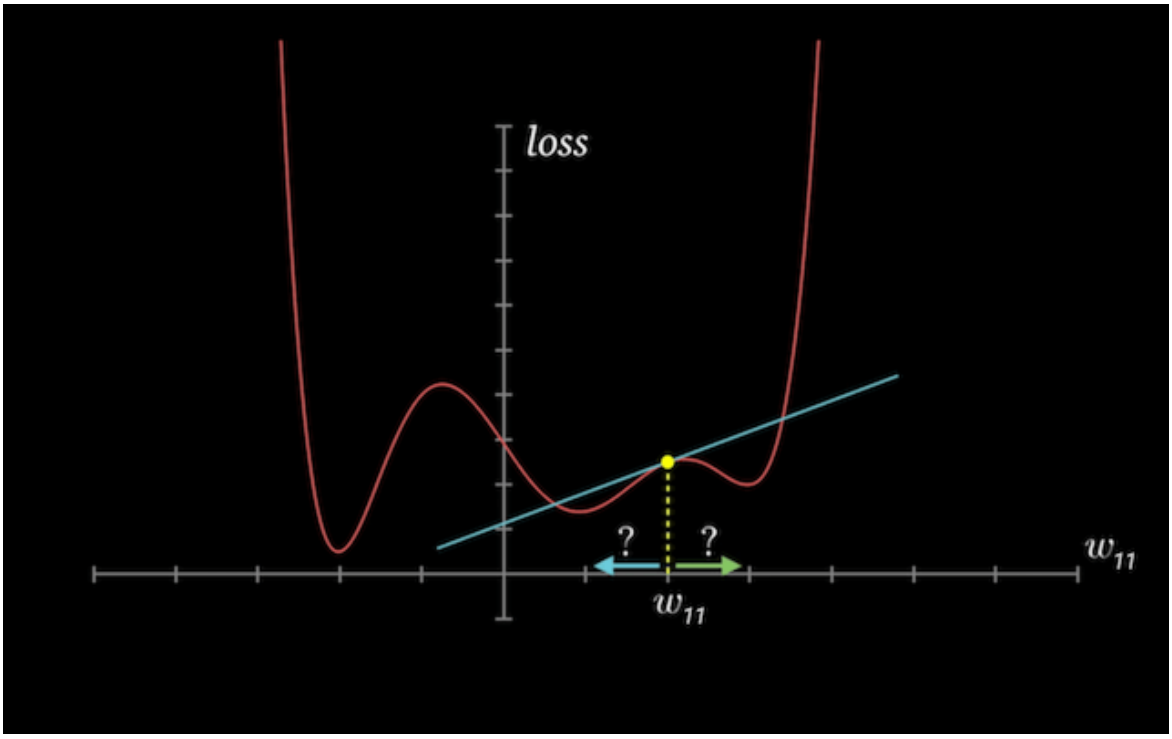
```
print(targets)
```

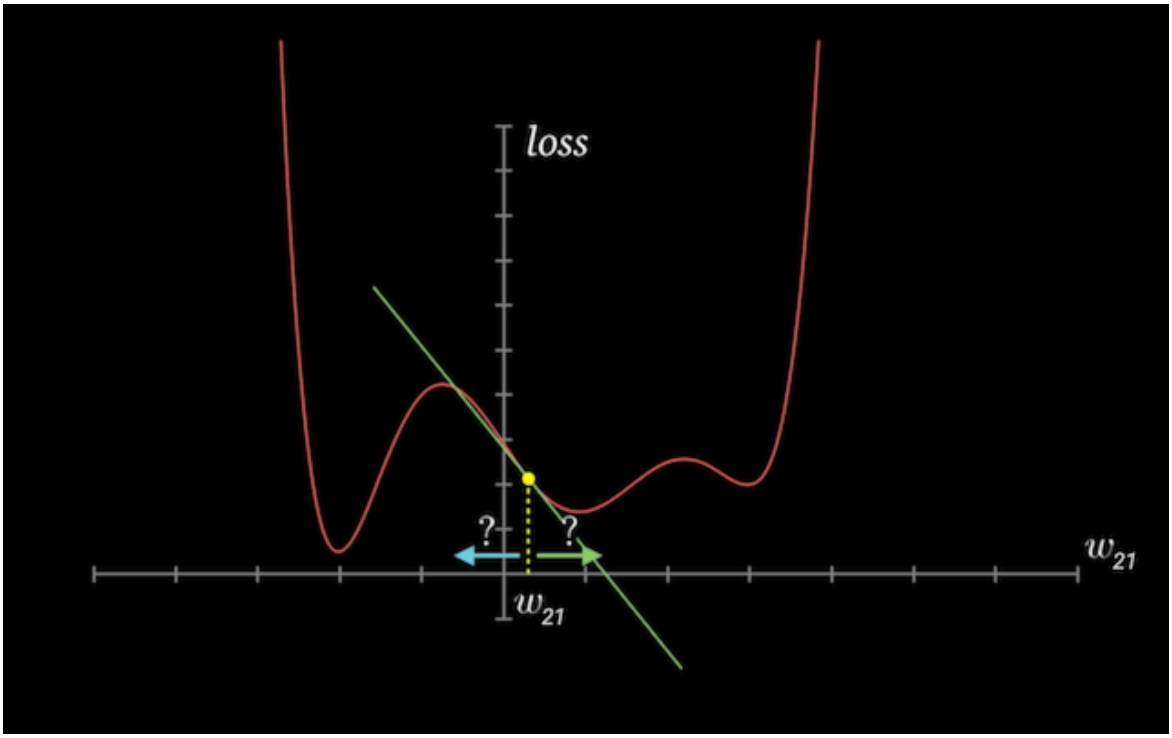
```
    tensor([[ 56.,  70.],
            [ 81., 101.],
            [119., 133.],
            [ 22.,  37.],
            [103., 119.]])
```

```
# MSE Loss
def mse(y1, y2):
    diff = y1 - y2
    return torch.sum(diff * diff) / diff.numel()


# Compute Loss
loss = mse(preds, targets)
print(f"Loss: {loss}")
```

```
Loss: 2438.819091796875
```

Adjust weights and biases to reduce the loss






```
loss.backward()
```

```
w, b
```

```
     (tensor([[ 1.2713,  0.0073, -0.4182],
              [ 0.9873, -0.4139,  1.3412]], requires_grad=True),
      tensor([0.1724, 0.6888], requires_grad=True))
```

```
print(w.grad)
print(b.grad)
```

```
     tensor([[1183.3378, -470.9612,  -11.6802],
             [2785.0959, 1382.1110, 1268.0994]])
     tensor([ 9.1448, 29.5532])
```

```
with torch.no_grad():
    w -= w.grad * 1e-5
    b -= b.grad * 1e-5
    w.grad.zero_()
    b.grad.zero_()
```

```
w, b
```

```
     (tensor([[ 1.2595,  0.0120, -0.4181],
              [ 0.9594, -0.4277,  1.3285]], requires_grad=True),
      tensor([0.1723, 0.6885], requires_grad=True))
```

```
# Compute Loss
preds = model(inputs)
```

```
loss = mse(preds, targets)
print(f"Loss: {loss}")
```

```
Loss: 2319.093017578125
```

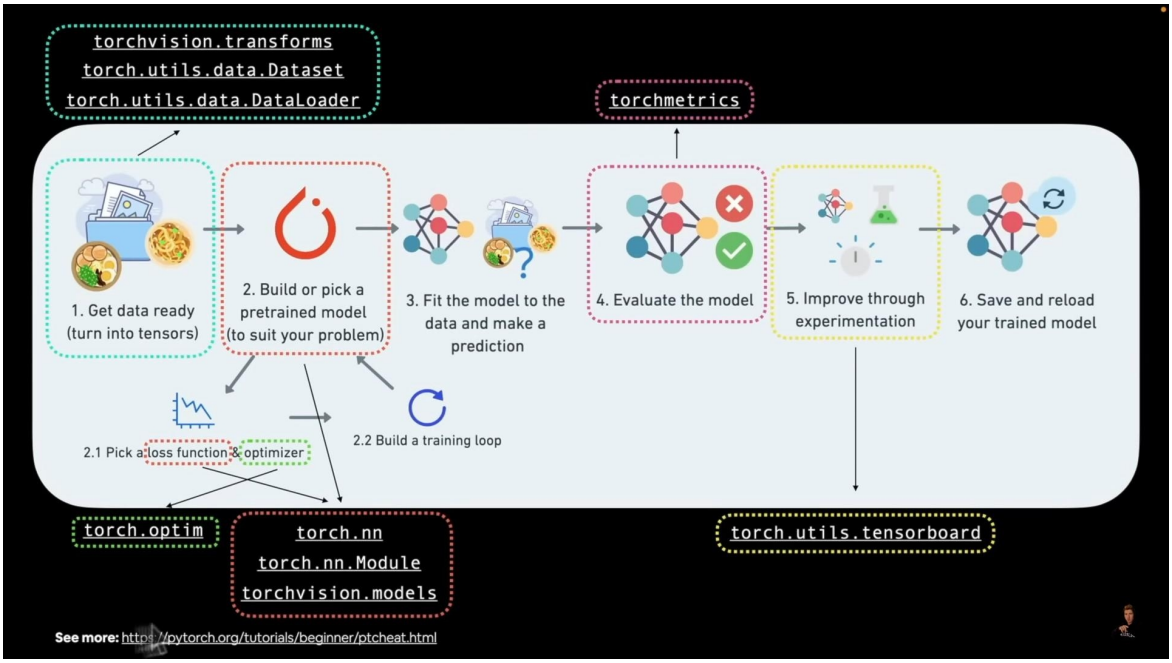
```
print(w.grad)
print(b.grad)
```

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

```
# Train for multiple epochs
epochs = 100
lr = 1e-5
for epoch in range(epochs):
    preds = model(inputs)
    loss = mse(preds, targets)
    loss.backward()
    with torch.no_grad():
        w -= w.grad * lr
        b -= b.grad * lr
        w.grad.zero_()
        b.grad.zero_()

print(f"Epoch: {epochs}\nLearning Rate: {lr}\nLoss: {loss}")
print(f"Predictions:\n{preds}")
print(f"targets:\n{targets}")
```

```
Epoch: 100
Learning Rate: 1e-05
Loss: 641.2437133789062
Predictions:
tensor([[ 65.8868,  77.2426],
        [ 81.6740, 109.5779],
        [105.8842, 101.5549],
        [ 70.9624,  77.0757],
        [ 71.9178, 111.3913]], grad_fn=<AddBackward0>)
targets:
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

Deel Learning Work Flow



Linear regression using PyTorch built-ins

- torch.nn
  https://pytorch.org/docs/stable/nn.html

Dataset

torch.utils.data
https://pytorch.org/docs/stable/data.html

- TensorDataset, which allows access to rows from inputs and targets as tuples, and provides standard APIs for working with many different types of datasets in PyTorch.

```
import torch.nn as nn
```

```
from torch.utils.data import TensorDataset
```

```
train_ds = TensorDataset(inputs, targets)
train_ds[0]
```

```
    (tensor([73., 67., 43.]), tensor([56., 70.]))
```

```
train_ds[0][0].dtype
```

```
    torch.float32
```

```
train_ds[0][1].dtype
```

```
torch.float32
```

DataLoader

- DataLoader, which can split the data into batches of a predefined size while training. It also provides other utilities like shuffling and random sampling of the data.

```
from torch.utils.data import DataLoader


batch_size =5

train_dl = DataLoader(train_ds,
                      batch_size,
                      shuffle=True)
```

torch.nn.linear

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

- Instead of initializing the weights & biases manually, we can define the model using the nn.Linear class from PyTorch, which does it automatically.

```
model = nn.Linear(in_features=3, out_features=2)
print(model.state_dict())
```

```
OrderedDict([('weight', tensor([[-0.3406,  0.1288, -0.1381],
        [ 0.4554, -0.4978, -0.0026]])), ('bias', tensor([-0.3868,  0.0035]))])
```

```
model.parameters()
```

```
<generator object Module.parameters at 0x7faf4d178ac0>
```

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

```
[Parameter containing:
 tensor([[-0.3406,  0.1288, -0.1381],
         [ 0.4554, -0.4978, -0.0026]], requires_grad=True),
 Parameter containing:
 tensor([-0.3868,  0.0035], requires_grad=True)]
```

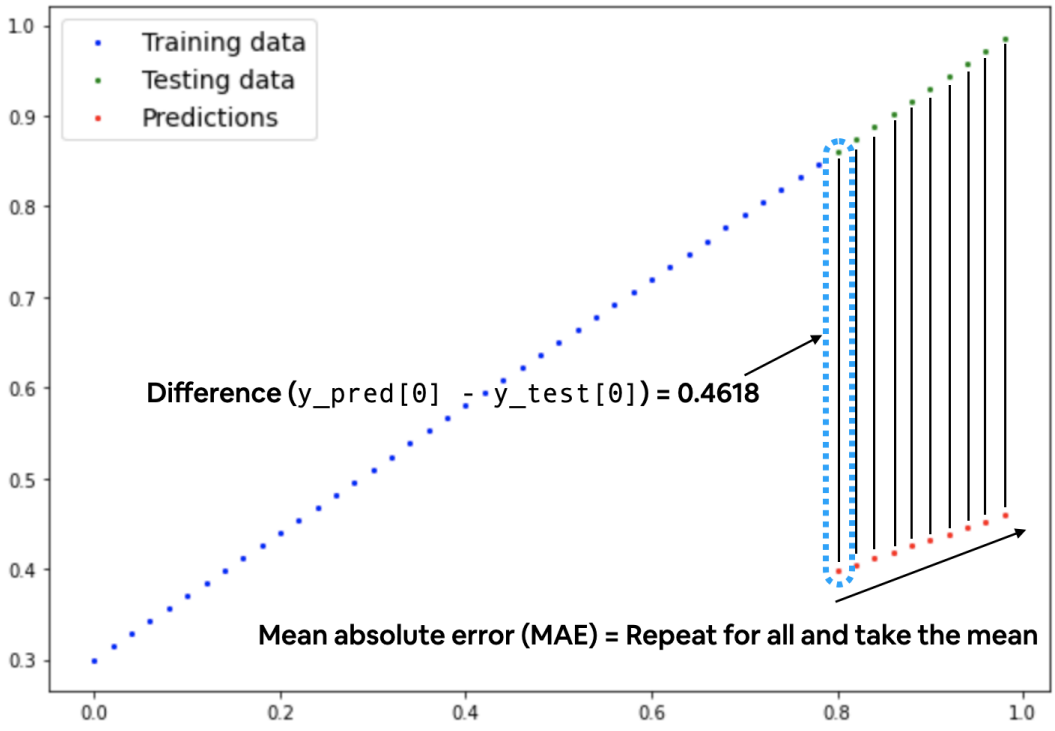
```
preds = model(inputs)
preds
```

```
tensor([[-22.5633,  -0.2155],
        [-28.8907,  -2.5262],
        [-20.7758, -27.2308],
        [-34.7032,  24.9537],
        [-21.1955, -16.5429]], grad_fn=<AddmmBackward0>)
```

Loss Function

https://pytorch.org/docs/stable/nn.html#loss-functions

- *Mean absolute error (MAE, in PyTorch: `torch.nn.L1Loss`) measures the absolute difference between two points (predictions and labels) and then takes the mean across all examples.*




```
from torch.nn import functional as F


# Define Loss Function
loss_fn = F.mse_loss


preds = model(inputs)
loss = loss_fn(preds, targets)
print(loss)
```

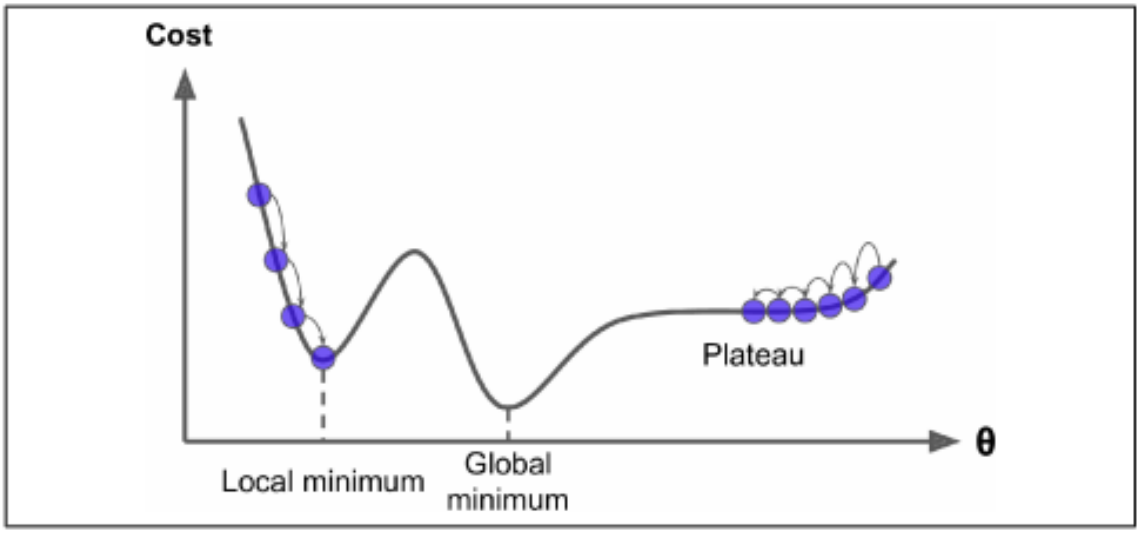
```
tensor(11626.3975, grad_fn=<MseLossBackward0>)
```

Optimizer

- torch.optim
  https://pytorch.org/docs/stable/optim.html?highlight=optimizer#torch.optim.Optimizer

- SGD
  https://pytorch.org/docs/stable/generated/torch.optim.SGD.html#torch.optim.SGD

- `params` is the target model parameters you'd like to optimize (e.g. the `weights` and `bias` values we randomly set before).
- `lr` is the **learning rate** you'd like the optimizer to update the parameters at, higher means the optimizer will try larger updates (these can sometimes be too large and the optimizer will fail to work), lower means the optimizer will try smaller updates (these can sometimes be too small and the optimizer will take too long to find the ideal values). The learning rate is considered a **hyperparamter** (because it's set by a machine learning engineer). Common starting values for the learning rate are `0.01`, `0.001`, `0.0001`, however, these can also be adjusted over time (this is called learning rate scheduling).

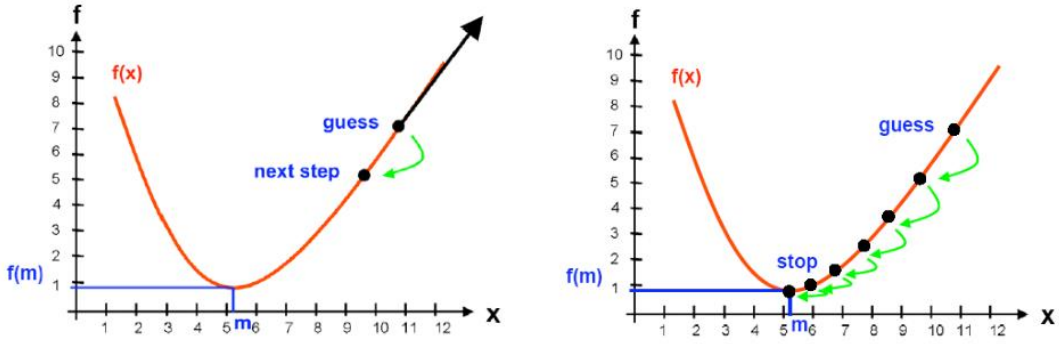


Gradient descent algorithm

- Given $x^{(k)}$.
- The vector $-\nabla f(x^{(k)})$ points in the direction of maximum rate of decrease.
- Makes sense to choose $d^{(k)} = -\nabla f(x^{(k)})$.
- Gradient algorithm:

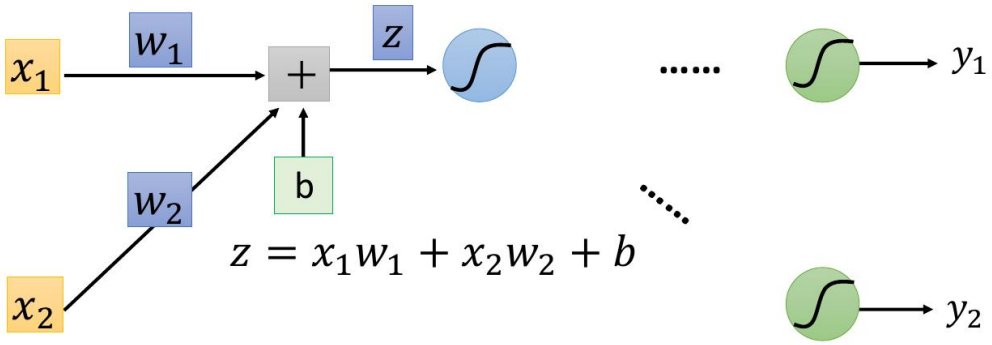
$$x^{(k+1)} = x^{(k)} - \alpha_k \nabla f(x^{(k)}).$$

- Step size α_k can be chosen in diverse ways.



Backpropagation – Forward pass

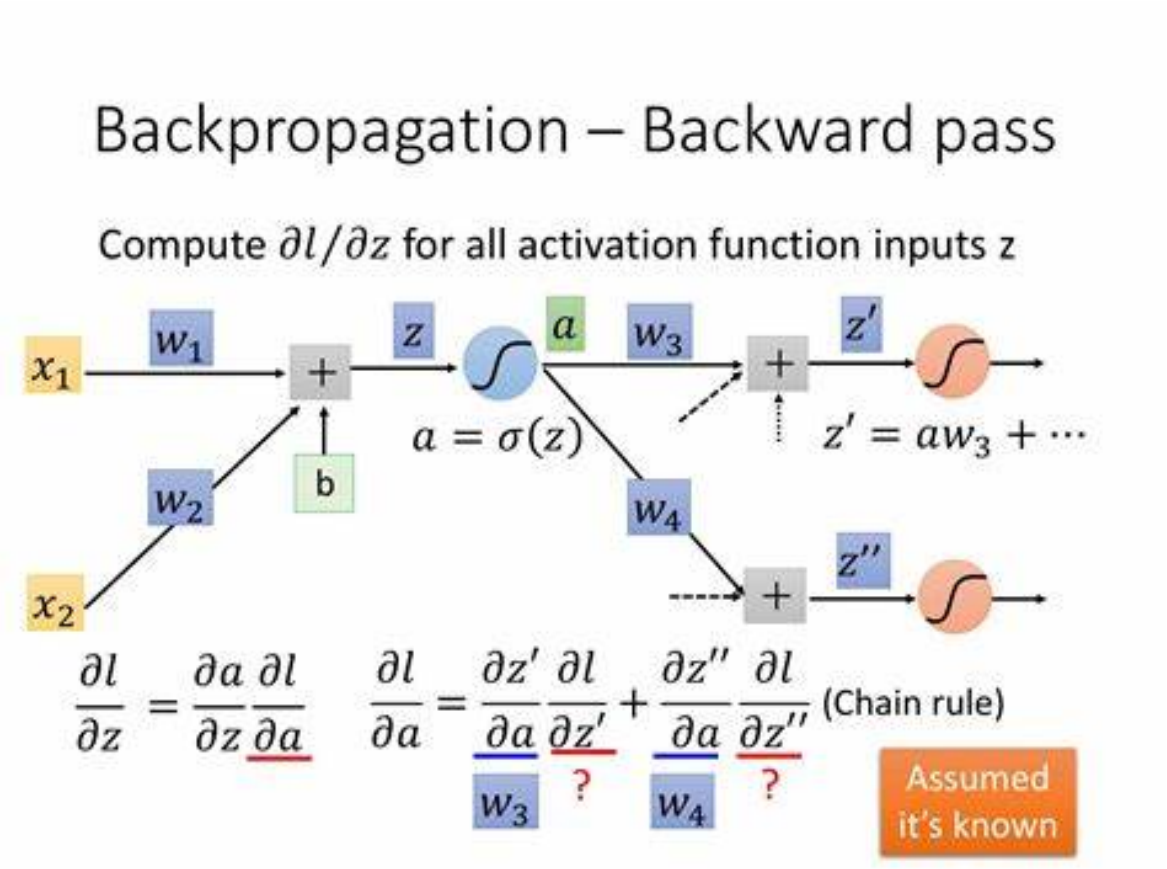
Compute $\partial z/\partial w$ for all parameters



$\partial z/\partial w_1 =?\ x_1$

$\partial z/\partial w_2 =?\ x_2$

The value of the input connected by the weight




