A Crash Course in Practical ML

Day 2: Introduction to Neural Networks

Gage DeZoort | October 28th, 2025

Course Timeline

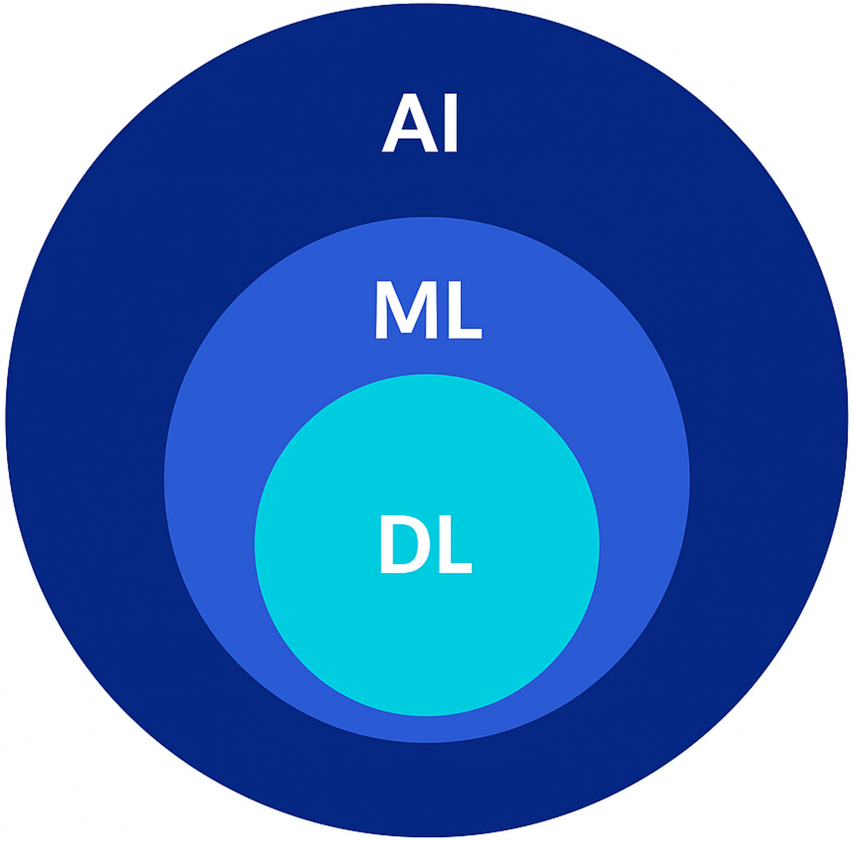
- ~~**Day 1** (10/27, 5pm, ISEB 1010):~~
  ~~- Linear regression, linear classification, clustering with K-means, clustering with DBScan~~

- **Day 2** (10/28, 5pm, Rowland Hall 160 + ISEB 4020)
  - Deep learning, introduction to neural networks (NNs)

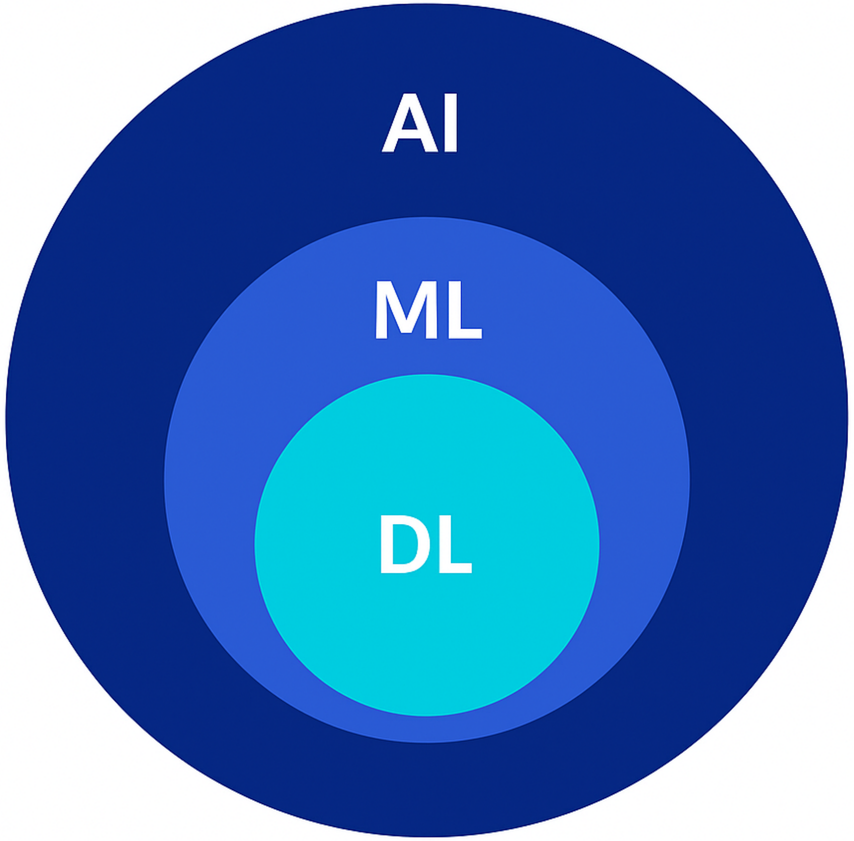
- **Day 3** (10/29, 5pm, ISEB 1010):
  - Convolutional NNs, Graph NNs, transformers and attention

- All materials will be posted in the course's Git repo:
  https://github.com/GageDeZoort/intro_ml_uci/tree/main

Artificial intelligence (AI), the broad endeavor to build systems that reason, plan, perceive, problem-solve, navigate etc.

