Introduction to Artificial Neural Networks

CSI 4106 - Fall 2025

Marcel Turcotte
Version: Oct 7, 2025 17:10

Preamble

Message of the Day

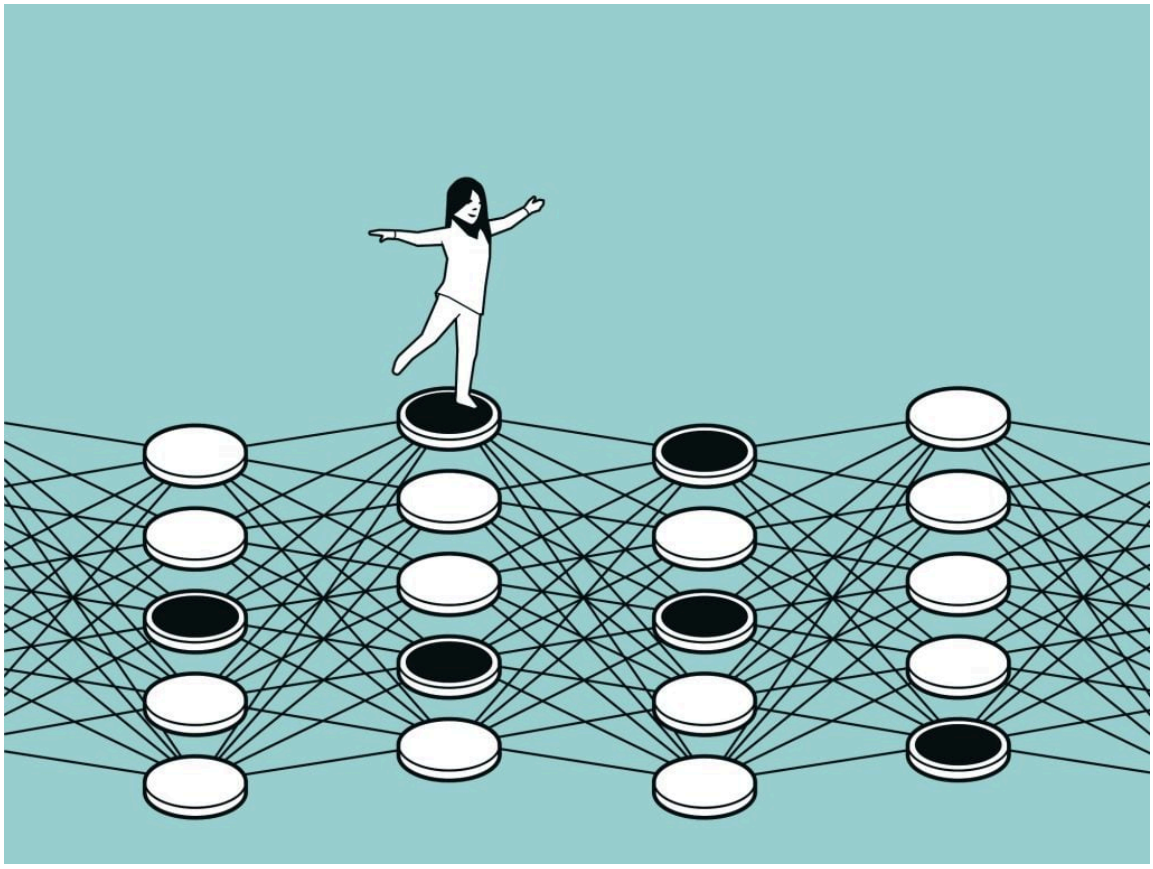
https://www.youtube.com/watch?v=NUAb6zHXqdI

These Numbers Can Make AI Dangerous [Subliminal Learning] by WelchLabs, 2025-09-04.

I appreciate the educational videos produced by WelchLabs, particularly their guest feature on the 3Blue1Brown channel, which elucidates the mechanism by which diffusion models convert text into images. This video assumes foundational knowledge of embeddings and latent space, thus catering to an audience with a background in these concepts.

- But how do AI images and videos actually work? | Guest video by Welch Labs

Message of the Day (2024)



Last year, the timing for this lecture could not have been better since John J. Hopfield and Geoffrey E. Hinton were awarded the 2024 Nobel Prize in Physics "for their fundamental discoveries and inventions enabling machine learning with artificial neural networks."

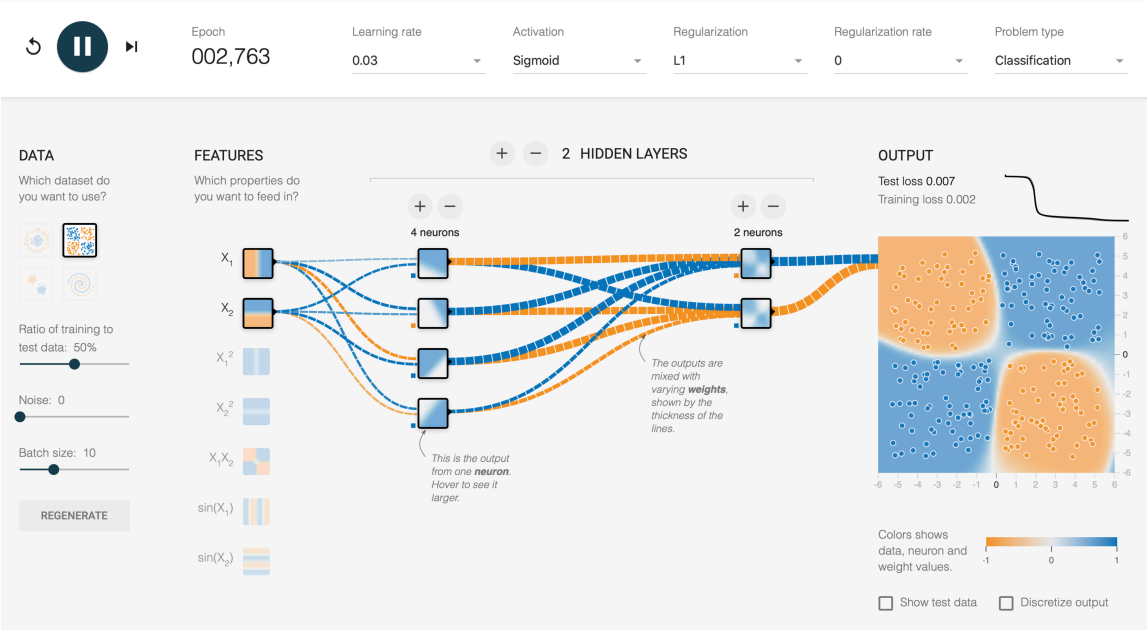
Learning objectives

- **Explain** perceptrons and MLPs: structure, function, history, and limitations.
- **Describe** activation functions: their role in enabling complex pattern learning.
- **Implement** a feedforward neural network with Keras on Fashion-MNIST.
- **Interpret** neural network training and results: visualization and evaluation metrics.
- **Familiarize** with deep learning frameworks: PyTorch, TensorFlow, and Keras for model building and deployment.

As stated at the beginning of this course, there are two primary schools of thought in artificial intelligence: **symbolic AI** and **connectionism**. While the symbolic approach initially dominated the field, the connectionist approach is now more prevalent. We will now focus on **connectionism**.

Introduction

TensorFlow Playground



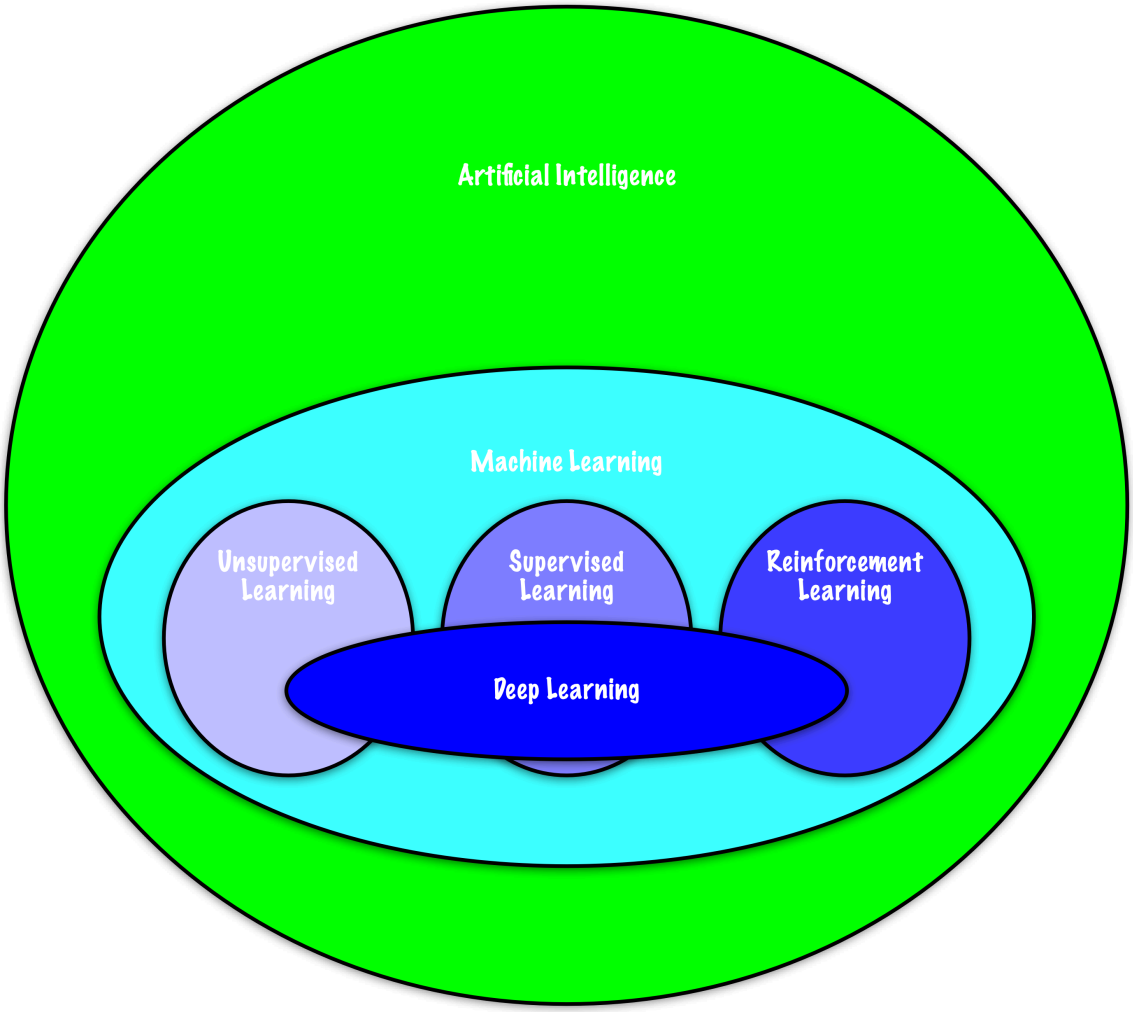
playground.tensorflow.org

Neural Networks (NN)

We now shift our focus to a family of **machine learning models** that draw inspiration from the structure and function of **biological neural networks** found in animals.

