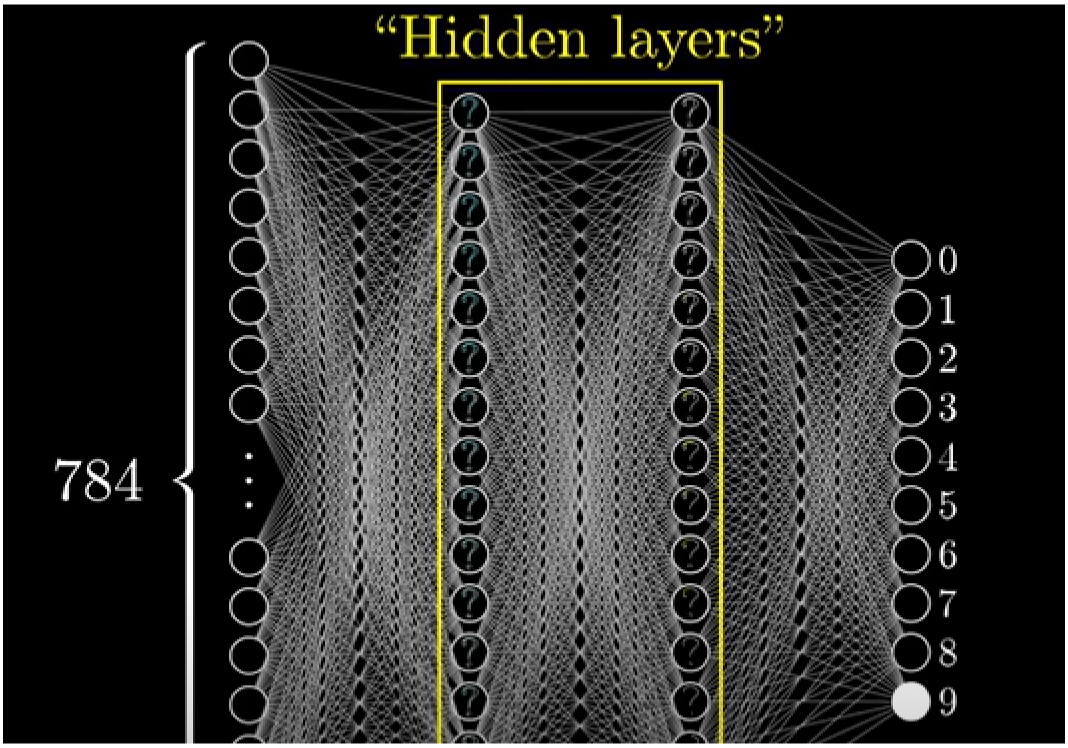
SOC ‘25 LLM

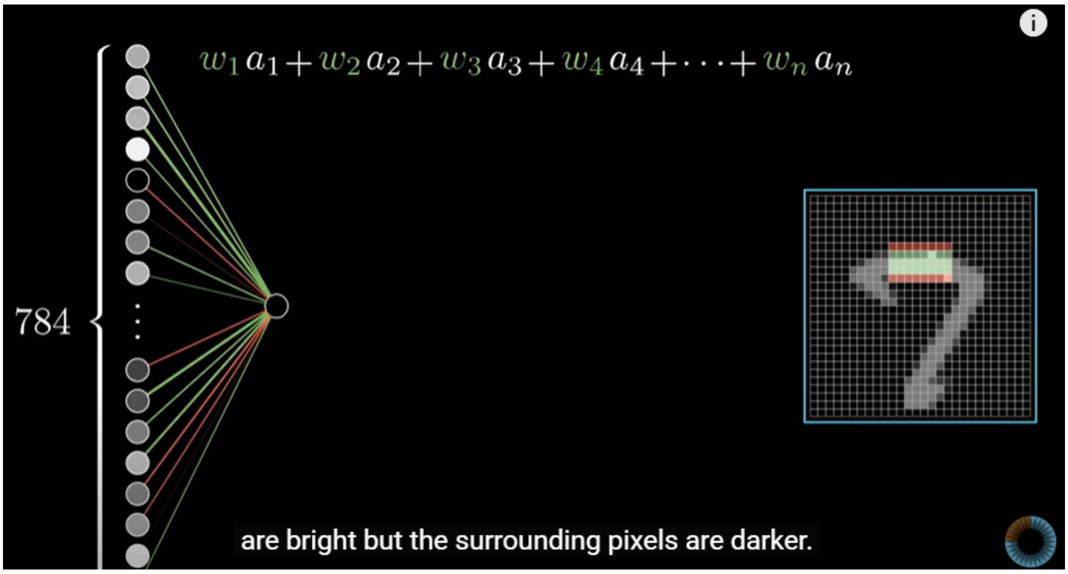
▼ Neural Network

The neural network used in the recognition of handwritten digits is the most basic one i.e., a multilayer perceptron.



What is a Neural Network?

Neuron —> Acts like a function

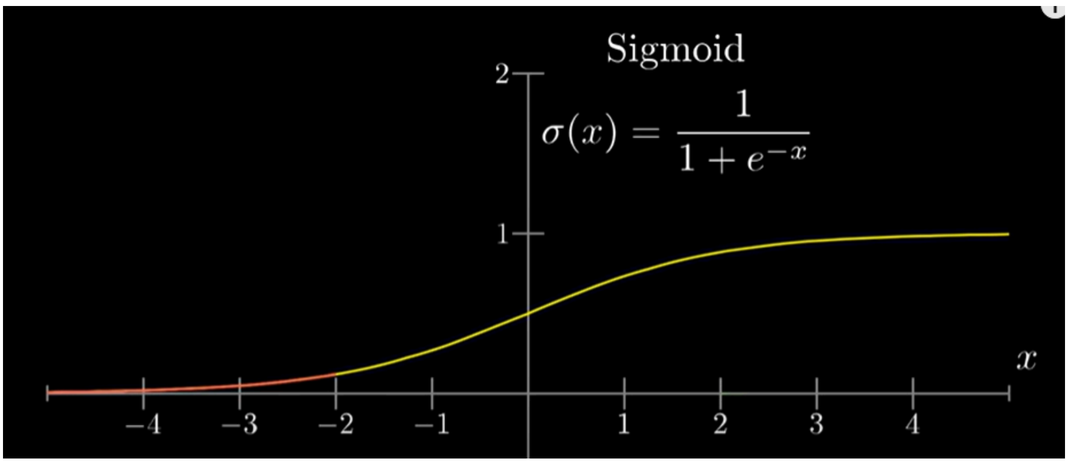


Here w are weights assigned to particular neuron in determining the activation of the neuron of the next layer and a are the activation of the neurons in that layer.

