

1 LAB7 tutorial for Machine Learning Neural NetWork & Pytorch

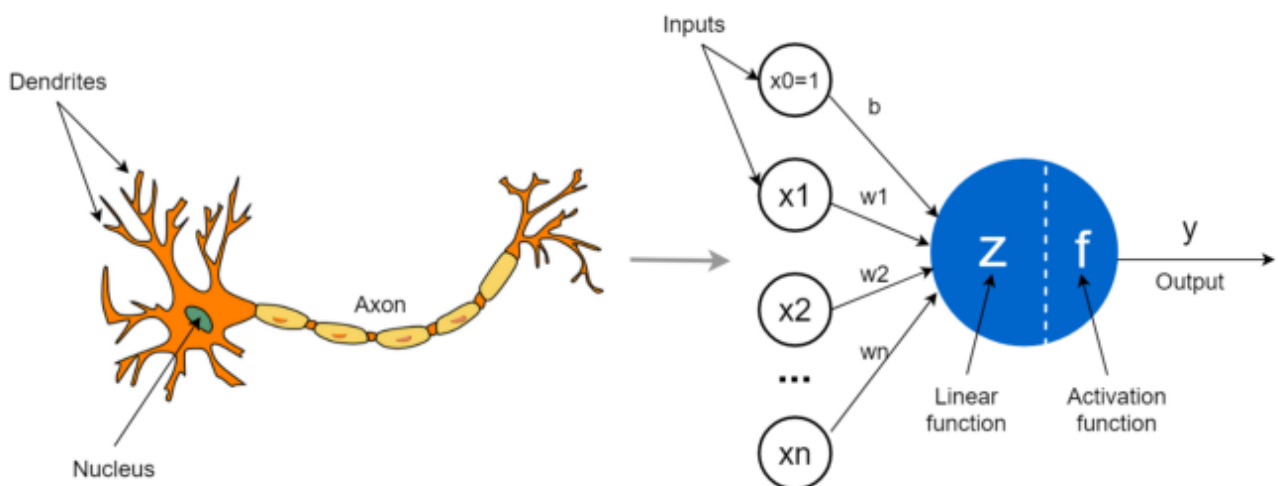
The document description are designed by Jla Yanhong in 2022. Oct. 20th

1.1 Objective

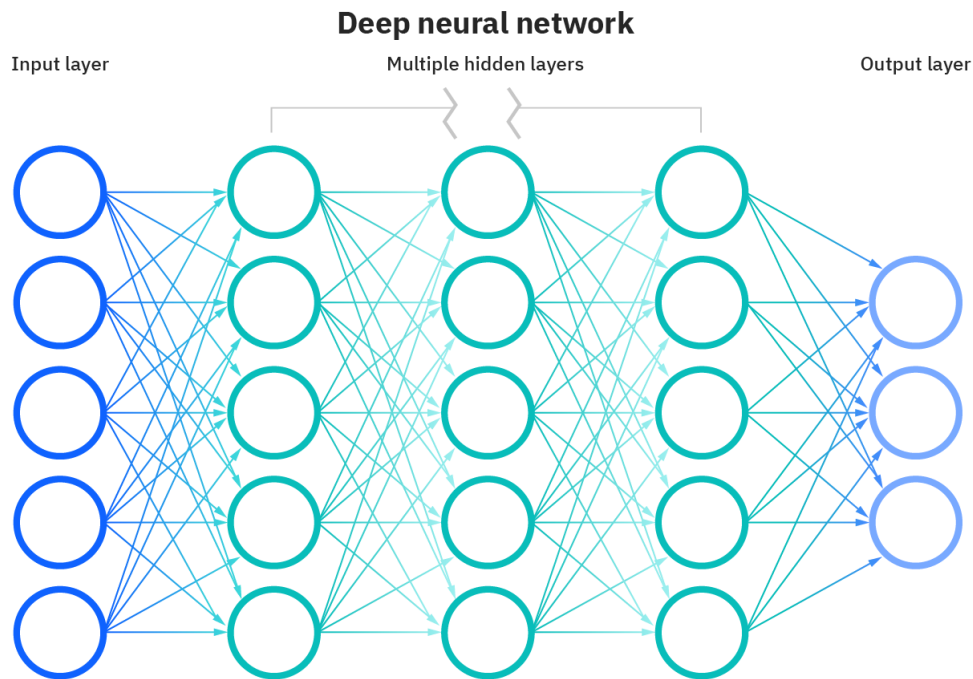
- Install the pytorch neural network framework for your computer.
- Learn to use pytorch.
- Implement a simple neural network using pytorch
- Complete the LAB assignment and submit it to BB.

1.2 Introduction

Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.

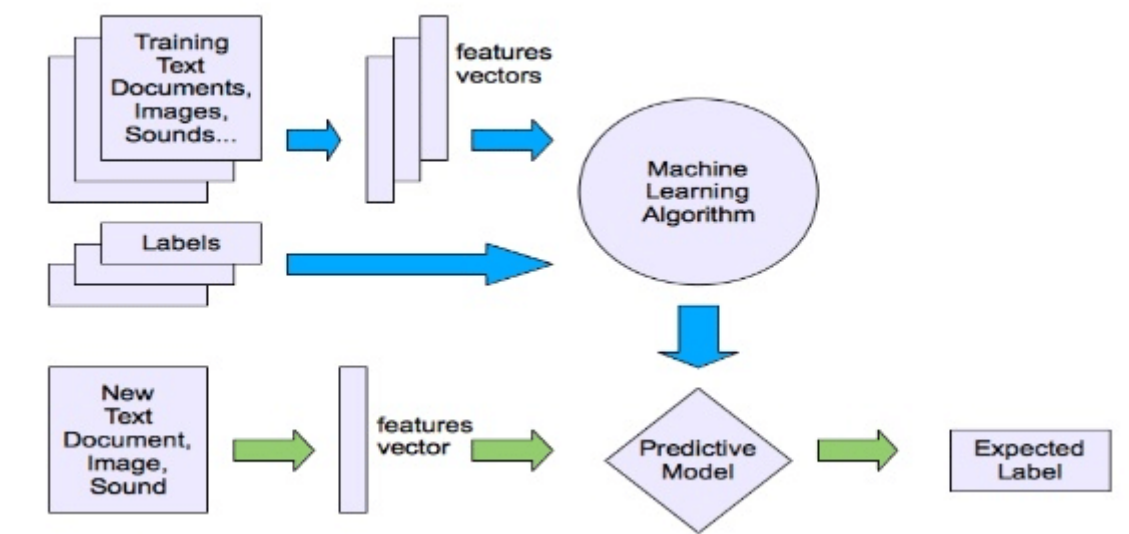


Artificial neural networks (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.