AKA **artificial neural networks** or **neural nets**, abbreviated as ANN or NN.

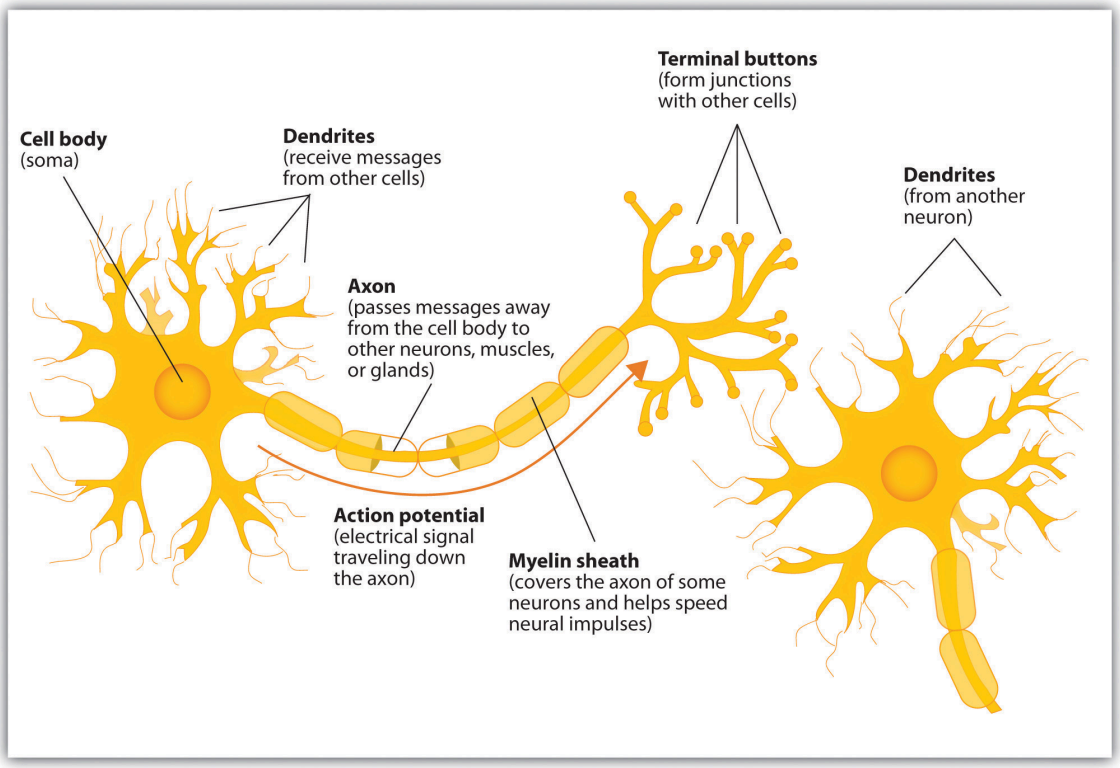
Machine Learning Problems



- **Supervised Learning**: Classification, Regression
- **Unsupervised Learning**: Autoencoders, Self-Supervised
- **Reinforcement Learning**: Now an Integral Component

We will begin our exploration within the framework of supervised learning.

A neuron



Attribution: Jennifer Walinga, CC BY-SA 4.0

In the study of artificial intelligence, it is logical to derive inspiration from the most well-understood form of intelligence: the human brain. The brain is composed of a complex network of neurons, which together form biological neural networks. Although each neuron exhibits relatively simple behavior, it is connected to thousands of other neurons, contributing to the intricate functionality of these networks.

A neuron can be conceptualized as a basic computational unit, and the complexity of brain function arises from the interconnectedness of these units.

Yann LeCun and other researchers have frequently noted that artificial neural networks used in machine learning resemble biological neural networks in much the same way that an airplane's wings resemble those of a bird.

Interconnected neurons

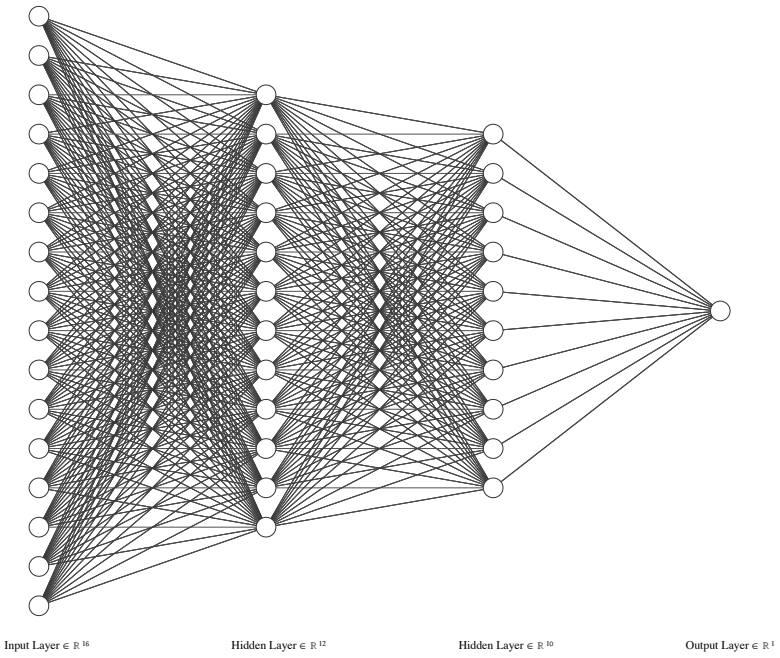
https://youtu.be/uDnHOUPRTYM

Attribution: Molecular Mechanism of Synaptic Function from the Howard Hughes Medical Institute (HHMI). Published on YouTube on 2018-11-15.

From biology, we essentially adopt the concept of simple computational units that are interconnected to form a network, which collectively performs complex computations.

While research into understanding biological neural networks is undeniably important, the field of artificial neural networks has incorporated only a limited number of key concepts from this research.

Connectionist



Attribution: LeNail, (2019). NN-SVG: Publication-Ready Neural Network Architecture Schematics. Journal of Open Source Software, 4(33), 747, https://doi.org/10.21105/joss.00747 (GitHub)

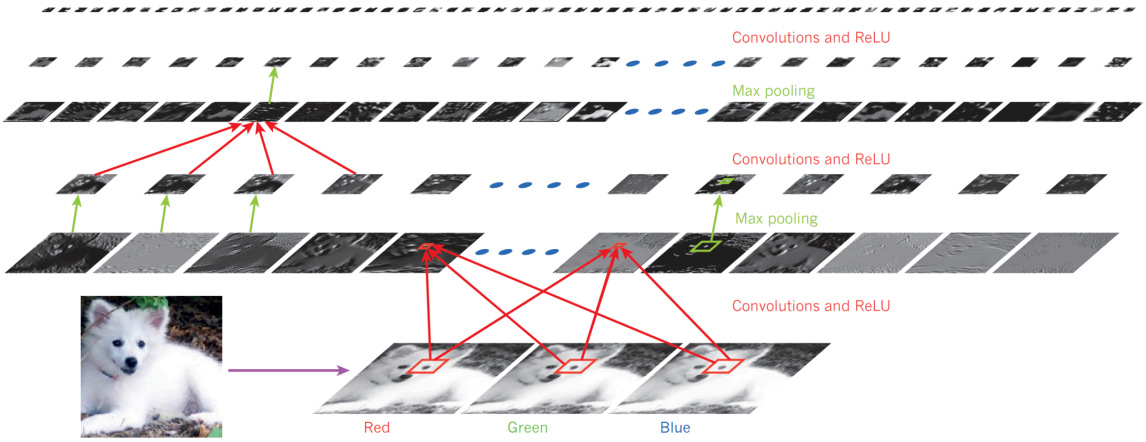
Another characteristic of biological neural networks that we adopt is the organization of neurons into layers, particularly evident in the cerebral cortex.

The term “connectionists” comes from the idea that nodes in these models are interconnected. Instead of being explicitly programmed, these models learn their behavior through training. Deep learning is a connectionist approach.

Neural networks (**NNs**) consist of layers of **interconnected nodes (neurons)**, each **connection** having an associated **weight**.

Neural networks process input data through these weighted connections, and **learning** occurs by **adjusting the weights** based on **errors** in the **training data**.

Hierarchy of concepts



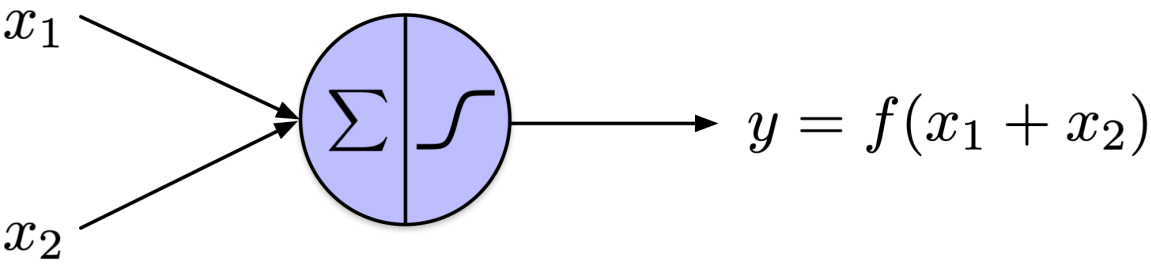
Attribution: LeCun, Bengio, and Hinton (2015)

In the book "Deep Learning" (Goodfellow, Bengio, and Courville 2016), authors Goodfellow, Bengio, and Courville define deep learning as a subset of machine learning that enables computers to "understand the world in terms of a hierarchy of concepts."

This hierarchical approach is one of deep learning's most significant contributions. It reduces the need for manual feature engineering and redirects the focus toward the engineering of neural network architectures.

Basics

Computations with neurodes



where $x_1, x_2 \in \{0, 1\}$ and $f(z)$ is an **indicator function**:

$$f(z) = \begin{cases} 0, & z < \theta \\ 1, & z \geq \theta \end{cases}$$

McCulloch and Pitts (1943) termed artificial neurons, **neurodes**, for **"neuron"** + **"node"**.

In mathematics, $f(z)$, as defined above, is known as an **indicator function** or a **characteristic function**.

These neurodes have one or more binary inputs, taking a value of 0 or 1, and one binary output.

They showed that such units could implement Boolean functions such as **AND**, **OR**, and **NOT**.

But also that networks of such units can compute any logical proposition.

Computations with neurodes

$$y = f(x_1 + x_2) = \begin{cases} 0, & x_1 + x_2 < \theta \\ 1, & x_1 + x_2 \geq \theta \end{cases}$$

- With $\theta = 2$, the neurode implements an **AND** logic gate.
- With $\theta = 1$, the neurode implements an **OR** logic gate.