```
# Define Optimizer
optimizer = torch.optim.SGD(model.parameters(),
                            lr=1e-5)
```

Train Model

- 1.Generate predictions
- 2.Calculate the loss
- 3.Compute gradients w.r.t the weights and biases
- 4.Adjust the weights by subtracting a small quantity proportional to the gradient
- 5.Reset the gradients to zero

```
# Utility function to train the model
# ! pip install tqdm

from tqdm.auto import tqdm

def fit(num_epochs:int,
        model: torch.nn.Module,
        loss_fn: torch.nn.Module,
        optimizer: torch.optim.Optimizer,
        train_dl: torch.utils.data.DataLoader,
        device: torch.device=None):
```

```
    for epoch in tqdm(range(num_epochs)):
        for Xb, yb in train_dl:

            # 1. Make Predictions
            y_preds = model(Xb)

            # 2. Calculate Loss
            loss = loss_fn(y_preds, yb)

            # 3. Compute Gardients
            loss.backward()

            # 4. Update Parameters
            optimizer.step()

            # 5. Reset the Gradients to Zero
            optimizer.zero_grad()

        # Print the progress
        if (epoch+1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')



fit(100, model, loss_fn, optimizer, train_dl)
```

```
100%                                                 100/100 [00:00<00:00, 809.23it/s]
Epoch [10/100], Loss: 1051.7415
Epoch [20/100], Loss: 662.2552
Epoch [30/100], Loss: 580.5877
Epoch [40/100], Loss: 513.6693
Epoch [50/100], Loss: 454.7893
Epoch [60/100], Loss: 402.8942
Epoch [70/100], Loss: 357.1491
Epoch [80/100], Loss: 316.8211
Epoch [90/100], Loss: 281.2643
Epoch [100/100], Loss: 249.9103
```

```
model(inputs)
```

```
tensor([[ 59.8817,  76.9015],
        [ 80.9560, 102.1582],
        [117.1694, 118.9759],
        [ 36.9318,  75.0273],
        [ 90.4192,  99.5453]], grad_fn=<AddmmBackward0>)
```

```
targets
```

```
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

```
model(torch.tensor([[75, 63, 44.]]))
```

```
tensor([[56.7405, 76.2888]], grad_fn=<AddmmBackward0>)
```

Model with non-linearity

Linear vs Nonlinear

Name _____

4-3 Patterns and Nonlinear Functions

A *nonlinear function* is a function whose graph is not a line or part of a line.

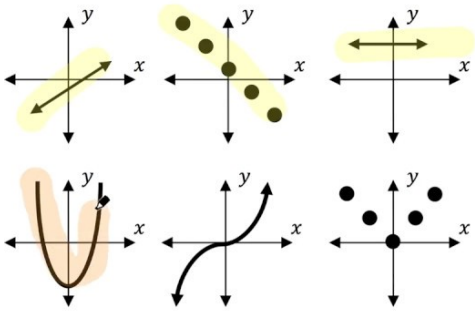
Linear and Nonlinear Functions

Linear Function

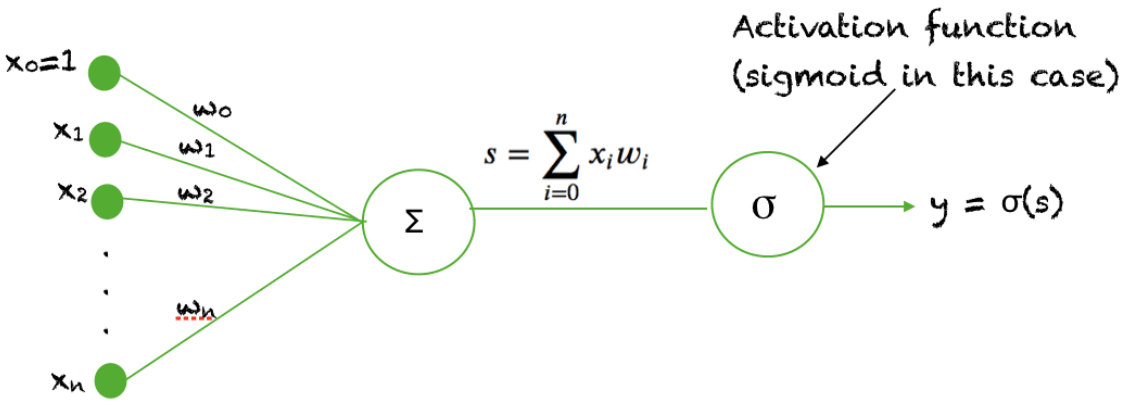
A linear function is a function whose graph is a nonvertical line or part of a nonvertical line.

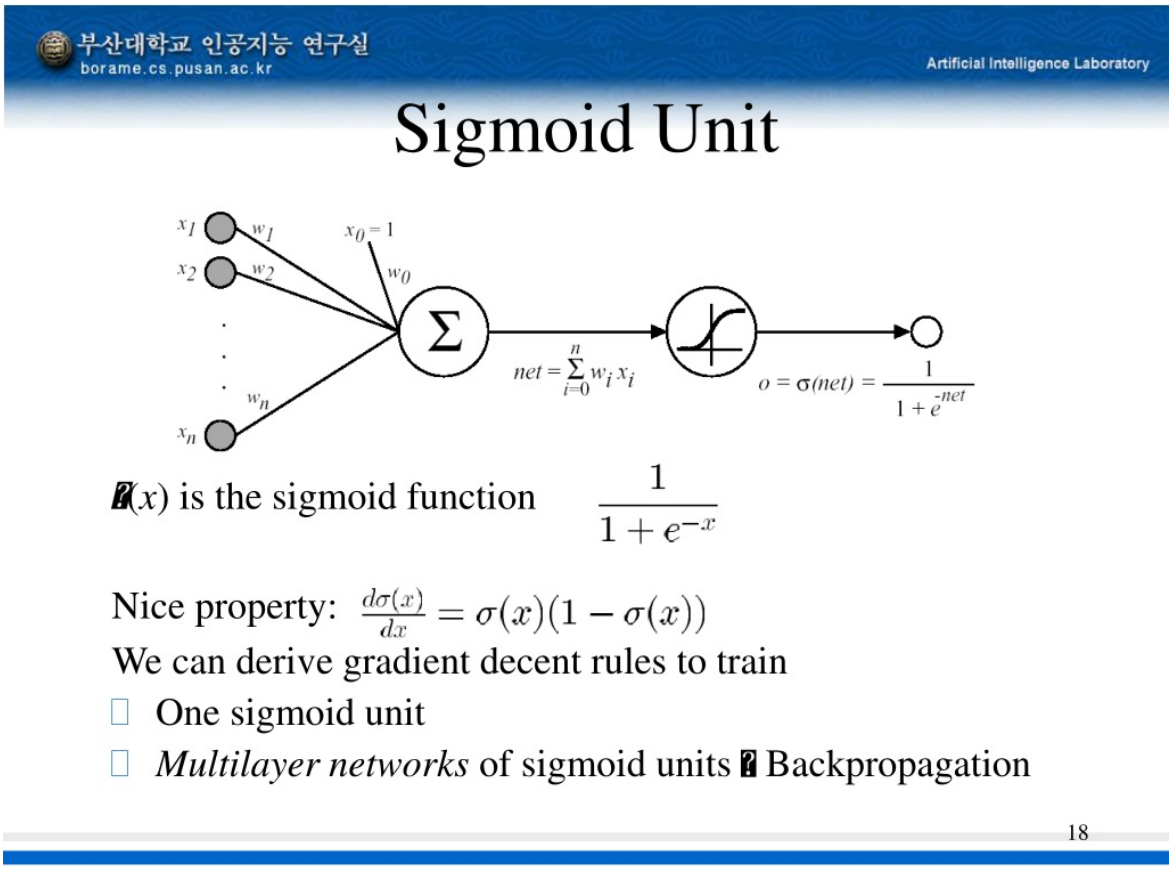
Nonlinear Function

A nonlinear function is a function whose graph is not a line or part of a line.

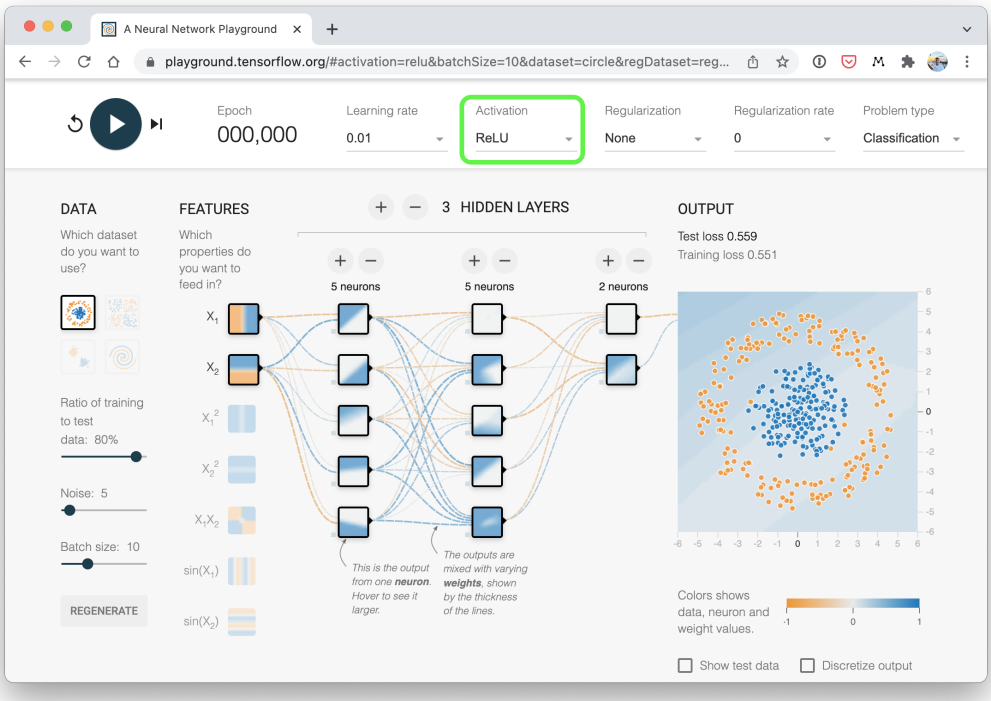


Neural Network with Sigmoid





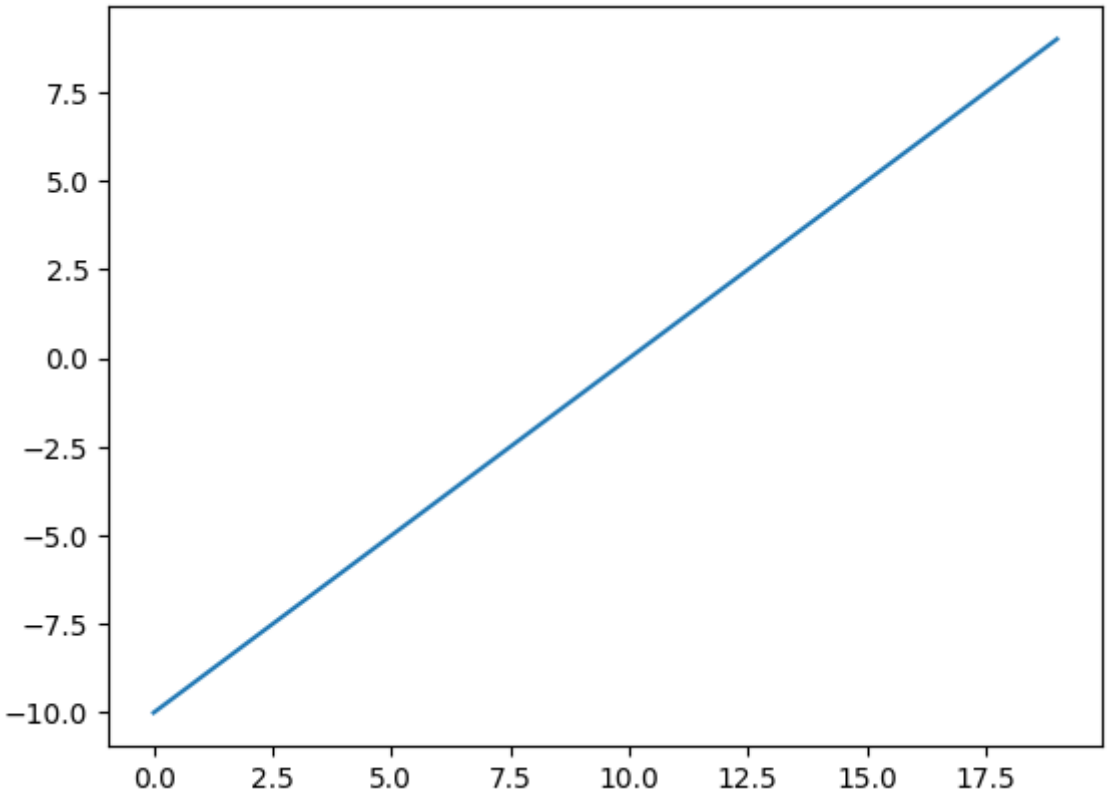
- TensorFlow Playground website https://playground.tensorflow.org/
- A classification neural network on TensorFlow playground with ReLU activation



Activation Function

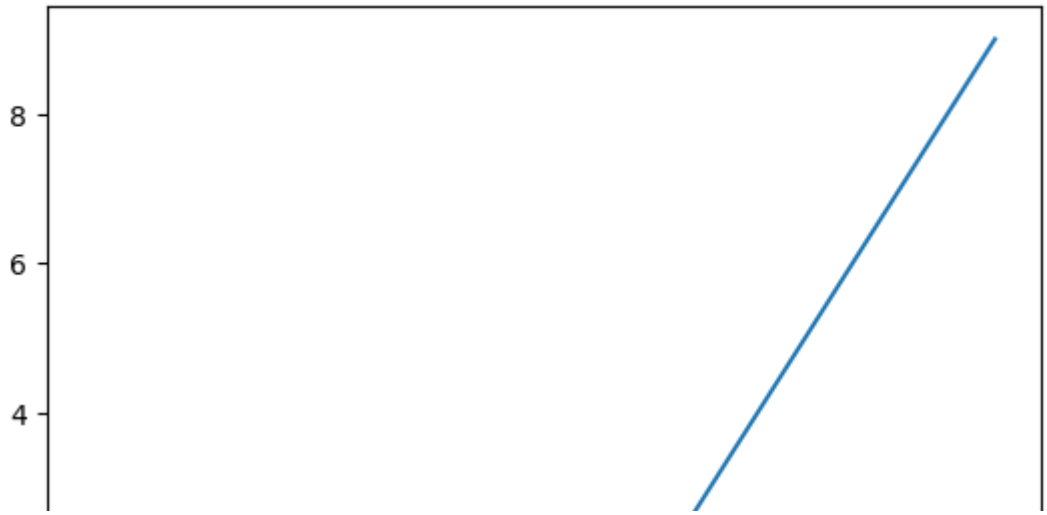
```
# Create a toy tensor (similar to the data going into our model(s))
A = torch.arange(-10, 10, 1, dtype=torch.float32)

# Pass toy tensor through ReLU function
print(A)

# Visualize the toy tensor
plt.plot(A);
```

```
tensor([-10.,  -9.,  -8.,  -7.,  -6.,  -5.,  -4.,  -3.,  -2.,  -1.,   0.,   1.,
          2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.])
```




```
# Create ReLU function by hand
def relu(x):
  return torch.maximum(torch.tensor(0), x) # inputs must be tensors

# Pass toy tensor through ReLU function
print(relu(A))

# Plot ReLU activated toy tensor
plt.plot(relu(A));
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 2., 3., 4., 5., 6., 7.,
        8., 9.])
```



