***Practical 7***

**Aim:** *Implement ANN.*

***Theory:***

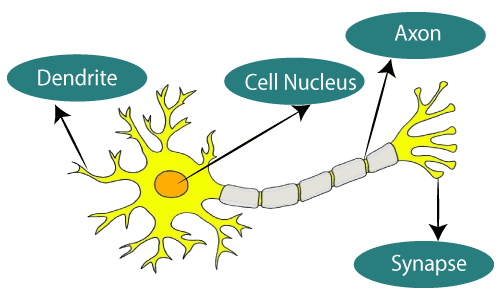
Artificial Neural Network Tutorial provides basic and advanced concepts of ANNs. Our Artificial Neural Network tutorial is developed for beginners as well as professions.

The term "Artificial neural network" refers to a biologically inspired sub-field of artificial intelligence modeled after the brain. An Artificial neural network is usually a computational network based on biological neural networks that construct the structure of the human brain. Similar to a human brain has neurons interconnected to each other, artificial neural networks also have neurons that are linked to each other in various layers of the networks. These neurons are known as nodes.

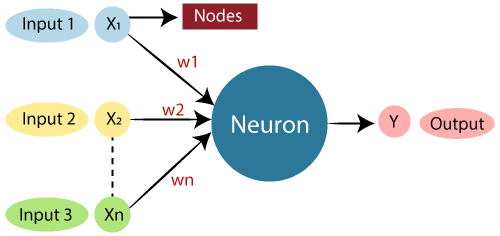
Artificial neural network tutorial covers all the aspects related to the artificial neural network. In this tutorial, we will discuss ANNs, Adaptive resonance theory, Kohonen self-organizing map, Building blocks, unsupervised learning, Genetic algorithm, etc.

**What is Artificial Neural Network?**

The term "**Artificial Neural Network**" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.



**The typical Artificial Neural Network looks something like the given figure.**



Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output.

Relationship between Biological neural network and artificial neural network:

|  |  |
| --- | --- |
| **Biological Neural Network** | **Artificial Neural Network** |
| Dendrites | Inputs |
| Cell nucleus | Nodes |
| Synapse | Weights |
| Axon | Output |

An **Artificial Neural Network** in the field of **Artificial intelligence** where it attempts to mimic the network of neurons makes up a human brain so that computers will have an option to understand things and make decisions in a human-like manner. The artificial neural network is designed by programming computers to behave simply like interconnected brain cells.

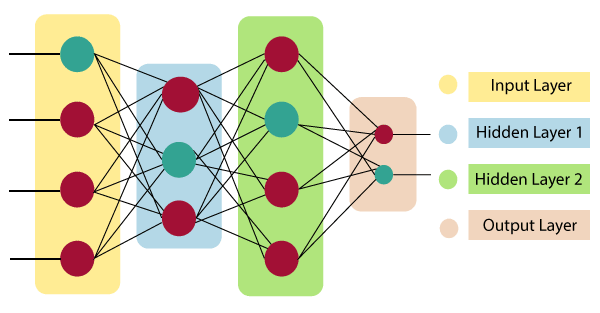
There are around 1000 billion neurons in the human brain. Each neuron has an association point somewhere in the range of 1,000 and 100,000. In the human brain, data is stored in such a manner as to be distributed, and we can extract more than one piece of this data when necessary from our memory parallelly. We can say that the human brain is made up of incredibly amazing parallel processors.

We can understand the artificial neural network with an example, consider an example of a digital logic gate that takes an input and gives an output. "OR" gate, which takes two inputs. If one or both the inputs are "On," then we get "On" in output. If both the inputs are "Off," then we get "Off" in output. Here the output depends upon input. Our brain does not perform the same task. The outputs to inputs relationship keep changing because of the neurons in our brain, which are "learning."

***The architecture of an artificial neural network:***

To understand the concept of the architecture of an artificial neural network, we have to understand what a neural network consists of. In order to define a neural network that consists of a large number of artificial neurons, which are termed units arranged in a sequence of layers. Lets us look at various types of layers available in an artificial neural network.

Artificial Neural Network primarily consists of three layers:



**Input Layer:**

As the name suggests, it accepts inputs in several different formats provided by the programmer.

