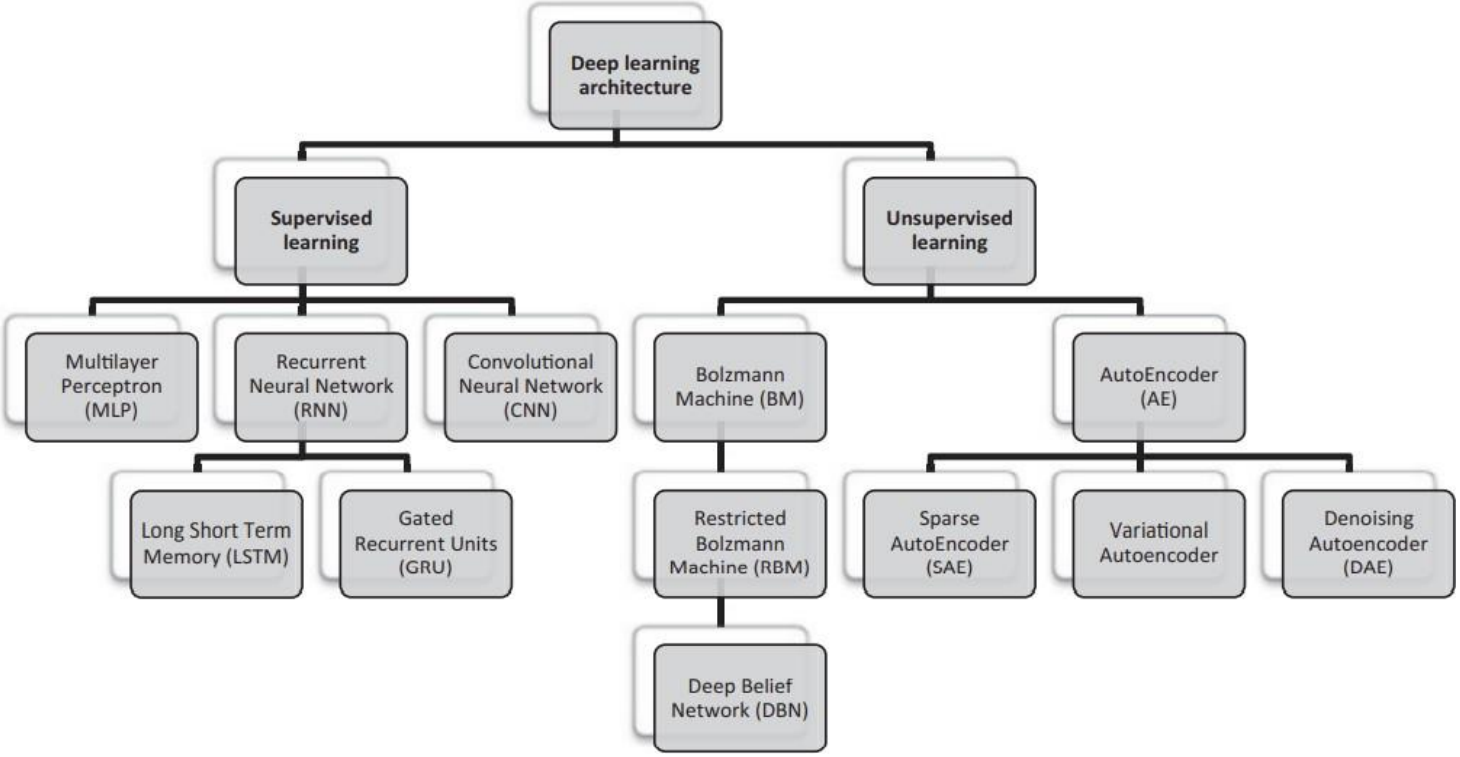
Deep Learning

Deep Learning Architectures

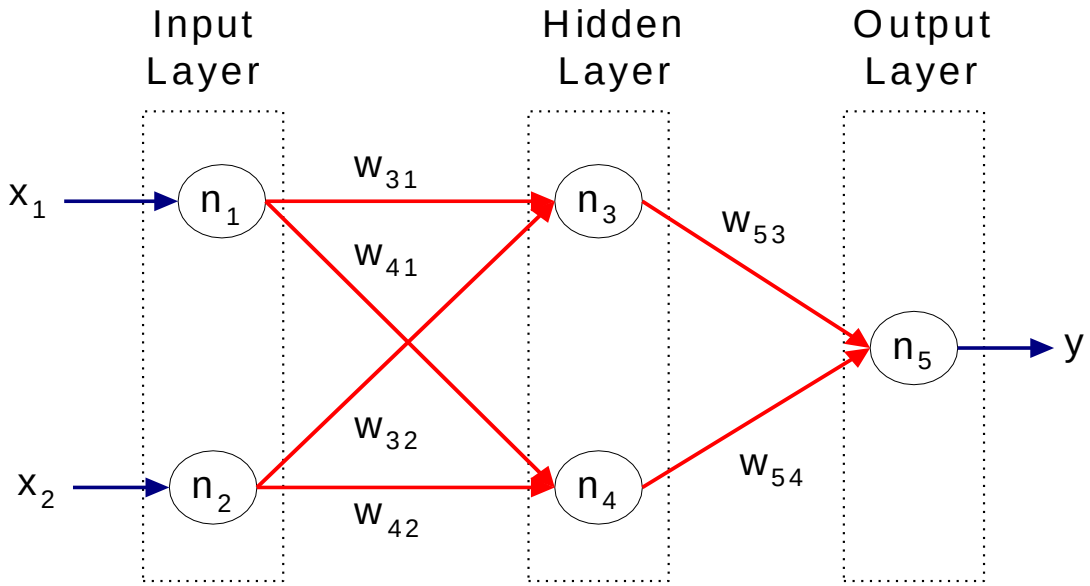
Deep Learning Architectures

- Deep learning algorithms are categorized into supervised & unsupervised techniques



Multi-Layer Perceptron (MLP)

- Created to overcome largest shortcoming of the single layer model
  - i.e., inability to effectively handle data that is not linearly separable
- MLP can solve any type of classification task involving nonlinear decision surfaces