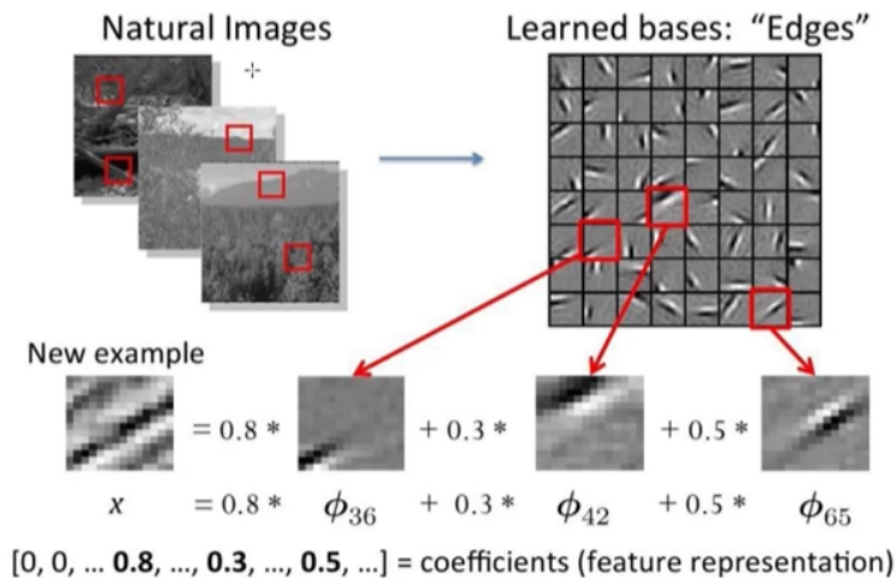
1.3 The role of neural networks in Machine learning

- Supervise machine learning process

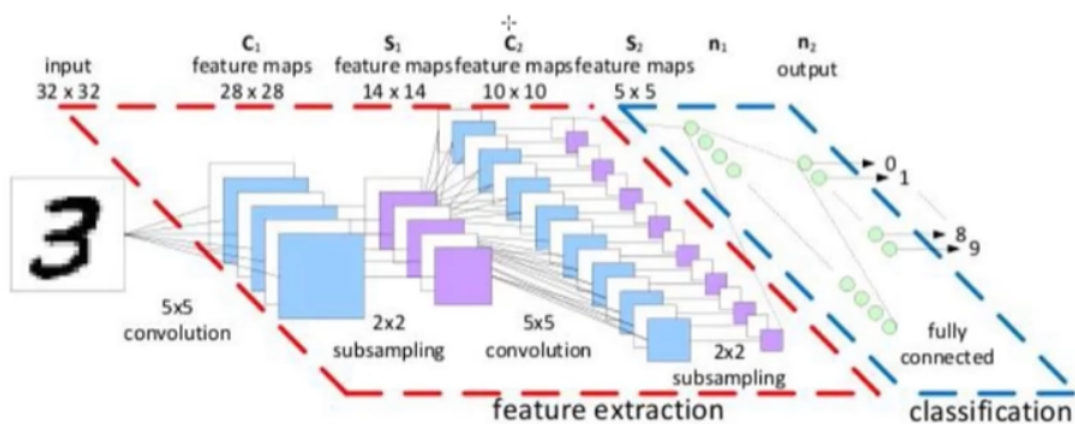


In the figure above, the hardest part is how to obtain valid feature vectors. This is a technique called **feature engineering**.

- **The Importance of Feature Engineering:**
 - Preprocessing and feature extraction determine the upper bound of the model
 - The algorithm and parameter selection approach this upper bound.
- **Traditional feature extraction methods:**



- Neural networks automatically extract features



1.4 Pytorch

1.4.1 Install Pytorch

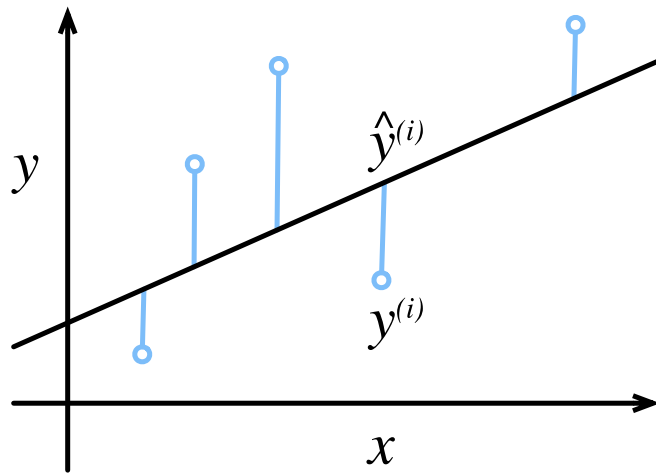
Please refer to **Installing PyTorch on Windows 10.md** for this section:

1.4.2 Learning pytorch with linear regression

1.4.2.1 Linear regression

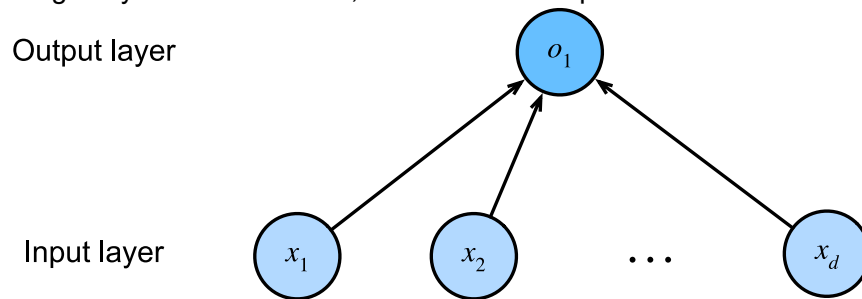
Do you remember the general form of linear regression model's prediction?

$$\hat{y} = h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n = \theta^T \cdot x$$



Fitting a linear regression model to one-dimensional data

Linear regression is a single-layer neural network, We used this simple network to learn how to use pytorch.



Linear regression is a single-layer neural network

1.4.2.2 Warm-up: [numpy \(https://numpy.org/learn/\)](https://numpy.org/learn/)

Before introducing PyTorch, we will first implement the network using numpy.