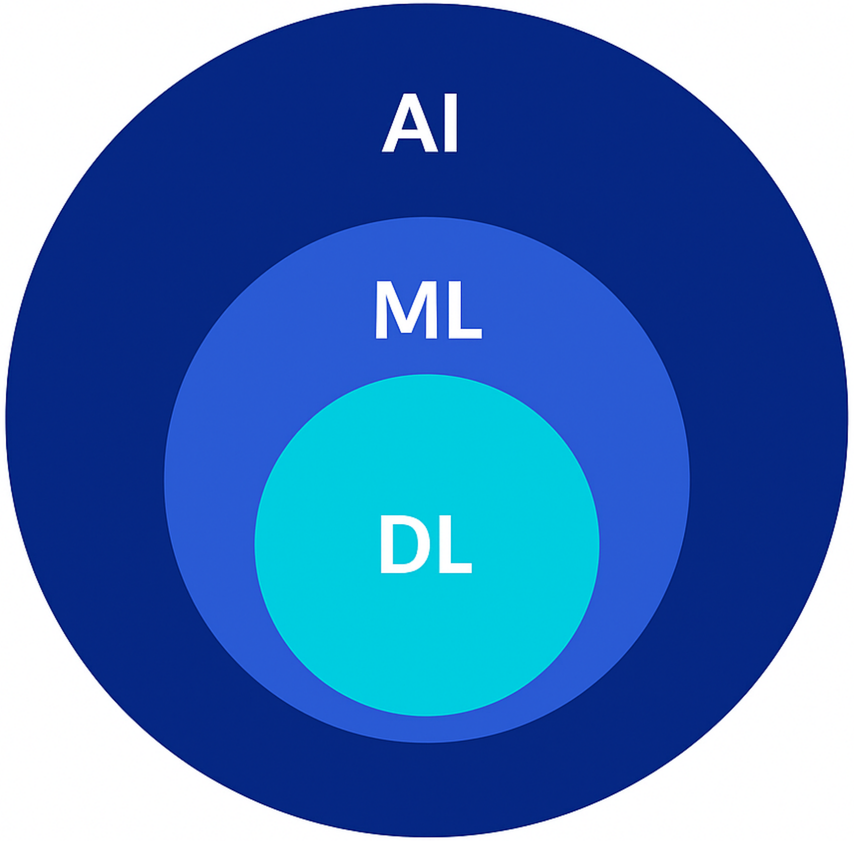


***Machine Learning (ML)*, algorithms that *learn* (without explicit programming) to model relationships in data and use them to make predictions**



Deep Learning (DL), algorithms based on deep neural networks (NNs)

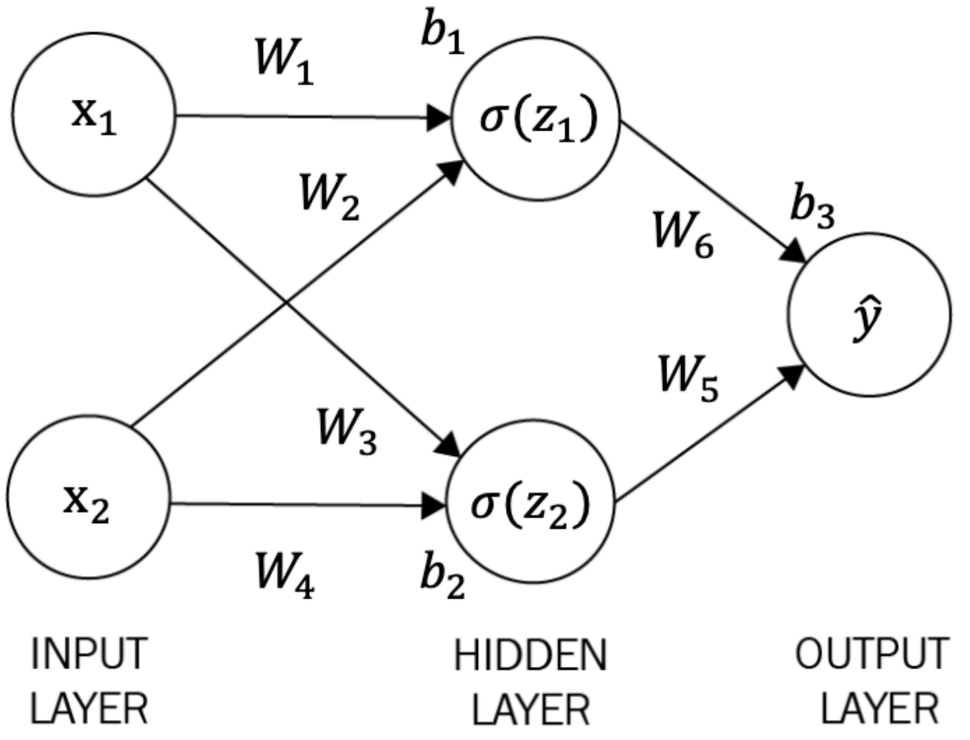
- Large layers of neurons feeding into each other sequentially; large stack of nonlinear transformations
- Flexible, applicable to many domains:
  Convolutional NNs (images)
  Recurrent NNs (sequences)
  Transformers (sequences / sets),
  Graph NNs (graphs)