The sigmoid function formula goes like so:

$$out_i = \frac{1}{1 + e^{-input_i}}$$

Or using x as input:

$$S(x) = \frac{1}{1 + e^{-x_i}}$$

Where S stands for sigmoid, e stands for exponential (`torch.exp()`) and i stands for a particular element in a tensor.

```
# Create a custom sigmoid function
def sigmoid(x):
  return 1 / (1 + torch.exp(-x))

# Test custom sigmoid on toy tensor
sigmoid(A)

# Plot sigmoid activated toy tensor
plt.plot(sigmoid(A));
```


```
model_2 = nn.Sequential(
    nn.Linear(in_features=3, out_features=3),
    nn.Sigmoid(),
    nn.Linear(in_features=3, out_features=2))
```

```
optimizer = torch.optim.SGD(model_2.parameters(), lr=0.001)
```

```
fit(100, model_2, F.mse_loss, optimizer, train_dl)
```

```
100%                                                  100/100 [00:00<00:00, 815.21it/s]
Epoch [10/100], Loss: 7941.4033
Epoch [20/100], Loss: 7563.4663
Epoch [30/100], Loss: 7205.2485
Epoch [40/100], Loss: 6865.7139
Epoch [50/100], Loss: 6543.2568
Epoch [60/100], Loss: 6206.6216
Epoch [70/100], Loss: 5914.1582
Epoch [80/100], Loss: 5638.7524
Epoch [90/100], Loss: 5379.4082
Epoch [100/100], Loss: 5135.1890
```

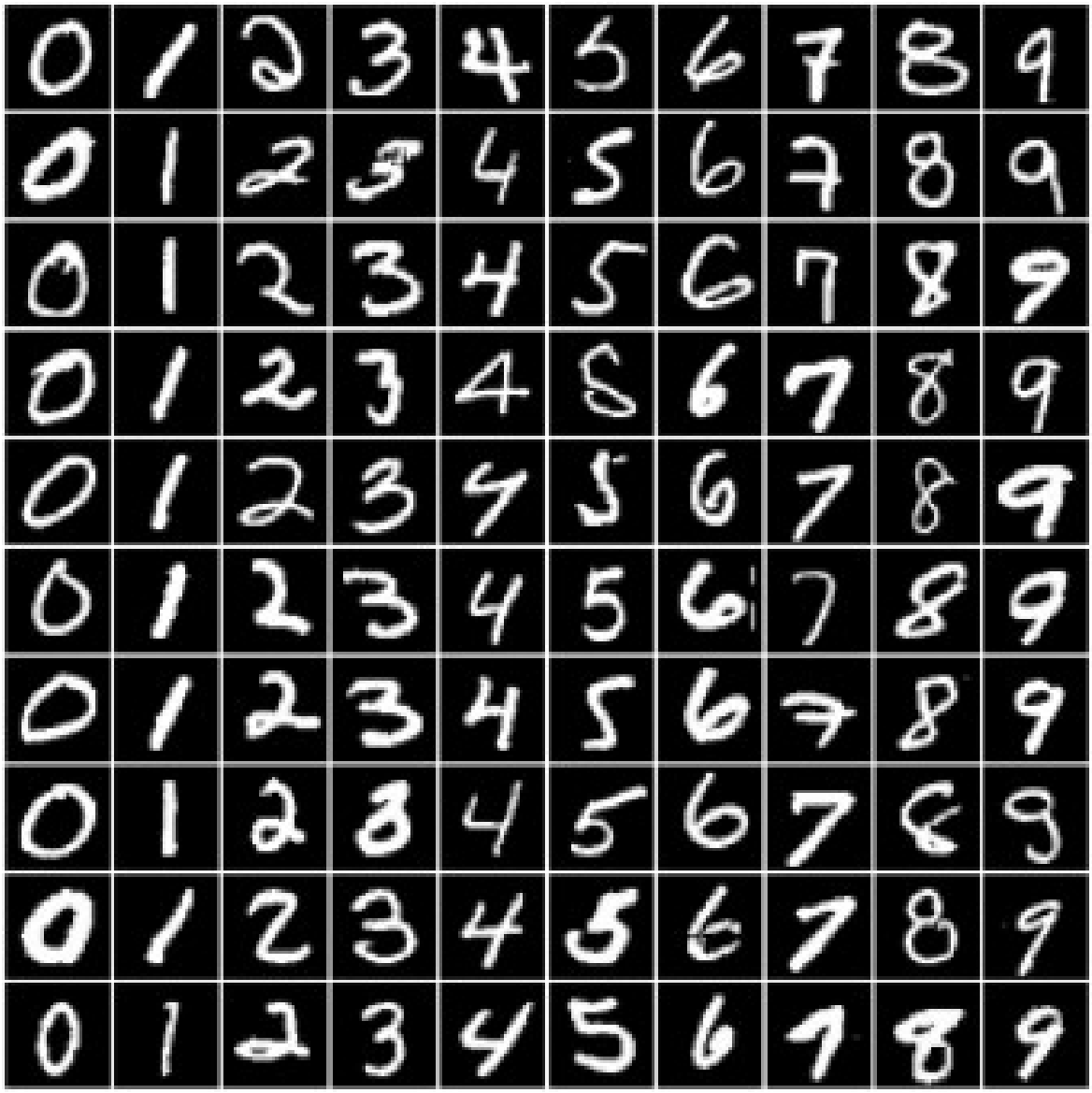
Project 1 - PyTorch Images and Logistic Regression

Computer vision libraries in PyTorch

PyTorch module	
torchvision	Contains datasets, model architectures and image transformations often used for computer vi
torchvision.datasets	Here you'll find many example computer vision datasets for a range of problems from image c
torchvision.models	This module contains well-performing and commonly used computer vision model architectur
torchvision.transforms	Often images need to be transformed (turned into numbers/processed/augmented) before bei
torch.utils.data.Dataset	Base dataset class for PyTorch.
torch.utils.data.DataLoader	Creates a Python iteralbe over a dataset (created with `torch.utils.data.Dataset`).

> - FashionMNIST in `torchvision.datasets.FashionMNIST()`.
> - `root: str` - which folder do you want to download the data to?
> - `train: Bool` - do you want the training or test split?
> - `download: Bool` - should the data be downloaded?

- `transform: torchvision.transforms` - what transformations would you like to do on the data?
- `target_transform` - you can transform the targets (labels) if you like too.



```
import torch
import torchvision
from torchvision.datasets import MNIST
```

Phase 1: Get Data and Explore Data

```
dataset = MNIST(root='data/', download=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to data/MNIS
100%|██████████| 9912422/9912422 [00:00<00:00, 264818507.64it/s]Extracting data/MNIS
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to data/MNIST
100%|██████████| 28881/28881 [00:00<00:00, 42090234.13it/s]
Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to data/MNIST/
100%|██████████| 1648877/1648877 [00:00<00:00, 141545055.19it/s]
Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to data/MNIST/
100%|██████████| 4542/4542 [00:00<00:00, 21998301.12it/s]Extracting data/MNIST/raw/t1
```

```
train_dataset = MNIST(root='data', train=True)
len(train_dataset)
```

```
60000
```

```
test_dataset = MNIST(root='data', train=False)
len(test_dataset)
```

```
10000
```

```
train_dataset.__dict__
```

```
{'root': 'data',
 'transform': None,
 'target_transform': None,
 'transforms': None,
 'train': True,
 'data': tensor([[[0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          ...,
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0]],

         [[0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          ...,
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0]],

         [[0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          ...,
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0],
          [0, 0, 0,  ..., 0, 0, 0]],
```

```
                ...,

                [[0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0],
                 ...,
                 [0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0]],

                [[0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0],
                 ...,
                 [0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0]],

                [[0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0],
                 ...,
                 [0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0],
                 [0, 0, 0,  ..., 0, 0, 0]]], dtype=torch.uint8),
       'targets': tensor([5, 0, 4,  ..., 5, 6, 8])}
```

```
train_dataset.data.shape
```

```
torch.Size([60000, 28, 28])
```

```
class_names= train_dataset.classes
class_names
```

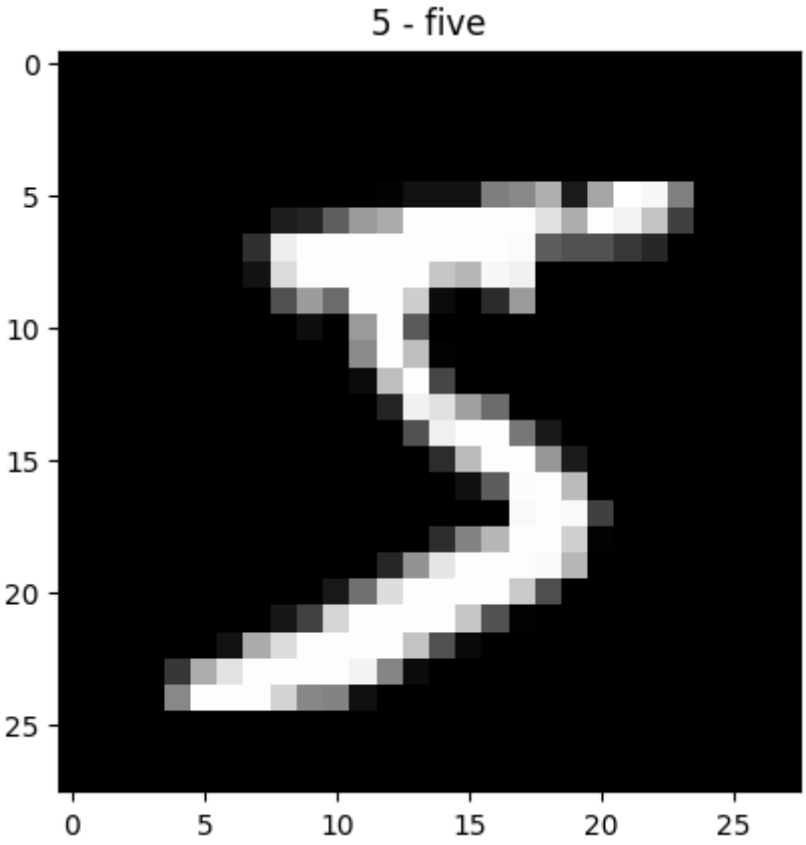
```
['0 - zero',
 '1 - one',
 '2 - two',
 '3 - three',
 '4 - four',
 '5 - five',
 '6 - six',
 '7 - seven',
 '8 - eight',
 '9 - nine']
```

```
train_dataset[0]
```

```
(<PIL.Image.Image image mode=L size=28x28 at 0x7FAF43441A20>, 5)
```

```
image, label = train_dataset[0]
print(image)
print(label)
plt.imshow(image, cmap="gray")
plt.title(class_names[label])
```

```
<PIL.Image.Image image mode=L size=28x28 at 0x7FAF4172BEB0>
5
Text(0.5, 1.0, '5 - five')
```



Convert the images into tensors

```
from torchvision.transforms import transforms


train_dataset = MNIST(root='data/',
                      train=True,
                      transform=transforms.ToTensor())

test_dataset = MNIST(root='data/',
                     train=False,
                     transform=transforms.ToTensor())


len(train_dataset), len(test_dataset)
```

```
(60000, 10000)
```

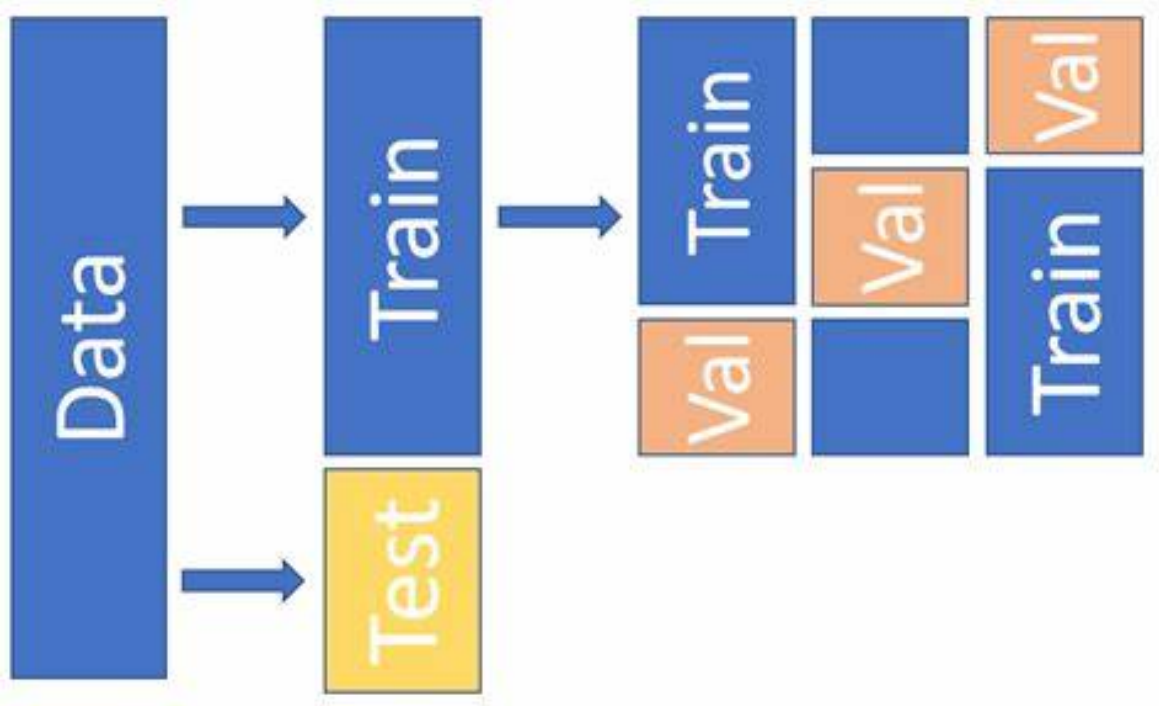
```
img_tensor, label = train_dataset[0]
img_tensor.shape, label
```

```
(torch.Size([1, 28, 28]), 5)
```

Training and Validation Datasets

While building real-world machine learning models, it is quite common to split the dataset into three parts:

`Training set` - used to train the model, i.e., compute the loss and adjust the model's weights using gradient descent.
`Validation set` - used to evaluate the model during training, adjust hyperparameters (learning rate, etc.), and pick the best version of the model.
`Test set` - used to compare different models or approaches and report the model's final accuracy.



torch.utils.data
https://pytorch.org/docs/stable/data.html

```
from torch.utils.data import random_split

train_ds, val_ds = random_split(train_dataset, [50000, 10000])
len(train_ds), len(val_ds)
```

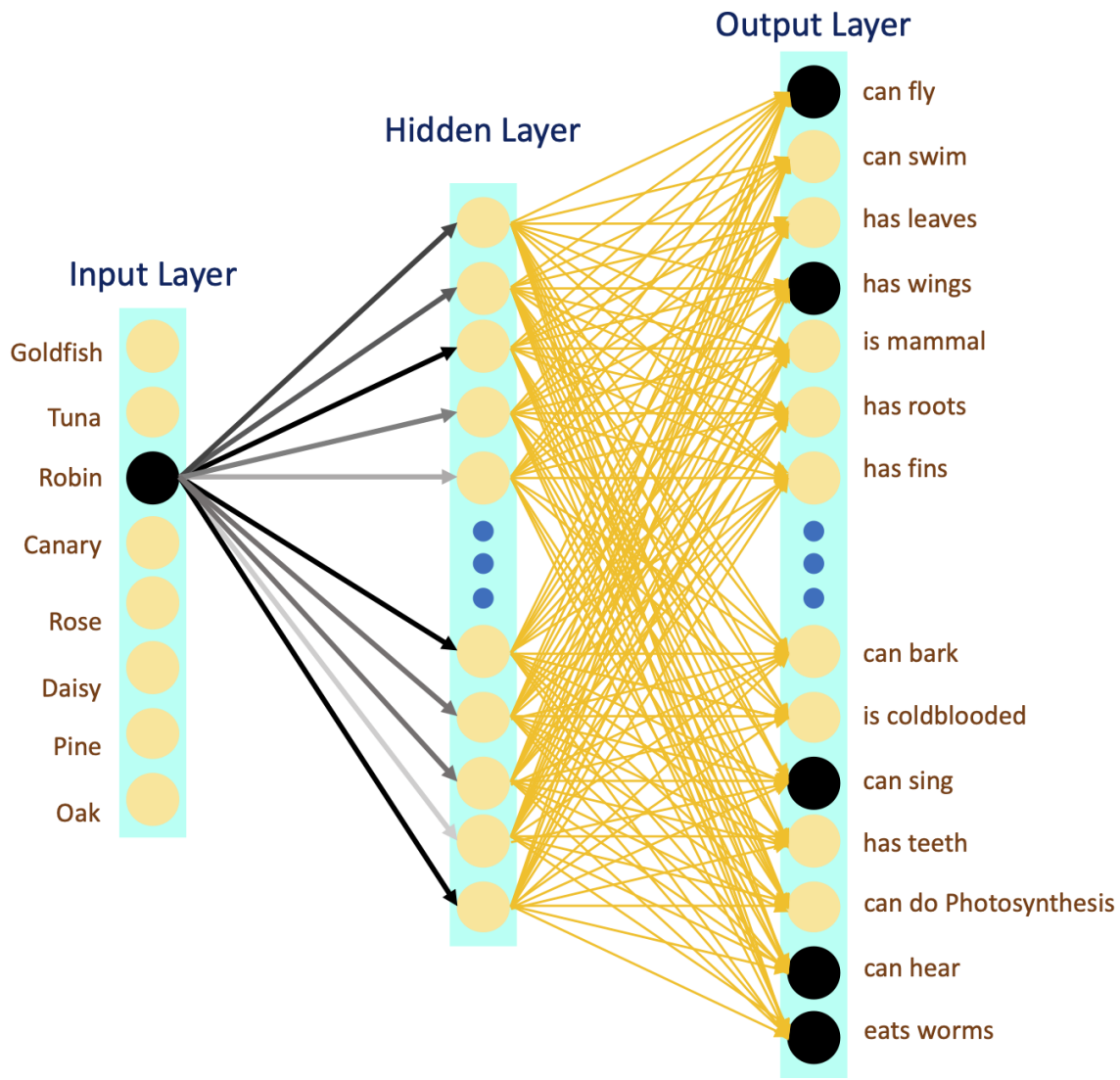
```
(50000, 10000)
```

```
from torch.utils.data import DataLoader

batch = 128

train_loader = DataLoader(train_ds, batch, shuffle=True)
val_loader = DataLoader(val_ds, batch)
```

```
len(train_loader), len(val_loader)
```

```
(391, 79)
```

Phase 2: Bulid Neural Network Model




```
img_tensor.shape, len(class_names)
```

```
(torch.Size([1, 28, 28]), 10)
```

```
import torch.nn as nn

input_size = 28 * 28
num_classes = 10

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

```
model.state_dict()
```

```
OrderedDict([('weight',
              tensor([[ 0.0252,  0.0183,  0.0188,  ..., -0.0286, -0.0162, -0.0320],
                      [ 0.0166, -0.0324, -0.0093,  ...,  0.0019, -0.0320, -0.0263],
                      [ 0.0046,  0.0214, -0.0133,  ..., -0.0291, -0.0116, -0.0336],
                      ...,
                      [ 0.0054,  0.0329,  0.0035,  ...,  0.0156,  0.0237, -0.0165],
                      [-0.0070, -0.0019, -0.0216,  ...,  0.0171,  0.0230, -0.0185],
                      [ 0.0136, -0.0350,  0.0130,  ...,  0.0278,  0.0178,
0.0234]])),
             ('bias',
              tensor([-0.0062, -0.0280,  0.0176, -0.0104, -0.0321, -0.0032, -0.0243,
-0.0147,
                      -0.0074, -0.0041]))])
```

```
model.weight.shape
```

```
torch.Size([10, 784])
```

```
model.bias.shape
```

```
torch.Size([10])
```

```
for image, label in train_loader:
  print(image.shape)
  print(image.reshape(batch, input_size).shape)
  outputs = model(image.reshape(batch, input_size))
  print(outputs)
  print(label)
  break
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 784])
tensor([[ 0.0436, -0.0277,  0.0918,  ...,  0.1782, -0.2836,  0.0223],
        [-0.2073, -0.2013,  0.1310,  ...,  0.3535, -0.2310, -0.0471],
        [-0.2035,  0.0217, -0.0017,  ..., -0.0250, -0.1910, -0.0597],
        ...,
        [-0.1946,  0.3217, -0.0909,  ...,  0.2880, -0.0879,  0.2790],
        [-0.2550,  0.1317,  0.4776,  ...,  0.3013,  0.0261,  0.3203],
        [-0.2159,  0.0822,  0.1122,  ..., -0.2291,  0.1017, -0.1112]],
       grad_fn=<AddmmBackward0>)
tensor([8, 5, 1, 8, 7, 6, 1, 6, 9, 7, 9, 2, 0, 7, 6, 7, 9, 0, 2, 8, 8, 3, 9, 5,
        3, 9, 3, 8, 8, 8, 6, 6, 1, 2, 2, 8, 2, 6, 4, 4, 1, 0, 0, 6, 1, 0, 1, 9,
        0, 4, 6, 5, 9, 3, 4, 7, 1, 7, 7, 2, 3, 0, 7, 5, 7, 9, 5, 2, 5, 2, 6, 8,
        3, 3, 6, 2, 7, 2, 7, 9, 8, 9, 0, 7, 0, 2, 5, 1, 6, 1, 7, 3, 6, 3, 2, 4,
        1, 3, 7, 4, 7, 8, 5, 9, 6, 3, 7, 6, 6, 9, 9, 7, 0, 9, 4, 9, 5, 3, 1, 7,
        5, 1, 4, 9, 4, 5, 7, 4])
```

```
class MnistModel(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear = nn.Linear(in_features=input_size,
                            out_features=num_classes)

  def forward(self, xb):
```

```
        xb = xb.reshape(-1, 784)
        out = self.linear(xb)
        return out

    model = MnistModel()
    model.linear
```

```
    Linear(in_features=784, out_features=10, bias=True)
```

> - xb.reshape(-1, 28*28) indicates to PyTorch that we want a view of the xb tensor with two dimensions. The length along the 2nd dimension is 28*28 (i.e., 784). One argument to .reshape can be set to -1 (in this case, the first dimension) to let PyTorch figure it out automatically based on the shape of the original tensor.