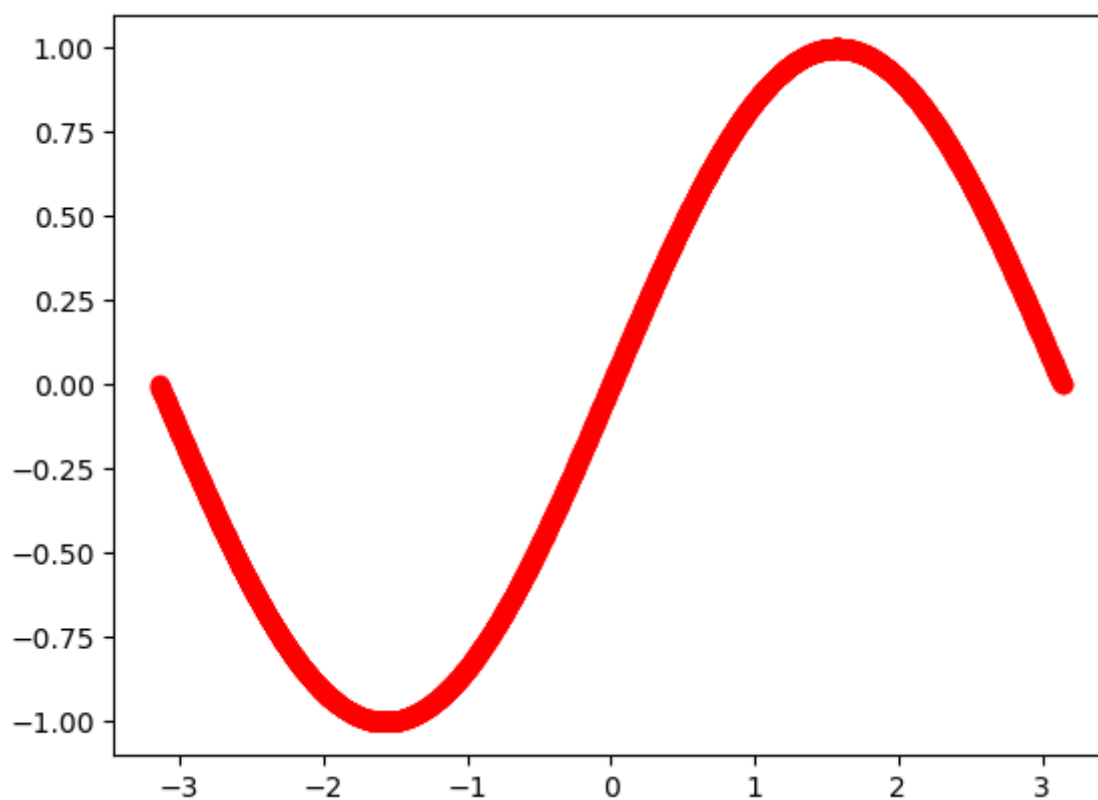
Numpy provides an n-dimensional array object, and many functions for manipulating these arrays. Numpy is a generic framework for scientific computing; it does not know anything about computation graphs, or deep learning, or gradients. However we can easily use numpy to fit a third order polynomial to sine function by manually implementing the forward and backward passes through the network using numpy operations:

In [1]:

```
1 # -*- coding: utf-8 -*-
2 import numpy as np
3 import math
4 import matplotlib.pyplot as plt
5 # Create random input and output data
6 x = np.linspace(-math.pi, math.pi, 2000)
7 y = np.sin(x)
8
9 plt.scatter(x, y, c = 'r')
```

Out[1]:

<matplotlib.collections.PathCollection at 0x1c2c6bf3580>



In [2]:

```

1  # Randomly initialize weights
2  a = np.random.randn()
3  b = np.random.randn()
4  c = np.random.randn()
5  d = np.random.randn()
6
7  learning_rate = 1e-6
8  for t in range(2000):
9      # Forward pass: compute predicted y
10     #  $y = a + b x + c x^2 + d x^3$ 
11     y_pred = a + b * x + c * x ** 2 + d * x ** 3
12
13     # Compute and print loss
14     loss = np.square(y_pred - y).sum()
15     if t % 100 == 99:
16         print(t, loss)
17
18     # Backprop to compute gradients of a, b, c, d with respect to loss
19     grad_y_pred = 2.0 * (y_pred - y)
20     grad_a = grad_y_pred.sum()
21     grad_b = (grad_y_pred * x).sum()
22     grad_c = (grad_y_pred * x ** 2).sum()
23     grad_d = (grad_y_pred * x ** 3).sum()
24
25     # Update weights
26     a -= learning_rate * grad_a
27     b -= learning_rate * grad_b
28     c -= learning_rate * grad_c
29     d -= learning_rate * grad_d
30
31 print(f'Result: y = {a} + {b} x + {c} x^2 + {d} x^3')
```

99 219.75987278604913

199 148.58343528337207

299 101.43604011225801

399 70.20181185376865

499 49.50712628592717

599 35.7937155020376

699 26.70516580291521

799 20.680815526991793

899 16.686920030443595

999 14.038672272495969

1099 12.282363729041458

1199 11.11735701266532

1299 10.344415058173519

1399 9.83148057616177

1499 9.491010056353504

1599 9.264959227655064

1699 9.11483583543545

1799 9.015108695797782

1899 8.94883998952236

1999 8.904790434080176

Result: y = 0.0030797666561155302 + 0.86538964081298 x + -0.0005313109957182931 x^2
+ -0.09456060520460607 x^3

1.4.2.3 PyTorch: Tensors

Numpy is a great framework, but it cannot utilize GPUs to accelerate its numerical computations. For modern deep neural networks, GPUs often provide speedups of 50x or greater, so unfortunately numpy won't be enough for modern deep learning.

Here we introduce the most fundamental PyTorch concept: the Tensor.

In [3]:

```

1  # -*- coding: utf-8 -*-
2
3  import torch
4  import math
5
6
7  dtype = torch.float
8  device = torch.device("cpu")
9  # device = torch.device("cuda:0") # Uncomment this to run on GPU
10
11 # Create random input and output data
12 x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype)
13 y = torch.sin(x)
14
15 # Randomly initialize weights
16 a = torch.randn((), device=device, dtype=dtype)
17 b = torch.randn((), device=device, dtype=dtype)
18 c = torch.randn((), device=device, dtype=dtype)
19 d = torch.randn((), device=device, dtype=dtype)
20
21 learning_rate = 1e-6
22 for t in range(2000):
23     # Forward pass: compute predicted y
24     y_pred = a + b * x + c * x ** 2 + d * x ** 3
25
26     # Compute and print loss
27     loss = (y_pred - y).pow(2).sum().item()
28     if t % 100 == 99:
29         print(t, loss)
30
31     # Backprop to compute gradients of a, b, c, d with respect to loss
32     grad_y_pred = 2.0 * (y_pred - y)
33     grad_a = grad_y_pred.sum()
34     grad_b = (grad_y_pred * x).sum()
35     grad_c = (grad_y_pred * x ** 2).sum()
36     grad_d = (grad_y_pred * x ** 3).sum()
37
38     # Update weights using gradient descent
39     a -= learning_rate * grad_a
40     b -= learning_rate * grad_b
41     c -= learning_rate * grad_c
42     d -= learning_rate * grad_d
43
44
45 print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

d:\miniconda3\envs\pytorch\lib\site-packages\tqdm\auto.py:22: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html (https://ipywidgets.readthedocs.io/en/stable/user_install.html)

```
from .autonotebook import tqdm as notebook_tqdm
```

```

99 1308.72705078125
199 886.845458984375
299 602.4786987304688
399 410.6216735839844
499 281.0531311035156
599 193.46322631835938
699 134.19088745117188
```