The sum would be maximum if the pixels in the green(+ve) area are brighter and the pixels in the red(-ve) area are darker.

Now, we wanted the wieghted sum to lie between 0 and 1.

SIgmoid function:



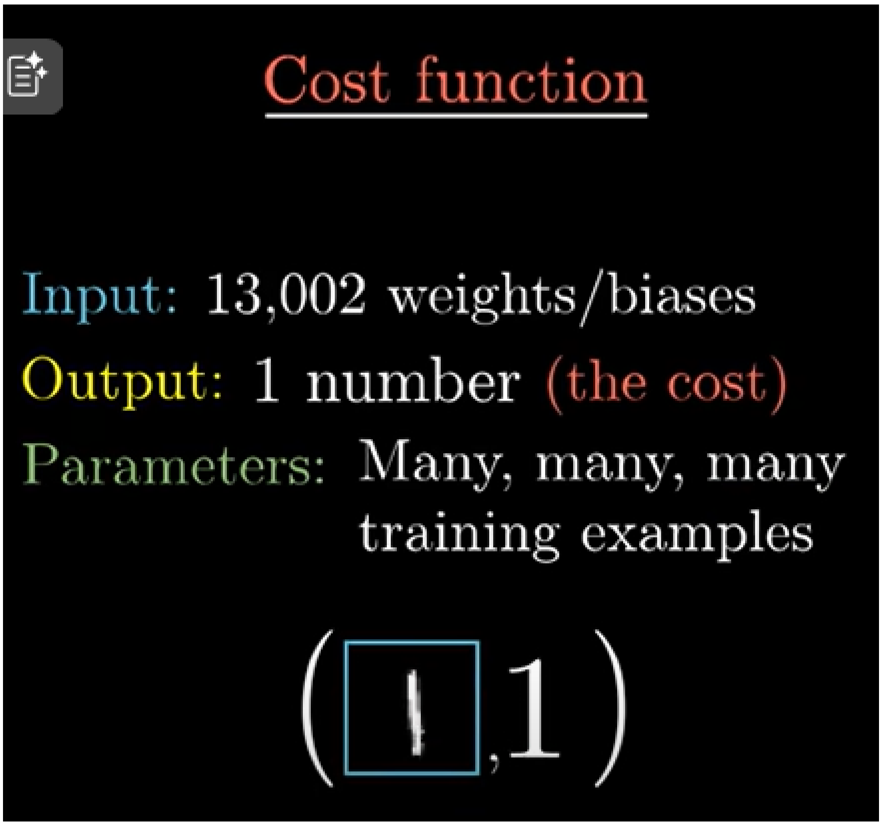
Learning refers to finding the right weights and biases.

$$a_0^{(1)} = \overset{\text{Sigmoid}}{\sigma}\left(w_{0,0}\, a_0^{(0)} + w_{0,1}\, a_1^{(0)} + \cdots + w_{0,n}\, a_n^{(0)} + \underset{\text{Bias}}{b_0}\right)$$

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}\mathbf{a}^{(0)} + \mathbf{b}\right)$$

ReLU –> Rectified linear unit –> ReLU(a) = max(0,a)



We want to minimise the cost function output so as to find the best combination of weights and biases. But, it is hard to find the global minimum of a function instead of the local minimum.

Backpropagation is the algorithm, how to change the weights and biases to make the most rapiod decrease to the cost.

We use stochastic gradient decent in which we randomly shuffle the data points and then we divide them into mini-batches after which we perform gradient descent using backpropagation separately on the batches.

▼ PyTorch

It is a machine learning and deep learning framework.

In PyTorch, everything is based on tensor operations.

▼ Basics

```
import torch
x = torch.zeros(2,3)
```

x = torch.ones(2)

x = torch.rand()

—> Generates a random number between{0, 1)

x = torch.randn() —> generates random numbers with mean 0 and std dev of 1

x.dtype —> returns the datatype of the variable

BY default, the tensor datatype is float32

x = torch.ones(2,2, dtype=torch.________)

x.size() —> returns the size of the tensor

x = torch.tensor([___, ____])

z = x + y OR **z = torch.add(x, y)**—> Element wise addition

y.add_(x) —> does inplace operation on y

Every function that has a trailing underscore does an inplace operation.

z = x - y OR **z = torch.sub(x ,y)** OR **y.sub_(x)**—> Element wise subtraction

z = x * y OR **z = torch.mul(x, y)** OR **y.mul_(x)**

z = x / y OR **z = torch.div(x ,y)**

Slicing can also be done similarly to the numpy array operation

x[:, 0] —> All the rows in the first column

x[a ,b] —> prints the single element in the tensor format

x[a ,b].item() —> gives the values of the element (only usable when there is only one value)

Reshaping :

`y = x.view (__)` —> the number of element must be the same as in the tensor earlier

`y = x.view(-1, ___)` —> It automatically adjust the dimension value for -1

Converting from numpy to tensor and vice-versa:

```
a = torch.ones(5)
b = a.numpy()
```

If the tensor is on CPU instead of GPU then both the object will share the same memory location and if it is on GPU it can't be converted

```
a = numpy.ones(5)
```