```
print(model.linear.weight.shape, model.linear.bias.shape)
list(model.parameters())
```

```
    torch.Size([10, 784]) torch.Size([10])
    [Parameter containing:
     tensor([[ 0.0349, -0.0223, -0.0235,  ...,  0.0286,  0.0206,  0.0347],
             [-0.0079, -0.0213,  0.0328,  ...,  0.0260, -0.0023,  0.0090],
             [-0.0335,  0.0177,  0.0139,  ..., -0.0136,  0.0324, -0.0076],
             ...,
             [-0.0137,  0.0057, -0.0253,  ..., -0.0299,  0.0032,  0.0195],
             [ 0.0047, -0.0141, -0.0291,  ..., -0.0268,  0.0164,  0.0284],
             [-0.0091,  0.0126,  0.0312,  ...,  0.0214, -0.0138,  0.0308]],
            requires_grad=True),
     Parameter containing:
     tensor([ 0.0069,  0.0214,  0.0113,  0.0212, -0.0060, -0.0060,  0.0291, -0.0096,
             -0.0124, -0.0353], requires_grad=True)]
```

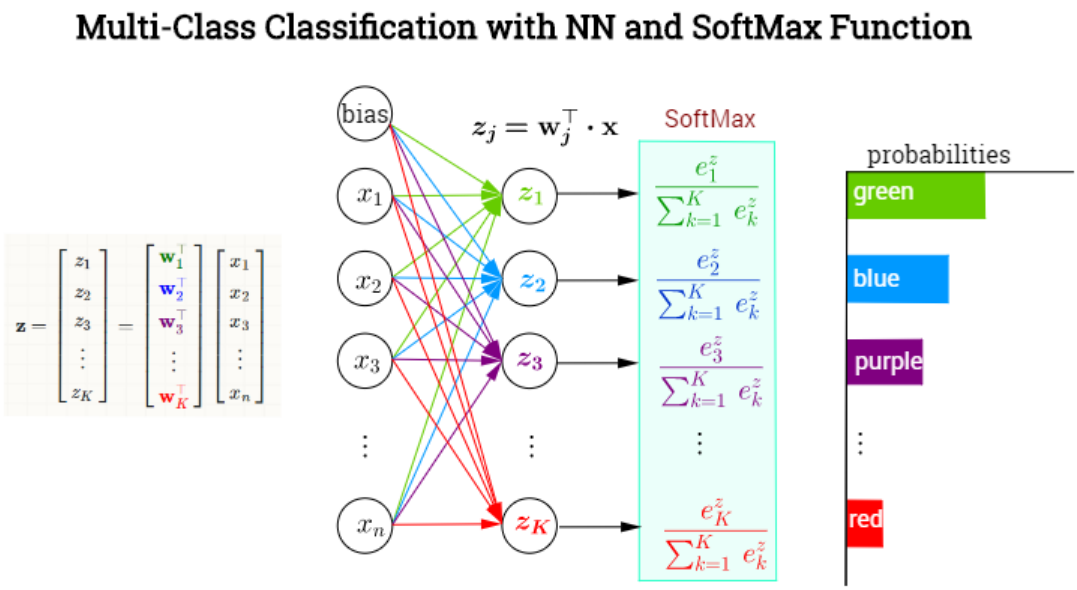
```
for images, labels in train_loader:
  print(images.shape)
  outputs = model(images)
  break

print(f"Outputs.shape: {outputs.shape}")
print(f"Sample outputs: {outputs[:2].data}\n")
```

```
    torch.Size([128, 1, 28, 28])
    Outputs.shape: torch.Size([128, 10])
    Sample outputs: tensor([[ 0.1708, -0.0211,  0.0441,  0.1748,  0.0962, -0.0727, -0.062
             -0.0577, -0.0970],
            [ 0.2255, -0.0677, -0.2205,  0.3410, -0.2618, -0.3074, -0.2947,  0.3727,
             -0.2510, -0.0339]])
```

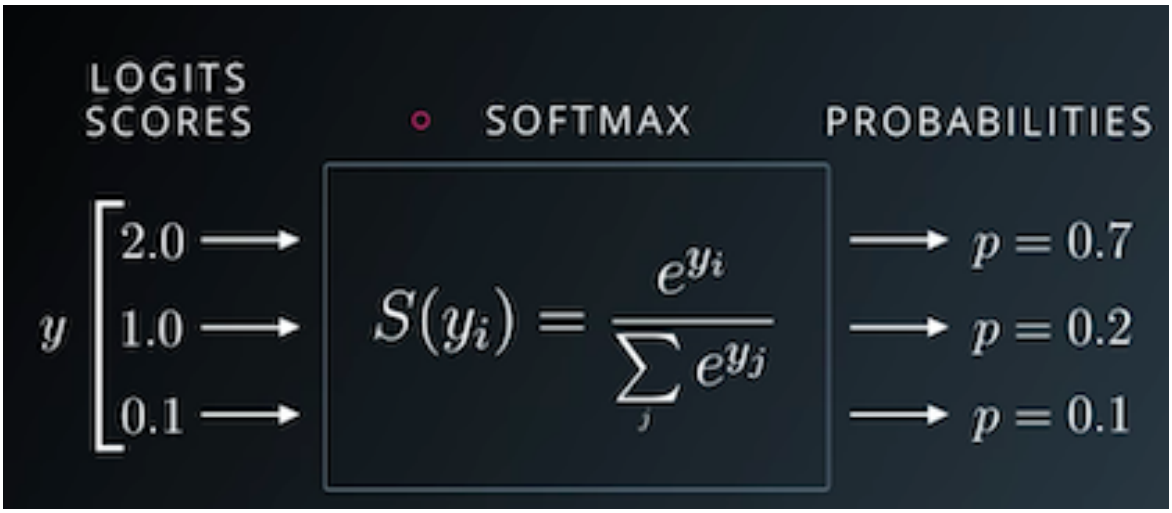
```
print(outputs[0].shape)
print(outputs[:2])
```

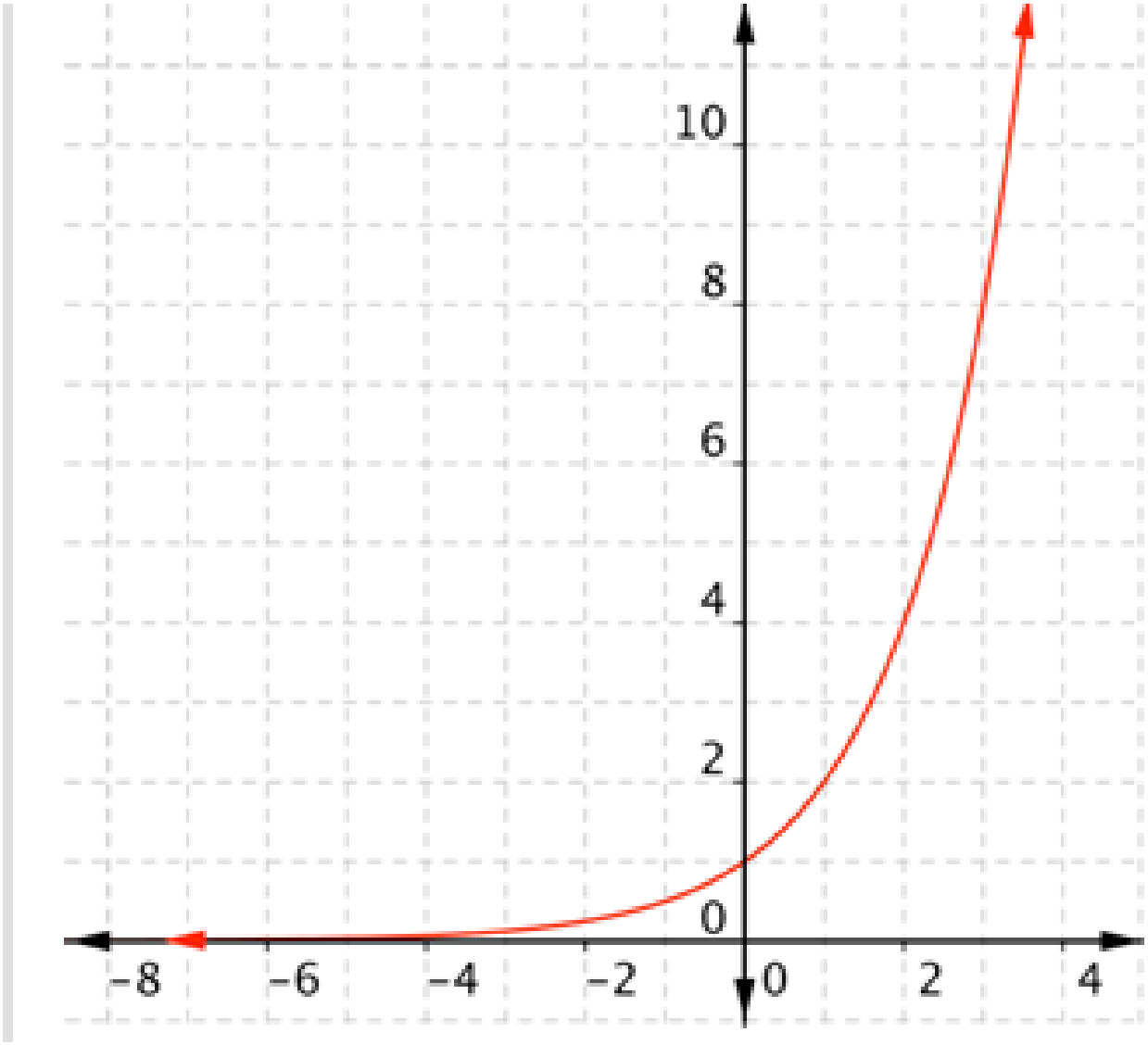
```
torch.Size([10])
tensor([[ 0.1708, -0.0211,  0.0441,  0.1748,  0.0962, -0.0727, -0.0623,  0.2833,
         -0.0577, -0.0970],
        [ 0.2255, -0.0677, -0.2205,  0.3410, -0.2618, -0.3074, -0.2947,  0.3727,
         -0.2510, -0.0339]], grad_fn=<SliceBackward0>)
```

Neural Network with Softmax



- To convert the output rows into probabilities, we use the softmax function, which has the following formula:
- First, we replace each element yi in an output row by e^yi, making all the elements positive.
- Then, we divide them by their sum to ensure that they add up to 1. The resulting vector can thus be interpreted as probabilities.






```
# Apply softmax for each output
probs = F.softmax(outputs, dim=1)
probs[:2]
```

```
tensor([[0.1124, 0.0928, 0.0990, 0.1129, 0.1043, 0.0881, 0.0891, 0.1258, 0.0895,
         0.0860],
        [0.1273, 0.0949, 0.0815, 0.1429, 0.0782, 0.0747, 0.0757, 0.1475, 0.0791,
         0.0982]], grad_fn=<SliceBackward0>)
```

```
print(probs.shape)
```

```
torch.Size([128, 10])
```

```
torch.max(probs, dim=1)
```

```
torch.return_types.max(
values=tensor([0.1258, 0.1475, 0.1377, 0.1365, 0.1211, 0.1453, 0.1547, 0.1299,
0.1201,
        0.1264, 0.1263, 0.1214, 0.1416, 0.1162, 0.1683, 0.1233, 0.1491, 0.1343,
        0.1232, 0.1277, 0.1291, 0.1467, 0.1209, 0.1692, 0.1126, 0.1222, 0.1121,
        0.1398, 0.1218, 0.1415, 0.1335, 0.1230, 0.1321, 0.1426, 0.1381, 0.1386,
        0.1323, 0.1268, 0.1230, 0.1265, 0.1316, 0.1258, 0.1373, 0.1320, 0.1212,
        0.1181, 0.1243, 0.1568, 0.1215, 0.1303, 0.1287, 0.1570, 0.1180, 0.1266,
```

```
        0.1255, 0.1420, 0.1162, 0.1213, 0.1735, 0.1289, 0.1517, 0.1375, 0.1225,
        0.1248, 0.1250, 0.1229, 0.1211, 0.1382, 0.1552, 0.1157, 0.1260, 0.1170,
        0.1501, 0.1297, 0.1179, 0.1837, 0.1393, 0.1594, 0.1144, 0.1372, 0.1236,
        0.1273, 0.1261, 0.1174, 0.1450, 0.1504, 0.1146, 0.1343, 0.1510, 0.1421,
        0.1273, 0.1678, 0.1319, 0.1280, 0.1189, 0.1252, 0.1499, 0.1285, 0.1150,
        0.1152, 0.1258, 0.1383, 0.1451, 0.1327, 0.1148, 0.1313, 0.1225, 0.1492,
        0.1189, 0.1753, 0.1388, 0.1276, 0.1176, 0.1199, 0.1321, 0.1238, 0.1156,
        0.1157, 0.1395, 0.1118, 0.1348, 0.1206, 0.1512, 0.1123, 0.1185, 0.1314,
        0.1393, 0.1481], grad_fn=<MaxBackward0>),
indices=tensor([7, 7, 7, 3, 3, 3, 7, 7, 7, 3, 7, 3, 3, 5, 3, 0, 3, 7, 3, 1, 3, 3, 7,
3,
        0, 7, 8, 3, 0, 7, 3, 5, 7, 7, 3, 7, 7, 0, 7, 0, 3, 7, 3, 3, 0, 7, 7, 3,
        7, 3, 0, 3, 0, 6, 3, 3, 3, 3, 7, 7, 7, 3, 3, 3, 7, 3, 3, 3, 7, 3, 3, 7,
        3, 3, 7, 7, 3, 7, 0, 3, 7, 3, 3, 5, 7, 7, 5, 3, 3, 3, 9, 3, 7, 3, 3, 3,
        3, 7, 0, 0, 2, 3, 7, 3, 5, 3, 3, 3, 3, 3, 3, 5, 7, 0, 7, 3, 0, 0, 7, 3,
        0, 5, 3, 0, 7, 7, 7, 7]))
```

```
probs_max, preds = torch.max(probs, dim=1)
print(probs_max)
print(preds)
```

```
tensor([0.1258, 0.1475, 0.1377, 0.1365, 0.1211, 0.1453, 0.1547, 0.1299, 0.1201,
        0.1264, 0.1263, 0.1214, 0.1416, 0.1162, 0.1683, 0.1233, 0.1491, 0.1343,
        0.1232, 0.1277, 0.1291, 0.1467, 0.1209, 0.1692, 0.1126, 0.1222, 0.1121,
        0.1398, 0.1218, 0.1415, 0.1335, 0.1230, 0.1321, 0.1426, 0.1381, 0.1386,
        0.1323, 0.1268, 0.1230, 0.1265, 0.1316, 0.1258, 0.1373, 0.1320, 0.1212,
        0.1181, 0.1243, 0.1568, 0.1215, 0.1303, 0.1287, 0.1570, 0.1180, 0.1266,
        0.1255, 0.1420, 0.1162, 0.1213, 0.1735, 0.1289, 0.1517, 0.1375, 0.1225,
        0.1248, 0.1250, 0.1229, 0.1211, 0.1382, 0.1552, 0.1157, 0.1260, 0.1170,
        0.1501, 0.1297, 0.1179, 0.1837, 0.1393, 0.1594, 0.1144, 0.1372, 0.1236,
        0.1273, 0.1261, 0.1174, 0.1450, 0.1504, 0.1146, 0.1343, 0.1510, 0.1421,
        0.1273, 0.1678, 0.1319, 0.1280, 0.1189, 0.1252, 0.1499, 0.1285, 0.1150,
        0.1152, 0.1258, 0.1383, 0.1451, 0.1327, 0.1148, 0.1313, 0.1225, 0.1492,
        0.1189, 0.1753, 0.1388, 0.1276, 0.1176, 0.1199, 0.1321, 0.1238, 0.1156,
        0.1157, 0.1395, 0.1118, 0.1348, 0.1206, 0.1512, 0.1123, 0.1185, 0.1314,
        0.1393, 0.1481], grad_fn=<MaxBackward0>)
tensor([7, 7, 7, 3, 3, 3, 7, 7, 7, 3, 7, 3, 3, 5, 3, 0, 3, 7, 3, 1, 3, 3, 7, 3,
        0, 7, 8, 3, 0, 7, 3, 5, 7, 7, 3, 7, 7, 0, 7, 0, 3, 7, 3, 3, 0, 7, 7, 3,
        7, 3, 0, 3, 0, 6, 3, 3, 3, 3, 7, 7, 7, 3, 3, 3, 7, 3, 3, 3, 7, 3, 3, 7,
        3, 3, 7, 7, 3, 7, 0, 3, 7, 3, 3, 5, 7, 7, 5, 3, 3, 3, 9, 3, 7, 3, 3, 3,
        3, 7, 0, 0, 2, 3, 7, 3, 5, 3, 3, 3, 3, 3, 3, 5, 7, 0, 7, 3, 0, 0, 7, 3,
        0, 5, 3, 0, 7, 7, 7, 7])
```

Evaluation Metric and Loss Function

```
labels
```

```
tensor([1, 0, 8, 9, 1, 2, 6, 8, 4, 0, 2, 9, 2, 7, 9, 7, 3, 6, 7, 6, 8, 3, 3, 9,
        4, 4, 2, 7, 4, 9, 7, 4, 6, 6, 3, 5, 1, 6, 1, 2, 8, 8, 3, 0, 7, 1, 6, 2,
        1, 7, 4, 0, 1, 6, 7, 3, 1, 1, 2, 1, 3, 3, 9, 9, 5, 4, 7, 0, 3, 4, 5, 4,
        3, 9, 1, 8, 3, 3, 1, 4, 8, 9, 7, 7, 2, 7, 7, 9, 0, 7, 0, 0, 5, 3, 4, 3,
        0, 1, 6, 4, 0, 2, 5, 1, 4, 4, 4, 2, 7, 0, 0, 4, 4, 9, 1, 0, 1, 1, 6, 1,
        0, 7, 3, 1, 1, 2, 1, 5])
```