```
799 94.03910827636719
899 66.81092071533203
999 48.32662582397461
1099 35.76449203491211
1199 27.217470169067383
1299 21.395824432373047
1399 17.42598533630371
1499 14.715751647949219
1599 12.863423347473145
1699 11.59592056274414
1799 10.727645874023438
1899 10.13214111328125
1999 9.723262786865234
Result: y = 0.024994973093271255 + 0.8385941982269287 x + -0.004312050063163042 x^
2 + -0.09074918925762177 x^3
```

1.4.2.4 PyTorch: Tensors and autograd

In the above examples, we had to manually implement both the forward and backward passes of our neural network. Manually implementing the backward pass is not a big deal for a small two-layer network, but can quickly get very hairy for large complex networks.

Thankfully, we can use automatic differentiation to automate the computation of backward passes in neural networks. The autograd package in PyTorch provides exactly this functionality.

When using autograd, the forward pass of your network will define a computational graph; nodes in the graph will be Tensors, and edges will be functions that produce output Tensors from input Tensors. Backpropagating through this graph then allows you to easily compute gradients.

This sounds complicated, it's pretty simple to use in practice. Each Tensor represents a node in a computational graph. If `x` is a Tensor that has `x.requires_grad=True` then `x.grad` is another Tensor holding the gradient of `x` with respect to some scalar value.

In [4]:

```

1  # -*- coding: utf-8 -*-
2
3  from importlib.metadata import requires
4  import torch
5  from torch.autograd import Variable as V
6  import math
7
8
9  dtype = torch.float
10 device = torch.device("cpu")
11 # device = torch.device("cuda:0") # Uncomment this to run on GPU
12
13 # Create random input and output data
14 x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype)
15 y = torch.sin(x)
16
17
18 # Randomly initialize weights
19 a = torch.randn(), device=device, dtype=dtype, requires_grad=True)
20 b = torch.randn(), device=device, dtype=dtype, requires_grad=True)
21 c = torch.randn(), device=device, dtype=dtype, requires_grad=True)
22 d = torch.randn(), device=device, dtype=dtype, requires_grad=True)
23 # a = V(a, requires_grad=True)
24 # #a.requires_grad = True
25 # b.requires_grad = True
26 # c.requires_grad = True
27 # d.requires_grad = True
28
29
30 learning_rate = 1e-6
31 for t in range(2000):
32     # Forward pass: compute predicted y
33     y_pred = a + b * x + c * x ** 2 + d * x ** 3
34
35     # Compute and print loss
36     loss = (y_pred - y).pow(2).sum()
37     #loss = (y_pred - y).pow(2).sum().item()
38     if t % 100 == 99:
39         print(t, loss.data.item())
40
41     # Backprop to compute gradients of a, b, c, d with respect to loss
42     # grad_y_pred = 2.0 * (y_pred - y)
43     # grad_a = grad_y_pred.sum()
44     # grad_b = (grad_y_pred * x).sum()
45     # grad_c = (grad_y_pred * x ** 2).sum()
46     # grad_d = (grad_y_pred * x ** 3).sum()
47     loss.backward()
48
49     # Update weights using gradient descent
50     a.data -= learning_rate * a.grad.data
51     b.data -= learning_rate * b.grad.data
52     c.data -= learning_rate * c.grad.data
53     d.data -= learning_rate * d.grad.data
54
55     a.grad.data.zero_()
56     b.grad.data.zero_()
57     c.grad.data.zero_()
58     d.grad.data.zero_()
59

```

```
60  
61 print(f'Result: y = {a.data.item()} + {b.data.item()} x + {c.data.item()} x^2 + {d.data.item()}'
```

99 866.6287231445312

199 590.111083984375

299 403.1534729003906

399 276.6170349121094

499 190.88546752929688

599 132.7378692626953

699 93.25631713867188

799 66.41924285888672

899 48.156795501708984

999 35.71524429321289

1099 27.22962188720703

1199 21.4354305267334

1299 17.474504470825195

1399 14.763716697692871

1499 12.906318664550781

1599 11.632223129272461

1699 10.75724983215332

1799 10.155684471130371

1899 9.74163818359375

1999 9.456335067749023

Result: $y = 0.021818608045578003 + 0.8425113558769226 x + -0.00376407103613019 x^2 + -0.09130636602640152 x^3$

1.4.2.5 Pytorch:[nn module \(https://pytorch.org/docs/stable/nn.html\)](https://pytorch.org/docs/stable/nn.html)

we use the nn package to implement our polynomial model network

- Using torch.nn.Sequential

In [5]:

```

1  # -*- coding: utf-8 -*-
2  import torch
3  import math
4
5
6  # Create Tensors to hold input and outputs.
7  x = torch.linspace(-math.pi, math.pi, 2000)
8  y = torch.sin(x)
9
10 # For this example, the output y is a linear function of (x, x^2, x^3), so
11 # we can consider it as a linear layer neural network. Let's prepare the
12 # tensor (x, x^2, x^3).
13 p = torch.tensor([1, 2, 3])
14 xx = x.unsqueeze(-1).pow(p)
15
16 # In the above code, x.unsqueeze(-1) has shape (2000, 1), and p has shape
17 # (3,), for this case, broadcasting semantics will apply to obtain a tensor
18 # of shape (2000, 3)
19
20 # Use the nn package to define our model as a sequence of layers. nn.Sequential
21 # is a Module which contains other Modules, and applies them in sequence to
22 # produce its output. The Linear Module computes output from input using a
23 # linear function, and holds internal Tensors for its weight and bias.
24 # The Flatten layer flattens the output of the linear layer to a 1D tensor,
25 # to match the shape of `y`.
26 model = torch.nn.Sequential(
27     torch.nn.Linear(3, 1),
28     torch.nn.Flatten(0, 1)
29 )
30
31 # The nn package also contains definitions of popular loss functions; in this
32 # case we will use Mean Squared Error (MSE) as our loss function.
33 loss_fn = torch.nn.MSELoss(reduction='sum')
34
35 learning_rate = 1e-6
36 for t in range(2000):
37
38     # Forward pass: compute predicted y by passing x to the model. Module objects
39     # override the __call__ operator so you can call them like functions. When
40     # doing so you pass a Tensor of input data to the Module and it produces
41     # a Tensor of output data.
42     y_pred = model(xx)
43
44     # Compute and print loss. We pass Tensors containing the predicted and true
45     # values of y, and the loss function returns a Tensor containing the
46     # loss.
47     loss = loss_fn(y_pred, y)
48     if t % 100 == 99:
49         print(t, loss.item())
50
51     # Zero the gradients before running the backward pass.
52     model.zero_grad()
53
54     # Backward pass: compute gradient of the loss with respect to all the learnable
55     # parameters of the model. Internally, the parameters of each Module are stored
56     # in Tensors with requires_grad=True, so this call will compute gradients for
57     # all learnable parameters in the model.
58     loss.backward()
59

```