Toy Example

Classification problem,

$\mathbf{x}_i \in \mathbb{R}^2$ and $y_i \in \{0,1\}$

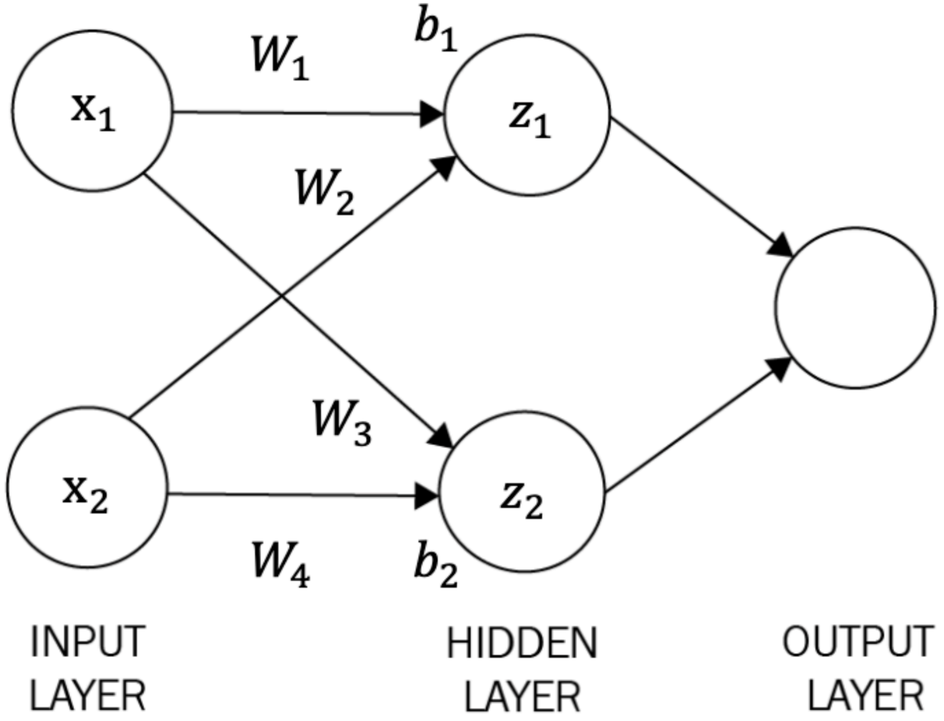


Neural Network:

First, compute **preactivations**

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} W_1 & W_2 \\ W_3 & W_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Linear operation! Weights mix up the inputs, biases add a constant

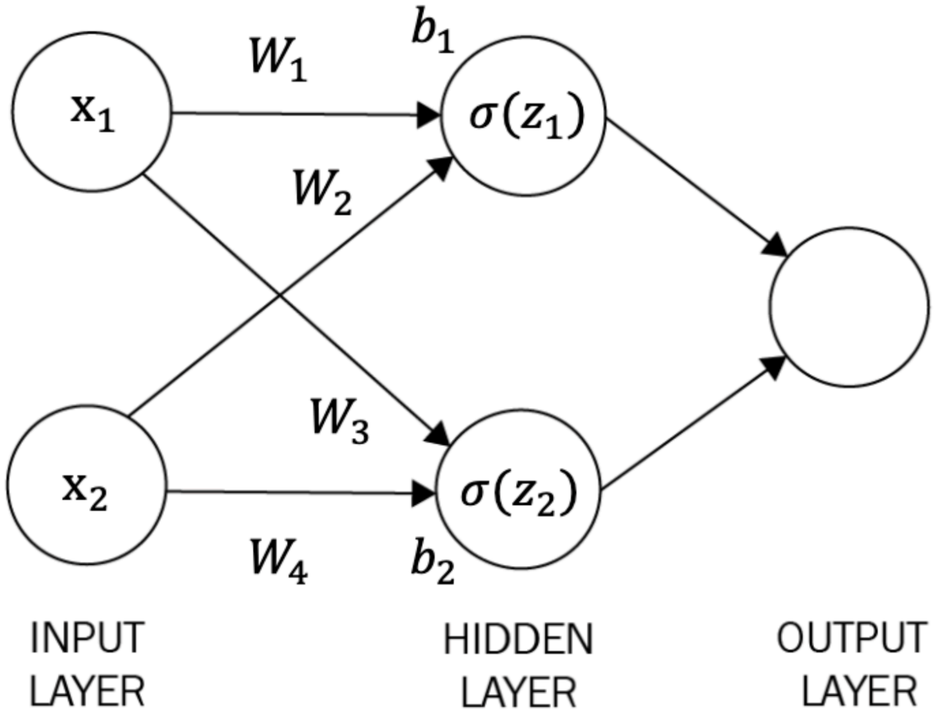


Neural Network:

Next, compute **postactivations** using a *nonlinear* **activation function** $\sigma(\cdot)$

$$\begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix} = \begin{bmatrix} \sigma(z_1) \\ \sigma(z_2) \end{bmatrix}$$

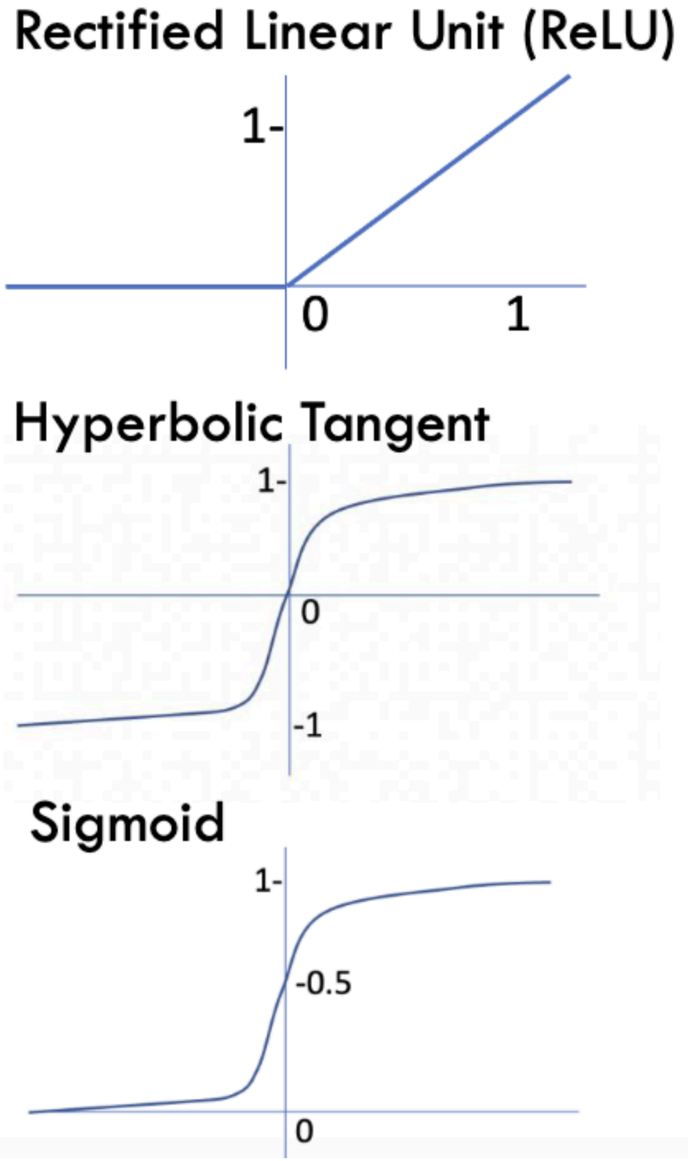
Nonlinear = induces a nonlinear relationship between the weights in the model