```
outputs[:2]
```

```
tensor([[ 0.1708, -0.0211,  0.0441,  0.1748,  0.0962, -0.0727, -0.0623,  0.2833,
         -0.0577, -0.0970],
        [ 0.2255, -0.0677, -0.2205,  0.3410, -0.2618, -0.3074, -0.2947,  0.3727,
         -0.2510, -0.0339]], grad_fn=<SliceBackward0>)
```

```
preds == labels
```

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

```
print(torch.sum(preds == labels))
```

```
tensor(14)
```

```
def accuracy(outputs, labels):
  _, preds = torch.max(outputs, dim=1)
  return torch.tensor(torch.sum(preds == labels).item() / len(preds))
```

```
accuracy(outputs, labels)
```

```
tensor(0.1094)
```

Accuracy is an excellent way for us (humans) to evaluate the model. However, we can't use it as a loss function for optimizing our model using gradient descent for the following reasons:

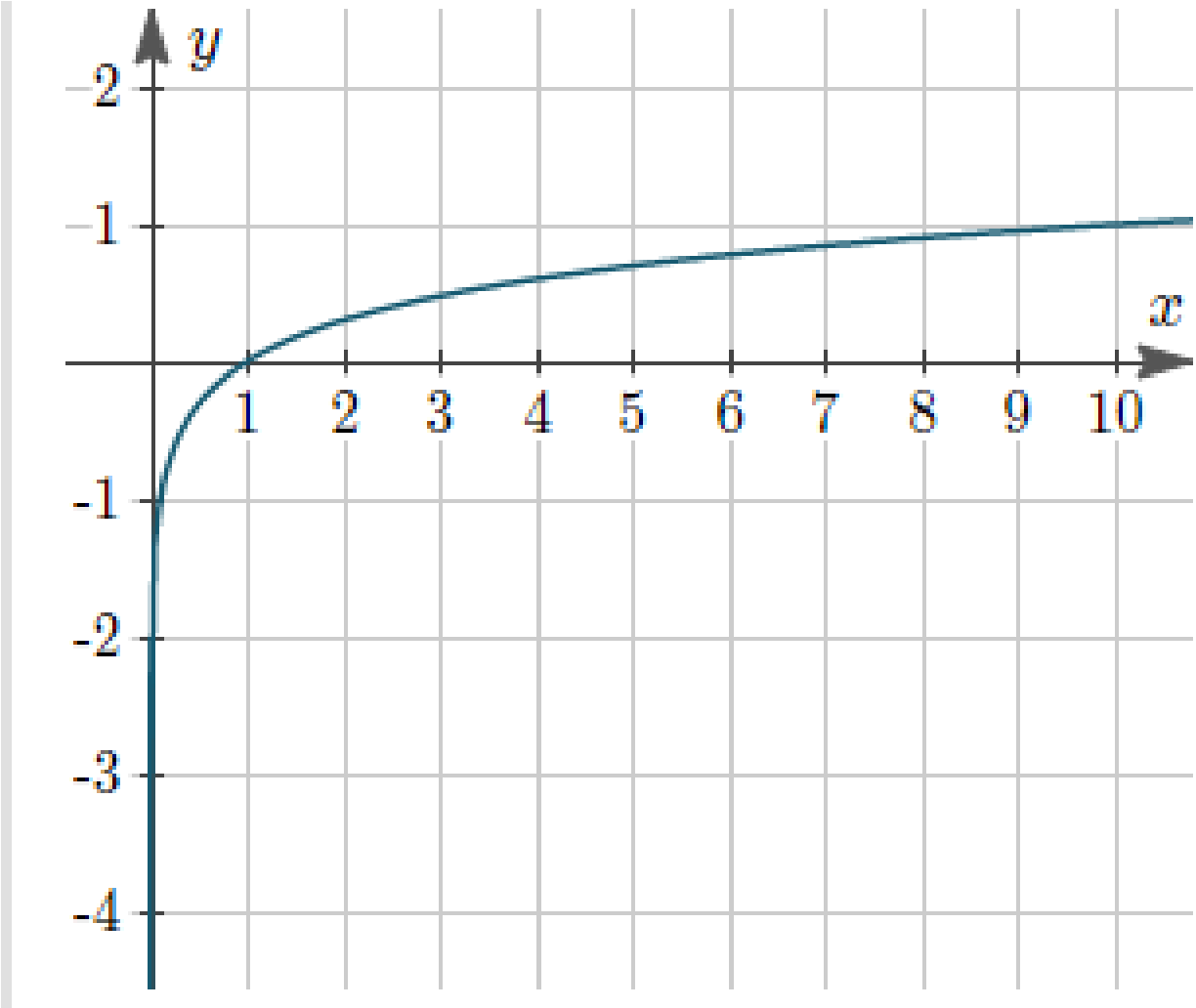
1. It's not a differentiable function. `torch.max` and `==` are both non-continuous and non-differentiable operations, so we can't use the accuracy for computing gradients w.r.t the weights and biases.
2. It doesn't take into account the actual probabilities predicted by the model, so it can't provide sufficient feedback for incremental improvements.

For these reasons, accuracy is often used as an **evaluation metric** for classification, but not as a loss function. A commonly used loss function for classification problems is the **cross-entropy**, which has the following formula:

$$\begin{bmatrix} 0.1 \\ 0.5 \\ 0.4 \end{bmatrix} \quad D(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_j y_j \ln \hat{y}_j \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

While it looks complicated, it's actually quite simple:

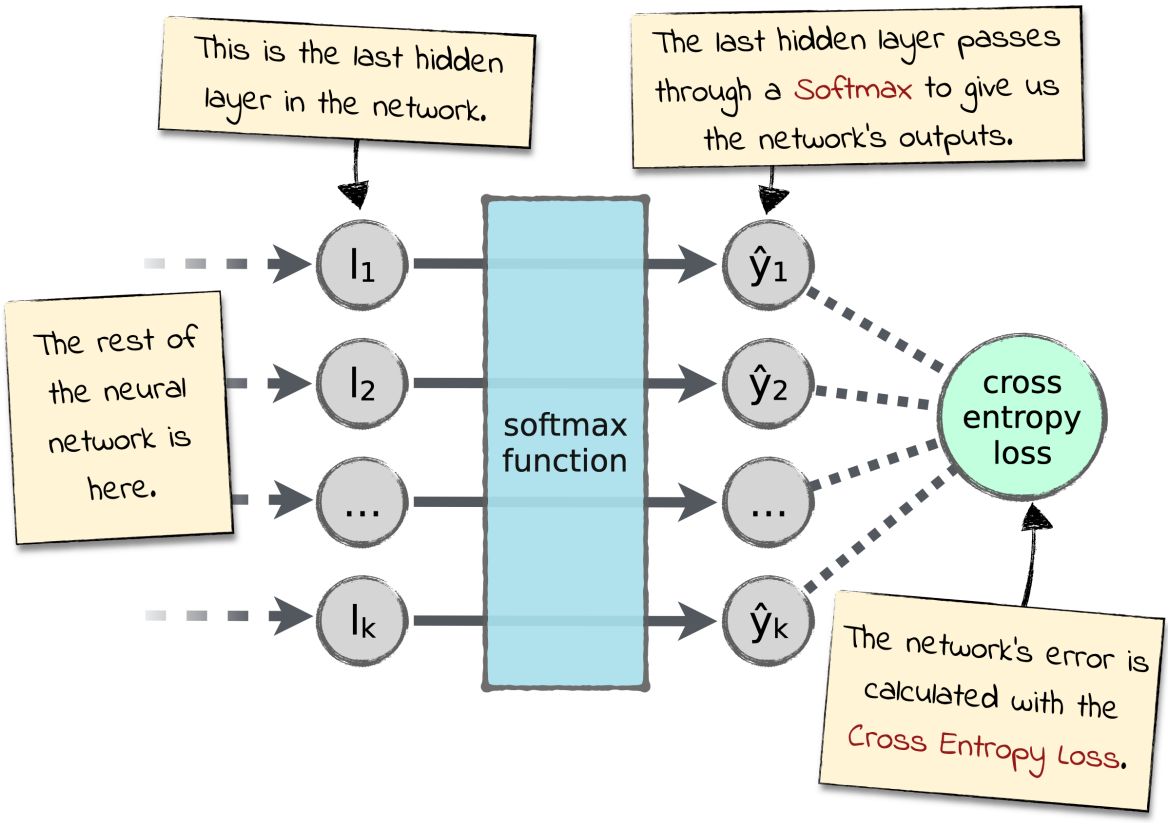
- For each output row, pick the predicted probability for the correct label. E.g., if the predicted probabilities for an image are `[0.1, 0.3, 0.2, ...]` and the correct label is `1`, we pick the corresponding element `0.3` and ignore the rest.
- Then, take the [logarithm](https://) of the picked probability. If the probability is high, i.e., close to 1, then its logarithm is a very small negative value, close to 0. And if the probability is low (close to 0), then the logarithm is a very large negative value. We also multiply the result by -1, which results is a large postive value of the loss for poor predictions.
- Finally, take the average of the cross entropy across all the output rows to get the overall loss for a batch of data.

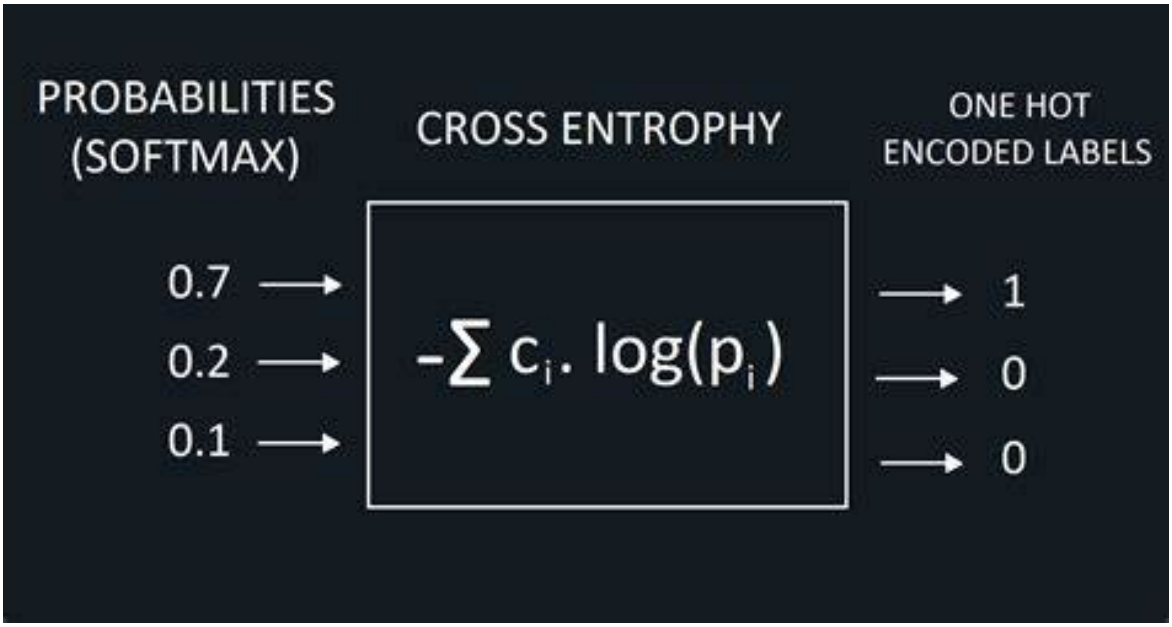


Unlike accuracy, cross-entropy is a continuous and differentiable function. It also provides useful feedback for incremental improvements in the model (a slightly higher probability for the correct

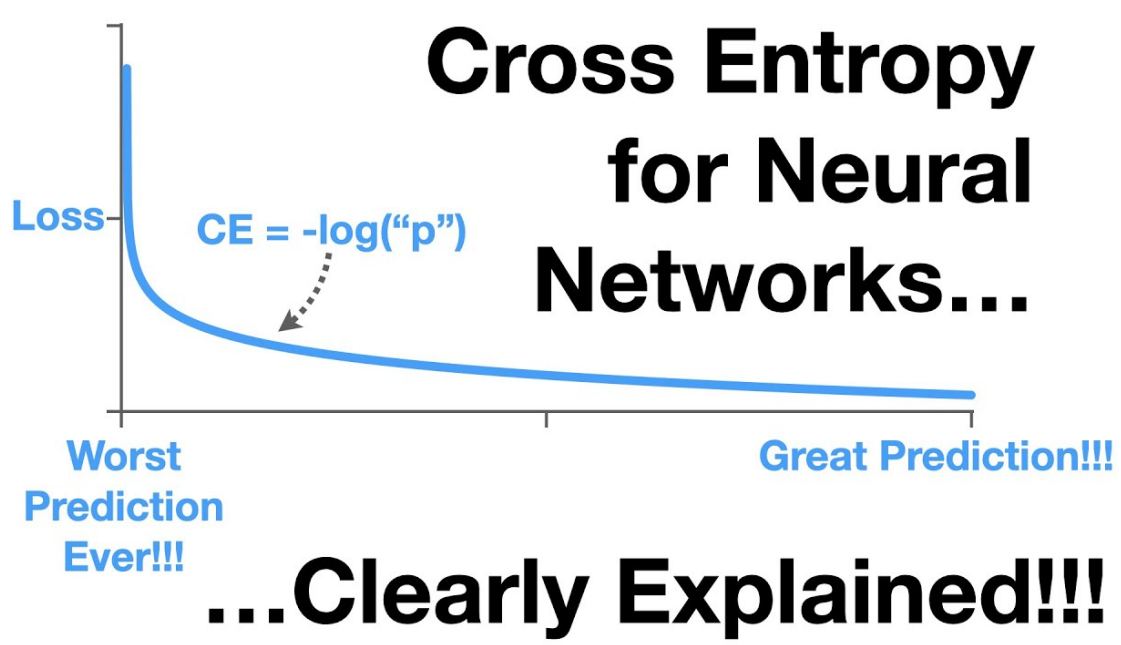
label leads to a lower loss). These two factors make cross-entropy a better choice for the loss function.

> https://pytorch.org/docs/stable/generated/torch.nn.functional.cross_entropy.html





> As you might expect, PyTorch provides an efficient and tensor-friendly implementation of cross-entropy as part of the torch.nn.functional package. Moreover, it also performs softmax internally, so we can directly pass in the model's outputs without converting them into probabilities.




```
loss_fn = F.cross_entropy


# Loss for batch of data
loss = loss_fn(outputs, labels)
print(loss)
```

```
tensor(2.2993, grad_fn=<NllLossBackward0>)
```

Phase 3: Train Model and Evaluate Model

```
def accuracy(outputs, labels):
  _, preds = torch.max(outputs, dim=1)
  return torch.tensor(torch.sum(preds == labels).item() / len(preds))


class MnistModel(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear = nn.Linear(in_features=input_size,
                            out_features=num_classes)
```

```
  def forward(self, xb):
    #xb = xb.reshape(-1, 784) # input_size=784
    xb = xb.view(xb.size(0), -1)
    outputs = self.linear(xb) # output_size=10
    return outputs

  def train_step(self, batch):
    images, labels = batch
    outputs = self(images) # Make Predictions, self===model
    loss = F.cross_entropy(outputs, labels)
    return loss

  def validation_step(self, batch):
    images, labels = batch
    outputs = self(images) # Make predictions, self===model
    loss = F.cross_entropy(outputs, labels)
    acc = accuracy(outputs, labels)
    return {'val_loss': loss,
            'val_acc': acc}

  def validation_epoch_end(self, outputs):
    batch_losses = [x['val_loss'] for x in outputs]
    epoch_loss = torch.stack(batch_losses).mean() # combine losses
    batch_accs = [x['val_acc'] for x in outputs]
    epoch_acc = torch.stack(batch_accs).mean() # combine accuracies
    return {'val_loss': epoch_loss.item(),
            'val_acc': epoch_acc.item()}

  def epoch_end(self, epoch, result):
    print(f"Epoch [{epoch}], val_loss: {result['val_loss']:.4f}, val_acc: {result['val_acc
```

```
model = MnistModel()
print(model.linear)
print(model.state_dict())
```

```
    Linear(in_features=784, out_features=10, bias=True)
    OrderedDict([('linear.weight', tensor([[ 0.0260, -0.0268, -0.0302,  ...,  0.0225, -0
            [-0.0292, -0.0346,  0.0333,  ...,  0.0098, -0.0331,  0.0012],
            [-0.0327,  0.0154, -0.0347,  ..., -0.0319, -0.0346, -0.0332],
            ...,
            [ 0.0248, -0.0269,  0.0025,  ..., -0.0273, -0.0034, -0.0003],
            [ 0.0222, -0.0048,  0.0171,  ..., -0.0204, -0.0249, -0.0107],
            [-0.0028, -0.0141, -0.0318,  ...,  0.0249,  0.0298, -0.0109]])), ('linear.bia
            -0.0101,  0.0057]))])
```

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

```
    [Parameter containing:
     tensor([[ 0.0260, -0.0268, -0.0302,  ...,  0.0225, -0.0356,  0.0114],
             [-0.0292, -0.0346,  0.0333,  ...,  0.0098, -0.0331,  0.0012],
             [-0.0327,  0.0154, -0.0347,  ..., -0.0319, -0.0346, -0.0332],
             ...,
```

```
                [ 0.0248, -0.0269,  0.0025,  ..., -0.0273, -0.0034, -0.0003],
                [ 0.0222, -0.0048,  0.0171,  ..., -0.0204, -0.0249, -0.0107],
                [-0.0028, -0.0141, -0.0318,  ...,  0.0249,  0.0298, -0.0109]],
               requires_grad=True),
       Parameter containing:
       tensor([-0.0234, -0.0241, -0.0037,  0.0252, -0.0283,  0.0354, -0.0243, -0.0115,
                -0.0101,  0.0057], requires_grad=True)]
```

```
def evaluate(model, val_loader):
  outputs = [model.validation_step(batch) for batch in val_loader]
  return model.validation_epoch_end(outputs)


result_0 = evaluate(model, val_loader)
result_0
```

```
     {'val_loss': 2.327208995819092, 'val_acc': 0.0768394023180008}
```

```
def fit(epochs,
        lr,
        model,
        trian_loader,
        val_loader,
        opt_func=torch.optim.SGD):

  optimizer = opt_func(model.parameters(), lr)

  history = [] # for recording epoch-wise results

  for epoch in range(epochs):
    # Training phase
    for batch in train_loader:
      loss = model.train_step(batch)
      loss.backward()
      optimizer.step()
      optimizer.zero_grad()

    # Validation phase
    result  = evaluate(model, val_loader)
    model.epoch_end(epoch, result)
    history.append(result)

  return result



history1 = fit(5, 0.001, model, train_loader, val_loader)
```

```
     Epoch [0], val_loss: 1.9636, val_acc: 0.6059
     Epoch [1], val_loss: 1.6915, val_acc: 0.7258
     Epoch [2], val_loss: 1.4885, val_acc: 0.7627
     Epoch [3], val_loss: 1.3355, val_acc: 0.7840
     Epoch [4], val_loss: 1.2181, val_acc: 0.7965
```

```
history2 = fit(5, 0.001, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 1.1257, val_acc: 0.8072
Epoch [1], val_loss: 1.0517, val_acc: 0.8151
Epoch [2], val_loss: 0.9913, val_acc: 0.8203
Epoch [3], val_loss: 0.9410, val_acc: 0.8293
Epoch [4], val_loss: 0.8984, val_acc: 0.8329
```

```
history3 = fit(5, 0.001, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 0.8621, val_acc: 0.8362
Epoch [1], val_loss: 0.8304, val_acc: 0.8396
Epoch [2], val_loss: 0.8027, val_acc: 0.8417
Epoch [3], val_loss: 0.7784, val_acc: 0.8435
Epoch [4], val_loss: 0.7566, val_acc: 0.8454
```