**Hidden Layer:**

The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

**Output Layer:**

The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

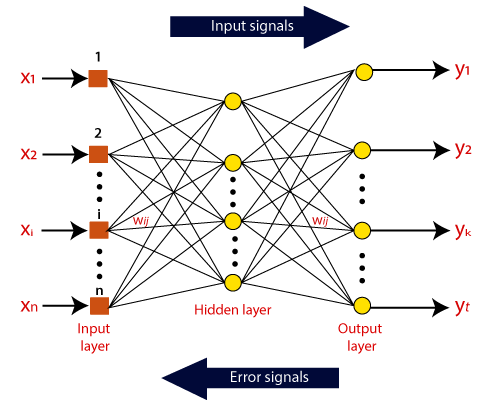
The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

What is Artificial Neural Network

It determines weighted total is passed as an input to an activation function to produce the output. Activation functions choose whether a node should fire or not. Only those who are fired make it to the output layer. There are distinctive activation functions available that can be applied upon the sort of task we are performing.

***How do artificial neural networks work?***

Artificial Neural Network can be best represented as a weighted directed graph, where the artificial neurons form the nodes. The association between the neurons outputs and neuron inputs can be viewed as the directed edges with weights. The Artificial Neural Network receives the input signal from the external source in the form of a pattern and image in the form of a vector. These inputs are then mathematically assigned by the notations x(n) for every n number of inputs.



Afterward, each of the input is multiplied by its corresponding weights ( these weights are the details utilized by the artificial neural networks to solve a specific problem ). In general terms, these weights normally represent the strength of the interconnection between neurons inside the artificial neural network. All the weighted inputs are summarized inside the computing unit.

If the weighted sum is equal to zero, then bias is added to make the output non-zero or something else to scale up to the system's response. Bias has the same input, and weight equals to 1. Here the total of weighted inputs can be in the range of 0 to positive infinity. Here, to keep the response in the limits of the desired value, a certain maximum value is benchmarked, and the total of weighted inputs is passed through the activation function.

The activation function refers to the set of transfer functions used to achieve the desired output. There is a different kind of the activation function, but primarily either linear or non-linear sets of functions. Some of the commonly used sets of activation functions are the Binary, linear, and Tan hyperbolic sigmoidal activation functions. Let us take a look at each of them in details:

***Binary:***

In binary activation function, the output is either a one or a 0. Here, to accomplish this, there is a threshold value set up. If the net weighted input of neurons is more than 1, then the final output of the activation function is returned as one or else the output is returned as 0.

***Sigmoidal Hyperbolic:***

The Sigmoidal Hyperbola function is generally seen as an "**S**" shaped curve. Here the tan hyperbolic function is used to approximate output from the actual net input. The function is defined as:

**F(x) = (1/1 + exp(-????x))**

Where ???? is considered the Steepness parameters.

***Code:***

***minst.py -driver code***

import numpy as np

import torch

import torchvision

from torch.utils.data import DataLoader

import layers

import loss

import optimizers

from model import Model

def get\_dataset(batch\_size):

    train\_loader = DataLoader(

        torchvision.datasets.MNIST('./data/', train=True, download=True,

                                   transform=torchvision.transforms.Compose([

                                       torchvision.transforms.ToTensor(),

                                   ])), shuffle=True, batch\_size=batch\_size)

    test\_loader = DataLoader(

        torchvision.datasets.MNIST('./data/', train=False, download=True,

                                   transform=torchvision.transforms.Compose([

                                       torchvision.transforms.ToTensor(),

                                   ])), shuffle=True, batch\_size=batch\_size)

    return train\_loader, test\_loader

if \_\_name\_\_ == '\_\_main\_\_':

    torch.random.manual\_seed(1234)

    np.random.seed(1234)

    epochs = 10

    lr = 0.01

    batch\_size = 32

    optimizer = optimizers.SGD(learning\_rate=lr)

    criterion = loss.CrossEntropy()

    layers = [

        layers.LinearLayer(784, 512),

        layers.ReLU(),

        layers.Dropout(keep\_rate=0.8),

        layers.LinearLayer(512, 512),

        layers.ReLU(),

        layers.Dropout(keep\_rate=0.8),

        layers.LinearLayer(512, 10)

    ]

    model = Model(layers, optimizer, criterion)

    train\_loader, test\_loader = get\_dataset(batch\_size)

    for epoch\_id in range(epochs):

        model.train()

        total = 0

        correct = 0

        for i, (x, y) in enumerate(train\_loader):

            x = x.numpy().reshape(y.shape[0], -1, 1)

            y = y.numpy()

            model.optimizer.zero\_grad()

            loss, pred, logits = model.forward(x, y)

            model.backward(y, logits)

            correct += np.sum(y == pred.flatten())

            total += y.shape[0]

            if i % 100 == 0:

                print("Loss:", loss.mean())

        print("Accuracy (train) epoch {}: {} %".format(epoch\_id + 1, correct / total \* 100.0))

        model.eval()

        total = 0

        correct = 0

        for i, (x, y) in enumerate(test\_loader):

            x = x.numpy().reshape(y.shape[0], -1, 1)

            y = y.numpy()