Activation Functions

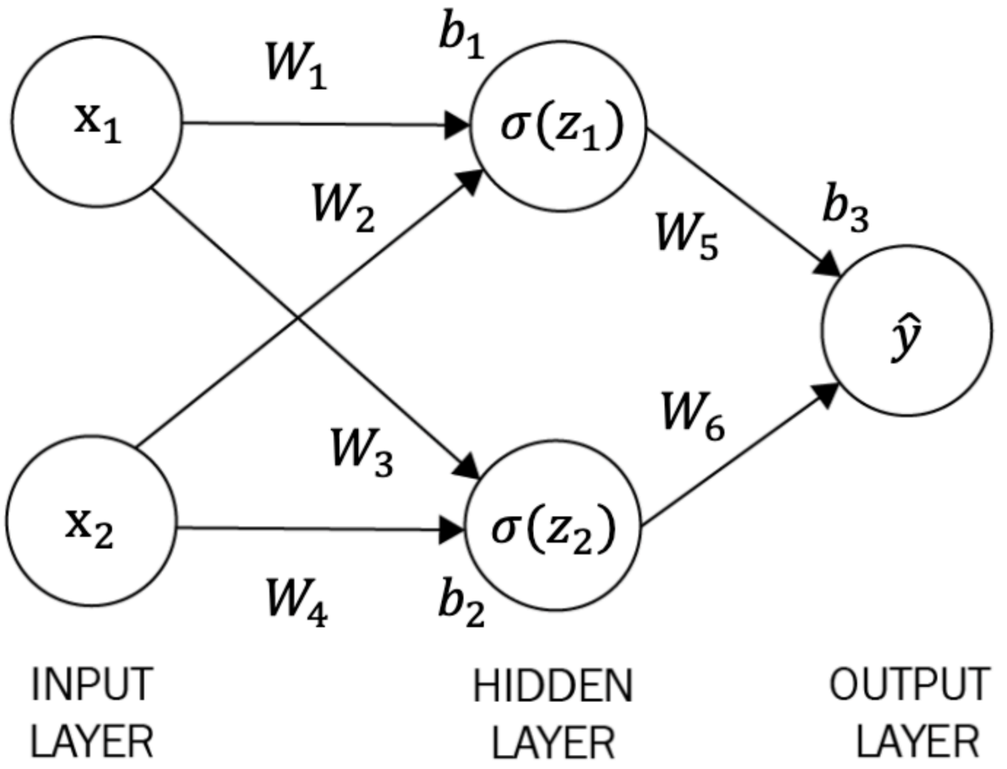
Think about them being “on-off” switches in analogy to firing neurons. The upshot is that they induce a non-linear hypothesis in the forward pass of the neural network.

$$\begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix} = \begin{bmatrix} \sigma(z_1) \\ \sigma(z_2) \end{bmatrix}$$

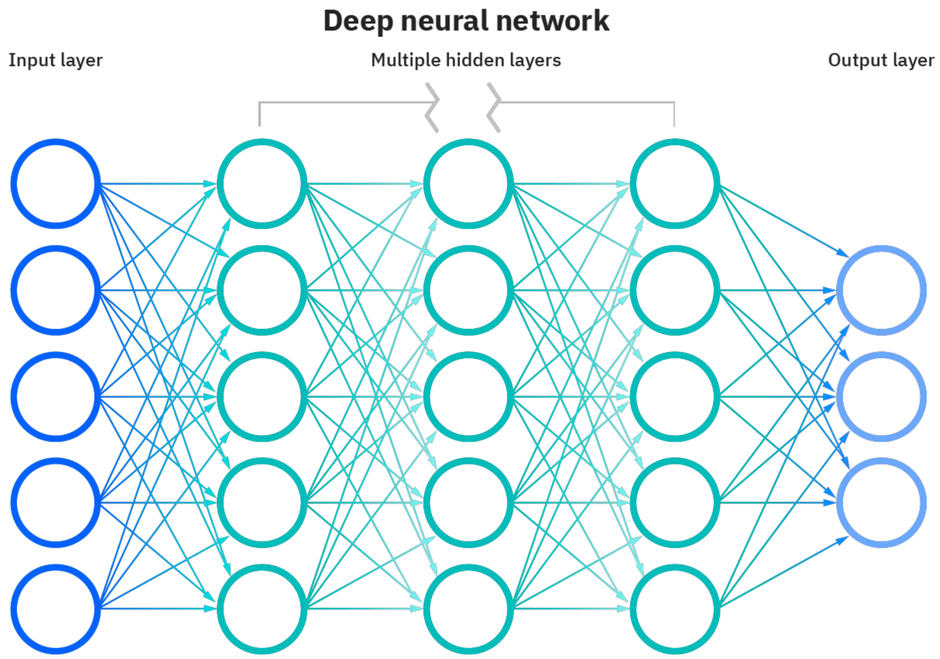


Outputs

$$\hat{y} = \begin{bmatrix} W_5 & W_6 \end{bmatrix} \begin{bmatrix} \sigma(z_1) \\ \sigma(z_2) \end{bmatrix} + b_3$$