```
history4 = fit(5, 0.001, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 0.7371, val_acc: 0.8469
Epoch [1], val_loss: 0.7194, val_acc: 0.8481
Epoch [2], val_loss: 0.7035, val_acc: 0.8494
Epoch [3], val_loss: 0.6889, val_acc: 0.8512
Epoch [4], val_loss: 0.6756, val_acc: 0.8525
```

Phase 4: Testing

```
img, label = test_dataset[0]
plt.imshow(img[0], cmap='gray')
print("Shape: ", img.shape)
print("Label: ", label)
```

```
Shape:  torch.Size([1, 28, 28])
Label:  7
```

```
def predict_image(img, model):
  xb = img.unsqueeze(0)
  yb = model(xb)
  _, preds = torch.max(yb, dim=1)
  return preds[0].item()
```

```
img.shape
```

```
torch.Size([1, 28, 28])
```

```
img.unsqueeze(0).shape # we need batch, image
```

```
torch.Size([1, 1, 28, 28])
```

```
yb = model(img[0].unsqueeze(0))
yb
```

```
tensor([[-0.3728, -2.1433, -0.7344,  0.1679, -0.0335, -0.4728, -1.7151,  4.4532,
         -0.3150,  1.4413]], grad_fn=<AddmmBackward0>)
```

```
_, preds = torch.max(yb, dim=1)
print(preds)
```

```
tensor([7])
```

```
preds[0].item()
```

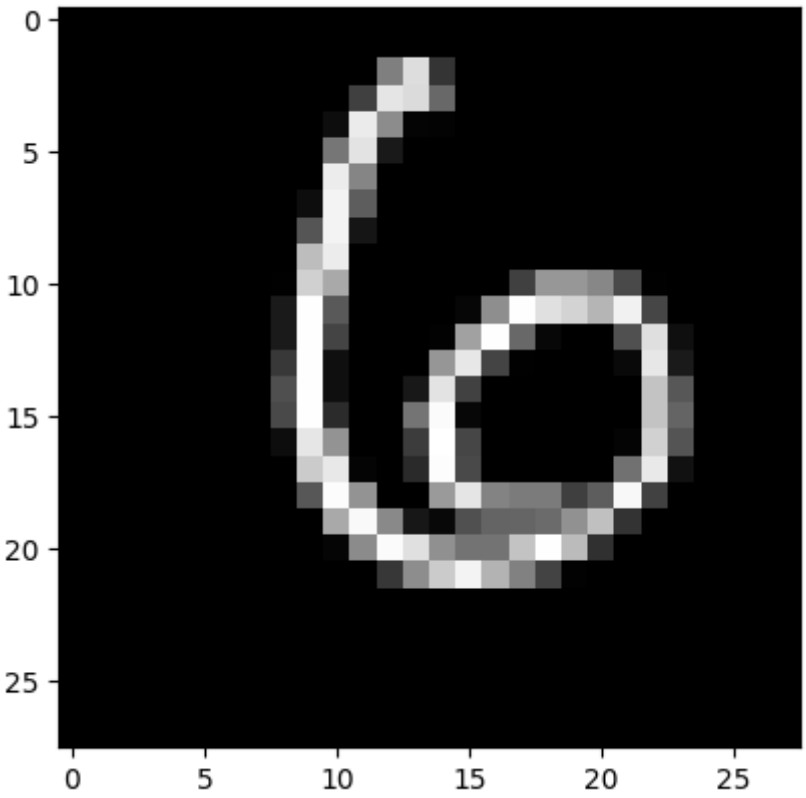
```
7
```

```
img, label = test_dataset[0]
print("Lable: ", label, "Prediction: ", predict_image(img[0], model))
plt.imshow(img[0], cmap='gray')
```

```
Lable:  7 Prediction:  7
<matplotlib.image.AxesImage at 0x7faf40c10220>
```

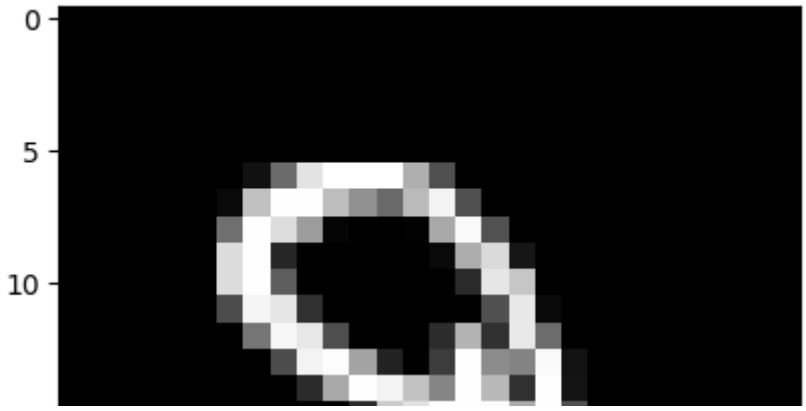

```
img, label = test_dataset[100]
print("Lable: ", label, "Prediction: ", predict_image(img[0], model))
plt.imshow(img[0], cmap='gray')
```

```
Lable:  6 Prediction:  6
<matplotlib.image.AxesImage at 0x7faf40b091b0>
```




```
img, label = test_dataset[193]
print("Lable: ", label, "Prediction: ", predict_image(img[0], model))
plt.imshow(img[0], cmap='gray')
```

```
Lable:  9 Prediction:  9
<matplotlib.image.AxesImage at 0x7faf40ac2410>
```




```
test_loader = DataLoader(test_dataset, batch_size=256)
result = evaluate(model, test_loader)
result
```

```
{'val_loss': 0.6411768198013306, 'val_acc': 0.861328125}
```

Phase 5: Saving and Loading the Model

```
torch.save(model.state_dict(), 'mnist-logistic.pth')
```

```
model.state_dict()
```

```
OrderedDict([('linear.weight',
              tensor([[ 0.0260, -0.0268, -0.0302,  ...,  0.0225, -0.0356,  0.0114],
                      [-0.0292, -0.0346,  0.0333,  ...,  0.0098, -0.0331,  0.0012],
                      [-0.0327,  0.0154, -0.0347,  ..., -0.0319, -0.0346, -0.0332],
                      ...,
                      [ 0.0248, -0.0269,  0.0025,  ..., -0.0273, -0.0034, -0.0003],
                      [ 0.0222, -0.0048,  0.0171,  ..., -0.0204, -0.0249, -0.0107],
                      [-0.0028, -0.0141, -0.0318,  ...,  0.0249,  0.0298,
-0.0109]])),
             ('linear.bias',
              tensor([-0.0694,  0.0740, -0.0268,  0.0006,  0.0009,  0.0766, -0.0365,
0.0283,
                      -0.1027, -0.0040]))])
```

```
model_load = MnistModel()
```

```
model_load.load_state_dict(torch.load('mnist-logistic.pth'))
```

```
<All keys matched successfully>
```

```
model_load.state_dict()
```

```
OrderedDict([('linear.weight',
              tensor([[ 0.0260, -0.0268, -0.0302,  ...,  0.0225, -0.0356,  0.0114],
                      [-0.0292, -0.0346,  0.0333,  ...,  0.0098, -0.0331,  0.0012],
```

```
                          [-0.0327,  0.0154, -0.0347,  ..., -0.0319, -0.0346, -0.0332],
                          ...,
                          [ 0.0248, -0.0269,  0.0025,  ..., -0.0273, -0.0034, -0.0003],
                          [ 0.0222, -0.0048,  0.0171,  ..., -0.0204, -0.0249, -0.0107],
                          [-0.0028, -0.0141, -0.0318,  ...,  0.0249,  0.0298,
    -0.0109]])),
             ('linear.bias',
              tensor([-0.0694,  0.0740, -0.0268,  0.0006,  0.0009,  0.0766, -0.0365,
    0.0283,
                          -0.1027, -0.0040]))])
```

```
test_loader = DataLoader(test_dataset, batch_size=256)
result = evaluate(model_load, test_loader)
result
```

```
    {'val_loss': 0.6411768198013306, 'val_acc': 0.861328125}
```

Project 2 - Training Deep Neural Networks on a GPU with PyTorch

Phase 0: Project Preparing

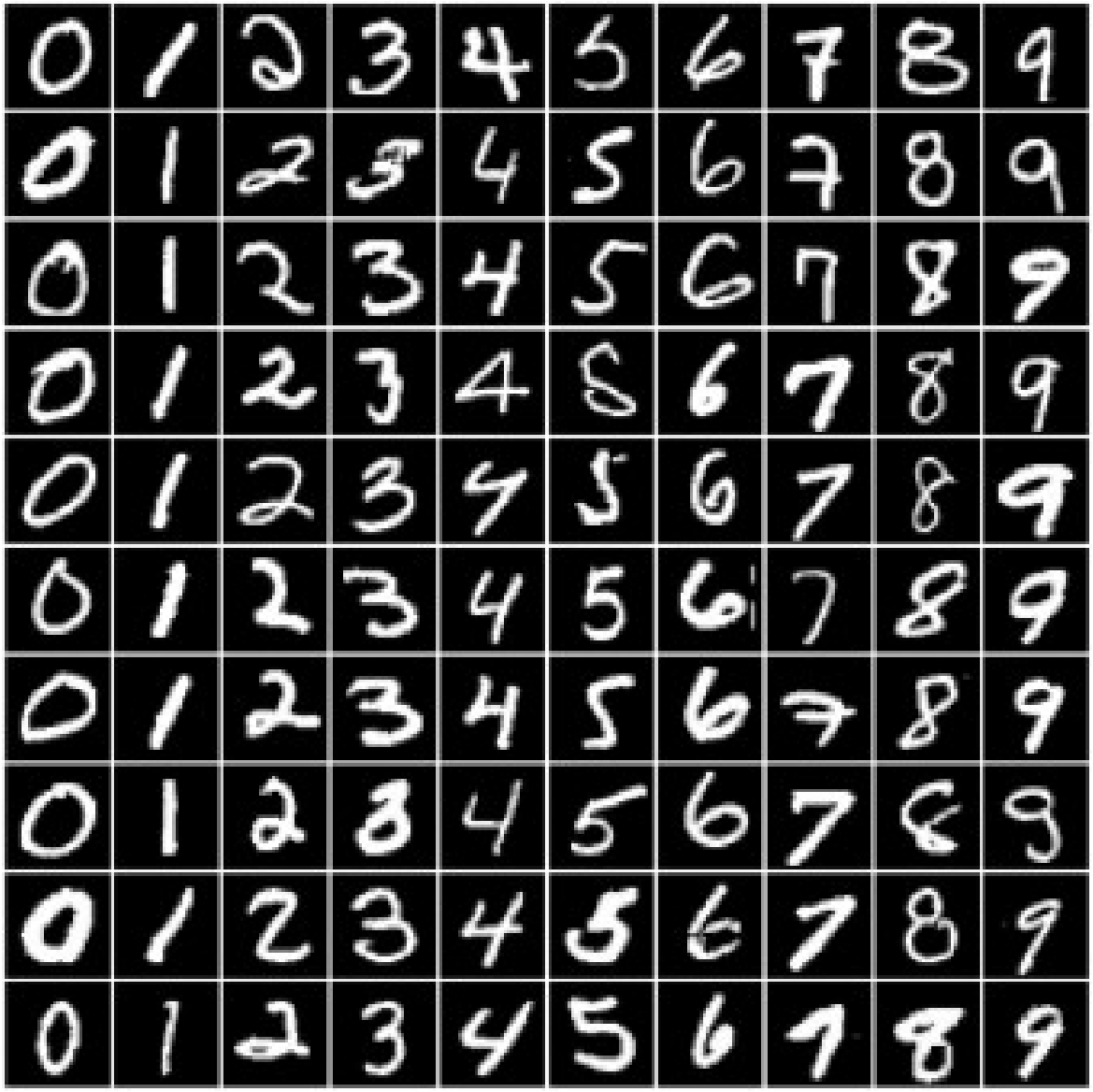
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
import torch
torch.__version__
```

```
    '2.0.0+cu118'
```

```
import torch.nn as nn
import torch.nn.functional as F
```

Phase 1: Get Data and Explore Data



```
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
```

```
datasets = MNIST(root='data/',
                 download=True,
                 transform=ToTensor())
```

```
len(datasets)
```

```
60000
```

```
datasets[0][1]
```

```
5
```

```
datasets[0][0].shape # he dimensions represent color channels, width and height
```

```
torch.Size([1, 28, 28])
```

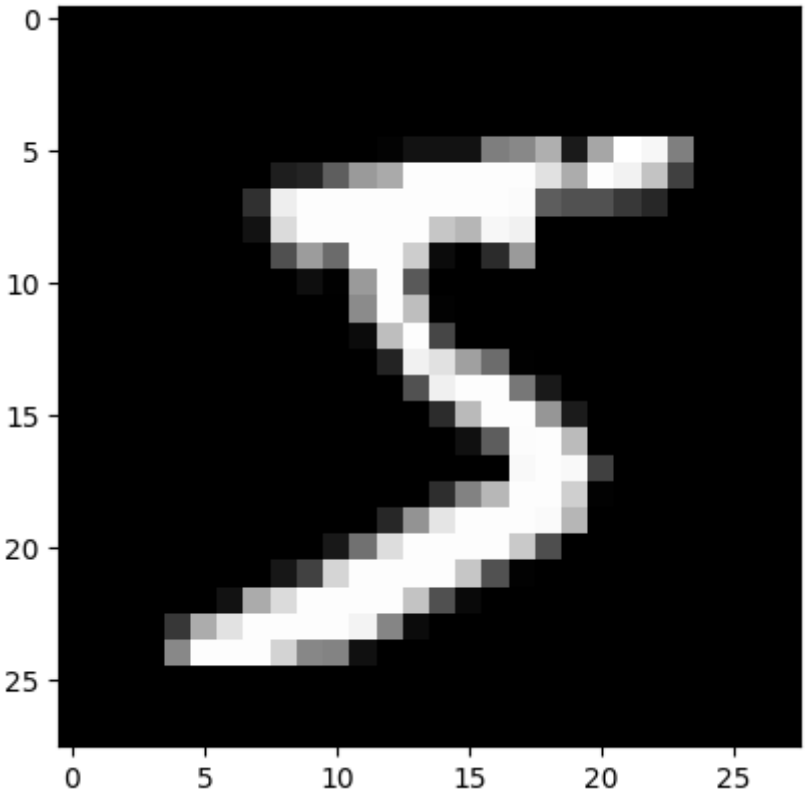
```
datasets[0][0].permute(1, 2, 0).shape
```

```
torch.Size([28, 28, 1])
```

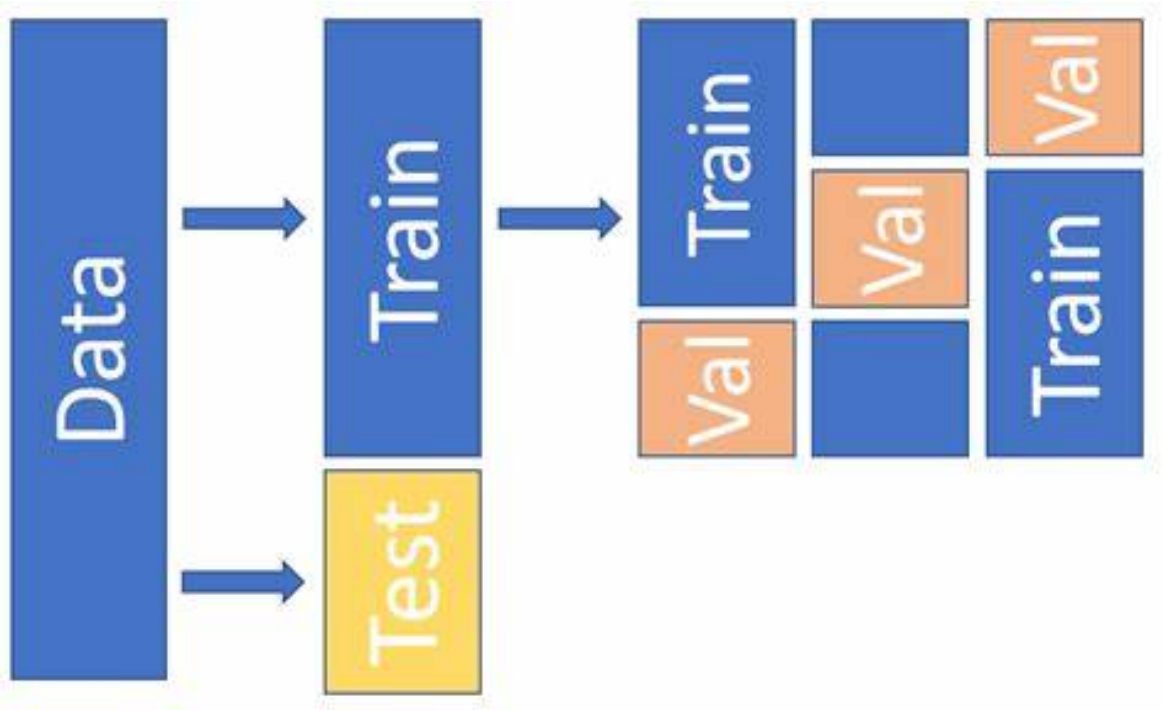
> - plt.imshow expects channels to be last dimension in an image tensor, so we use the permute method to reorder the dimensions of the image.

```
image, label = datasets[0]
print("image.shape: ", image.shape)
print("lable: ", label)
plt.imshow(image.permute(1, 2, 0), cmap="gray")
```

```
image.shape:  torch.Size([1, 28, 28])
lable:  5
<matplotlib.image.AxesImage at 0x7faf40a5ca30>
```



Setup train_dataset and test_dataset




```
from torchvision.transforms import transforms

train_dataset = MNIST(root='data/',
                      train=True,
                      transform=transforms.ToTensor())

test_dataset = MNIST(root='data/',
                     train=False,
                     transform=transforms.ToTensor())


print(len(train_dataset))
print(len(test_dataset))
```

```
60000
10000
```

Split train_dataset to train_ds and validate_ds

```
from torch.utils.data import random_split

val_size = 10000
train_size = len(train_dataset) - val_size


train_ds, val_ds = random_split(train_dataset, [train_size, val_size])
print("len(train_ds): ", len(train_ds))
print("len(val_ds): ", len(val_ds))
```

```
len(train_ds):  50000
len(val_ds):  10000
```

Setup the train_loader and val_loader for train

```
from torch.utils.data.dataloader import DataLoader

batch_size = 128

train_loader = DataLoader(train_ds,
                          batch_size,
                          shuffle=True,
                          num_workers=2,
                          pin_memory=True)

val_loader = DataLoader(val_ds,
                        batch_size*2,
                        num_workers=2,
                        pin_memory=True)


print(train_loader)
print(val_loader)
print("len(train_loader): ", len(train_loader))
print("len(val_loader): ", len(val_loader))
```

```
    <torch.utils.data.dataloader.DataLoader object at 0x7faf408cec20>
    <torch.utils.data.dataloader.DataLoader object at 0x7faf408cdc30>
    len(train_loader):  391
    len(val_loader):  40
```

Visualize the batch of data

```
from torchvision.utils import make_grid

for image, _ in train_loader:
  print("image.shape: ", image.shape) # he dimensions represent color channels, width and
  print("grid.shape: ", make_grid(image, nrow=16).shape)
  break
```

```
    image.shape:  torch.Size([128, 1, 28, 28])
    grid.shape:  torch.Size([3, 242, 482])
```

```
make_grid(image, nrow=16).shape
```

```
    torch.Size([3, 242, 482])
```

```
plt.imshow(make_grid(image, nrow=16).permute(1, 2, 0))
```