More **complex logic** can be constructed by multiplying the inputs by **-1**, which is interpreted as **inhibitory**. Namely, this allows building a logical NOT.

With $\theta = 1$, $x_1 1 and x\_2 multiplied by (-1)$,y = $0 when x\_2 = 1,y = 1, if x\_2 = 0$.

$$y = f(x_1 + (-1)x_2) = \begin{cases} 0, & x_1 + x_2 < \theta \\ 1, & x_1 + (-1)x_2 \geq \theta \end{cases}$$

Neurons can be broadly categorized into two primary types: **excitatory** and **inhibitory**.

Computations with neurodes

- **Digital computations** can be broken down into a **sequence of logical operations**, enabling neurode networks to **execute any computation**.
- McCulloch and Pitts (1943) did **not** focus on **learning** parameter θ.
- They introduced a machine that **computes any function** but **cannot learn**.

The period roughly from 1930 to 1950 marked a transformative shift in mathematics toward the formalization of computation. Pioneering work by **Gödel**, **Church**, and **Turing** not only established the theoretical limits and capabilities of computation—with Gödel's **incompleteness theorems**, Church's **λ-calculus and thesis**, and Turing's model of **universal machines**—but also set the stage for later developments in computer science.

McCulloch and Pitts' 1943 model of neural networks was inspired by this early mathematical framework linking computation to aspects of intelligence, prefiguring later research in artificial intelligence.

From this work, we take the idea that networks of such units perform computations. Signal propagates from one end of the network to compute a result.

Perceptron

https://youtu.be/cNxadbrN_aI

The buzz around artificial intelligence isn't a new trend; in fact, it was already generating excitement back in 1958, as highlighted by a quote from the New York Times.

> ** 1958, New York Times, July 8**
>
> The Navy revealed the embryo of an electronic computer today that it expects will be able to **walk**, **talk**, **see**, **write**, **reproduce itself**, and be **conscious** of its existence.

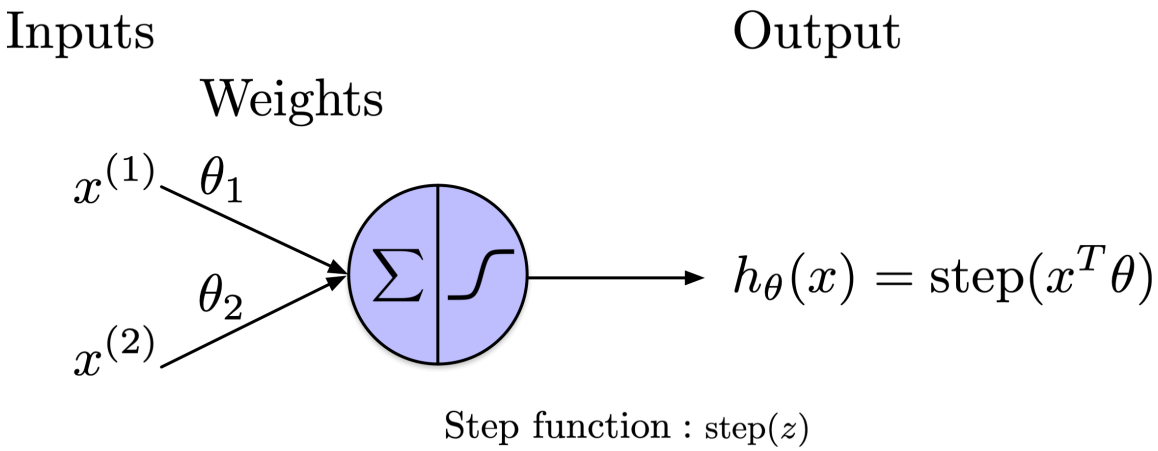
- The Machine That Changed the World - TV Series - 1992

It is observed that artificial neural networks were introduced very early, and as early as the 1960s, they were not considered promising. Additionally, biases in the datasets used for training these networks were already noticeable.

Perceptron

https://www.youtube.com/watch?v=Suevq-kZdIw

Threshold logic unit



$$\text{Weigthed sum} : z = x^T\theta$$

Rosenblatt (1958)

In 1957, Frank Rosenblatt developed a conceptually distinct model of a neuron known as the **threshold logic unit**, which he published in 1958.

In this model, both the inputs and the output of the neuron are represented as **real values**. Notably, each input connection has an associated weight.

The left section of the neuron, denoted by the sigma symbol, represents the computation of a weighted sum of its inputs, expressed as $\theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_D x_D + b$.

This sum is then processed through a step function, right section of the neuron, to generate the output.

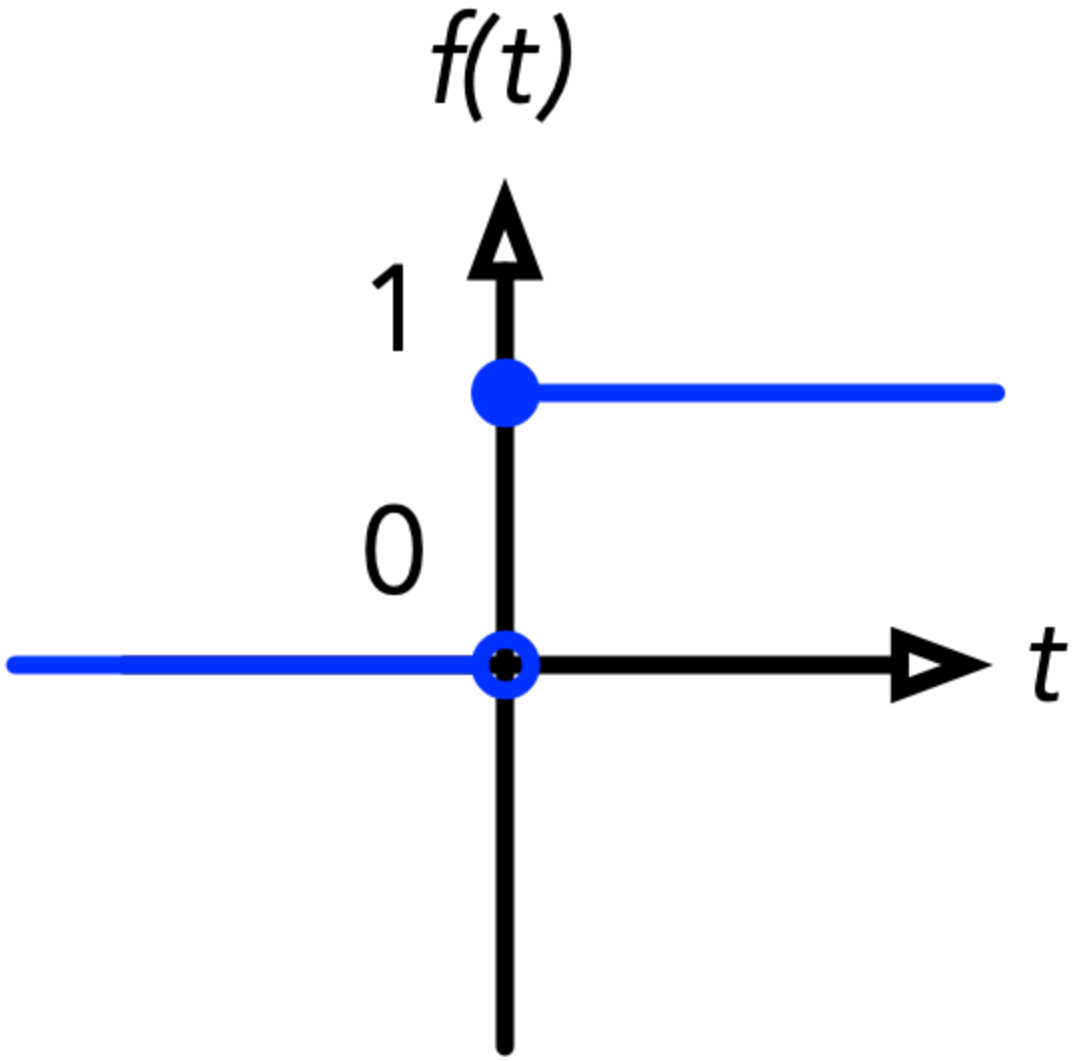
Here, $x^T\theta$ represents the dot product of two vectors: x and θ. Here, x^T denotes the transpose of the vector x, converting it from a row vector to a column vector, allowing the dot product operation to be performed with the vector θ.

The dot product $x^T\theta$ is then a scalar given by:

$$x^T\theta = x^{(1)}\theta_1 + x^{(2)}\theta_2 + \cdots + x_{(D)}\theta_D$$

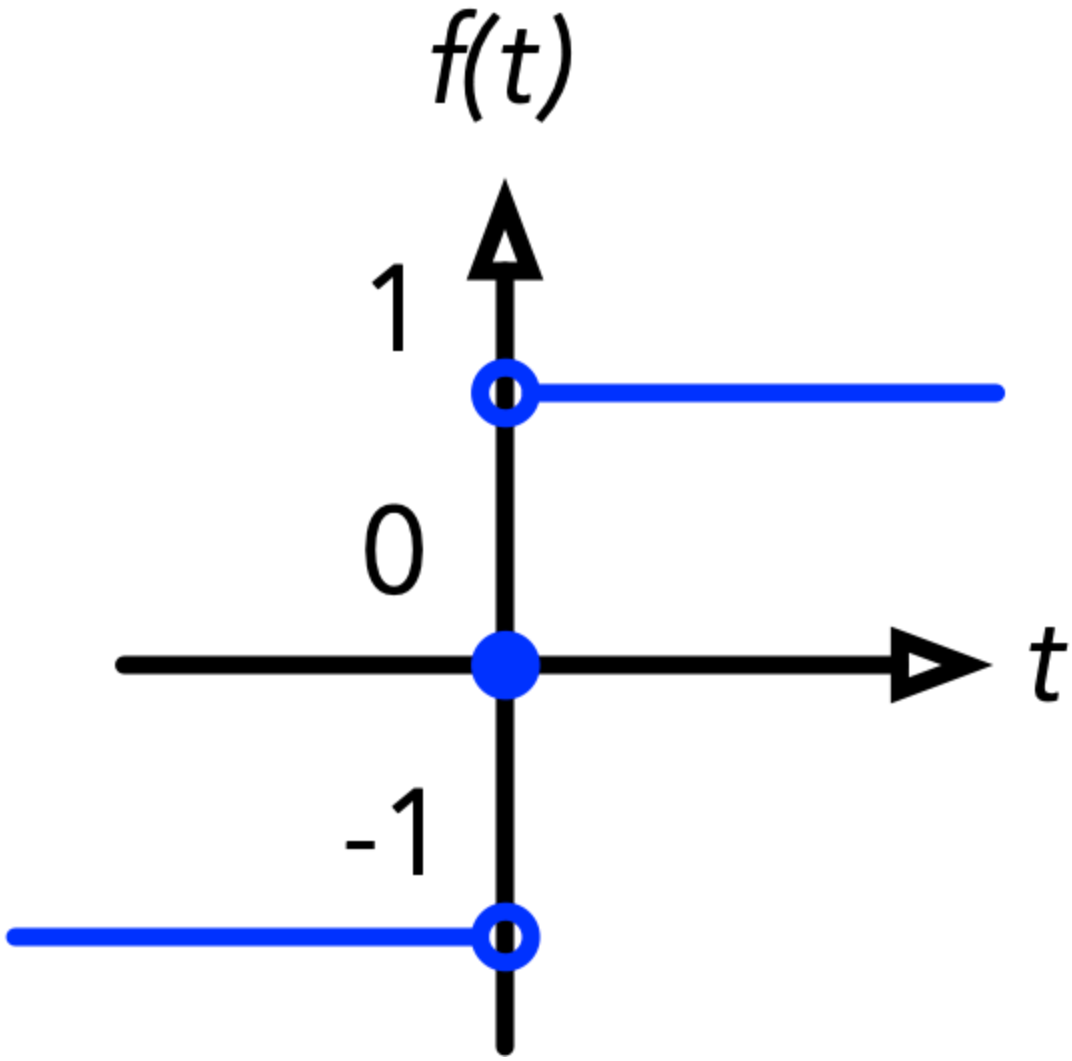
where $x^{(j)}$ and $theta_j$ are the components of the vectors x and θ, respectively.