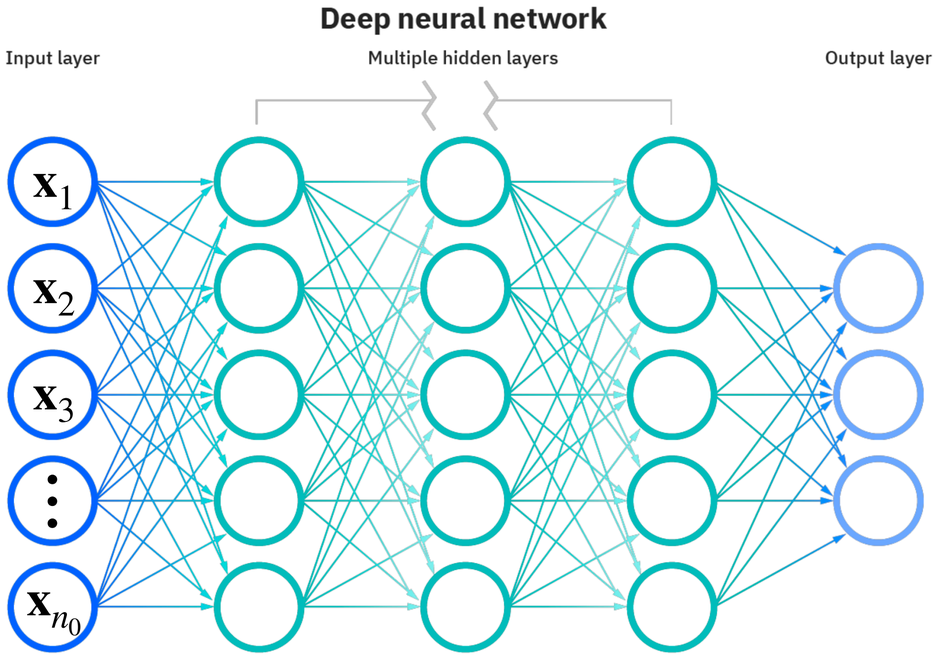


We've just described the **forward pass** of a **deep neural network (DNN).** You may also hear DNNs referred to as **feedforward NNs** or **multilayer perceptrons (MLPs).**

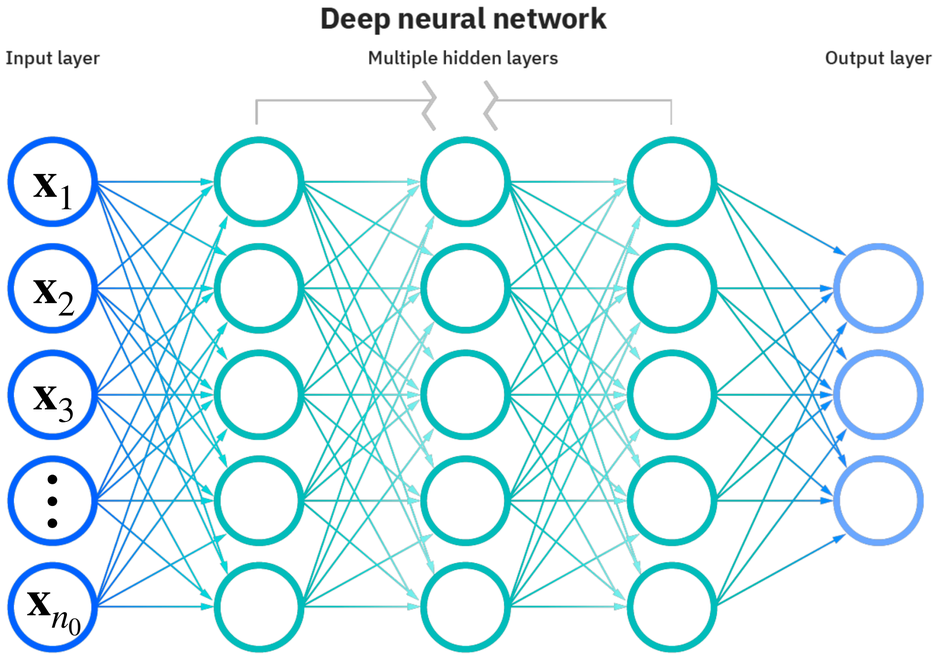


Now we'll describe the forward pass of an arbitrarily-shaped DNN.