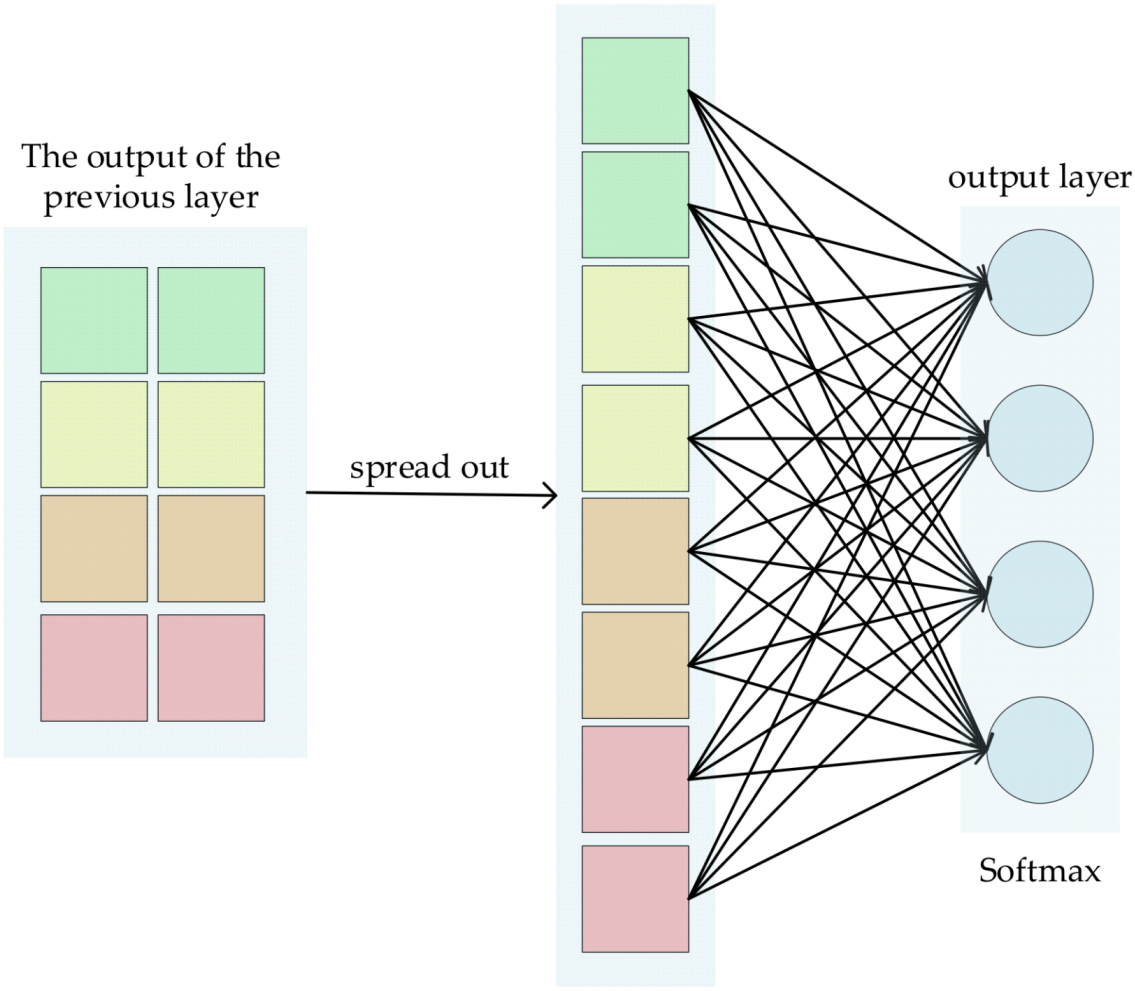
```
<matplotlib.image.AxesImage at 0x7faf408ccbe0>
```

```
for image, _ in train_loader:
  print("image.shape: ", image.shape)
  plt.figure(figsize=(12, 8))
  plt.axis('off')
  plt.imshow(make_grid(image, nrow=16).permute(1, 2, 0))
  break
```

```
image.shape:  torch.Size([128, 1, 28, 28])
```

Phase 2: Building Model

Hidden Layers, Activation Functions and Non-Linearity




```
for images, labels in train_loader:
  print('images.shape: ', images.shape)
  inputs = images.reshape(-1, 784) # 784=28*28
  print('inputs.shape: ', inputs.shape)
  break
```

```
images.shape:  torch.Size([128, 1, 28, 28])
inputs.shape:  torch.Size([128, 784])
```

```
input_size = inputs.shape[-1]
input_size
```

```
784
```

```
layer_1 = nn.Linear(in_features=input_size, out_features=32)
layer_1_outputs = layer_1(inputs)
print('layer_1_outputs.shape: ', layer_1_outputs.shape)
```

```
layer_1_outputs.shape:  torch.Size([128, 32])
```

```
layer_1.weight
```

```
Parameter containing:
tensor([[ 1.8308e-02,  3.4869e-02,  3.3369e-02,  ..., -2.4700e-02,
          3.5433e-02,  1.9859e-02],
        [ 2.5368e-02,  8.1104e-03, -3.4772e-02,  ..., -2.0382e-02,
         -2.6465e-02,  1.4379e-02],
        [ 2.0516e-02, -1.9166e-02,  4.7158e-03,  ...,  3.0958e-02,
         -2.8142e-04, -7.1198e-03],
        ...,
        [ 3.2112e-02,  2.7692e-02,  1.5236e-02,  ..., -2.9458e-02,
         -8.0666e-03,  2.2408e-03],
        [-3.1442e-02, -2.1792e-03,  2.7468e-02,  ...,  2.1704e-02,
         -2.6792e-02,  2.1368e-02],
        [-3.2642e-02, -2.6746e-02,  6.2006e-03,  ..., -4.0455e-05,
         -4.2589e-03,  9.1436e-03]], requires_grad=True)
```

```
layer_1.bias
```

```
Parameter containing:
tensor([ 0.0008, -0.0190,  0.0118, -0.0297, -0.0143, -0.0009,  0.0022,  0.0248,
        -0.0106,  0.0354, -0.0027, -0.0252, -0.0298, -0.0064,  0.0335, -0.0320,
        -0.0276,  0.0075,  0.0113,  0.0021, -0.0216,  0.0201,  0.0036, -0.0123,
         0.0036, -0.0336,  0.0021,  0.0204, -0.0276, -0.0232,  0.0251,  0.0113],
       requires_grad=True)
```

$$X \times W^T + b$$

$$\begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \\ b_1 & b_2 \\ \vdots & \vdots \\ b_1 & b_2 \end{bmatrix}$$

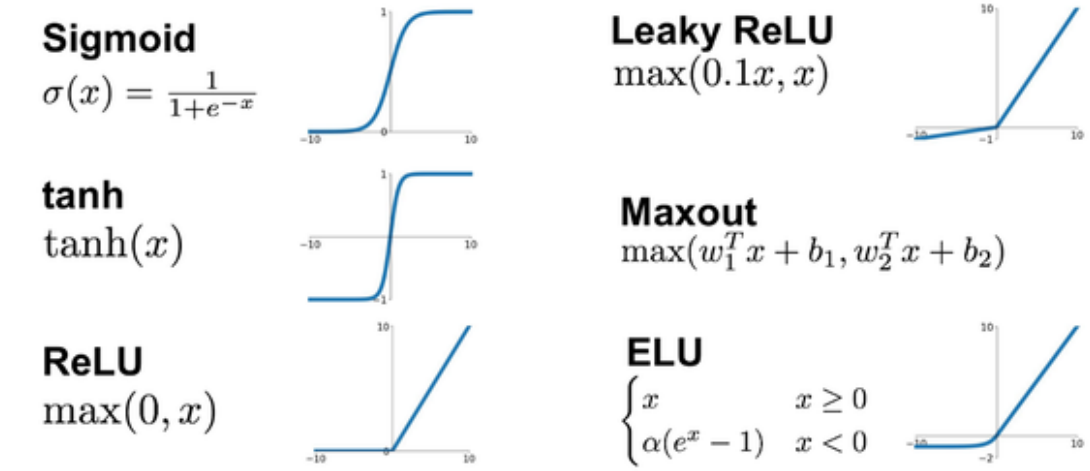
```
layer_1_outputs_direct = inputs @ layer_1.weight.t() + layer_1.bias
print("layer_1_outputs_direct.shape: ", layer_1_outputs_direct.shape)
```

```
layer_1_outputs_direct.shape:  torch.Size([128, 32])
```

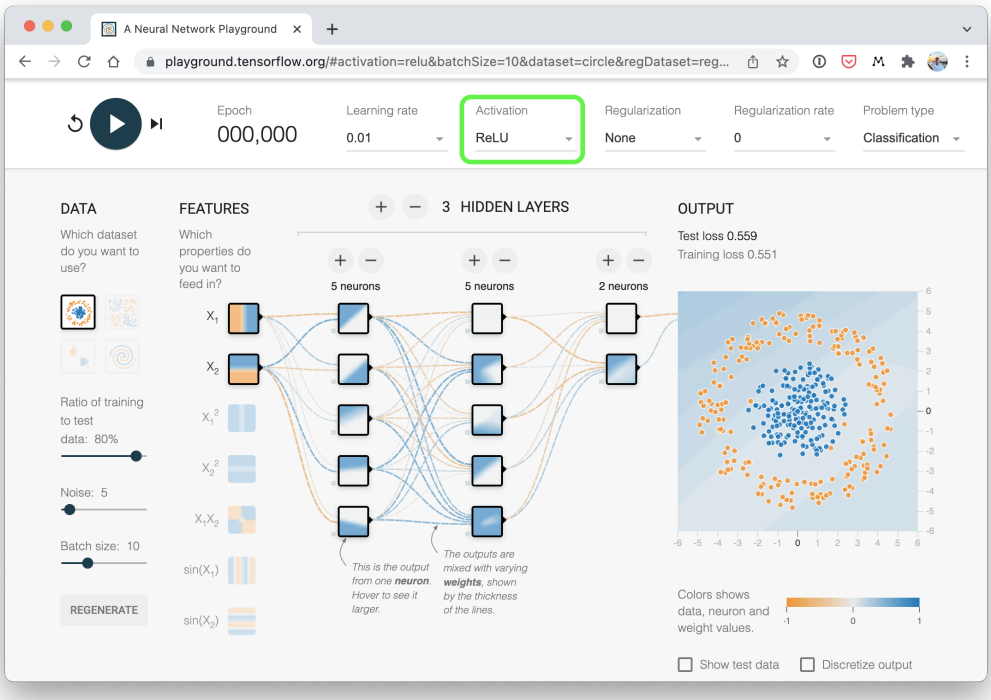
```
# Make sure the value is close

torch.allclose(layer_1_outputs, layer_1_outputs_direct, 1e-3)
```

```
True
```

Activation Functions



- TensorFlow Playground website https://playground.tensorflow.org/
- A classification neural network on TensorFlow playground with ReLU activation




```
F.relu(torch.tensor([[1., 3, -2],
                     [0.1, -3, -2]]))
```

```
tensor([[1.0000, 3.0000, 0.0000],
        [0.1000, 0.0000, 0.0000]])
```

```
layer_1_outputs_relu = F.relu(layer_1_outputs)
print('layer_1_outputs_relu.shape: ', layer_1_outputs_relu.shape)
```

```
layer_1_outputs_relu.shape:  torch.Size([128, 32])
```

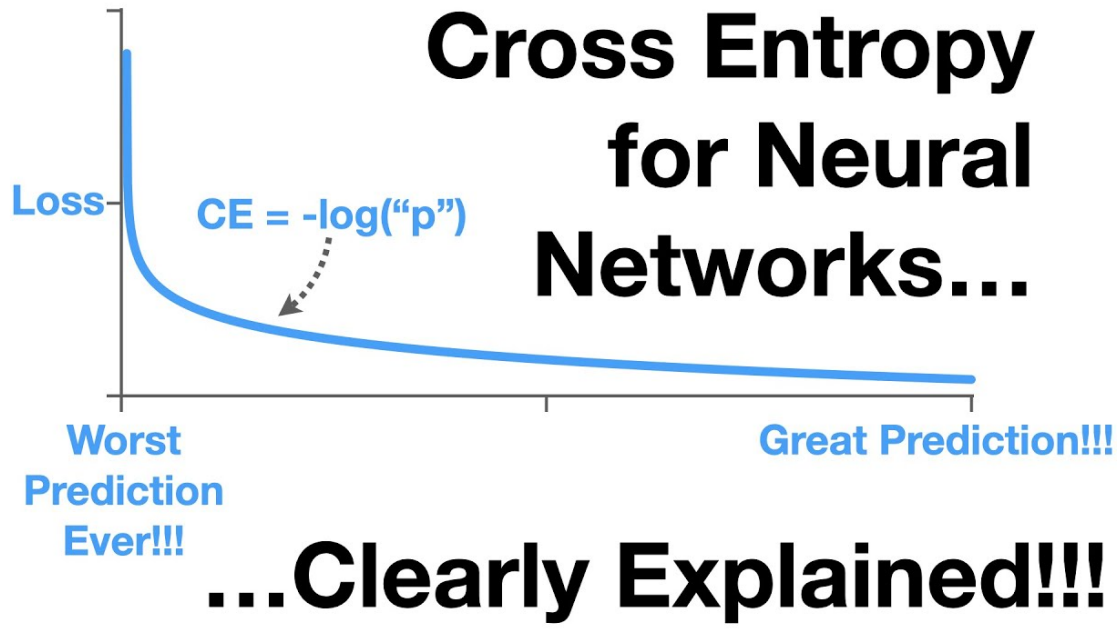
```
print("min(layer_1_outputs): ", torch.min(layer_1_outputs))
print("min(layer_1_outputs_relu): ", torch.min(layer_1_outputs_relu))
```

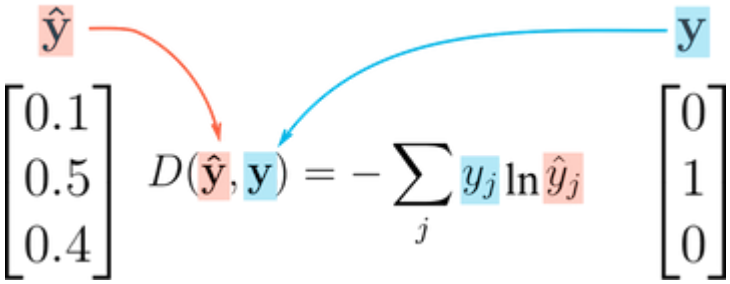
```
min(layer_1_outputs):  tensor(-0.8164, grad_fn=<MinBackward1>)
min(layer_1_outputs_relu):  tensor(0., grad_fn=<MinBackward1>)
```

```
layer_2 = nn.Linear(in_features=32, out_features=10)
layer_2_outputs = layer_2(layer_1_outputs_relu)
layer_2_outputs_relu = F.relu(layer_2_outputs)

print('layer_2_outputs_relu.shape: ', layer_2_outputs_relu.shape)
print("min(layer_2_outputs): ", torch.min(layer_2_outputs))
print("min(layer_2_outputs_relu): ", torch.min(layer_2_outputs_relu))
```

```
layer_2_outputs_relu.shape:  torch.Size([128, 10])
min(layer_2_outputs):  tensor(-0.4557, grad_fn=<MinBackward1>)
min(layer_2_outputs_relu):  tensor(0., grad_fn=<MinBackward1>)
```

Loss Function






```
loss = F.cross_entropy(layer_2_outputs_relu, labels)
loss
```

```
tensor(2.3095, grad_fn=<NllLossBackward0>)
```

```
layer_2_outputs_direct = F.relu(inputs @ layer_1.weight.t() + layer_1.bias) @ layer_2.weig
print("outputs.shape: ", layer_2_outputs_direct.shape)
```

```
outputs.shape:  torch.Size([128, 10])
```

```
# Make sure the value is close
torch.allclose(layer_2_outputs, layer_2_outputs_direct, 1e-3)
```

```
True
```

Combine Linear Function

```
layer_2_outputs = (inputs @ layer_1.weight.t() + layer_1.bias) @ layer_2.weight.t() + laye

# Create a single layer to replace the two linear layers
combined_layer = nn.Linear(in_features=input_size, out_features=10)
combined_layer.weight.data = layer_2.weight @ layer_1.weight
combined_layer.bias.data = layer_1.bias @ layer_2.weight.t() + layer_2.bias
```

```
# Same as combined_layer(inputs)
outputs_f = inputs @ combined_layer.weight.t() + combined_layer.bias

# Make sure the value is close
torch.allclose(outputs_f, layer_2_outputs, 1e-3)
```

```
True
```

```
def accuracy(outputs, labels):
  _, preds = torch.max(outputs, dim=1)
  return torch.tensor(torch.sum(preds == labels).item() / len(preds))


class MnistModel(nn.Module):
  def __init__(self, input_size, hidden_size, output_size):
    super().__init__()
    self.linear_1 = nn.Linear(in_features=input_size,
                              out_features=hidden_size)
    self.linear_2 = nn.Linear(in_features=hidden_size,
                              out_features=output_size)

  def forward(self, xb):
    xb = xb.view(xb.size(0), -1)
    outputs = self.linear_1(xb) # output_size=10
    outputs = F.relu(outputs)
    outputs = self.linear_2(outputs)
    return outputs

  def train_step(self, batch):
    images, labels = batch
    outputs = self(images) # Make Predictions, self===model
    loss = F.cross_entropy(outputs, labels)
    return loss

  def validation_step(self, batch):
    images, labels = batch
    outputs = self(images) # Make predictions, self===model
    loss = F.cross_entropy(outputs, labels)
    acc = accuracy(outputs, labels)
    return {'val_loss': loss,
            'val_acc': acc}

  def validation_epoch_end(self, outputs):
    batch_losses = [x['val_loss'] for x in outputs]
    epoch_loss = torch.stack(batch_losses).mean() # combine losses
    batch_accs = [x['val_acc'] for x in outputs]
    epoch_acc = torch.stack(batch_accs).mean() # combine accuracies
    return {'val_loss': epoch_loss.item(),
            'val_acc': epoch_acc.item()}

  def epoch_end(self, epoch, result):
    print(f"Epoch [{epoch}], val_loss: {result['val_loss']:.4f}, val_acc: {result['val_acc
```

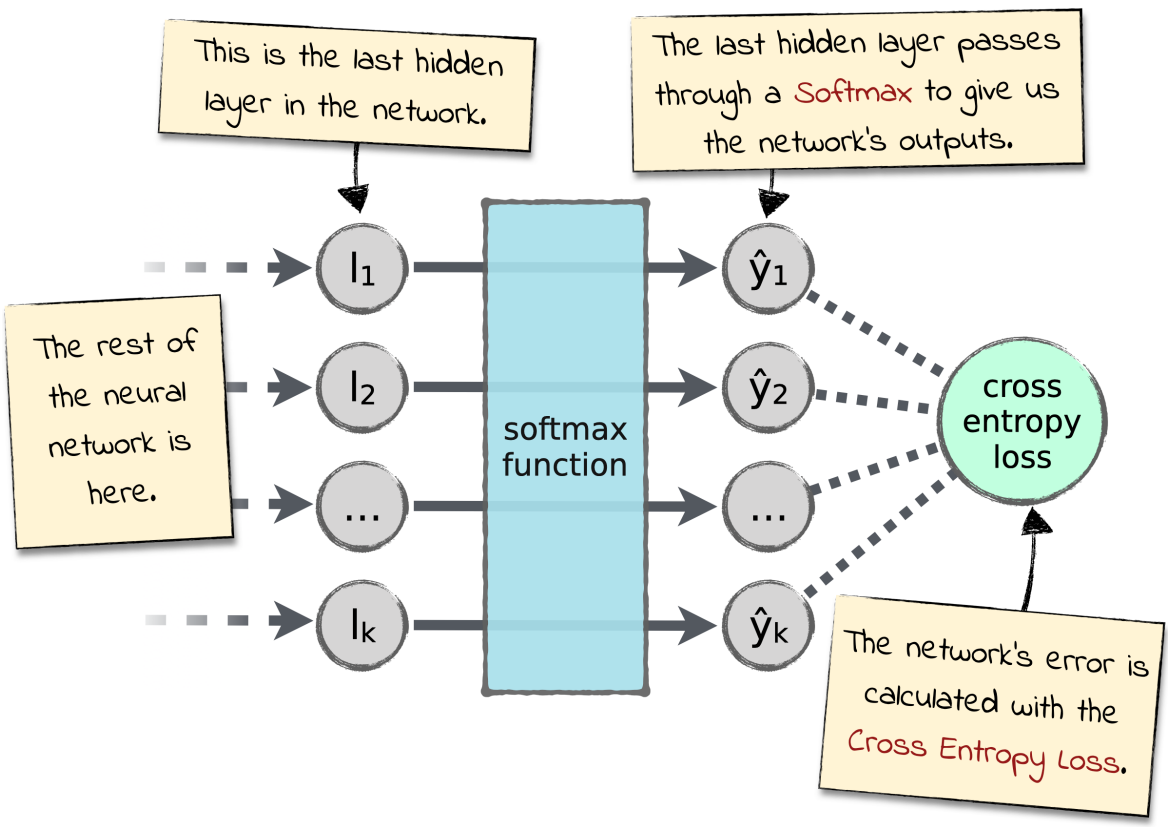
```
model = MnistModel(input_size=784, hidden_size=32, output_size=10)


for parameter in model.parameters():
  print(parameter.shape)
```