Multi-Layer Perceptron (MLP)

- Multi-layer perception is also known as MLP. It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together so that the outputs of some neurons are inputs of other neurons.
- A multi-layer perceptron has one input layer and for each input, there is one neuron(or node), it has one output layer with a single node for each output and it can have any number of hidden layers and each hidden layer can have any number of nodes.

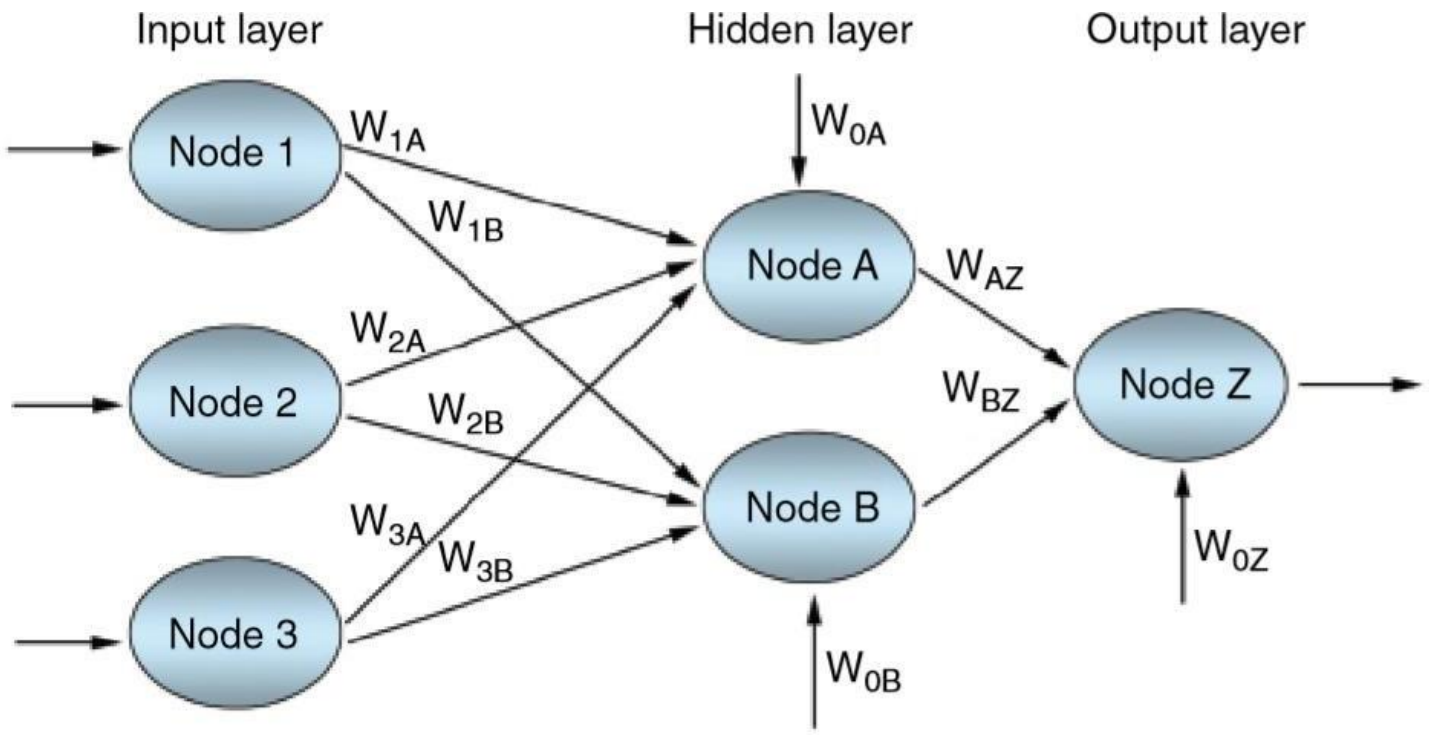
Multi-Layer Perceptron (MLP)