`b = torch.from_numpy(a)` —> By default it is float 64

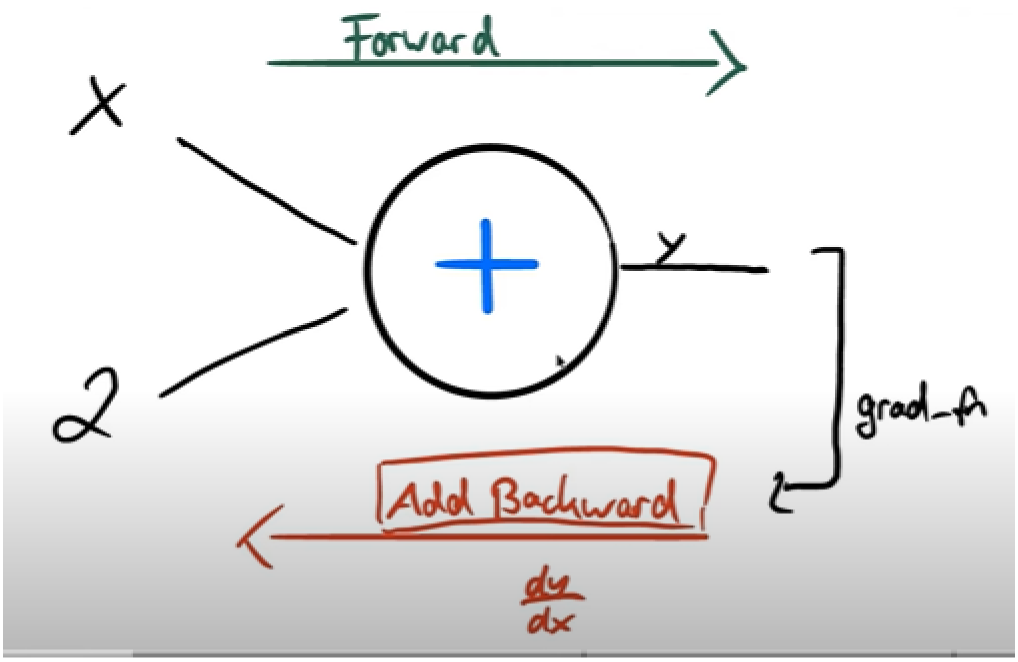
Same memory location factor here also

`x = torch.ones(5, requires_grad=True)` — > this is needed if in future we want to calculate the gradient for optimization, BY default it is false

▼ Calculating gradients:

`x = torch.ones(5, requires_grad=True)`

whenever we do operation on this tensor, it will create computational graph



In the forward pass, it will calculate the output and since we said that we need gradient, it will automatically create and store a function. This function is used in the backpropagation and to get the gradients.

The attribute grad_fn points to the function.

`z.backward()` —> dz/dx

`x.grad` —> the gradients are stored

The above works only for scalar single value

```
v = torch.tensor([0.1, 1.0, 0.001], dtype = torch.float32)
```

```
z.backward(v)
```

```
x.grad
```

In the background, it will create a vector Jacobian product to calculate gradients

▼ Preventing autograd from tracking histroy:

```
x.requires_grad_(False)
```

```
 y = x.detach()
```

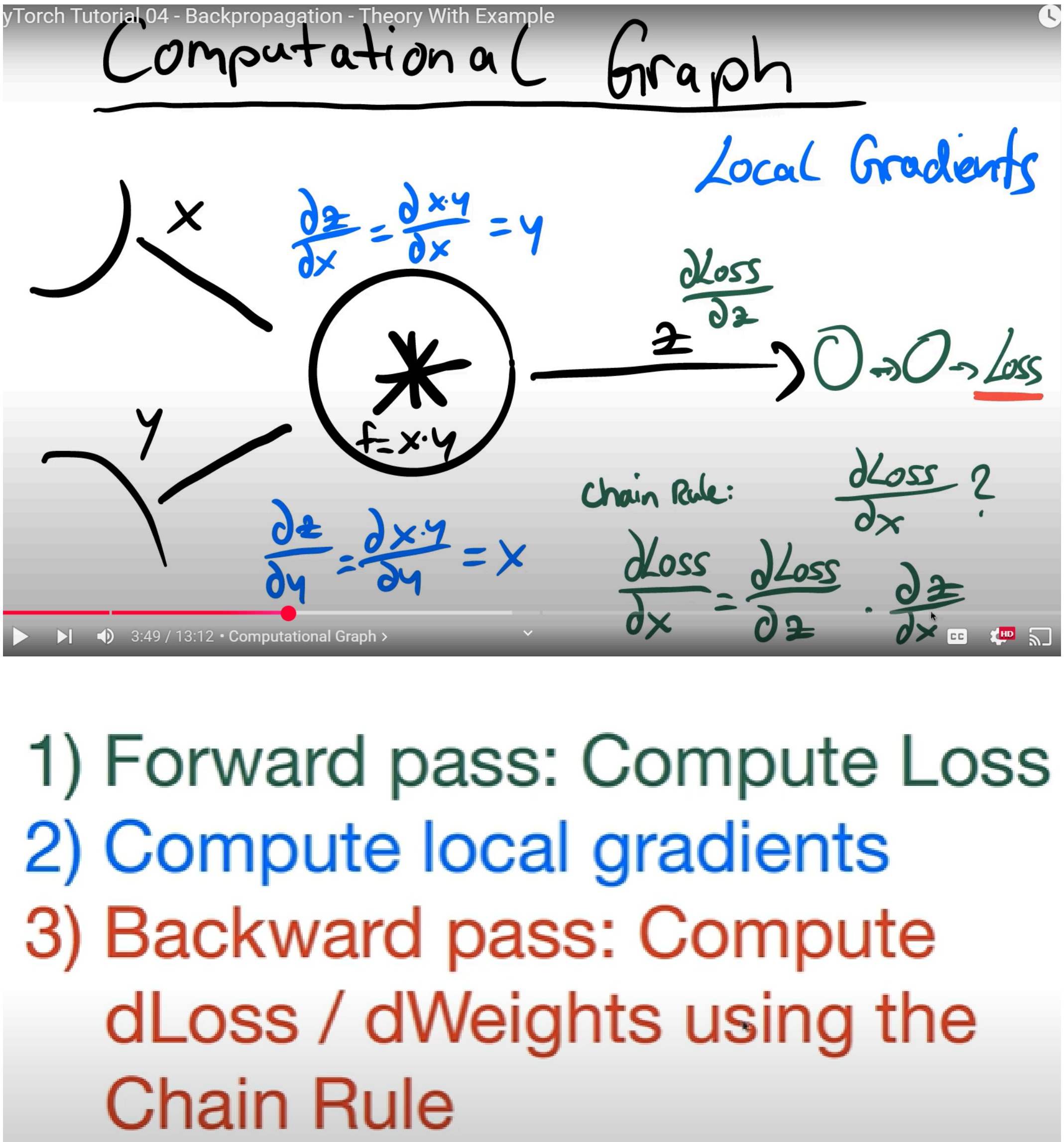
`with torch.no_grad():`

`y = x * 2` —> This does not have the gradient function

SO, we use `_____.grad.zero_()` before running next iteration

▼ FOR Optimization:

```
optimizer = torch.optim.SGD(weights, lr=0.01)
optimizer . step()
optimizer . zero_grad()
```