            \_, pred, \_ = model.forward(x, y)

            correct += np.sum(y == pred.flatten())

            total += y.shape[0]

        print("Accuracy (test) epoch {}: {} %".format(epoch\_id + 1, correct / total \* 100.0))

    total = 0

    correct = 0

    for i, (x, y) in enumerate(train\_loader):

        x = x.numpy().reshape(y.shape[0], -1, 1)

        y = y.numpy()

        \_, pred, \_ = model.forward(x, y)

        correct += np.sum(y == pred.flatten())

        total += y.shape[0]

    print("Accuracy final (train) epoch {}: {} %".format(epochs, correct / total \* 100.0))

***layers.py***

from typing import Tuple, List, Dict, Any

import numpy as np

class Param:

    def \_\_init\_\_(self, data: np.ndarray):

        self.data = data

class Layer:

    def \_\_init\_\_(self, train: bool = True):

        """Creates layer.

        :param train: bool deciding whether the layer is in train/eval mode

        """

        self.train = train

    def forward(self, x: np.ndarray) -> Tuple[np.ndarray, List[Any], Dict[str, Any]]:

        """Forward pass.

        :param x: input of the layer

        :return: output of the layer, \*args and \*\*kwargs as a tuple

        Args and kwargs are passed as arguments for backward pass.

        """

        pass

    def backward(self, x: np.ndarray, dy: np.ndarray, \*args, \*\*kwargs) -> Tuple[np.ndarray, List[np.ndarray]]:

        """Backward pass.

        :param x: the layer input

        :param dy: upstream gradient

        :param args: optional args

        :param kwargs: optional kwargs

        :return: tuple of downstream gradient, then list of gradients with respect to parameters (in order)

        defined in weights method

        """

        pass

    def weights(self) -> List[Param]:

        """Learnable parameters of the layer - order must be the same as in backward."""

        return []

def xavier\_uniform\_init(input\_dim, output\_dim, gain: float = 1.0):

    r = gain \* np.sqrt(6.0 / (input\_dim + output\_dim))

    return np.random.uniform(-r, r, (input\_dim, output\_dim))

class LinearLayer(Layer):

    def \_\_init\_\_(self, input\_dim, output\_dim):

        super().\_\_init\_\_()

        self.W = Param(xavier\_uniform\_init(input\_dim, output\_dim))

        self.b = Param(np.zeros(shape=(1, output\_dim)))

        self.output\_dim = output\_dim

    def forward(self, x: np.ndarray) -> Tuple[np.ndarray, List[Any], Dict[str, Any]]:

        y = (np.matmul(x.transpose((0, 2, 1)), self.W.data) + self.b.data).transpose((0, 2, 1))

        assert y.shape == (x.shape[0], self.output\_dim, 1)

        return y, [], {}

    def backward(self, x: np.ndarray, dy: np.ndarray, \*args, \*\*kwargs) -> Tuple[np.ndarray, List[np.ndarray]]:

        batch\_size = x.shape[0]

        dx = dy.transpose((0, 2, 1)).dot(self.W.data.T).transpose((0, 2, 1))

        assert dx.shape == x.shape

        dW = np.matmul(dy, x.transpose((0, 2, 1))).transpose((0, 2, 1))

        assert dW.shape == (batch\_size, \*self.W.data.shape)

        db = dy.transpose((0, 2, 1))

        assert db.shape == (batch\_size, \*self.b.data.shape)

        return dx, [dW, db]

    def weights(self) -> List[Param]:

        return [self.W, self.b]

class ReLU(Layer):

    def forward(self, x: np.ndarray) -> Tuple[np.ndarray, List[Any], Dict[str, Any]]:

        return np.maximum(x, 0), [], {}

    def backward(self, x: np.ndarray, dy: np.ndarray, \*args, \*\*kwargs) -> Tuple[np.ndarray, List[np.ndarray]]:

        dx = np.maximum(x, 0) \* dy

        assert dy.shape == dx.shape

        return dx, []

class Dropout(Layer):

    def \_\_init\_\_(self, keep\_rate: float):

        super().\_\_init\_\_()

        self.keep\_rate = keep\_rate

    def forward(self, x: np.ndarray) -> Tuple[np.ndarray, List[Any], Dict[str, Any]]:

        mask = (np.random.binomial(1, self.keep\_rate, size=x.shape) / self.keep\_rate

                if self.train else np.ones\_like(x))

        return mask \* x, [], {"mask": mask}

    def backward(self, x: np.ndarray, dy: np.ndarray, \*args, \*\*kwargs) -> Tuple[np.ndarray, List[np.ndarray]]:

        assert "mask" in kwargs

        return kwargs["mask"] \* dy, []