- In the multi-layer perceptron diagram above, we can see that there are three inputs and thus three input nodes and the hidden layer has three nodes. The output layer gives two outputs, therefore there are two output nodes. The nodes in the input layer take input and forward it for further process, in the diagram above the nodes in the input layer forwards their output to each of the three nodes in the hidden layer, and in the same way, the hidden layer processes the information and passes it to the output layer.
- Every node in the multi-layer perception uses a sigmoid activation function. The sigmoid activation function takes real values as input and converts them to numbers between 0 and 1 using the sigmoid formula.

$$(x) = 1/(1 + exp(-x))$$

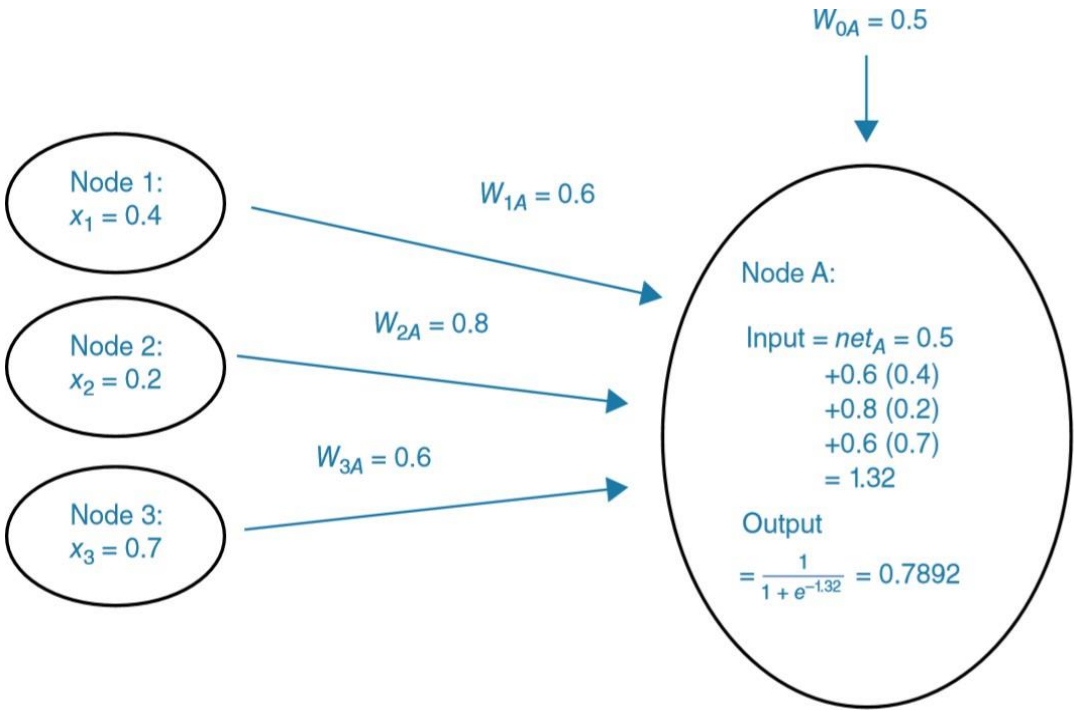
Example: Multi-Layer Perceptron

1 hidden layer, 3 input nodes Neural Network



Weights & Combination Function

- Nodes in the hidden & output layer take values from the input layer & combine using a *combination function* Σ.
- x_{ij} i^{th} input to node j
- W_{ij} weight associated with the i^{th} input to node j.
- X_o a constant input
- $x_1, x_2, \ldots, x_I$ inputs from upstream nodes



$$net_j = \sum_i W_{ij} x_{ij} = W_{0j} x_{0j} + W_{1j} x_{1j} + \cdots + W_{Ij} x_{Ij}$$

- The weights are typically initialized to random values in the range -0.05 to +0.05

Ex.: Weights & Combination Function

Toy Sample Data

$x_0 = 1.0$	$W_{0A} = 0.5$	$W_{0B} = 0.7$	$W_{0Z} = 0.5$
$x_1 = 0.4$	$W_{1A} = 0.6$	$W_{1B} = 0.9$	$W_{AZ} = 0.9$
$x_2 = 0.2$	$W_{2A} = 0.8$	$W_{2B} = 0.8$	$W_{BZ} = 0.9$
$x_3 = 0.7$	$W_{3A} = 0.6$	$W_{3B} = 0.4$	—

For node A in the hidden layer

$$net_A = \sum_i W_{iA} x_{iA} = W_{0A}(1) + W_{1A}x_{1A} + W_{2A}x_{2A} + W_{3A}x_{3A}$$

$$= 0.5 + 0.6(0.4) + 0.80(0.2) + 0.6(0.7) = 1.32$$

Within node A, the combination function *netA = 1.32* is then used as an input to an activation function

Sigmoid Function

- When the combination of inputs to a particular neuron crosses a certain threshold, the neuron is activated