▼ Backpropagation: (Example)

```
import torch
x = torch.tensor(1.0)
y = torch.tensor(2.0)
w = torch.tensor(1.0, requires_grad=True)

y_hat = w * x
loss = (y_hat - y)**2
loss.backward()
w.grad
```

▼ Typical pytorch pipeline:

1. Design model(input ,output, forward pass)
2. Contruct loss and optimizer
3. training loop
    a. forward pass: compute prediction and loss
    b. backward pass: gradients
    c. update weights

▼ Step 1: Prediction, gradients computation, loss computation, parameters updates all manually

```
import numpy as np
% f = w * x
% f = 2 * x
X = np.array([1, 2, 3, 4], dtype=np.float32)
Y = np.array([2, 4, 6, 8], dtype=np.float32)
```

```
w = 0.0
% model prediction
def forward(x) :
  return w * x
% loss = MSE
def loss(y, y_predicted) :
  return((y_predicted-y)**2).mean( )
% gradient
% MSE = 1/N *  (w*x - y)**2
% dJ/dw = 1/N * 2 * (w*x - y)
def gradient(x, y, y_predicted)
  return np.dot(2*x, y_predicted-y).mean()
print(f'Prediction before training : f(5) = {forward(5):.3f}')
%Training
learning_rate = 0.01
n_iters = 10
for epoch in range(n_iters) :
  y_pred = forward(X)
  l = loss(Y, y_pred)
  dw = gradient(X, Y, y_pred)
  w -= learning_rate * dw
  if epoch % 1 == 0:
    print(f'epoch {epoch+1}: w = {w:.3f}, loss = {l:.8f}')
  print(f'Prediction after training: f(5) = {forward(5):.3f}')
```

▼ Step 2 : Gradients Computation through autograd, rest same

```
import torch
% f = w * x
% f = 2 * x
X = torch.tensor([1, 2, 3, 4], dtype=torch.float32)
Y =torch.tensor([2, 4, 6, 8], dtype=torch.float32)
w = torch.tensor(0.0, dtype=torch.float32, requires_grad
= True)
% model prediction
def forward(x) :
   return w * x
% loss = MSE
def loss(y, y_predicted) :
   return((y_predicted-y)**2).mean()
% gradient
% MSE = 1/N *  (w*x - y)**2
% dJ/dw = 1/N * 2 * (w*x - y)
def gradient(x, y, y_predicted)
   return np.dot(2*x, y_predicted-y).mean()
print(f'Prediction before training : f(5) =
{forward(5):.3f}')
%Training
learning_rate = 0.01
n_iters = 10
for epoch in range(n_iters) :
   y_pred = forward(X)
   l = loss(Y, y_pred)
   l.backward()
   with torch.no_grad():
```

```
        w -= learning_rate * w.grad
    w.grad.zero_()
    if epoch % 1 == 0:
        print(f'epoch {epoch+1}: w = {w:.3f}, loss =
        {l:.8f}')
    print(f'Prediction after training: f(5) =
    {forward(5):.3f}')
```

▼ Step 3: Prediction: manually, Gradients Computation: Autograd, Loss Computation: pytorch Loss, Parameter Updates: pytorch optimizer

```
import torch
import torch.nn as nn
% f = w * x
% f = 2 * x
X = torch.tensor([1, 2, 3, 4], dtype=torch.float32)
Y =torch.tensor([2, 4, 6, 8], dtype=torch.float32)
w = torch.tensor(0.0, dtype=torch.float32, requires_grad
= True)
% model prediction
def forward(x) :
    return w * x
% loss = MSE
def loss(y, y_predicted) :
    return((y_predicted-y)**2).mean( )
print(f'Prediction before training : f(5) =
{forward(5):.3f}')
%Training
```

```
learning_rate = 0.01
n_iters = 10
loss = nn.MSEloss()
optimizer = torch.optim.SGD([w], lr=learning_rate)
for epoch in range(n_iters) :
    y_pred = forward(X)
    l = loss(Y, y_pred)
    l.backward()
    with torch.no_grad():
        w -= learning_rate * w.grad
    optimizer.step()
    w.grad.zero_()
    optimizer.zero_grad()
    if epoch % 1 == 0:
        print(f'epoch {epoch+1}: w = {w:.3f}, loss = {l:.8f}')
    print(f'Prediction after training: f(5) = {forward(5):.3f}')
```

▼ Step = 4: Prediction: pytorch model, Gradients Computation: Autograd, Loss Computation: pytorch Loss, Parameter Updates: pytorch optimizer

```
import torch
import torch.nn as nn
% f = w * x
% f = 2 * x
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
w = torch.tensor(0.0, dtype=torch.float32, requires_grad
= True)
% model prediction
def forward(x) :
  return w * x
input_size = n_features
output_size = n_features
model = nn.Linear(input_size, output_size)
print(f'Prediction before training : f(5) =
{model(X_test).item( ):.3f}')
%Training
learning_rate = 0.01
n_iters = 10
loss = nn.MSEloss()
optimizer = torch.optim.SGD(model.parameters( ) ,
lr=learning_rate)
for epoch in range(n_iters) :
  y_pred = model(X)
  l = loss(Y, y_pred)
  l.backward()
  optimizer.step()
  optimizer.zero_grad()
  if epoch % 1 == 0:
    [w, b] = model.parameters()
    print(f'epoch {epoch+1}: w = {w[0][0].item( ):.3f},
    loss = {l:.8f}')
  print(f'Prediction after training: f(5) =
  {model(X_test).item():.3f}')
```