Inputs $\mathbf{x} \in \mathbf{R}^{n_0}$

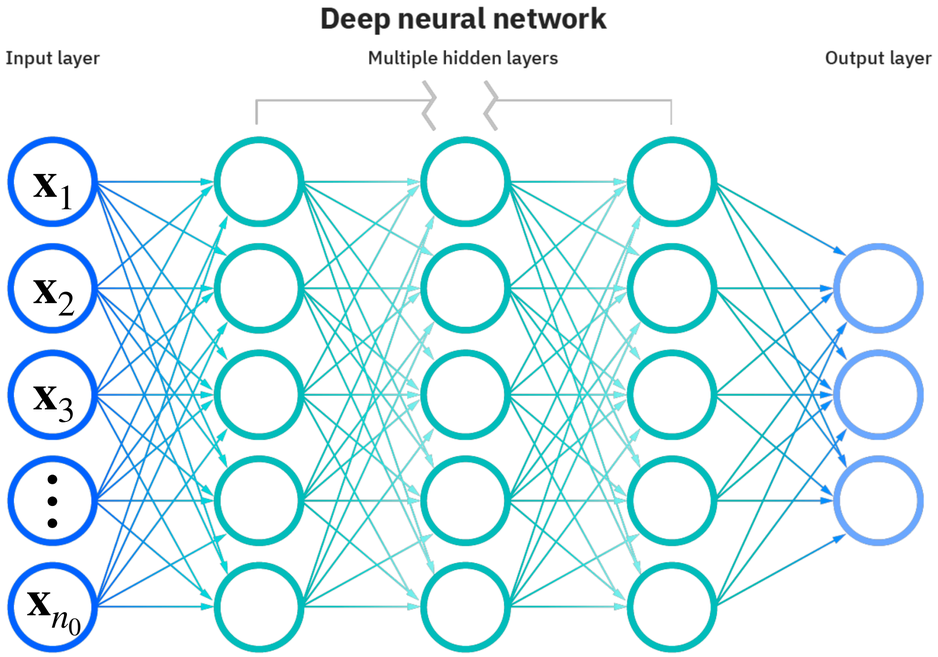


Inputs $\mathbf{x} \in \mathbf{R}^{n_0}$



Hidden layers of **neurons**

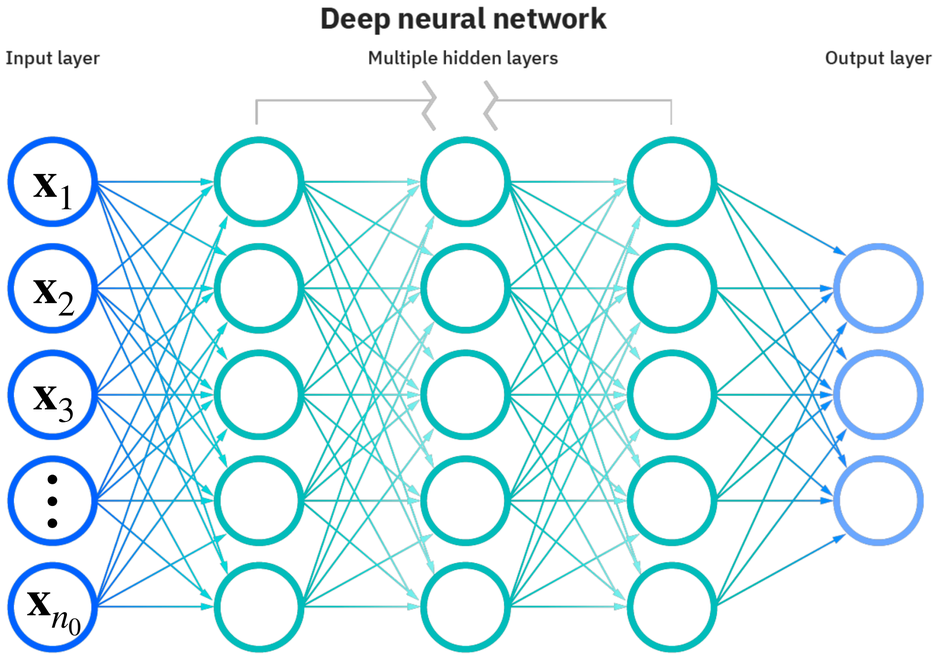
Inputs $\mathbf{x} \in \mathbf{R}^{n_0}$