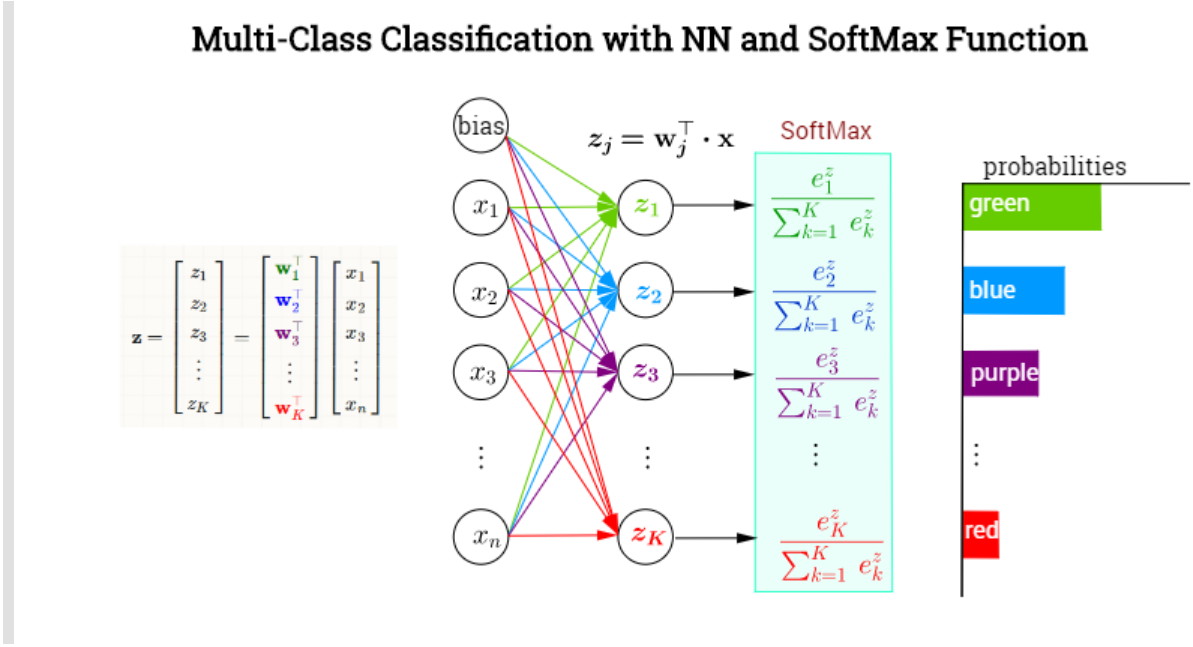
```
    torch.Size([32, 784])
    torch.Size([32])
    torch.Size([10, 32])
    torch.Size([10])
```

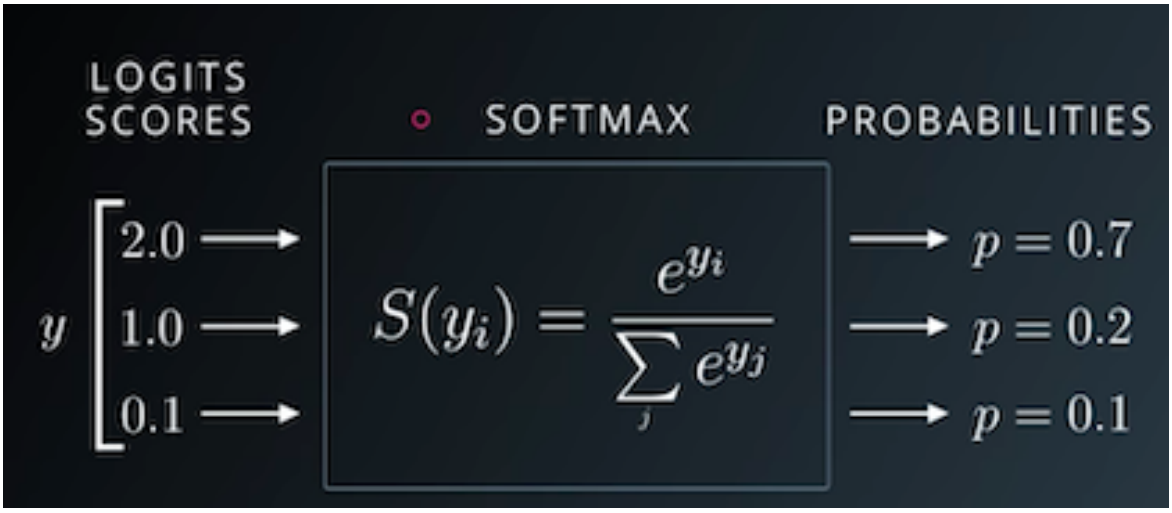
```
for images, labels in train_loader:
  outputs = model(images)
  break

print("Loss: ", F.cross_entropy(outputs, labels))
print("outputs.shape: ", outputs.shape)
print("Sample outputs: \n", outputs[:2].data)
```

```
    Loss:  tensor(2.3197, grad_fn=<NllLossBackward0>)
    outputs.shape:  torch.Size([128, 10])
    Sample outputs:
     tensor([[-0.2810,  0.0544, -0.1321,  0.1336,  0.1861,  0.1692,  0.0824,  0.1779,
              0.1448, -0.2456],
            [-0.3211, -0.0034, -0.2279,  0.1778,  0.2079,  0.1378, -0.0623,  0.1950,
              0.0739, -0.1150]])
```

Neural Network with Softmax








```
# Apply softmax for each output
probs = F.softmax(outputs.data, dim=1)
probs[:2]
```

```
tensor([[0.0723, 0.1012, 0.0840, 0.1095, 0.1154, 0.1135, 0.1040, 0.1145, 0.1107,
         0.0749],
        [0.0710, 0.0976, 0.0779, 0.1169, 0.1205, 0.1124, 0.0920, 0.1190, 0.1054,
         0.0873]])
```

```
print(probs.shape)
```

```
torch.Size([128, 10])
```

```
torch.max(probs, dim=1)
```

```
torch.return_types.max(
values=tensor([0.1154, 0.1205, 0.1165, 0.1181, 0.1286, 0.1130, 0.1178, 0.1214,
0.1143,
        0.1276, 0.1192, 0.1223, 0.1204, 0.1210, 0.1234, 0.1190, 0.1177, 0.1197,
        0.1182, 0.1214, 0.1206, 0.1281, 0.1302, 0.1159, 0.1185, 0.1181, 0.1264,
        0.1177, 0.1170, 0.1143, 0.1203, 0.1220, 0.1238, 0.1243, 0.1163, 0.1249,
        0.1155, 0.1186, 0.1145, 0.1168, 0.1226, 0.1181, 0.1254, 0.1206, 0.1174,
```

```
               0.1212, 0.1199, 0.1183, 0.1197, 0.1168, 0.1171, 0.1195, 0.1194, 0.1228,
               0.1184, 0.1182, 0.1316, 0.1292, 0.1163, 0.1195, 0.1251, 0.1179, 0.1221,
               0.1180, 0.1201, 0.1204, 0.1257, 0.1177, 0.1187, 0.1323, 0.1173, 0.1205,
               0.1250, 0.1216, 0.1177, 0.1213, 0.1226, 0.1177, 0.1220, 0.1206, 0.1179,
               0.1197, 0.1204, 0.1192, 0.1209, 0.1178, 0.1166, 0.1275, 0.1269, 0.1180,
               0.1172, 0.1204, 0.1162, 0.1218, 0.1276, 0.1213, 0.1205, 0.1231, 0.1170,
               0.1170, 0.1233, 0.1181, 0.1193, 0.1200, 0.1224, 0.1203, 0.1186, 0.1208,
               0.1216, 0.1196, 0.1160, 0.1122, 0.1215, 0.1223, 0.1201, 0.1197, 0.1169,
               0.1143, 0.1133, 0.1181, 0.1208, 0.1274, 0.1224, 0.1284, 0.1290, 0.1155,
               0.1177, 0.1183]),
     indices=tensor([4, 4, 3, 4, 5, 5, 4, 4, 5, 5, 4, 7, 3, 7, 7, 5, 4, 4, 8, 4, 8, 7, 5,
     5,
               4, 7, 5, 7, 3, 5, 8, 5, 7, 5, 3, 7, 3, 8, 5, 7, 3, 8, 7, 5, 7, 4, 4, 5,
               7, 7, 5, 4, 5, 7, 5, 7, 5, 7, 4, 7, 7, 8, 4, 4, 4, 7, 7, 7, 8, 7, 5, 7,
               4, 7, 7, 5, 3, 5, 7, 7, 8, 8, 7, 4, 7, 4, 7, 7, 8, 8, 7, 5, 7, 7, 5, 7,
               5, 4, 4, 4, 5, 4, 5, 5, 7, 5, 7, 4, 7, 4, 7, 4, 7, 7, 7, 8, 8, 7, 8, 4,
               8, 5, 7, 4, 7, 5, 5, 7]))
```

```
probs_max, preds = torch.max(probs, dim=1)
print(probs_max)
print(preds)
```

```
     tensor([0.1154, 0.1205, 0.1165, 0.1181, 0.1286, 0.1130, 0.1178, 0.1214, 0.1143,
               0.1276, 0.1192, 0.1223, 0.1204, 0.1210, 0.1234, 0.1190, 0.1177, 0.1197,
               0.1182, 0.1214, 0.1206, 0.1281, 0.1302, 0.1159, 0.1185, 0.1181, 0.1264,
               0.1177, 0.1170, 0.1143, 0.1203, 0.1220, 0.1238, 0.1243, 0.1163, 0.1249,
               0.1155, 0.1186, 0.1145, 0.1168, 0.1226, 0.1181, 0.1254, 0.1206, 0.1174,
               0.1212, 0.1199, 0.1183, 0.1197, 0.1168, 0.1171, 0.1195, 0.1194, 0.1228,
               0.1184, 0.1182, 0.1316, 0.1292, 0.1163, 0.1195, 0.1251, 0.1179, 0.1221,
               0.1180, 0.1201, 0.1204, 0.1257, 0.1177, 0.1187, 0.1323, 0.1173, 0.1205,
               0.1250, 0.1216, 0.1177, 0.1213, 0.1226, 0.1177, 0.1220, 0.1206, 0.1179,
               0.1197, 0.1204, 0.1192, 0.1209, 0.1178, 0.1166, 0.1275, 0.1269, 0.1180,
               0.1172, 0.1204, 0.1162, 0.1218, 0.1276, 0.1213, 0.1205, 0.1231, 0.1170,
               0.1170, 0.1233, 0.1181, 0.1193, 0.1200, 0.1224, 0.1203, 0.1186, 0.1208,
               0.1216, 0.1196, 0.1160, 0.1122, 0.1215, 0.1223, 0.1201, 0.1197, 0.1169,
               0.1143, 0.1133, 0.1181, 0.1208, 0.1274, 0.1224, 0.1284, 0.1290, 0.1155,
               0.1177, 0.1183])
     tensor([4, 4, 3, 4, 5, 5, 4, 4, 5, 5, 4, 7, 3, 7, 7, 5, 4, 4, 8, 4, 8, 7, 5, 5,
               4, 7, 5, 7, 3, 5, 8, 5, 7, 5, 3, 7, 3, 8, 5, 7, 3, 8, 7, 5, 7, 4, 4, 5,
               7, 7, 5, 4, 5, 7, 5, 7, 5, 7, 4, 7, 7, 8, 4, 4, 4, 7, 7, 7, 8, 7, 5, 7,
               4, 7, 7, 5, 3, 5, 7, 7, 8, 8, 7, 4, 7, 4, 7, 7, 8, 8, 7, 5, 7, 7, 5, 7,
               5, 4, 4, 4, 5, 4, 5, 5, 7, 5, 7, 4, 7, 4, 7, 4, 7, 7, 7, 8, 8, 7, 8, 4,
               8, 5, 7, 4, 7, 5, 5, 7])
```

Run on GPU

```
torch.cuda.is_available()
```

```
     True
```

```
!nvidia.smi
```

```
     /bin/bash: nvidia.smi: command not found
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"
device

'''
def get_default_device():
  if torch.cuda.is_available():
    return torch.device('cuda')
  else:
    return torch.device("cpu")

device = get_default_device()
device
'''
```

```
'\ndef get_default_device():\n  if torch.cuda.is_available():\n    return torch.devi
ce(\'cuda\')\n  else:\n    return torch.device("cpu")\n\ndevice = get_default_device
()\ndevice\n'
```

```
# Define a function that can move data and model to a chosen device
def to_device(data, device):
  if isinstance(data, (list, tuple)):
    return [to_device(x, device) for x in data]
  return data.to(device, non_blocking=True)


for images, labels in train_loader:
  print(images.shape)
  print(images.device)
  images=to_device(images, device)
  print(images.device)
  break
```

```
torch.Size([128, 1, 28, 28])
cpu
cuda:0
```

```
class DeviceDataLoader():
  def __init__(self, dl, device):
    self.dl = dl
    self.device = device

  def __iter__(self):
    for b in self.dl:
      yield to_device(b, self.device)

  def __len__(self):
    return len(self.dl)


train_loader = DeviceDataLoader(train_loader, device)
val_loader = DeviceDataLoader(val_loader, device)

print(train_loader)
print(val_loader)
```

```
<__main__.DeviceDataLoader object at 0x7faf40e28c40>
<__main__.DeviceDataLoader object at 0x7faf40e2a200>
```

```
print(dir(train_loader))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__
```

```
for xb, yb in val_loader:
  print('xb.device: ', xb.device)
  print("yb: ", yb.shape)
  break
```

```
xb.device:  cuda:0
yb:  torch.Size([256])
```

Phase 3: Training and Evalating Model

```
def evaluate(model, val_loader):
  outputs = [model.validation_step(batch) for batch in val_loader]
  return model.validation_epoch_end(outputs)
```

```
def fit(epochs,
        lr,
        model,
        trian_loader,
        val_loader,
        opt_func=torch.optim.SGD):

  optimizer = opt_func(model.parameters(), lr)

  history = [] # for recording epoch-wise results

  for epoch in range(epochs):
    # Training phase
    for batch in train_loader:
      loss = model.train_step(batch)
      loss.backward()
      optimizer.step()
      optimizer.zero_grad()

    # Validation phase
    result  = evaluate(model, val_loader)
    model.epoch_end(epoch, result)
    history.append(result)

  return history
```

```
model_GPU = MnistModel(input_size,
```

```
                        hidden_size=32,
                        output_size=10)

to_device(model_GPU, device)
```

```
MnistModel(
  (linear_1): Linear(in_features=784, out_features=32, bias=True)
  (linear_2): Linear(in_features=32, out_features=10, bias=True)
)
```

```
history_GPU = [evaluate(model_GPU, val_loader)]
history_GPU
```

```
[{'val_loss': 2.2997612953186035, 'val_acc': 0.0810546875}]
```

```
device
```

```
'cuda'
```

```
history_GPU += fit(5, 0.5, model_GPU, train_loader, val_loader)
```

```
Epoch [0], val_loss: 0.2295, val_acc: 0.9302
Epoch [1], val_loss: 0.1778, val_acc: 0.9472
Epoch [2], val_loss: 0.1410, val_acc: 0.9591
Epoch [3], val_loss: 0.1361, val_acc: 0.9606
Epoch [4], val_loss: 0.1454, val_acc: 0.9551
```

```
history_GPU += fit(5, 0.1, model_GPU, train_loader, val_loader)
```

```
Epoch [0], val_loss: 0.1131, val_acc: 0.9671
Epoch [1], val_loss: 0.1110, val_acc: 0.9676
Epoch [2], val_loss: 0.1117, val_acc: 0.9678
Epoch [3], val_loss: 0.1105, val_acc: 0.9681
Epoch [4], val_loss: 0.1091, val_acc: 0.9682
```

Phase 4: Testing Model

```
test_dataset = MNIST(root='data/',
                     train=False,
                     transform=ToTensor())
```

```
image.shape
```

```
torch.Size([128, 1, 28, 28])
```

```
yb = model(image[0].unsqueeze(0))
yb
```

```
tensor([[-0.2273,  0.0470, -0.0745,  0.1854,  0.1730,  0.2614,  0.0241,  0.1614,
          0.1093, -0.1715]], grad_fn=<AddmmBackward0>)
```

```
torch.max(yb)
```

```
tensor(0.2614, grad_fn=<MaxBackward1>)
```

```
torch.max(yb, dim=1)
```

```
torch.return_types.max(
values=tensor([0.2614], grad_fn=<MaxBackward0>),
indices=tensor([5]))
```

```
_, preds = torch.max(yb, dim=1)
print(preds)
```

```
tensor([5])
```

```
preds[0].item()
```

```
5
```

```
test_dataset.targets
```

```
tensor([7, 2, 1,  ..., 4, 5, 6])
```

```
def predict_image(image, model):
  print(image)
  xb = to_device(image.unsqueeze(0), device)
  yb = model(xb)
  _, preds = torch.max(yb, dim=1)
  return preds[0].item()
```

```
image, label = test_dataset[0]
print("Lable: ", label, "Prediction: ", predict_image(image[0], model_GPU))
plt.imshow(image[0], cmap='gray')
```

```
tensor([[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.3294, 0.7255, 0.6235,
          0.5922, 0.2353, 0.1412, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.8706, 0.9961, 0.9961,
          0.9961, 0.9961, 0.9451, 0.7765, 0.7765, 0.7765, 0.7765, 0.7765, 0.7765,
          0.7765, 0.7765, 0.6667, 0.2039, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.2627, 0.4471, 0.2824,
          0.4471, 0.6392, 0.8902, 0.9961, 0.8824, 0.9961, 0.9961, 0.9961, 0.9804,
          0.8980, 0.9961, 0.9961, 0.5490, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0667, 0.2588, 0.0549, 0.2627, 0.2627, 0.2627, 0.2314,
          0.0824, 0.9255, 0.9961, 0.4157, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.3255, 0.9922, 0.8196, 0.0706, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0863,
          0.9137, 1.0000, 0.3255, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.5059,
          0.9961, 0.9333, 0.1725, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.2314, 0.9765,
          0.9961, 0.2431, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
```

```
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000
```

```
image, label = test_dataset[193]
print("Lable: ", label, "Prediction: ", predict_image(image[0], model_GPU))
plt.imshow(image[0], cmap='gray')
```

```
tensor([[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0667, 0.4196,
          0.8902, 0.9961, 0.9961, 0.9961, 0.6824, 0.3137, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0353, 0.7490, 0.9922,
          0.9922, 0.7412, 0.5608, 0.4118, 0.7255, 0.9529, 0.3137, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.4353, 0.9922, 0.8627,
          0.6078, 0.0196, 0.0000, 0.0000, 0.0078, 0.6549, 0.9725, 0.3176, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.8588, 0.9922, 0.1490,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0353, 0.6745, 0.8510, 0.0863,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.8588, 0.9922, 0.3647,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1647, 0.8980, 0.7725,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.2980, 0.9569, 0.8980,
          0.1882, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.3137, 0.9059,
```

✓ 0s completed at 11:42 PM