▼ Linear Regression:

```
import torch
import torch.nn as nn
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt

% prepare data
X_numpy, y_numpy = datasets.make_regression(n_samples =
100, n_features = 1, noise = 20, random_state = 1)
X = torch.from_numpy(X_numpy.astype(np.float32))
y = torch.from_numpy(y_numpy.astype(np.float32))
y = y.view(y.shape[0], 1)
n_samples, n_features = X.shape

% model
input_size = n_features
output_size = 1
model = nn.Linear(input_size, output_size)

% loss and optimizer
learning_rate = 0.01
criterion = nn.MSEloss()
optimizer = torch.optim.SGD(model.parameters(), lr =
learning_rate)

% Training Loop
num_epoch = 100
```

```
for epoch in range(num_epoch):
    y_predicted = model(X)
    loss = criterion(y_predicted, y)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()

    if(epoch+1) % 10 == 0:
        print(f'epoch: {epoch+1}, loss = {loss.item():.4f}')
% Plot
predicted = model(X).detach().numpy()
plt.plot(X_numpy, y_numpy, 'ro')
plt.plot(X_numpy, predicted, 'b')
plt.show()
```

▼ Logistic Regerssion:

```
import torch
import torch.nn as nn
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler —> to scale features
from sklear.model_selection import train_test_split —> separation of training, testing dataset

% prepare data
```

```
bc = datasets.load_breast_cancer()
X, y = bc.data, bc.target
n_samples, n_features = X.shape
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.2, random_state = 1234)

% scale
sc = StandarScaler() —> maen 0 and std var = 1, preferred
in case of logitsic regression
X_train = sc.fit_transform(X_train) —> used for learning
transformation paramters and apply
X_test = sc.transform(X_test) —> Apply the same
transformation, learned from training data

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))

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

% model
% f = wx + b, sigmoid at the end
class LogisticRegression(nn.Module):
  def __init__(self, n_input_features):
    super(LogisticRegression, self).__inti__()
    self.Linear = nn.Linear(n_input_features, 1)

  def forward(self, x):
```

```
        y_predicted = torch.sigmoid(self, linear(x))
        return y_predicted

model = LogisticRegression(n_features)

% loss and optimizer
learning_rate = 0.01
criterion = nn.BGELoss() —> Binary cross entropy loss
optimizer = torch.optim.SGD(model.parameters,
lr=learning_rate)

% Training Loop
num_epoch = 100
for epoch in range(num_epoch):
    y_predicted = model(X)
    loss = criterion(y_predicted, y_train)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()

    if(epoch+1) % 10 == 0:
        print(f'epoch: {epoch+1}, loss = {loss.item():.4f}')

with torch.no_grad():
    y_predicted = model(X_test)
    y_predicted_cls = y_predicted.round()
```

```
acc = y_predicted_cls.eq(y_test).sum() / float(y_test.shape[0])

print(f'accuracy = {acc:.4f}')
```

To improve the accuracy, we need to try with the optimizer or learning rate or number of epochs

▼ Dataset and Dataloader:

It requires much time to calculate the gradients on the whole data, so instead we divide it into batches.

epoch = 1 forward and backward pass of ALL training samples

batch_size = number of training samples in one forward anf backward pass

number of iterations = number of passes, each pass using (batch_size) number of samples

```
import torch

import torchvision

from torch.utils.data import Dataset, DataLoader

import numpy as np

import math


class WineDataset(Dataset):

    def __init__(self):

        % data loading

        xy = np.loadtxt('file address', delimiter=',', dtype=np.float32, skiprows=1) —> delimiter is used if the file is comma separated and skiprows is used if the first row contains headers

        self.x = torch.from_numpy(xy[:,1:])
```

```
        self.y = torch.from_numpy(xy[:,[0]])
        self.n_samples = xy.shape[0]

    def __getitem__(self, index):
        % dataset[0] —> Allows indexing
        return self.x[index], self.y[index]

    def __len__(self):
        % len(dataset)
        return self.n_samples

dataset = WineDataset()
dataloader = DataLoader(dataset=dataset, batch_size=4,
shuffle=True, num_workers=2) —> num_workers controls the
number of subprocesses used for data loading and it
speeds up data loading

num_epochs = 2
total_samples = len(dataset)
n_iterations = math.ceil(total_samples/4) —> it just
round off number to the upper side

for epoch in range(num_epoch):
    for i, (input, labels) in enumerate(dataloader): —>
    enumerate function will return the index
    if(i+1) % 5 == 0:
        print(f'epoch {epoch+1}/{num_epochs}, step
        {i+1}/{n_iterations}, inputs {inputs.shape}')
```

▼ Dataset Transform:

```
import torch
import torchvision
from torch.utils.data import Dataset, DataLoader
import numpy as np

class WineDataset(Dataset):
  def __init__(self, transform=None):
    % data loading
    xy = np.loadtxt('file address', delimiter=',',
    dtype=np.float32, skiprows=1)
    self.x = xy[:,1:]
    self.y = xy[:,[0]]
    self.n_samples = xy.shape[0]
    self.transform = transform

  def __getitem__(self, index):
    % dataset[0] —> Allows indexing
    sample = self.x[index], self.y[index]
    if self.transform:
      sample = self.transform(sample)

    return sample

  def __len__(self):
    % len(dataset)
    return self.n_samples

% Custom transform
class ToTensor:
```

```
    def __call__(self, sample):
        inputs, targets = sample
        return torch.from_numpy(inputs),
        torch.from_numpy(targets)

% Another Custom transform
class MulTransform:
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, factor):
        inputs, target = sample
        inputs *= self.factor
        return inputs, target

dataset = WineDataset(transform=ToTensor())

composed = torchvision.transform.Compose([ToTensor(),
MulTransform(2)]) —> compose is used to chain multiple
transforms
dataset = WineDataset(transform=composed())
```