Simple Step Functions



$\text{heaviside}(t)$ =

- 1, if $t \geq 0$
- 0, if $t < 0$

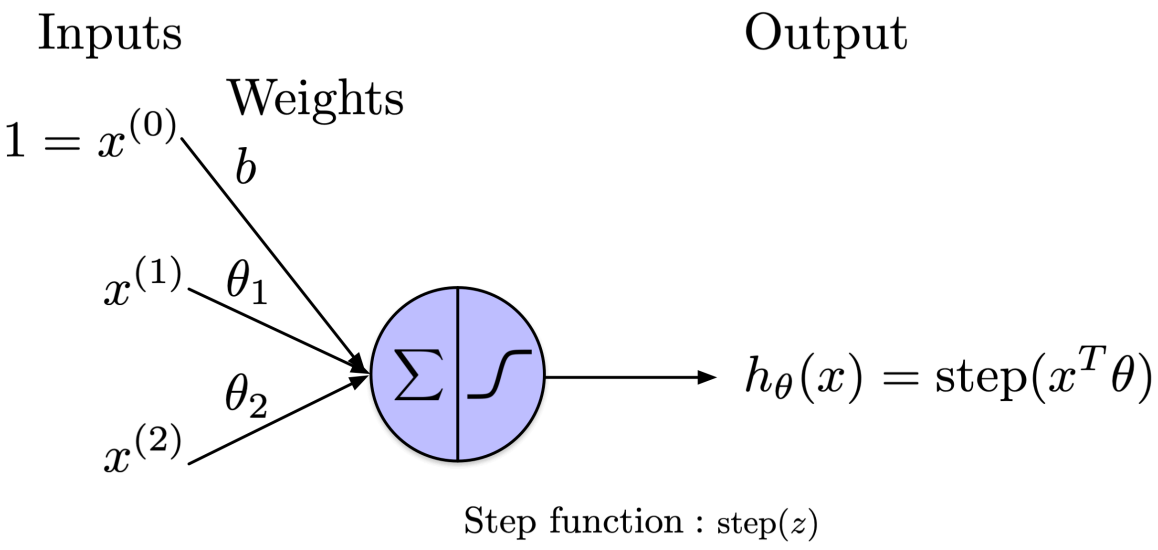


$\text{sign}(t)$ =

- 1, if $t > 0$
- 0, if $t = 0$
- -1, if $t < 0$

Common **step functions** include the **heavyside function** (0 if the input is negative and 1 otherwise) or the **sign** function (-1 if the input is negative, 0 if the input is zero, 1 otherwise).

Notation

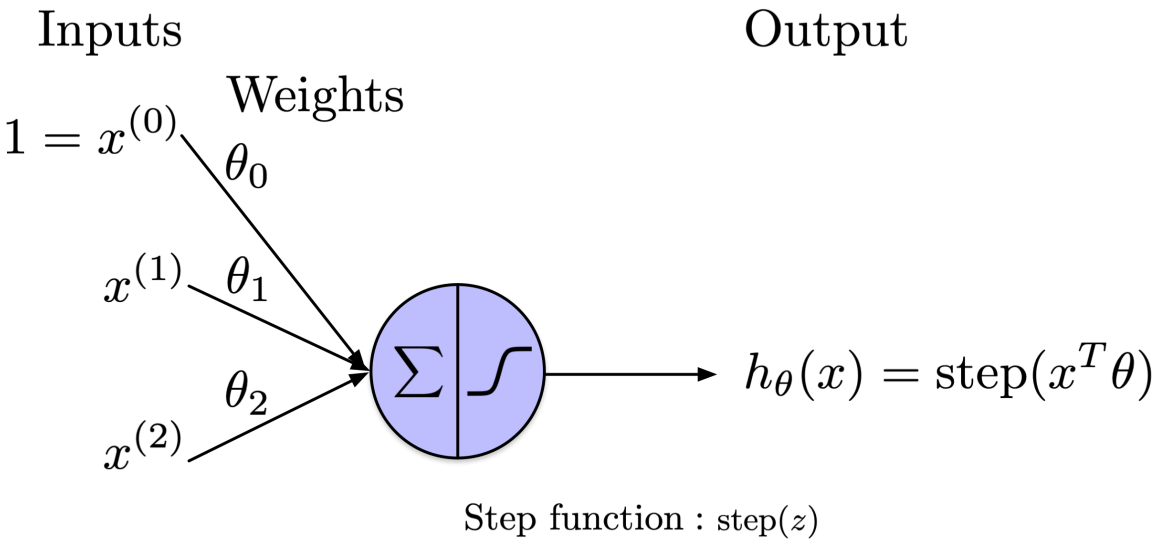


$$\text{Step function} : \text{step}(z)$$

$$\text{Weigthed sum} : z = x^T\theta$$

Add an extra feature with a fixed value of 1 to the input. Associate it with weight $b = \theta_0$, where b is the bias/intercept term.

Notation



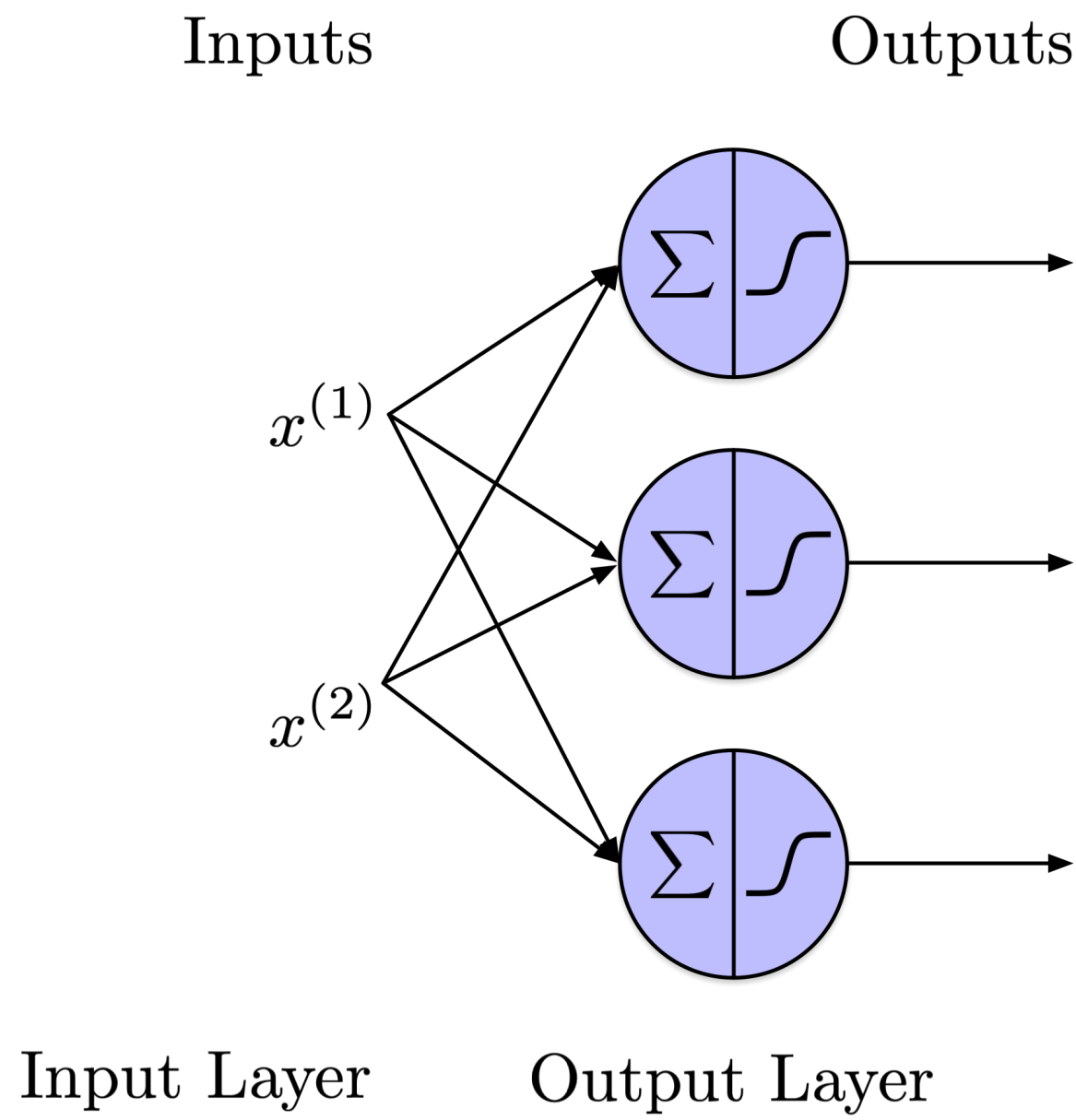
$$\text{Step function} : \text{step}(z)$$

$$\text{Weigthed sum} : z = x^T\theta$$

$\theta_0 = b$ is the bias/intercept term.

The threshold logic unit is analogous to logistic regression, with the primary distinction being the substitution of the logistic (sigmoid) function with a step function. Similar to logistic regression, the perceptron is employed for classification tasks.