Hidden layers of **neurons**

The **depth** (L) of the network
is the number of hidden layers

Inputs $\mathbf{x} \in \mathbf{R}^{n_0}$

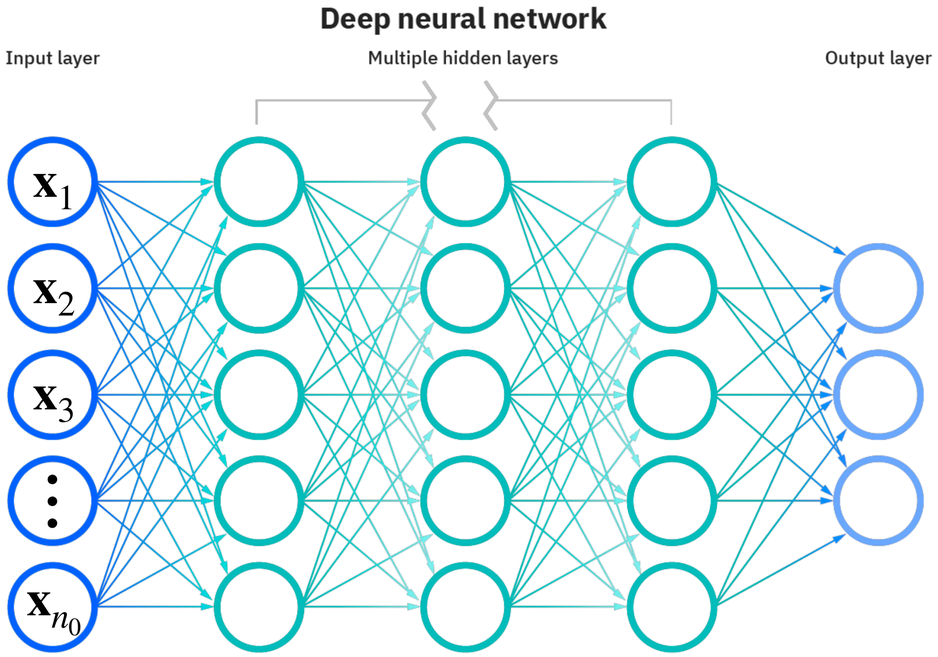


Hidden layers of **neurons**

The **width** (N) of the network is the number of neurons per hidden layer

The **depth** (L) of the network is the number of hidden layers

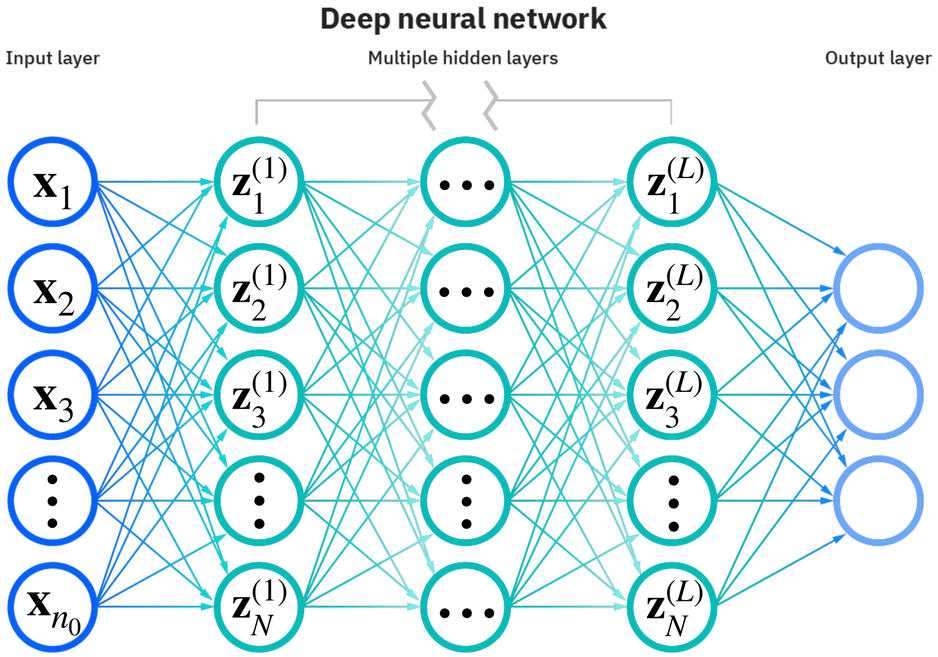
Inputs $\mathbf{x} \in \mathbf{R}^{n_0}$



Hidden layers of **neurons**

This structure is called **feedforward** because each layer of neurons feeds into the next

Inputs $\mathbf{x} \in \mathbf{R}^{n_0}$



Hidden layers of **neurons**

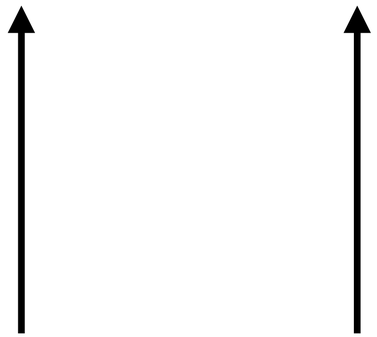
$$\mathbf{z}^{(l+1)} = \mathbf{W}^{(l+1)}\sigma(\mathbf{z}^{(l)}) + \mathbf{b}^{(l+1)} \in \mathbb{R}^N$$

Precisely, the **preactivations** in each layer are a depth wise recursion

Trainable / Learnable Parameters

Weights: $\mathbf{W}^{(\ell+1)} \in \mathbb{R}^{N \times N}$

Biases: $\mathbf{b}^{(\ell+1)} \in \mathbb{R}^{N}$



$$\mathbf{z}^{(l+1)} = \mathbf{W}^{(l+1)} \sigma(\mathbf{z}^{(l)}) + \mathbf{b}^{(l+1)} \in \mathbb{R}^{N}$$

Precisely, the **preactivations** in each layer are a depth wise recursion

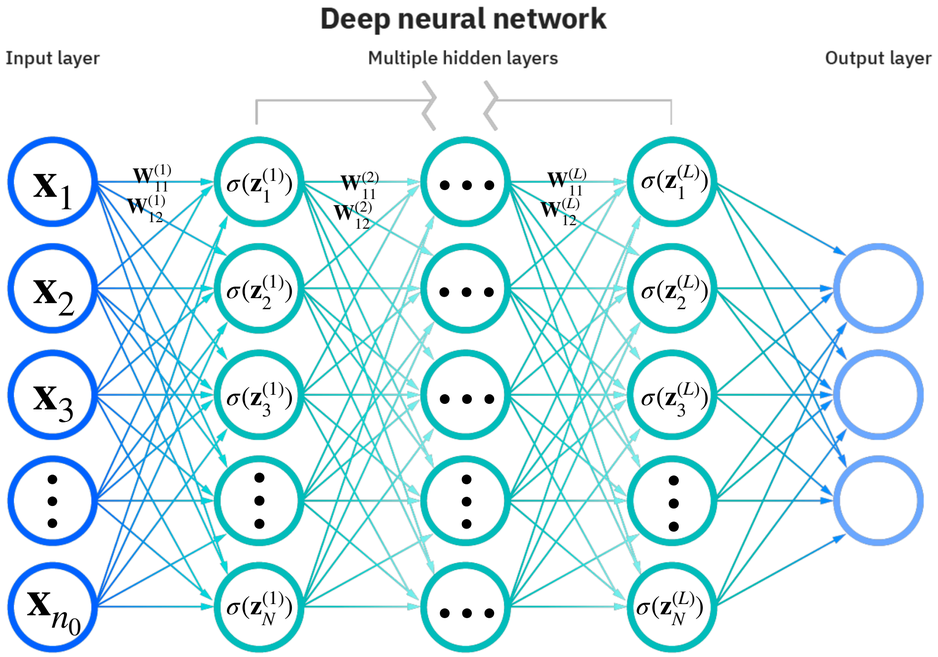
Activation Function (non-linearity)

$\sigma(\cdot)$, applied element-wise to every entry of $\mathbf{z}^{(l)}$

$$\mathbf{z}^{(l+1)} = \mathbf{W}^{(l+1)}\sigma(\mathbf{z}^{(l)}) + \mathbf{b}^{(l+1)} \in \mathbb{R}^N$$