```

60     # Update the weights using gradient descent. Each parameter is a Tensor, so
61     # we can access its gradients like we did before.
62     with torch.no_grad():
63         for param in model.parameters():
64             param -= learning_rate * param.grad
65
66 # You can access the first layer of `model` like accessing the first item of a list
67 linear_layer = model[0]
68
69 # For linear layer, its parameters are stored as `weight` and `bias`.
70 print(f'Result: y = {linear_layer.bias.item()} + {linear_layer.weight[:, 0].item()} x + {linear

```

```

99 222. 3236846923828
199 154. 76187133789062
299 108. 69402313232422
399 77. 2461929321289
499 55. 753963470458984
599 41. 04864501953125
699 30. 97553062438965
799 24. 067575454711914
899 19. 324832916259766
999 16. 0649356842041
1099 13. 82175350189209
1199 12. 276458740234375
1299 11. 210790634155273
1399 10. 475077629089355
1499 9. 96662712097168
1599 9. 614869117736816
1699 9. 371269226074219
1799 9. 202404975891113
1899 9. 085237503051758
1999 9. 00385856628418
Result: y = -0.012696003541350365 + 0.850379467010498 x + 0.002190270461142063 x^2 +
-0.09242553263902664 x^3

```

- Define the class

We define our neural network by subclassing `nn.Module`, and initialize the neural network layers in `init`. Every `nn.Module` subclass implements the operations on input data in the `forward()` method. The `forward()` method is in charge of conducting the **forward propagation**.

In [6]:

```

1 from torch import nn
2 class LinearModel(nn.Module):
3     def __init__(self):
4         super(LinearModel, self).__init__()
5         self.linear = nn.Linear(3, 1)
6         self.flatten = nn.Flatten(0, 1)
7         # self.model = torch.nn.Sequential(
8         #     torch.nn.Linear(3, 1),
9         #     torch.nn.Flatten(0, 1)
10        # )
11
12    def forward(self, x):
13        y = self.flatten(self.linear(x))
14
15        return y

```

In [7]:

```

1
2
3 # -*- coding: utf-8 -*-
4 import torch
5 import math
6
7
8 # Create Tensors to hold input and outputs.
9 x = torch.linspace(-math.pi, math.pi, 2000)
10 y = torch.sin(x)
11
12 # For this example, the output y is a linear function of (x, x^2, x^3), so
13 # we can consider it as a linear layer neural network. Let's prepare the
14 # tensor (x, x^2, x^3).
15 p = torch.tensor([1, 2, 3])
16 xx = x.unsqueeze(-1).pow(p)
17
18 # In the above code, x.unsqueeze(-1) has shape (2000, 1), and p has shape
19 # (3,), for this case, broadcasting semantics will apply to obtain a tensor
20 # of shape (2000, 3)
21
22
23 model = LinearModel()
24
25 # The nn package also contains definitions of popular loss functions; in this
26 # case we will use Mean Squared Error (MSE) as our loss function.
27 loss_fn = torch.nn.MSELoss(reduction='sum')
28
29 for t in range(2000):
30
31     # Forward pass: compute predicted y by passing x to the model. Module objects
32     # override the __call__ operator so you can call them like functions. When
33     # doing so you pass a Tensor of input data to the Module and it produces
34     # a Tensor of output data.
35     y_pred = model(xx)
36
37     # Compute and print loss. We pass Tensors containing the predicted and true
38     # values of y, and the loss function returns a Tensor containing the
39     # loss.
40     loss = loss_fn(y_pred, y)
41     if t % 100 == 99:
42         print(t, loss.item())
43
44     # Zero the gradients before running the backward pass.
45     model.zero_grad()
46
47     # Backward pass: compute gradient of the loss with respect to all the learnable
48     # parameters of the model. Internally, the parameters of each Module are stored
49     # in Tensors with requires_grad=True, so this call will compute gradients for
50     # all learnable parameters in the model.
51     loss.backward()
52
53     # Update the weights using gradient descent. Each parameter is a Tensor, so
54     # we can access its gradients like we did before.
55     with torch.no_grad():
56         for param in model.parameters():
57             param -= learning_rate * param.grad
58

```

```

59 # You can access the first layer of `model` like accessing the first item of a list
60 linear_layer = model.linear
61
62
63
64 # For linear layer, its parameters are stored as `weight` and `bias`.
65 print(f'Result: y = {linear_layer.bias.item()} + {linear_layer.weight[:, 0].item()} x + {linear

```

```

99 270.012939453125
199 185.647705078125
299 128.658203125
399 90.12274169921875
499 64.03877258300781
599 46.364288330078125
699 34.375492095947266
799 26.234413146972656
899 20.699996948242188
999 16.933467864990234
1099 14.367202758789062
1199 12.616707801818848
1299 11.421326637268066
1399 10.604074478149414
1499 10.044687271118164
1599 9.661358833312988
1699 9.398388862609863
1799 9.217771530151367
1899 9.093585968017578
1999 9.008096694946289
Result: y = -0.011800085194408894 + 0.848804771900177 x + 0.002035710262134671 x^2 +
-0.09220154583454132 x^3

```

1.4.2.6 PyTorch: optim

Up to this point we have updated the weights of our models by manually mutating the Tensors holding learnable parameters with `torch.no_grad()`. This is not a huge burden for simple optimization algorithms like stochastic gradient descent, but in practice we often train neural networks using more sophisticated optimizers like AdaGrad, RMSProp, Adam, etc.

The `optim` package in PyTorch abstracts the idea of an optimization algorithm and provides implementations of commonly used optimization algorithms.

In this example we will use the `nn` package to define our model as before, but we will optimize the model using the `RMSprop` algorithm provided by the `optim` package:

In [8]:

```

1  # -*- coding: utf-8 -*-
2  import torch
3  import math
4
5
6  # Create Tensors to hold input and outputs.
7  x = torch.linspace(-math.pi, math.pi, 2000)
8  y = torch.sin(x)
9
10 # For this example, the output y is a linear function of (x, x^2, x^3), so
11 # we can consider it as a linear layer neural network. Let's prepare the
12 # tensor (x, x^2, x^3).
13 p = torch.tensor([1, 2, 3])
14 xx = x.unsqueeze(-1).pow(p)
15
16 # In the above code, x.unsqueeze(-1) has shape (2000, 1), and p has shape
17 # (3,), for this case, broadcasting semantics will apply to obtain a tensor
18 # of shape (2000, 3)
19
20 # Use the nn package to define our model as a sequence of layers. nn.Sequential
21 # is a Module which contains other Modules, and applies them in sequence to
22 # produce its output. The Linear Module computes output from input using a
23 # linear function, and holds internal Tensors for its weight and bias.
24 # The Flatten layer flattens the output of the linear layer to a 1D tensor,
25 # to match the shape of `y`.
26 model = torch.nn.Sequential(
27     torch.nn.Linear(3, 1),
28     torch.nn.Flatten(0, 1)
29 )
30 model.requires_grad_()
31
32 # The nn package also contains definitions of popular loss functions; in this
33 # case we will use Mean Squared Error (MSE) as our loss function.
34 loss_fn = torch.nn.MSELoss(reduction='sum')
35
36 optimizer = torch.optim.RMSprop(params=model.parameters(), lr=0.001)
37 #optimizer = torch.optim.SGD(model.parameters(), lr=1e-6, momentum=0.9)
38
39 for t in range(2000):
40     # Zero the gradients before running the backward pass.
41     optimizer.zero_grad()
42     # Forward pass: compute predicted y by passing x to the model.
43     y_pred = model(xx)
44
45     # Compute and print loss. We pass Tensors containing the predicted and true
46     # values of y, and the loss function returns a Tensor containing the
47     # loss.
48
49     loss = loss_fn(y_pred, y)
50     if t % 100 == 99:
51         print(t, loss.item())
52
53     # Backward pass: compute gradient of the loss with respect to all the learnable
54     # parameters of the model.
55     loss.backward()
56
57     # Update the weights using gradient descent.
58     optimizer.step()
59

```