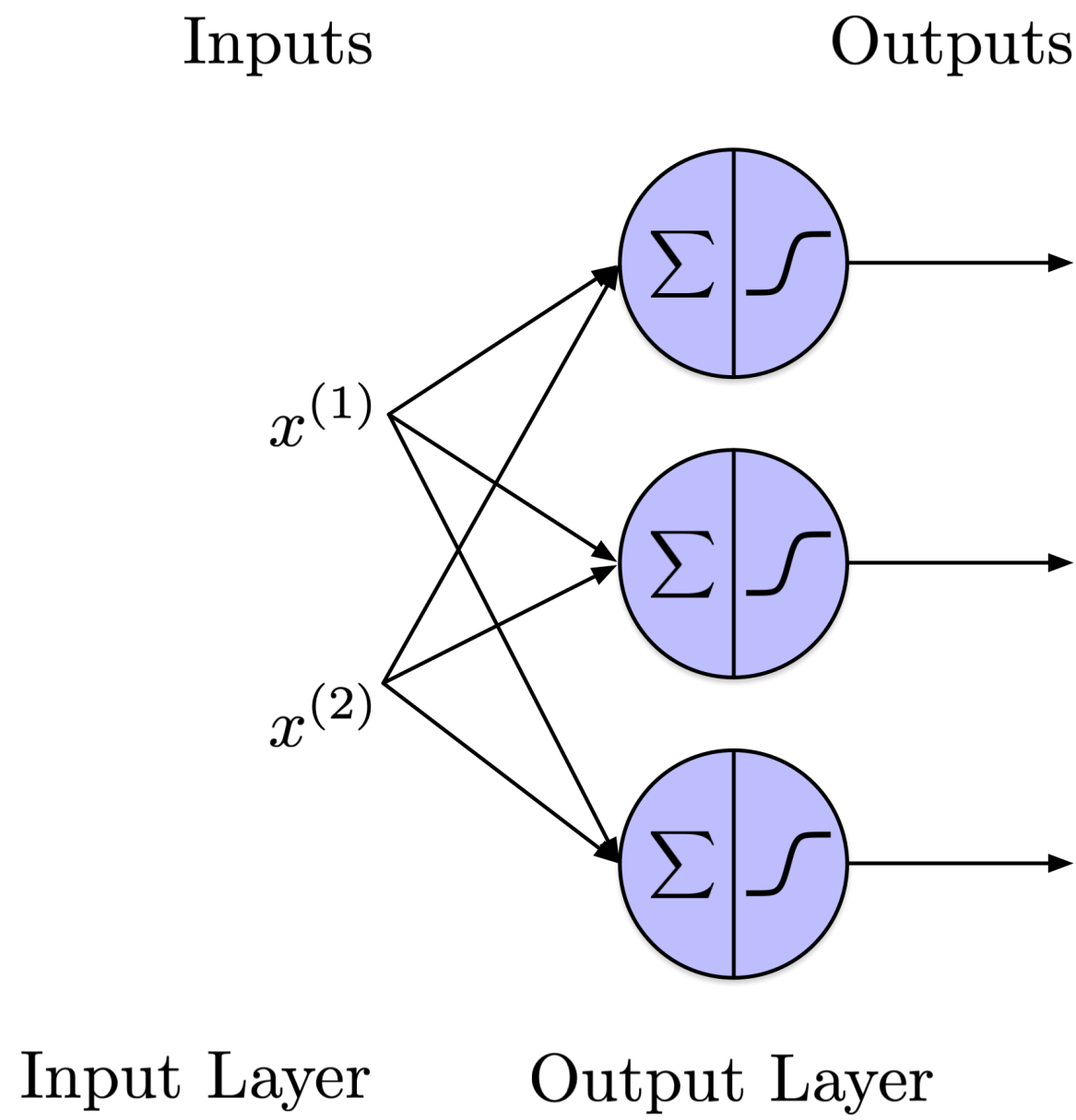
Perceptron



A **perceptron** consists of one or more **threshold logic units** arranged in a **single layer**, with each unit connected to all inputs. This configuration is referred to as **fully connected** or **dense**.

Since the threshold logic units in this single layer also generate the output, it is referred to as the **output layer**.

Perceptron



As this perceptron generates multiple outputs simultaneously, it performs **multiple binary predictions**, making it as a **multilabel classifier** (can also be used as multiclass classifier).

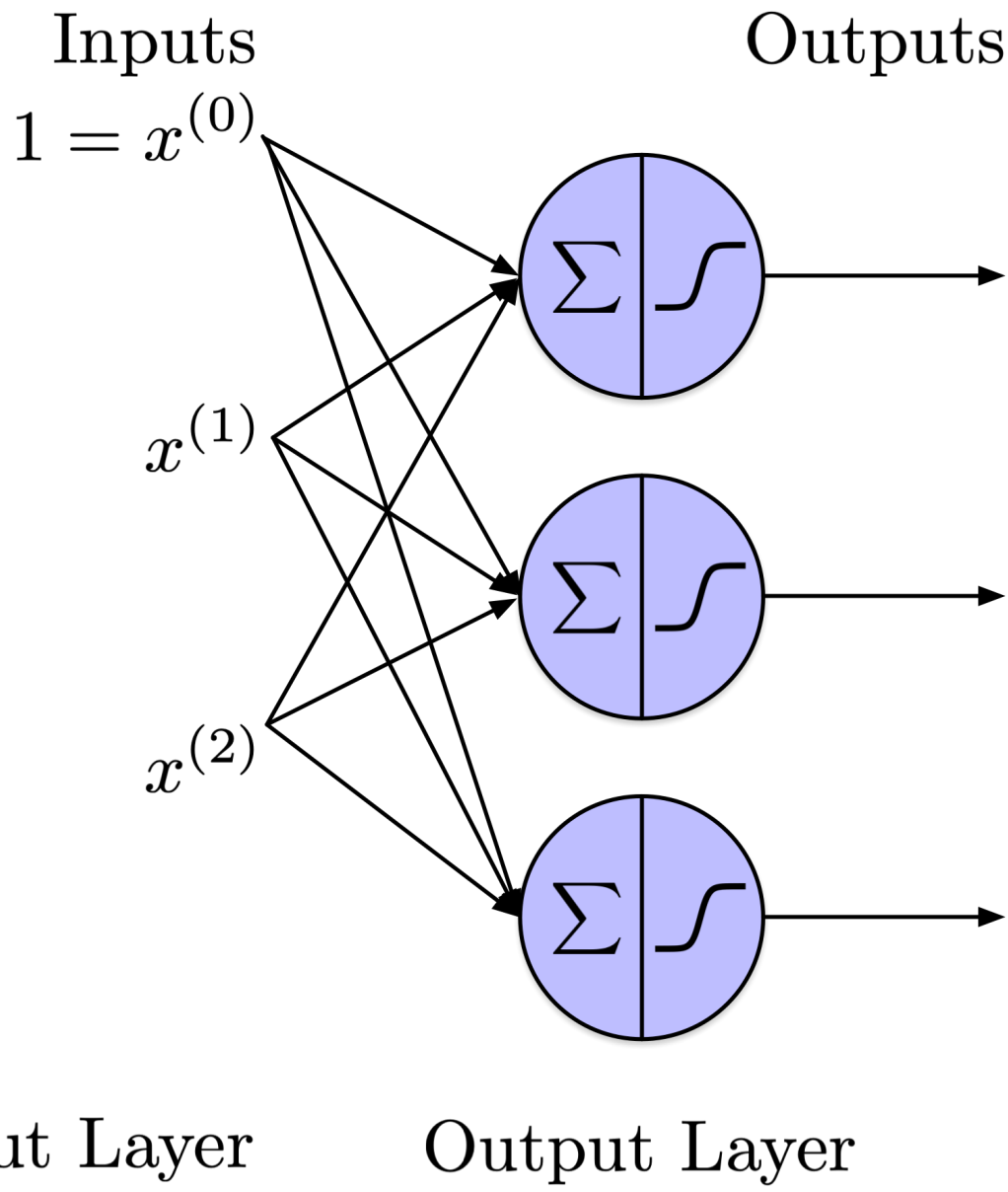
Classification tasks, can be further divided into **multilabel** and **multiclass** classification.

1. **Multiclass Classification:**
    - In multiclass classification, each instance is assigned to one and only one class out of a set of three or more possible classes. The classes are mutually exclusive, meaning that an instance cannot belong to more than one class at the same time.
    - **Example:** Classifying an image as either a cat, dog, or bird. Each image can only belong to one of these categories.
2. **Multilabel Classification:**

- In multilabel classification, each instance can be associated with multiple classes simultaneously. The classes are not mutually exclusive, allowing for the possibility that an instance can belong to several classes at once.
- **Example:** Tagging an image with multiple attributes such as “outdoor,” “sunset,” and “beach.” The image can simultaneously belong to all these labels.

The key difference lies in the relationship between classes: **multiclass classification deals with a single label per instance**, while **multilabel classification handles multiple labels for each instance**.

Notation



As before, introduce an additional feature with a value of 1 to the input. **Assign a bias b to each neuron.** Each incoming connection **implicitly** has an associated weight.

Notation

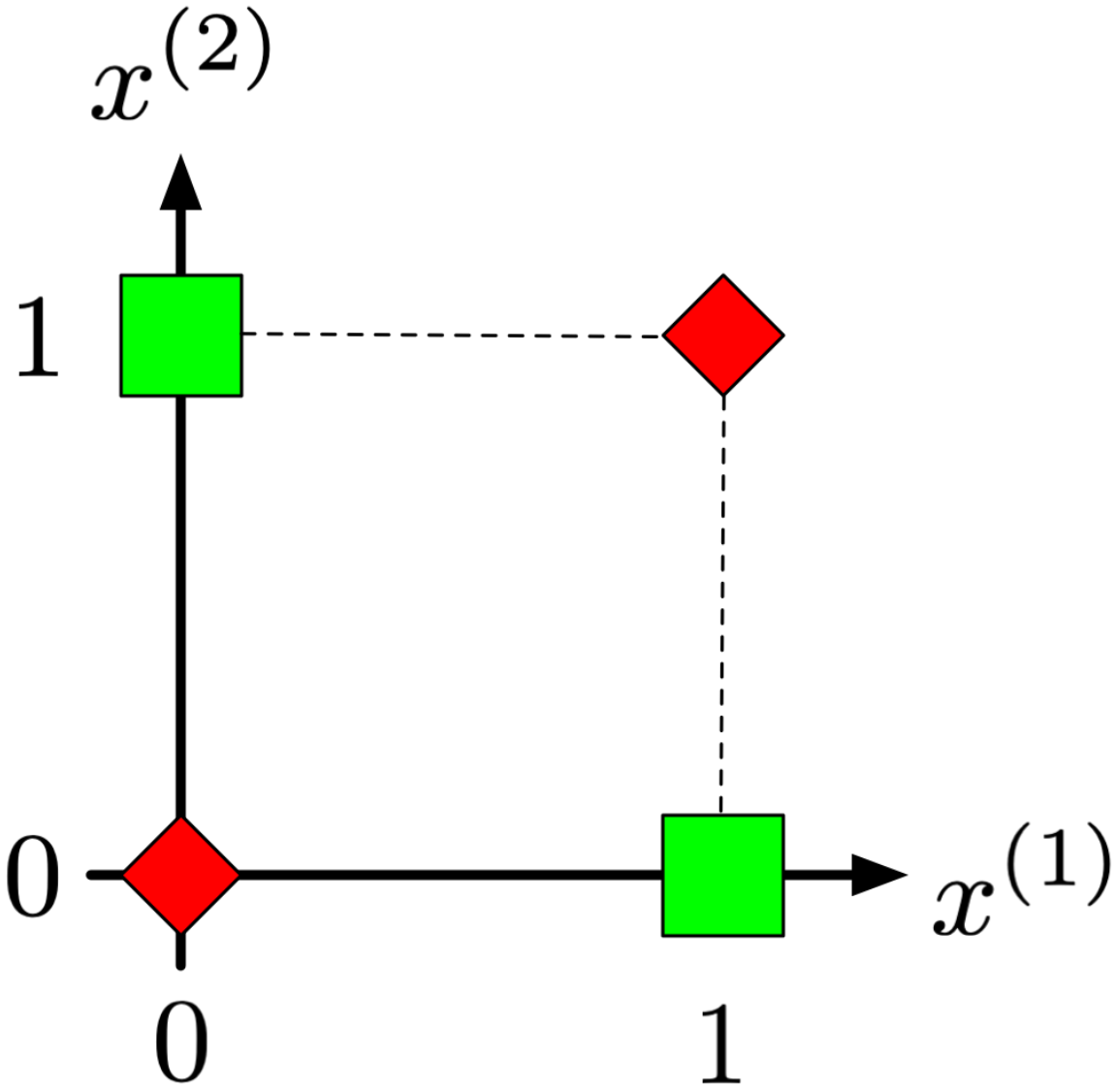
- X is the input **data matrix** where **each row corresponds to an example** and **each column represents one of the D features**.
- W is the **weight matrix**, structured with one **row per input (feature)** and **one column per neuron**.
- **Bias terms** can be represented separately; both approaches appear in the literature. Here, b is a vector with a **length equal to the number of neurons**.