Precisely, the **preactivations** in each layer are a depth wise recursion

Inputs $\mathbf{x} \in \mathbf{R}^{n_0}$

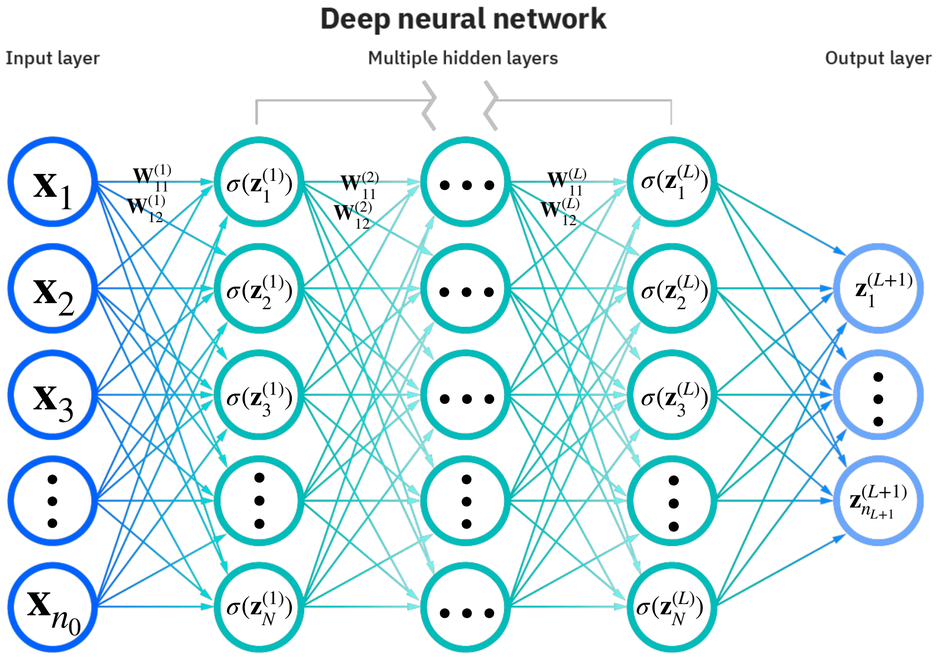


Hidden layers of **neurons**

$$\sigma(\mathbf{z}^{(l)}) = \sigma\left(\mathbf{W}^{(l)}\sigma(\mathbf{z}^{(l-1)}) + \mathbf{b}^{(l)}\right) \in \mathbb{R}^N$$

We can also think about the **postactivations** in every layer

Inputs $\mathbf{x} \in \mathbf{R}^{n_0}$



Hidden layers of **neurons**

$$\sigma(\mathbf{z}^{(l)}) = \sigma\left(\mathbf{W}^{(l)}\sigma(\mathbf{z}^{(l-1)}) + \mathbf{b}^{(l)}\right) \in \mathbb{R}^N$$

We can also think about the **postactivations** in every layer

Outputs

$$\mathbf{z}^{(L+1)} = \mathbf{W}^{(L+1)}\sigma(\mathbf{z}^{(L)}) + \mathbf{b}^{(L+1)}$$

$$\in \mathbb{R}^{n_{L+1}}$$

The Backward Pass

The **learnable parameters** in our model are weights $\mathbf{W}_{ij}^{(l)}$ and biases $\mathbf{b}_{i}^{(l)}$

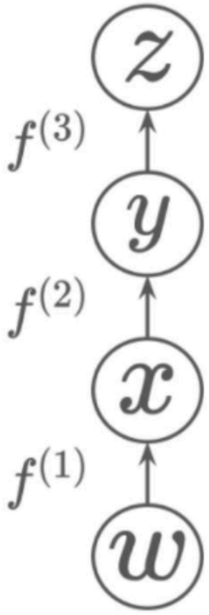
For any learnable parameter θ, the **gradient descent (GD)** updates is

$$\theta_{t+1} \leftarrow \theta_t - \eta \frac{\partial L}{\partial \theta}\Big|_{\theta=\theta_t}$$

where η is a **hyperparameter** called **learning rate.**

The Backward Pass

For a NN, the **gradients** $\partial L / \partial \theta$ can be complicated because the model has many layers. For this reason, we have to do **back propagation,** i.e. apply the chain rule in successive layers:



$$\frac{\partial z}{\partial w}$$

$$= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w}$$

$$= f^{(3)\prime}(y) f^{(2)\prime}(x) f^{(1)\prime}(w)$$

Training Loop Pseudocode

```
for each training epoch:
  for each (batch, labels) in train_data:
    prediction = NN(train_data)
    loss = loss_function(prediction, labels)
    gradients = backpropagate(loss)
    new_params = gradient_descent(gradients, NN.params, lr)
    model.update(new_params)
```

Training Loop Pseudocode

An **epoch** is one iteration through the entire dataset

```
for each training epoch:
  for each (batch, labels) in train_data:
    prediction = NN(train_data)
    loss = loss_function(prediction, labels)
    gradients = backpropagate(loss)
    new_params = gradient_descent(gradients, NN.params, lr)
    model.update(new_params)
```

Training Loop Pseudocode

```
for each training epoch:
  for each (batch, labels) in train_data:
    prediction = NN(train_data)
    loss = loss_function(prediction, labels)
    gradients = backpropagate(loss)
    new_params = gradient_descent(gradients, NN.params, lr)
    model.update(new_params)
```

An **batch** of data is a randomly drawn (without replacement) set of inputs

Training Loop Pseudocode

```
for each training epoch:
  for each (batch, labels) in train_data:
    prediction = NN(train_data)
    loss = loss_function(prediction, labels)
    gradients = backpropagate(loss)
    new_params = gradient_descent(gradients, NN.params, lr)
    model.update(new_params)
```

When we do gradient descent on random batches of data, it is called **stochastic gradient descent (SGD)**

https://github.com/GageDeZoort/intro_ml_uci/tree/main