**loss.py**

from typing import Tuple

import numpy as np

def softmax(x):

    e = np.exp(x - np.max(x, axis=1, keepdims=True))

    return e / np.sum(e, axis=1, keepdims=True)

class Loss:

    def forward(self, y\_true: np.ndarray, logits: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:

        """Returns loss value and prediction."""

        pass

    def backward(self, y\_true: np.ndarray, logits: np.ndarray):

        pass

class CrossEntropy(Loss):

    def forward(self, y\_true, logits):

        prob = softmax(logits)

        return - np.log(prob[range(logits.shape[0]), y\_true]), np.argmax(prob, axis=1)

    def backward(self, y\_true, logits):

        grad = softmax(logits)

        grad[range(logits.shape[0]), y\_true] -= 1

        return grad

**model.py**

from typing import List

from layers import Layer

from loss import Loss

from optimizers import Optimizer

class Model:

    def \_\_init\_\_(self, layers: List[Layer], optimizer: Optimizer, criterion: Loss):

        self.layers = layers

        self.optimizer = optimizer

        self.criterion = criterion

    def train(self):

        for l in self.layers:

            l.train = True

    def eval(self):

        for l in self.layers:

            l.train = False

    def forward(self, x, y):

        for idx, layer in enumerate(self.layers):

            new\_x, args, kwargs = layer.forward(x)

            self.optimizer.save(idx, (x, args, kwargs))

            x = new\_x

        logits = x

        loss, pred = self.criterion.forward(y, logits)

        return loss, pred, logits

    def backward(self, y, logits):

        upstream\_grad = self.criterion.backward(y, logits)

        for idx, layer in reversed(list(enumerate(self.layers))):

            x, args, kwargs = self.optimizer.load(idx)

            upstream\_grad, grad = layer.backward(x, upstream\_grad, \*args, \*\*kwargs)

            self.optimizer.update\_layer(grad, layer)

***optimizers.py***

import numpy as np

from layers import Layer, Param

class Optimizer:

    def \_\_init\_\_(self):

        self.cache = {}

    def load(self, idx):

        return self.cache[idx]

    def save(self, idx, x):

        self.cache[idx] = x

    def update\_layer(self, grad: np.ndarray, layer: Layer):

        weights = layer.weights()

        assert len(weights) == len(grad)

        for weight, grad in zip(weights, grad):

            self.step(grad, weight)

    def zero\_grad(self):

        self.cache = {}

    def step(self, grad: np.ndarray, weight: Param):

        pass

class SGD(Optimizer):

    def \_\_init\_\_(self, learning\_rate: float = 0.01):

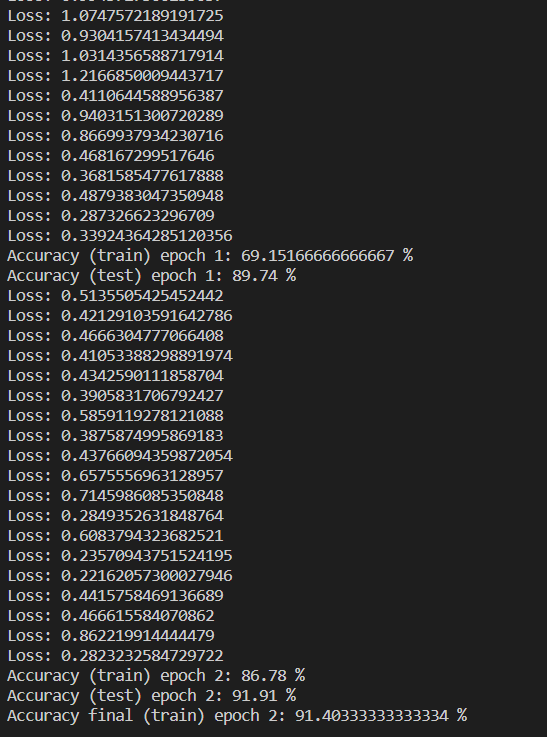
        super().\_\_init\_\_()

        self.learning\_rate = learning\_rate

    def step(self, grad: np.ndarray, weight: Param):

        weight.data -= self.learning\_rate \* grad.mean(axis=0)

***Output:***

******

***Conclusion:***

*Implemented ANN.*