With neural networks, the **parameters** of the model are often reffered to as w (vector) or W (matrix), rather than θ.

Discussion

- The algorithm to **train** the perceptron closely resembles stochastic gradient descent.
    - In the **interest of time** and to **avoid confusion**, we will skip this algorithm and focus on **multilayer perception** (MLP) and its training algorithm, **backpropagation**.

Historical Note and Justification

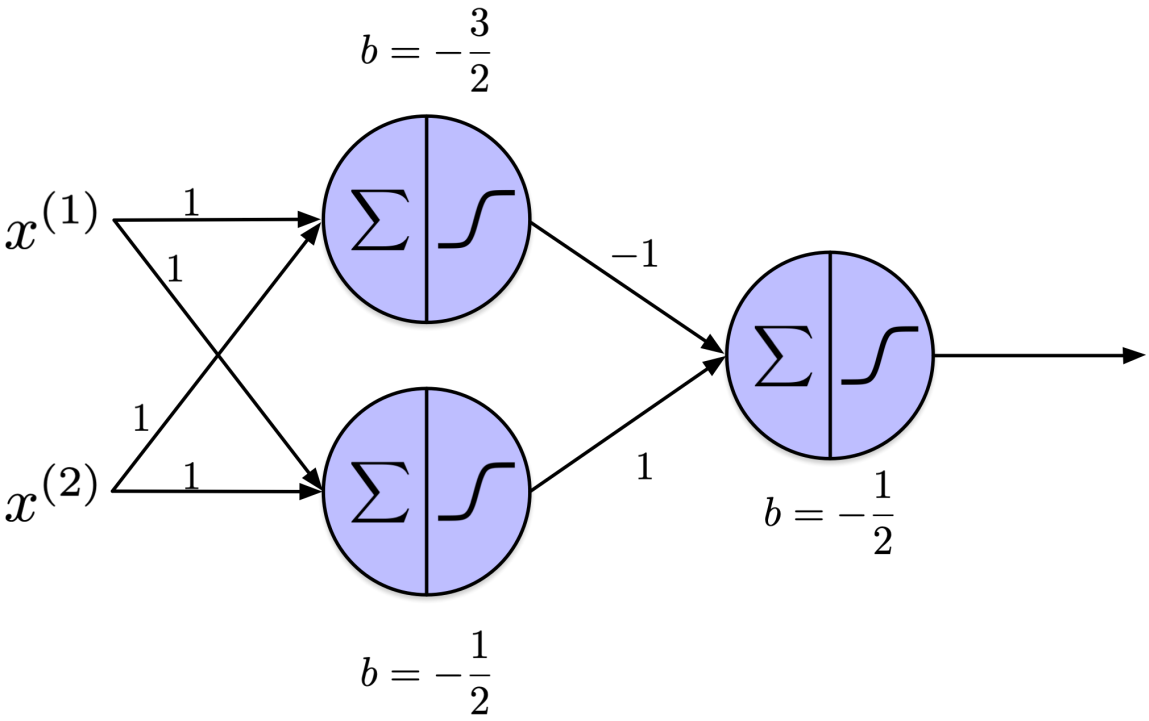


Minsky and Papert (1969) demonstrated the limitations of perceptrons, notably their inability to solve **exclusive OR** (XOR) classification problems:
$([0, 1], \text{true}), ([1, 0], \text{true}), ([0, 0], \text{false}), ([1, 1], \text{false})$.

This limitation also applies to other linear classifiers, such as logistic regression.

Consequently, due to these limitations and a lack of practical applications, some researchers abandoned the perceptron.

Multilayer Perceptron



A **multilayer perceptron** (MLP) includes an input layer and one or more layers of threshold logic units. Layers that are neither input nor output are termed **hidden layers**.

XOR Classification problem

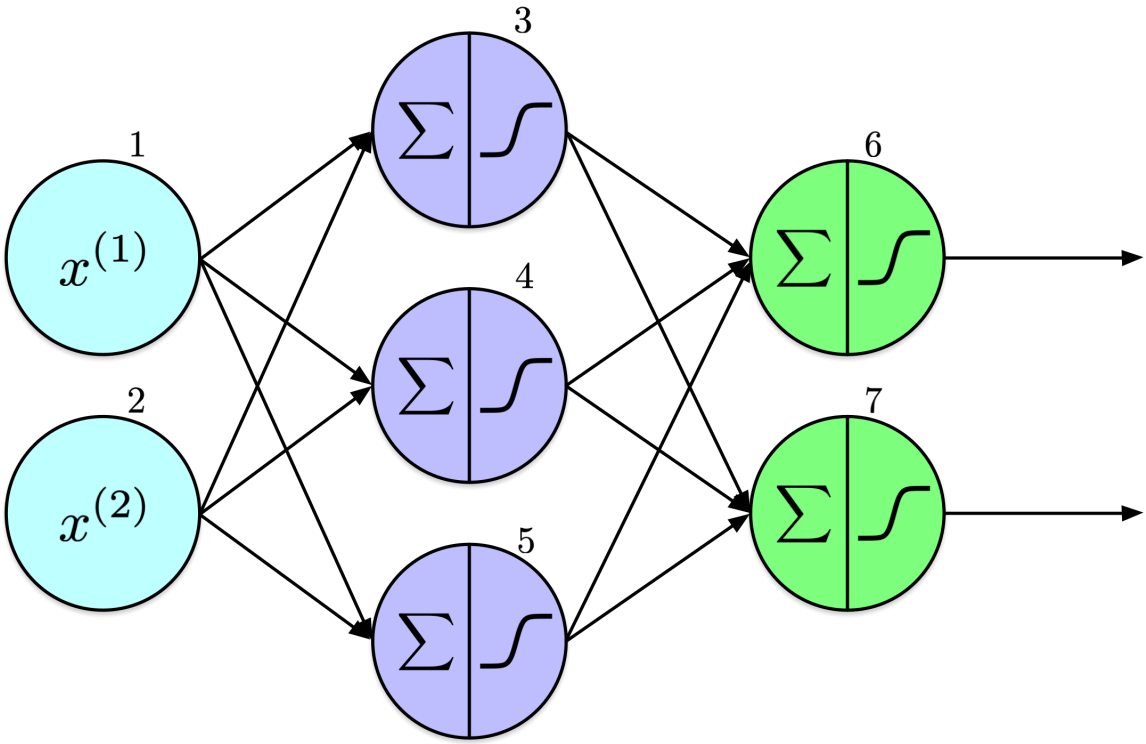
$x^{(1)}$	$x^{(2)}$	y	o_1	o_2	o_3
1	0	1	0	1	1
0	1	1	0	1	1
0	0	0	0	0	0
1	1	0	1	1	0

$x^{(1)}$ and $x^{(2)}$ are two attributes, y is the target, o_1, o_2, and $o_3 = h_\theta(x)$, are the output of the top left, bottom left, and right threshold units. Clearly $h_\theta(x) = y, \forall x \in X$. The challenge during Rosenblatt's time was the lack of algorithms to train multi-layer networks.

I developed an Excel spreadsheet to verify that the proposed multilayer perceptron effectively solves the XOR classification problem.

The step function used in the above model is the heavyside function.

Feedforward Neural Network (FNN)



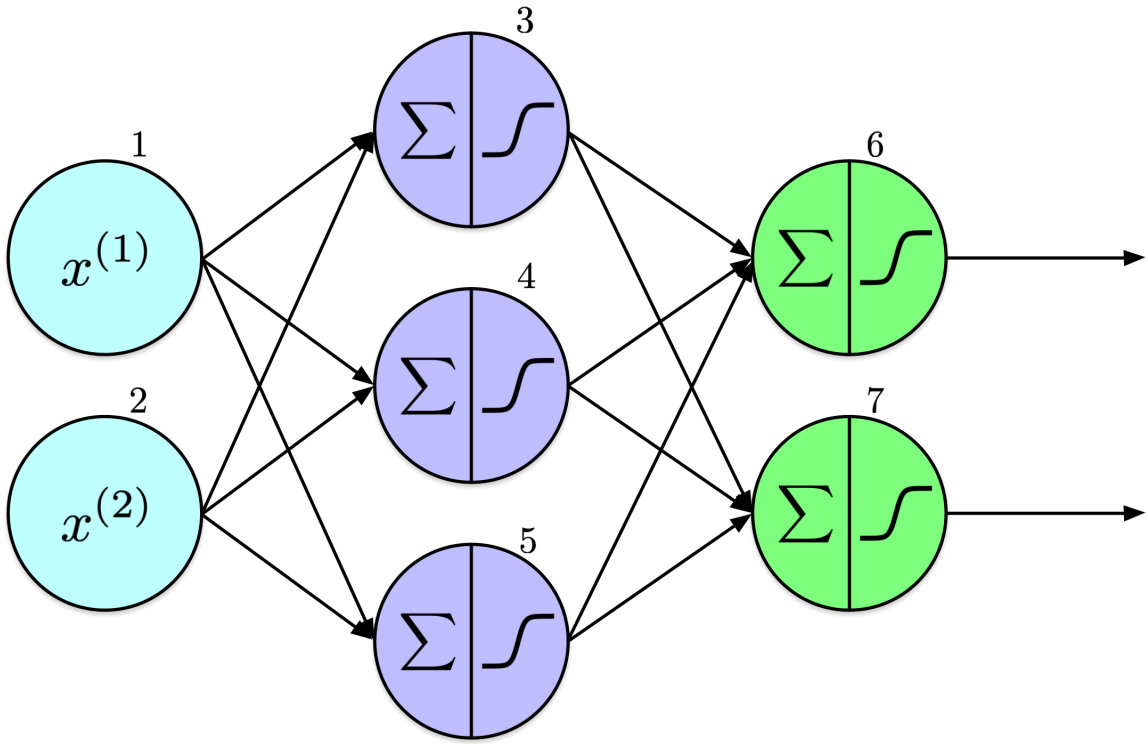
Information in this architecture flows unidirectionally—from left to right, moving from input to output. Consequently, it is termed a **feedforward neural network**.