```

60
61 # You can access the first layer of `model` like accessing the first item of a list
62 linear_layer = model[0]
63
64 # For linear layer, its parameters are stored as `weight` and `bias`.
65 print(f'Result: y = {linear_layer.bias.item()} + {linear_layer.weight[:, 0].item()} x + {linear

```

```

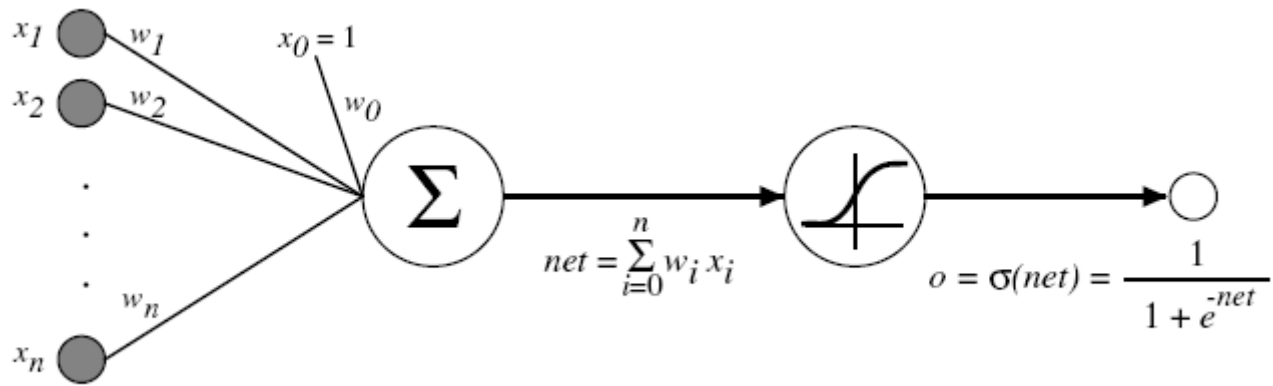
99 1641.3837890625
199 663.7893676757812
299 325.3468322753906
399 198.01614379882812
499 120.5510482788086
599 65.60163116455078
699 32.06513977050781
799 15.710921287536621
899 10.060104370117188
999 8.967170715332031
1099 8.900091171264648
1199 8.91170597076416
1299 8.906639099121094
1399 8.904470443725586
1499 8.907258987426758
1599 8.908171653747559
1699 8.90713119506836
1799 8.909259796142578
1899 8.934460639953613
1999 8.920280456542969
Result: y = -0.00045510393101722 + 0.8572410345077515 x + -0.0004552106838673353 x^2
+ -0.09283004701137543 x^3

```

1.4.3 Learning pytorch with logistic regression

If we use a single-layer network for classification, this is known as a logistic regression.

We need to add the sigmoid function to the output of the linear regression.



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

We can derive gradient decent rules to train

Perceptron

Let us define the number of epochs and the learning rate we want our model for training. As the data is a binary classification, we will use **Binary Cross Entropy** as the **loss function** used to optimize the model using an SGD optimizer .

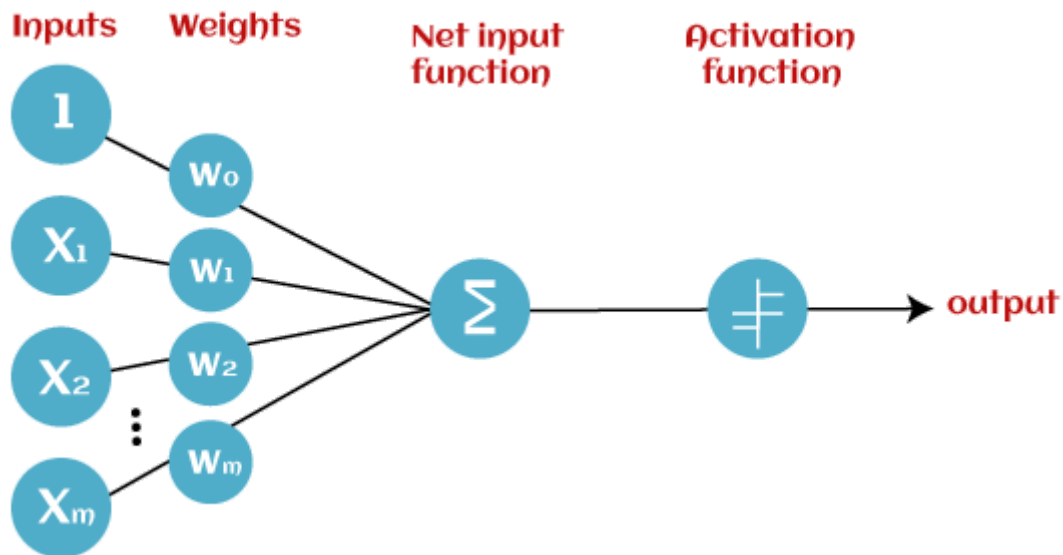
Please complete this part of the code!!

1.5 Neural Network

1.5.1 Perceptron-The basic unit of neural network

A simple model of a biological neuron in an artificial neural network is known as Perceptron. it is the primary step to learn Machine Learning and Deep Learning technologies.

we can consider it as a single-layer neural network with four main parameters, i.e., input values , weights and Bias , net sum , and an activation function .



- **Input Nodes or Input Layer:**

This is the primary component of Perceptron which accepts the initial data into the system for further processing.

- **Wight and Bias:**

Weight parameter represents the strength of the connection between units. Bias can be considered as the intercept in a linear equation.