- The most common activation function is the sigmoid function:

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

- *e* is the base of natural logarithms = 2.718281828.
- The activation would take *netA = 1.32 and netB = 1.5* as an input to the sigmoid activation function & produce an output value of

$$y = f(net_A) = \frac{1}{1+e^{-1.32}} = 0.7892 \qquad f(net_B) = \frac{1}{1+e^{-1.5}} = 0.8176.$$

- This would then be passed along the connection to the output *node Z*

Sigmoid Function

$$net_Z = \sum_i W_{iZ} x_{iZ} = W_{0Z}(1) + W_{AZ} x_{AZ} + W_{BZ} x_{BZ}$$
$$= 0.5 + 0.9(0.7892) + 0.9(0.8176) = 1.9461$$

- Finally, *netZ* is input into the sigmoid activation function in *node Z*, resulting in:

$$f(net_Z) = \frac{1}{1 + e^{-1.9461}} = 0.8750$$

- 0.8750 is the output from the Neural Network for the 1st pass through the network**, representing the predicted value** for the target variable.

Implementation Steps

Step 1: Import the necessary libraries.

```
# importing modules
import tensorflow as tf
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Activation
import matplotlib.pyplot as plt
```

Step 2: Download the dataset.

TensorFlow allows us to read the MNIST dataset and we can load it directly in the program as a train and test dataset.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

Implementation Steps

Step 3: Now we will convert the pixels into floating-point values.

```
# Cast the records into float values
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
# normalize image pixel values by dividing by 255
gray_scale = 255
x_train /= gray_scale
x_test /= gray_scale
```

We are converting the pixel values into floating-point values to make the predictions. Changing the numbers into **grayscale** values will be beneficial as the values become small and the computation becomes easier and faster. As the pixel values range from 0 to 256, apart from 0 the range is 255. So dividing all the values by 255 will convert it to range from 0 to 1