The network consists of **three layers**: input, hidden, and output. The **input layer** contains two nodes, the **hidden layer** comprises three nodes, and the **output layer** has two nodes. Additional hidden layers and nodes per layer can be added, which will be discussed later.

It is often useful to include explicit input nodes that do not perform calculations, known as **input units** or **input neurons**. These nodes act as placeholders to introduce input features into the network, passing data directly to the next layer without transformation. In the network diagram, these are the light blue nodes on the left, labeled 1 and 2. Typically, **the number of input units corresponds to the number of features**.

For clarity, nodes are labeled to facilitate discussion of the weights between them, such as $w_{1,5}$ between nodes 1 and 5. Similarly, the output of a node is denoted by o_k, where k represents the node's label. For example, for $k = 3$, the output would be o_3.

Forward Pass (Computatation)



$$o3 = \sigma(w_{13}x^{(1)} + w_{23}x^{(2)} + b_3)$$

$$o4 = \sigma(w_{14}x^{(1)} + w_{24}x^{(2)} + b_4)$$

$$o5 = \sigma(w_{15}x^{(1)} + w_{25}x^{(2)} + b_5)$$

$$o6 = \sigma(w_{36}o_3 + w_{46}o_4 + w_{56}o_5 + b_6)$$

$$o7 = \sigma(w_{37}o_3 + w_{47}o_4 + w_{57}o_5 + b_7)$$

First, it's important to **understand the information flow**: this network **computes** two outputs from its inputs.

To simplify the figure, I have opted not to display the bias terms, though they remain crucial components. Specifically, b_3 represents the bias term associated with node 3.

If bias terms were not significant, the training process would naturally reduce them to zero. Bias terms are essential as they enable the adjustment of the decision boundary, allowing the model to learn more complex patterns that weights alone cannot capture. By offering additional degrees of freedom, they also contribute to faster convergence during training.

Forward Pass (Computatation)

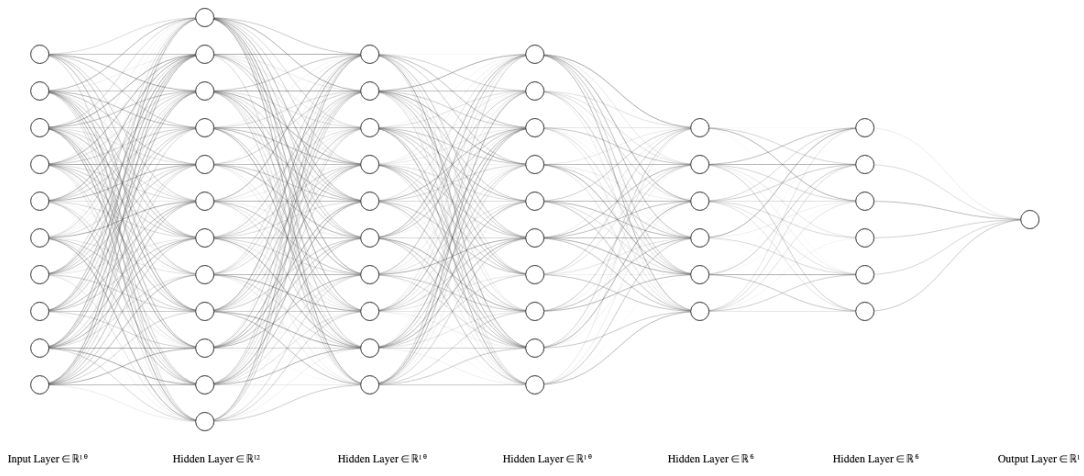
In [2]:
```
import numpy as np

# Sigmoid function
```

```python
def sigma(x):
    return 1 / (1 + np.exp(-x))

# Input (two attributes) vector, one example of our trainig set

x1, x2 = (0.5, 0.9)

# Initializing the weights of layers 2 and 3 to random values

w13, w14, w15, w23, w24, w25 = np.random.uniform(low=-1, high=1, size=6)
w36, w46, w56, w37, w47, w57 = np.random.uniform(low=-1, high=1, size=6)

# Initializing all 5 bias terms to random values

b3, b4, b5, b6, b7 = np.random.uniform(low=-1, high=1, size=5)

o3 = sigma(w13 * x1 + w23 * x2 + b3)
o4 = sigma(w14 * x1 + w24 * x2 + b4)
o5 = sigma(w15 * x1 + w25 * x2 + b5)
o6 = sigma(w36 * o3 + w46 * o4 + w56 * o5 + b6)
o7 = sigma(w37 * o3 + w47 * o4 + w57 * o5 + b7)

(o6, o7)
```

```
(np.float64(0.7184474803926196), np.float64(0.5083206266553244))
```

The example above illustrates the computation process with specific values. Before training a neural network, it is standard practice to initialize the weights and biases with random values. Gradient descent is then employed to iteratively adjust these parameters, aiming to minimize the loss function.

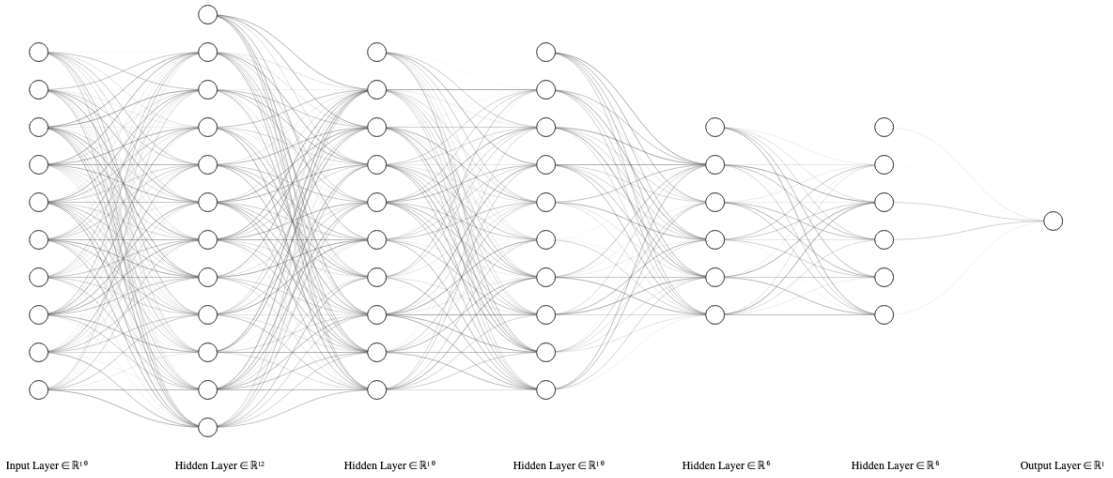
Forward Pass (Computatation)



The information flow remains consistent even in more **complex networks**. Networks with many layers are called **deep neural networks** (DNN).

Produced using **NN-SVG**, LeNail (2019).

Forward Pass (Computatation)



Same network with **bias terms shown**.

Produced using **NN-SVG**, LeNail (2019).

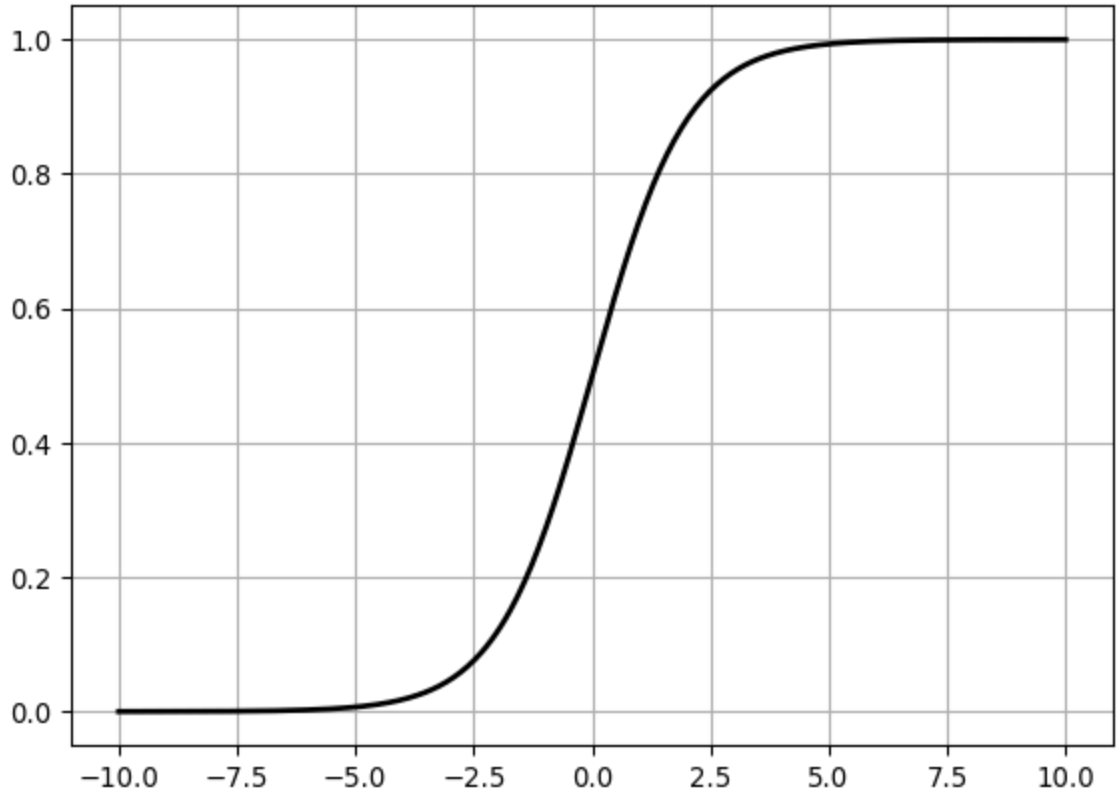
Activation Function

- As will be discussed later, the training algorithm, known as **backpropagation**, employs **gradient descent**, necessitating the calculation of the **partial derivatives of the loss function**.
- The **step function** in the multilayer perceptron had to be replaced, as it consists only of flat surfaces. **Gradient descent cannot progress on flat surfaces due to their zero derivative**.