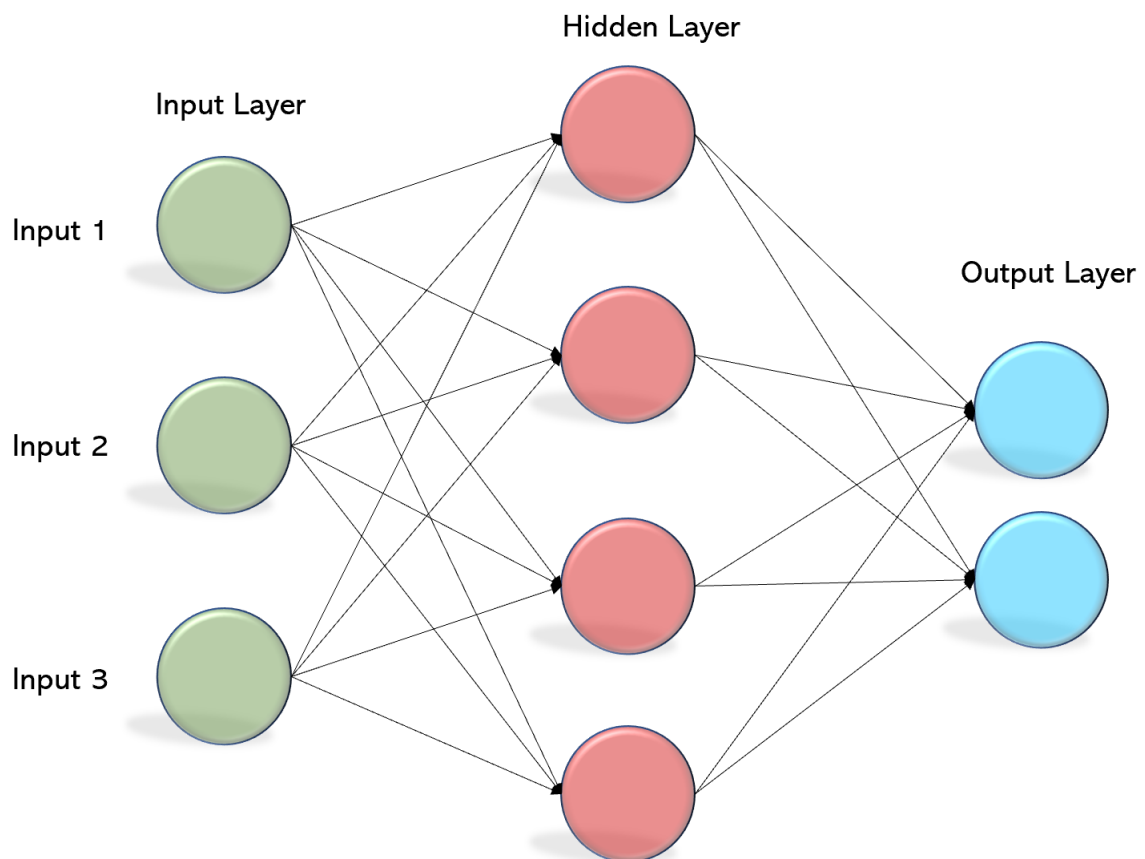
- **Activation Function:**

These are the final and important components that help to determine whether the neuron will fire or not. The activation function of perceptron is `sign function`

1.5.2 Multilayer Perceptron(MLP)

A neuron is a mathematical model of the behaviour of a single neuron in a biological nervous system.

A single neuron can solve some simple tasks, but the power of neural networks comes when many of them are arranged in layers and connected in a network architecture.

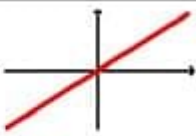

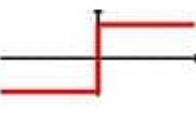






A Multilayered Perceptron is a Neural Network. A neural network having more than 3 hidden layers is called a **Deep Neural Network**.

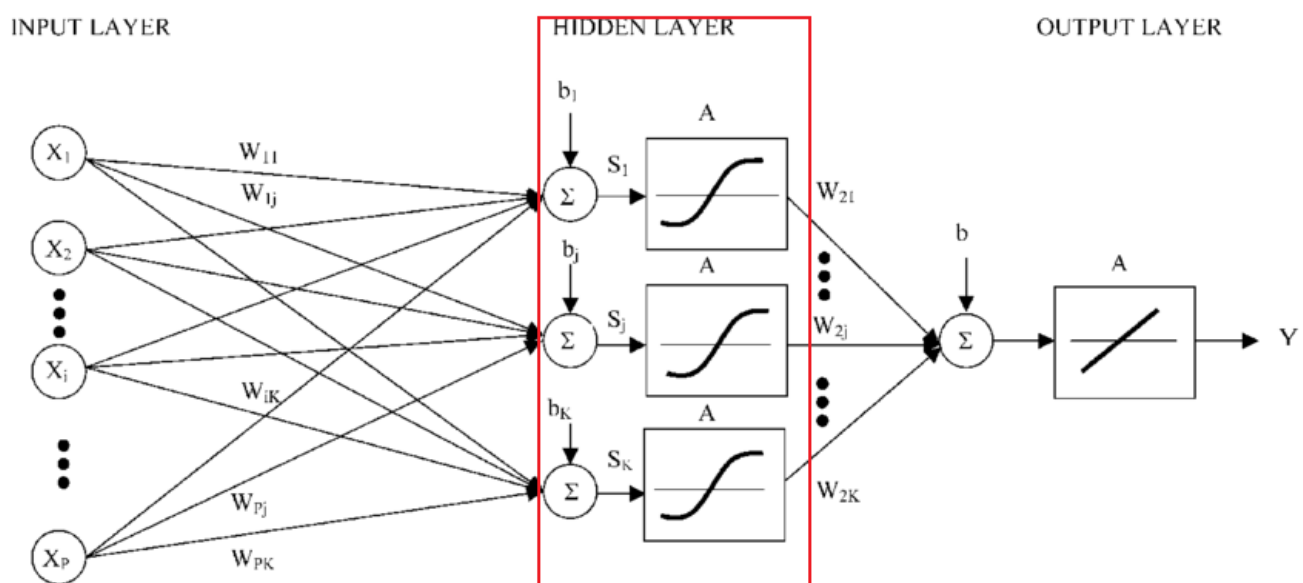
In this lab, Multilayer Perceptron and Neural Network mean the same thing.

1.5.3 Activation functions

Various activation functions that can be used with Perceptron.

Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit Step (Heaviside Function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise Linear	$\phi(z) = \begin{cases} 0 & z \leq -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \leq z \leq \frac{1}{2} \\ 1 & z \geq \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multilayer NN	
Hyperbolic Tangent (tanh)	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

Neural network without activation functions are simply linear regression model. The activation makes the input capable of learning and performing more complex tasks.



Therefore, when we write the neural network framework, the neurons in each hidden layer are most of the time followed by an activation function.

I recommend that you use the relu function as you build your neural network framework.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

1.5.4 Example for MLP

This Example uses dataset digit123.csv , which has 36 columns, and the last column is the dependent variable. We use this dataset to familiarize ourselves with MLP and solve the multi-classification problem.

Note that the values of the dependent variable are 1,2,3, and label coding is required.

1.5.4.1 MLP Model

- step 1 load datasets

In [9]:

```

1 import numpy as np
2 import pandas as pd
3
4 # 'Matplotlib' is a data visualization library for 2D and 3D plots, built on numpy
5 from matplotlib import pyplot as plt
6 %matplotlib inline
7
8 # to suppress warnings
9 import warnings
10 warnings.filterwarnings("ignore")
11 # ===== step 1/6 load datasets =====
12 df = pd.read_csv("datasets/digit123.csv", header=None)
13 df.head()

```

Out[9]:

	0	1	2	3	4	5	6	7	8	9	...	27	28	29	30	31	32	33	34	35	36
0	0	0	0	1	0	0	0	0	0	1	...	1	0	0	0	0	0	1	0	0	1
1	0	0	0	1	0	0	0	0	1	1	...	1	0	0	0	0	0	1	0	0	1
2	0	0	1	1	0	0	0	0	0	1	...	1	0	0	0	0	0	1	0	0	1
3	0	0	0	1	0	0	0	0	0	1	...	1	0	0	0	0	1	1	0	0	1
4	0	0	0	1	0	0	0	0	0	1	...	1	0	0	0	0	1	1	1	0	1

5 rows × 37 columns

In [10]:

```
1 df[36].value_counts()
```

Out[10]:

```

1    32
2    32
3    32
Name: 36, dtype: int64

```

In [11]:

```

1 y = df[36]
2 y.replace((1, 2, 3), (0, 1, 2), inplace=True)
3 X = df.drop(36, axis=1)
4 X.shape

```

Out[11]:

(96, 36)

In [12]:

```

1 ## Splitting for X and Y variables:
2 from sklearn.model_selection import train_test_split
3 ## Splitting dataset into 80% Training and 20% Testing Data:
4 X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, random_state =0)
5

```

In [13]:

```

1 import torch
2 #Converting them to tensors as PyTorch works on, we will use the torch.from_numpy() method:
3 X_train = torch.from_numpy(X_train.values).float()
4 X_test = torch.from_numpy(X_test.values).float()
5 y_train = torch.from_numpy(y_train.values).long()
6 y_test = torch.from_numpy(y_test.values).long()

```

- step 2 Define a MLP subclass of nn. Module.

In [14]:

```

1 # Define a MLP subclass of nn. Module.
2 # ===== step 2/6 define model =====
3 import torch
4
5 class MLP(nn.Module):
6     def __init__(self, n_i, n_h, n_o):
7         super(MLP, self).__init__()
8         self.flatten = nn.Flatten()
9         self.linear1 = nn.Linear(n_i, n_h)
10        self.relu = nn.ReLU()
11        self.linear2 = nn.Linear(n_h, n_o)
12    def forward(self, input):
13        return self.linear2(self.relu(self.linear1(self.flatten(input))))
14

```

- step 3 Create model

In [15]:

```

1
2 num_epochs = 10000
3 learning_rate = 0.001
4 # Create the model
5 # ===== step 3/6 Create model =====
6 models = MLP(X_train.shape[1], X_train.shape[1]//2, y_train.unique().size()[0])
7
8

```

- step 4 Loss function

In []:

```

1 # ===== step 4/6 Loss function =====
2 criterions = torch.nn.CrossEntropyLoss()
3

```

- step 5 The optimizer

In []:

```

1 # ===== step 4/6 The optimizer =====
2 optimizers = torch.optim.SGD(models.parameters(), lr=learning_rate)

```

- step 6 Train the Model

In [16]:

```

1 #Train the Model
2 # ===== step 5/6 training =====
3 for epoch in range(num_epochs):
4     models.train()
5     optimizers.zero_grad()
6     # Forward pass
7     y_pred = models(X_train)
8     # Compute Loss
9     loss = criterions(y_pred, y_train)
10    # Backward pass
11    loss.backward()
12    optimizers.step()
13    if (epoch+1) % 20 == 0:
14        # printing loss values on every 10 epochs to keep track
15        print(f'epoch: {epoch+1}, loss = {loss.item():.4f}')

```

```

epoch: 20, loss = 1.0930
epoch: 40, loss = 1.0871
epoch: 60, loss = 1.0813
epoch: 80, loss = 1.0755
epoch: 100, loss = 1.0698
epoch: 120, loss = 1.0642
epoch: 140, loss = 1.0586
epoch: 160, loss = 1.0531
epoch: 180, loss = 1.0476
epoch: 200, loss = 1.0421
epoch: 220, loss = 1.0367
epoch: 240, loss = 1.0313
epoch: 260, loss = 1.0260
epoch: 280, loss = 1.0208
epoch: 300, loss = 1.0157
epoch: 320, loss = 1.0106
epoch: 340, loss = 1.0056
epoch: 360, loss = 1.0006
epoch: 380, loss = 0.9958
epoch: 400, loss = 0.9910

```

Here,

- when you call `models(X_train)`, you automatically call `models.forward()` to propagate forward.
- Next, the loss is calculated. When `loss.backward()` is called, it computes the loss gradient with respect to the weights (of the layer).
- The weights are then updated by calling `optimizer.step()`.
- After this, the weights have to be emptied for the next iteration. So the `zero_grad()` method is called.

The above code prints the loss at each 20th epoch.

- step 7 Model Performance

Let us finally see the model accuracy:

In [17]:

```

1 with torch.no_grad():
2     logits = models(X_test)
3     y_pred = nn.Softmax(dim=1)(logits)
4     y_predicted_cls = y_pred.argmax(1)
5     acc = y_predicted_cls.eq(y_test).sum() / float(y_test.shape[0])
6     print(f'accuracy: {acc.item():.4f}')
```

accuracy: 0.9000

In [18]:

```

1 #classification report
2 from sklearn.metrics import classification_report
3 print(classification_report(y_test, y_predicted_cls))
```

	precision	recall	f1-score	support
0	1.00	0.78	0.88	9
1	0.80	1.00	0.89	4
2	0.88	1.00	0.93	7
accuracy			0.90	20
macro avg	0.89	0.93	0.90	20
weighted avg	0.92	0.90	0.90	20

1.6 LAB Assignment

1.6.1 Exercise 1 logistic regression (50 points)

This exercise uses dataset `digit01.csv` , which has 13 columns, and the last column is the dependent variable.

This part requires you to implement a `logistic regression` using the `pytorch` framework (defining a logistic regression class that inherits `nn.module`). To test your model, we provide a dataset `digit01.csv` which is in the **datasets** folder. This dataset requires you to divide the training set and the test set by yourself, and it is recommended that 80% of the training set and 20% of the test set be used.

1.6.2 Exercise 2 Handwriting recognition with MLP(50 points)

Like last week's lab , your task in this section is also about recognizing handwritten digits, but you are required to use MLP to complete the exercise. It is recommended that you define an MLP class, which is a subclass of `nn.module` .

Note that your accuracy in this section will directly determine your score.

For this exercise we use the `minist` dataset.

1.6.3 Exercise 3 Questions (10 points)

- 1.What's the difference between logistic regression and Perceptron?
- 2.Advantages and disadvantages of neural networks?

3.What is the role of Activation Function in Neural networks?