▼ Softmax and Cross-Entropy:

Softmax

$$S(y_i) = \frac{e^{y_i}}{\sum e^{y_j}}$$

Output between 0 and 1

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

def softmax(x):
   return np.exp(x) / np.sum(np.exp(x), axis=0)

% Using Numpy
x = np.array([2.0, 1.0, 0.1])
outputs = softmax(x)
print('softmax numpy:', outputs)

% Using Tensor
x = torch.tensor([2.0, 1.0, 0.1])
outputs = torch.softmax(x,dim=0)
```

Cross-Entropy

$$D(\hat{Y}, Y) = -\frac{1}{N} \cdot \sum Y_i \cdot \log(\hat{Y}_i)$$

Cross-entropy loss measures the performance of the classification model whose output is between 0 and 1.

The loss increases as the predicted probability diverges from actual label.

Y must be one hot encoded class labels.

For e.g. If we have three classes and the it belongs to the first class then Y = [1,0,0]

% Using Numpy

```
import numpy as np


def cross_entropy(actual, predicted):
    loss = -np.sum(actual * np.log(predicted))
    return loss or loss / float(predicted.shape[0])


Y = 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])
l1 = cross_entropy(Y, Y_pred_good)
l2 = cross_entropy(Y, Y_pred_bad)
```

nn.CrossEntropyLoss

Careful!

nn.CrossEntropyLoss applies
nn.LogSoftmax + nn.NLLLoss (negative log likelihood loss)

—> No Softmax in last layer!

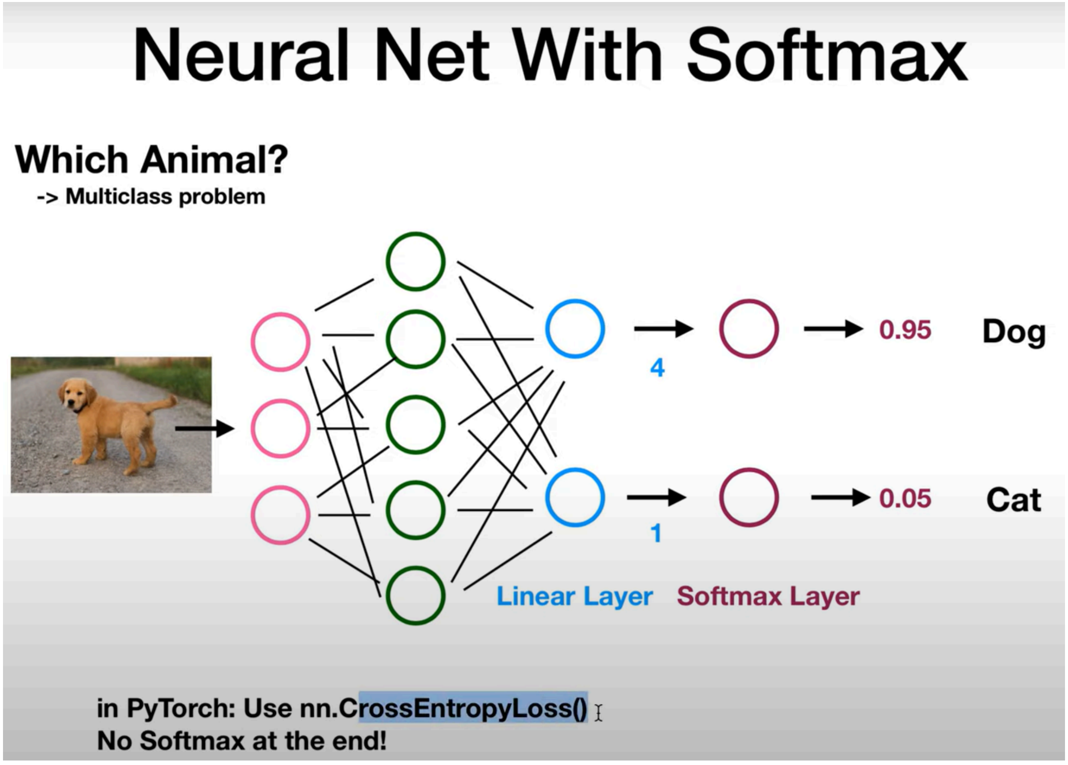
Y has class labels, not One-Hot!
Y_pred has raw scores (logits), no Softmax!

```
%Using tensor
import torch
import torch.nn as nn

loss = nn.CrossEntropyLoss()
Y = torch.tensor([0])
% nsamples x nclasses = 1x3
Y_pred_good = torch.tensor([2.0, 1.0, 0.1])
Y_pred_bad = torch.tensor([0.5, 2.0, 0.3])

l1 = loss(Y_pred_good, Y)
l2 = loss(Y_pred_bad, Y)