Implementation Steps

Step 4: Understand the structure of the dataset

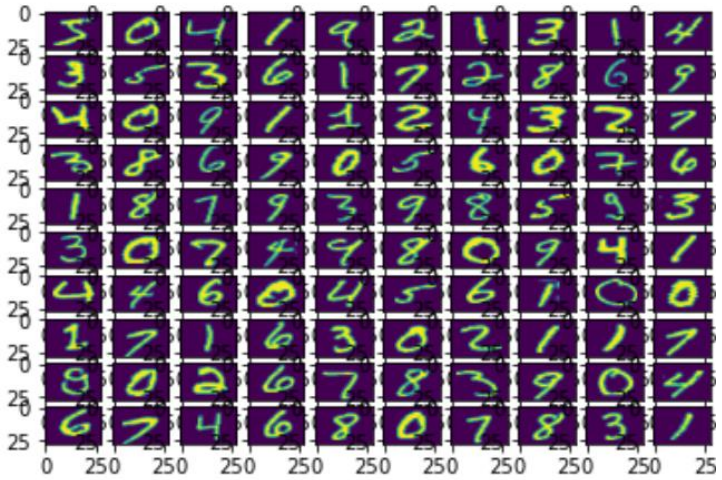
```
print("Feature matrix:", x_train.shape)
print("Target matrix:", x_test.shape)
print("Feature matrix:", y_train.shape)
print("Target matrix:", y_test.shape)
```

Thus we get that we have 60,000 records in the training dataset and 10,000 records in the test dataset and Every image in the dataset is of the size 28×28.

Implementation Steps

Step 5: Visualize the data.

```
fig, ax = plt.subplots(10, 10)
k = 0
for i in range(10):
        for j in range(10):
                ax[i][j].imshow(x_train[k].reshape(28, 28),
                                aspect='auto')
                k += 1
plt.show()
```



Implementation Steps

Step 6: Form the Input, hidden, and output layers.

```
model = Sequential([

        # reshape 28 row * 28 column data to 28*28 rows
        Flatten(input_shape=(28, 28)),

        # dense layer 1
        Dense(256, activation='sigmoid'),

        # dense layer 2
        Dense(128, activation='sigmoid'),

        # output layer
        Dense(10, activation='sigmoid'),
])
```

Implementation Steps

Some important points to note:

- **Sequential model** allows us to create models layer-by-layer as we need in a multi-layer perceptron and is limited to single-input, single-output stacks of layers.

- Flatten flattens the input provided without affecting the batch size. For example, If inputs are shaped (batch_size,) without a feature axis, then flattening adds an extra channel dimension and output shape is (batch_size, 1).

- Activation is for using the sigmoid activation function.

The first two **Dense** layers are used to make a fully connected model and are the hidden layers.

- **last Dense layer** is the output layer which contains 10 neurons that decide which category the image belongs to.

Implementation Steps

Step 7: Compile the model.

```
model.compile(optimizer='adam',
                              loss='sparse_categorical_crossentropy',
                              metrics=['accuracy'])
```

Compile function is used here that involves the use of loss, optimizers, and metrics. Here loss function used is **sparse_categorical_crossentropy**, optimizer used is **adam**.

Step 8: Fit the model.

```
model.fit(x_train, y_train, epochs=10,
                        batch_size=2000,
                        validation_split=0.2)
```

Implementation Steps

Some important points to note:

- **Epochs** tell us the number of times the model will be trained in forwarding and backward passes.

- **Batch Size** represents the number of samples, If it's unspecified, batch_size will default to 32.

- **Validation Split** is a float value between 0 and 1. The model will set apart this fraction of the training data to evaluate the loss and any model metrics at the end of each epoch. (The model will not be trained on this data)

Implementation Steps

Step 9: Find Accuracy of the model.

```
results = model.evaluate(x_test, y_test, verbose = 0)
print('test loss, test acc:', results)
```

We got the accuracy of our model 92% by using **model.evaluate()** on the test samples.

Advantages of MultiLayer Perceptron Neural Network

- MultiLayer Perceptron Neural Networks can easily work with non-linear problems.
- It can handle complex problems while dealing with large datasets.
- Developers use this model to deal with the fitness problem of Neural Networks.
- It has a higher accuracy rate and reduces prediction error by using backpropagation.
- After training the model, the Multilayer Perceptron Neural Network quickly predicts the output.

Disadvantages of MultiLayer Perceptron Neural Network

- This Neural Network consists of large computation, which sometimes increases the overall cost of the model.
- The model will perform well only when it is trained perfectly.
- Due to this model's tight connections, the number of parameters and node redundancy increases.