_, prediction1 = torch.max(Y_pred_good, 1)
_, prediction2 = torch.max(Y_pred_bad, 1)
```

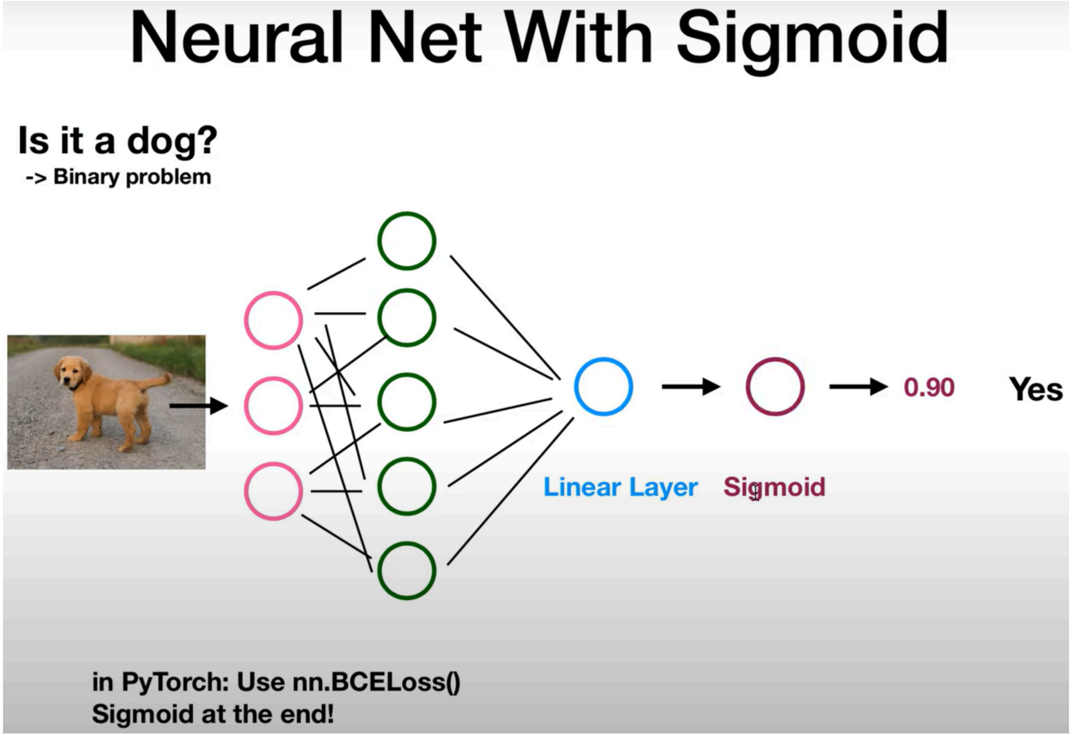



```python
import torch
import torch.nn as nn

# Multiclass problem
class NeuralNet2(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet2, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.linear1(x)
        out = self.relu(out)
        out = self.linear2(out)
        # no softmax at the end
        return out

model = NeuralNet2(input_size=28*28, hidden_size=5, num_classes=3)
criterion = nn.CrossEntropyLoss()  # (applies Softmax)
```




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

# Binary classification
class NeuralNet1(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(NeuralNet1, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out = self.linear1(x)
        out = self.relu(out)
        out = self.linear2(out)
        # sigmoid at the end
        y_pred = torch.sigmoid(out)
        return y_pred

model = NeuralNet1(input_size=28*28, hidden_size=5)
criterion = nn.BCELoss()
```

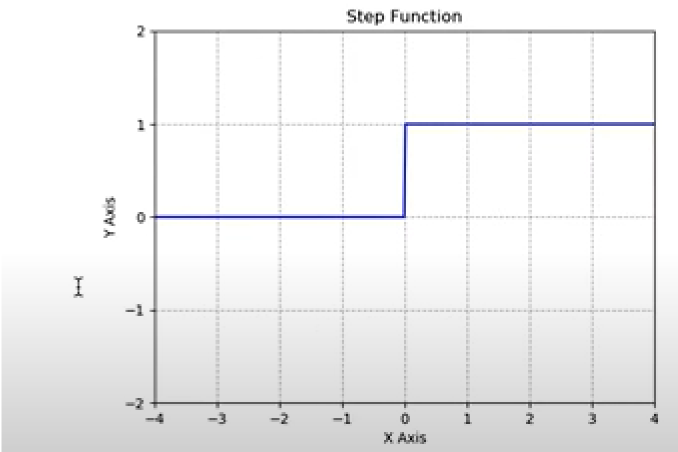
▼ Activation Functions:

Activation functions apply a non-linear transformation and decide whether a neuron should be activated or not.

1. **Step function**
2. **Sigmoid**
3. **TanH**
4. **ReLU**
5. **Leaky ReLU**
6. **Softmax**

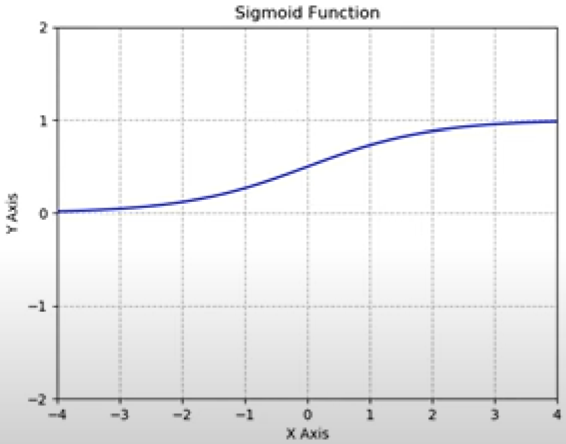
Step Function

$$f(x) = \begin{cases} 1 & \text{if } x \geq \theta \\ 0 & \text{otherwise} \end{cases}.$$



Sigmoid Function

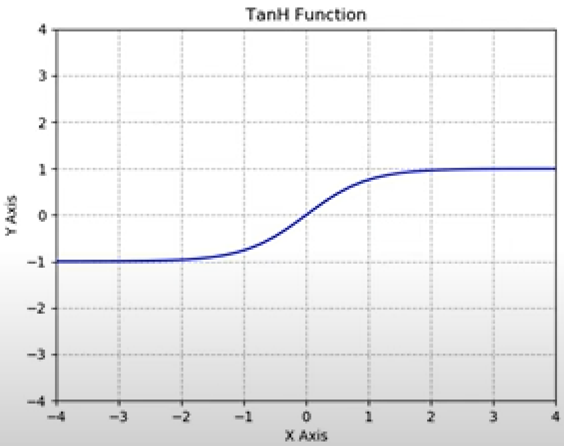
$$f(x) = \frac{1}{1 + e^{-x}}$$



—> Typically in the last layer of a binary classification problem

TanH Function

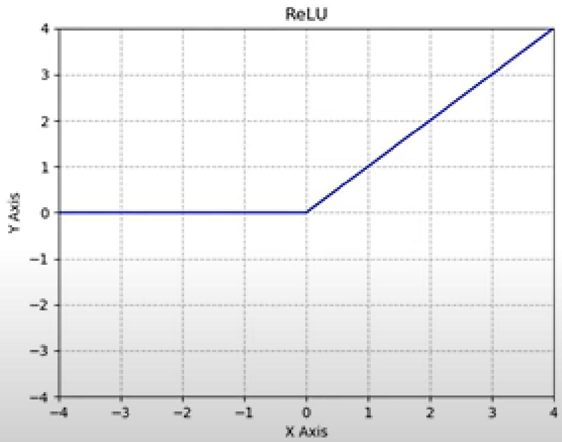
$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$



—> Hidden layers

ReLU Function

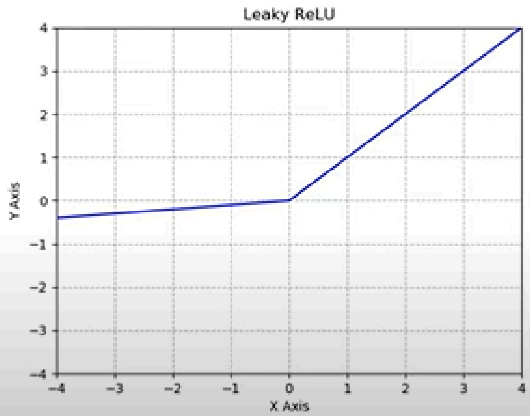
$$f(x) = max(0,\ x)$$



—> If you don't know what to use, just use a ReLU for hidden layers

Leaky ReLU Function

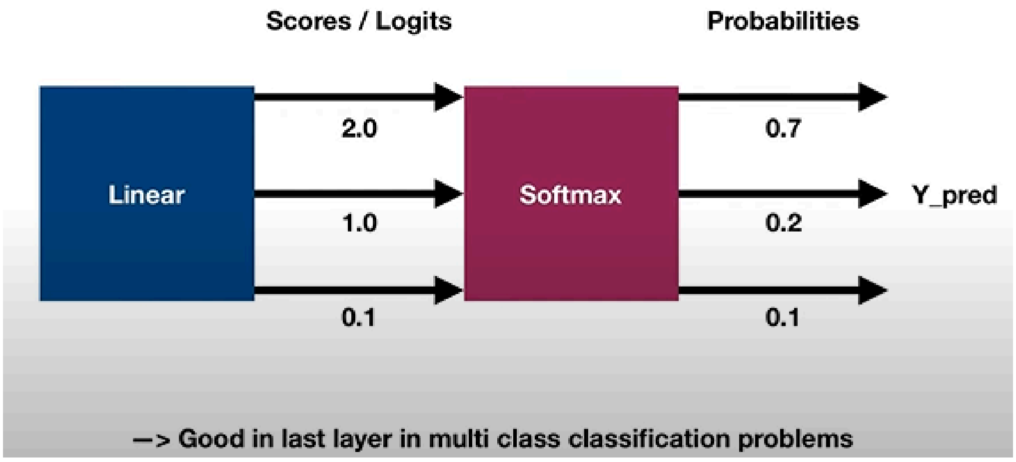
$$f(x) = \begin{cases} x & \textbf{if } x \geq 0 \\ a \cdot x & \textbf{otherwise}. \end{cases}$$



—> Improved version of ReLU. Tries to solve the vanishing gradient problem

Softmax

$$S(y_i) = \frac{e^{y_i}}{\sum e^{y_j}}$$




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

# option 1 (create nn modules)
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(NeuralNet, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(hidden_size, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out = self.linear1(x)
        out = self.relu(out)
        out = self.linear2(out)
        out = self.sigmoid(out)
        return out
```

```python
# option 2 (use activation functions directly in forward pass)
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(NeuralNet, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out = torch.relu(self.linear1(x))
        out = torch.sigmoid(self.linear2(out))
        return out
```

torch.relu === F.relu

▼ Feed Forward Neural Network(Hand written digit classification)

```python
# MNIST
# DataLoader, Transformation
# Multilayer Neural Net, activation function
# Loss and Optimizer
# Training Loop (batch training)
# Model evaluation
# GPU support
```

```python
import torch

import torch.nn as nn

import torchvision

import torchvision.transforms as transforms

import matplotlib.pyplot as plt


% Device configuration

device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')


% Hyper-parameters

input_size = 784 # 28x28
```

```
hidden_size = 500
num_classes = 10
num_epochs = 2
batch_size = 100
learning_rate = 0.001

% MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='./data',
train=True, transform=transforms.ToTensor(),
download=True)
test_dataset = torchvision.datasets.MNIST(root='./data',
train=False, transform=transforms.ToTensor())

% Data loader
train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=batch_size, shuffle=True)
test_loader =
torch.utils.data.DataLoader(dataset=test_dataset,
batch_size=batch_size, shuffle=False)
examples = iter(test_loader)
example_data, example_targets = next(examples)
for i in range(6):
    plt.subplot(2,3,i+1)
    plt.imshow(example_data[i][0], cmap='gray')
plt.show()

% Fully connected neural network with one hidden layer
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size,
    num_classes):
```

```
        super(NeuralNet, self).__init()__
        self.input_size = input_size
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out) # no activation and no softmax at
        the end
        return out

model = NeuralNet(input_size, hidden_size,
num_classes).to(device)

% Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),
lr=learning_rate)

% Train the model
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # origin shape: [100, 1, 28, 28]
        # resized: [100, 784]
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        # Forward pass
```

```
        outputs = model(images)
        loss = criterion(outputs, labels)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step
            [{i+1}/{n_total_steps}], Loss {loss.item():.4f}')

% Test the model
% In test phase, we don't need to compute gradients (for
memory efficiency)
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)

        # max returns (value ,index)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()
    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy of the network on the 10000 test
    images: {acc} %')
```

▼ LLM(Large Language Models)

The chatbots just predict the word that comes next in the sentence by assigning probability to different words and finding the best.

Step 1: Pretraining —> We feed in large amount of data so that it could adjust the parameters (weights) accordingly using the backpropagation technique.

Step 2: RLHF (Reinforcement learning with human feedback)

In this step, workers flag the unhelpful predictions, fine-tuning the parameters according to the predictions users prefer.

The whole computation is doing using GPUs which are capable of running multiple operations parallelly.

Initially, text was processed word by word and then transformers were introduced that could use all the text at once. They rely on attention operation. It also uses feedforward technique.

GPT — Generative Pre-trained Transformers