Activation Function

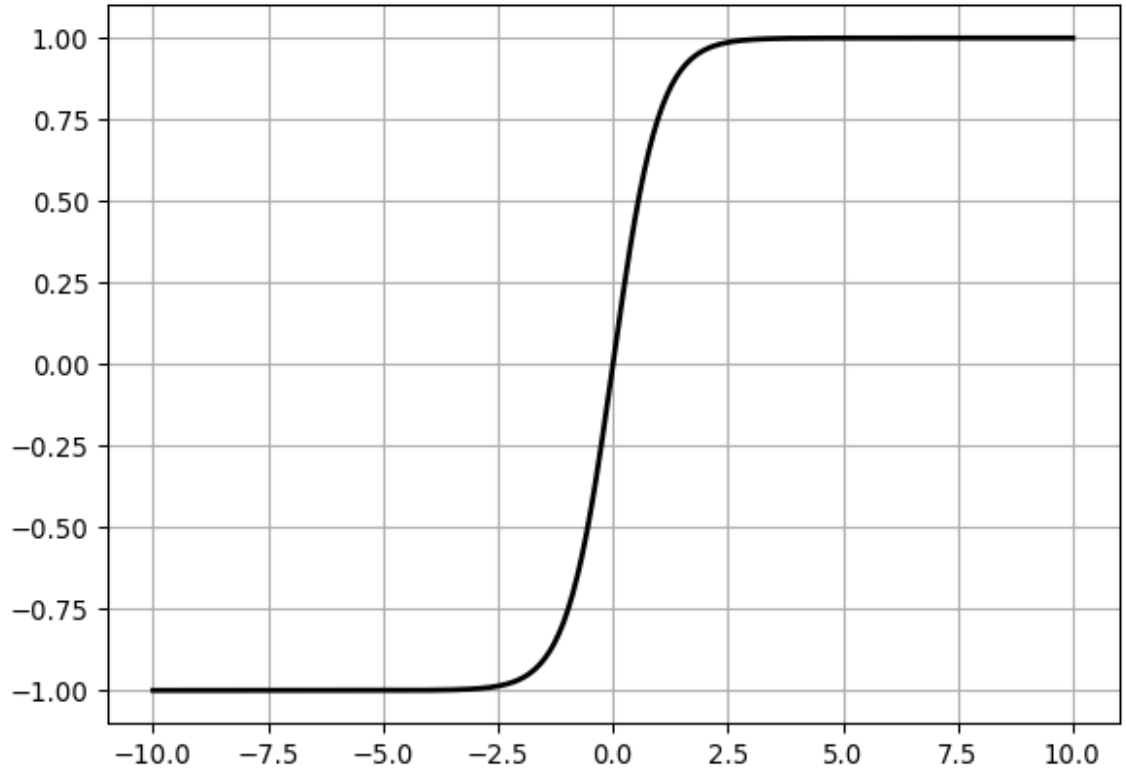
- **Nonlinear activation functions are paramount** because, without them, multiple layers in the network would only compute a linear function of the inputs.
- According to the **Universal Approximation Theorem**, sufficiently large deep networks with nonlinear activation functions can **approximate any continuous function**. See Universal Approximation Theorem.

Sigmoid



$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

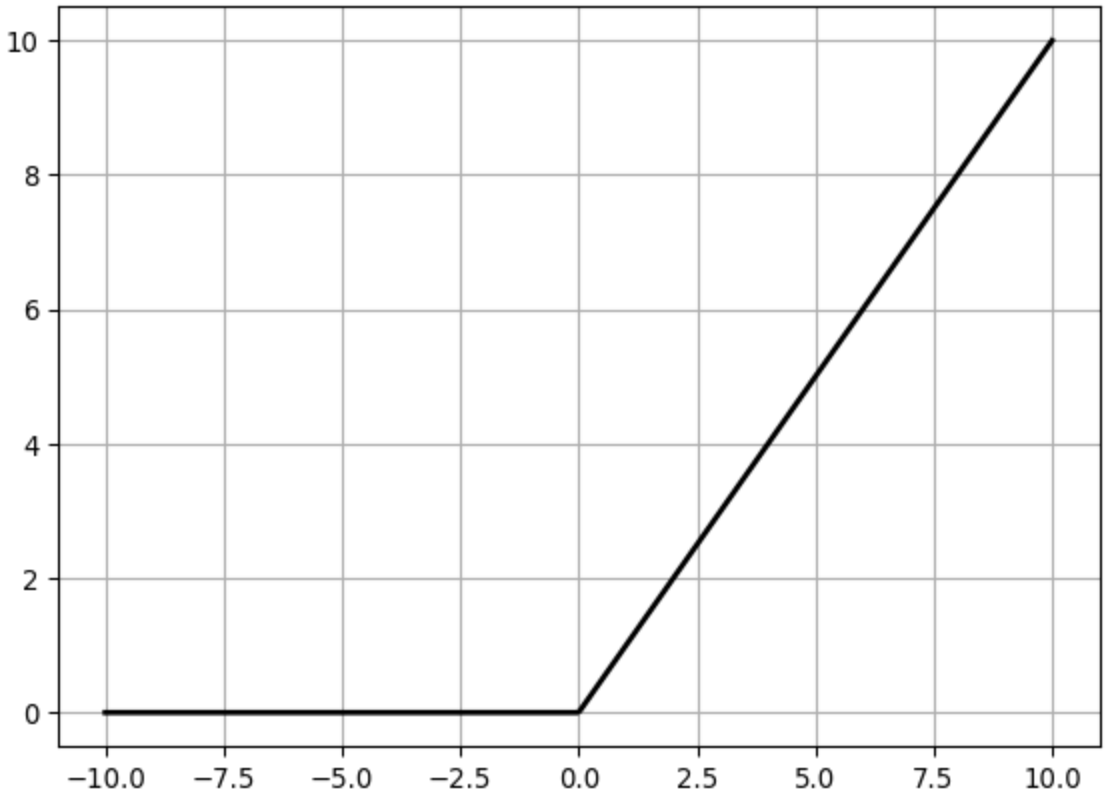
Hyperbolic Tangent Function



$$\tanh(t) = 2\sigma(2t) - 1$$

This S-shaped curve, similar to the sigmoid function, produces output values ranging from -1 to 1. According to Géron (2022), this range helps each layer's output to be approximately centered around 0 at the start of training, thereby **accelerating convergence**.

Rectified linear unit function (ReLU)



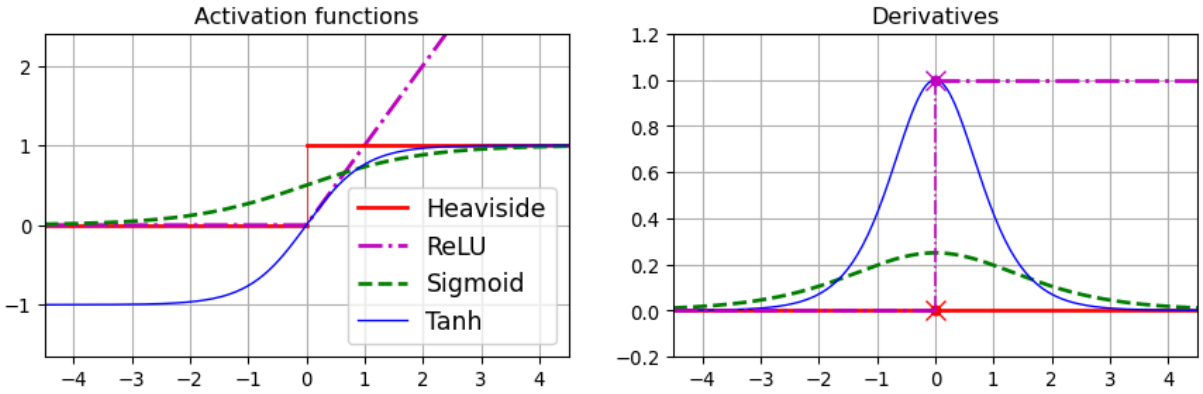
$$\text{ReLU}(t) = \max(0, t)$$

Although the **ReLU function** is not differentiable at $t = 0$ and has a derivative of 0 for $t < 0$, it performs quite well in practice and is computationally efficient. Consequently, it has become the **default activation function**.

Common Activation Functions

In [6]:
```
from scipy.special import expit as sigmoid

def relu(z):
    return np.maximum(0, z)

def derivative(f, z, eps=0.000001):
    return (f(z + eps) - f(z - eps))/(2 * eps)
```

```
max_z = 4.5
z = np.linspace(-max_z, max_z, 200)

plt.figure(figsize=(11, 3.1))

plt.subplot(121)
plt.plot([-max_z, 0], [0, 0], "r-", linewidth=2, label="Heaviside")
plt.plot(z, relu(z), "m-.", linewidth=2, label="ReLU")
plt.plot([0, 0], [0, 1], "r-", linewidth=0.5)
plt.plot([0, max_z], [1, 1], "r-", linewidth=2)
plt.plot(z, sigmoid(z), "g--", linewidth=2, label="Sigmoid")
plt.plot(z, np.tanh(z), "b-", linewidth=1, label="Tanh")
plt.grid(True)
plt.title("Activation functions")
plt.axis([-max_z, max_z, -1.65, 2.4])
plt.gca().set_yticks([-1, 0, 1, 2])
plt.legend(loc="lower right", fontsize=13)

plt.subplot(122)
plt.plot(z, derivative(np.sign, z), "r-", linewidth=2, label="Heaviside")
plt.plot(0, 0, "ro", markersize=5)
plt.plot(0, 0, "rx", markersize=10)
plt.plot(z, derivative(sigmoid, z), "g--", linewidth=2, label="Sigmoid")
plt.plot(z, derivative(np.tanh, z), "b-", linewidth=1, label="Tanh")
plt.plot([-max_z, 0], [0, 0], "m-.", linewidth=2)
plt.plot([0, max_z], [1, 1], "m-.", linewidth=2)
plt.plot([0, 0], [0, 1], "m-.", linewidth=1.2)
plt.plot(0, 1, "mo", markersize=5)
plt.plot(0, 1, "mx", markersize=10)
plt.grid(True)
plt.title("Derivatives")
plt.axis([-max_z, max_z, -0.2, 1.2])

plt.show()
```



Géron (2022) – 10_neural_nets_with_keras.ipynb

Universal Approximation

Definition

The **universal approximation theorem (UAT)** states that a feedforward neural network with a single hidden layer containing a finite number of neurons can **approximate any continuous function** on a compact subset of $\mathbb{R}^n$, given appropriate weights and activation functions.

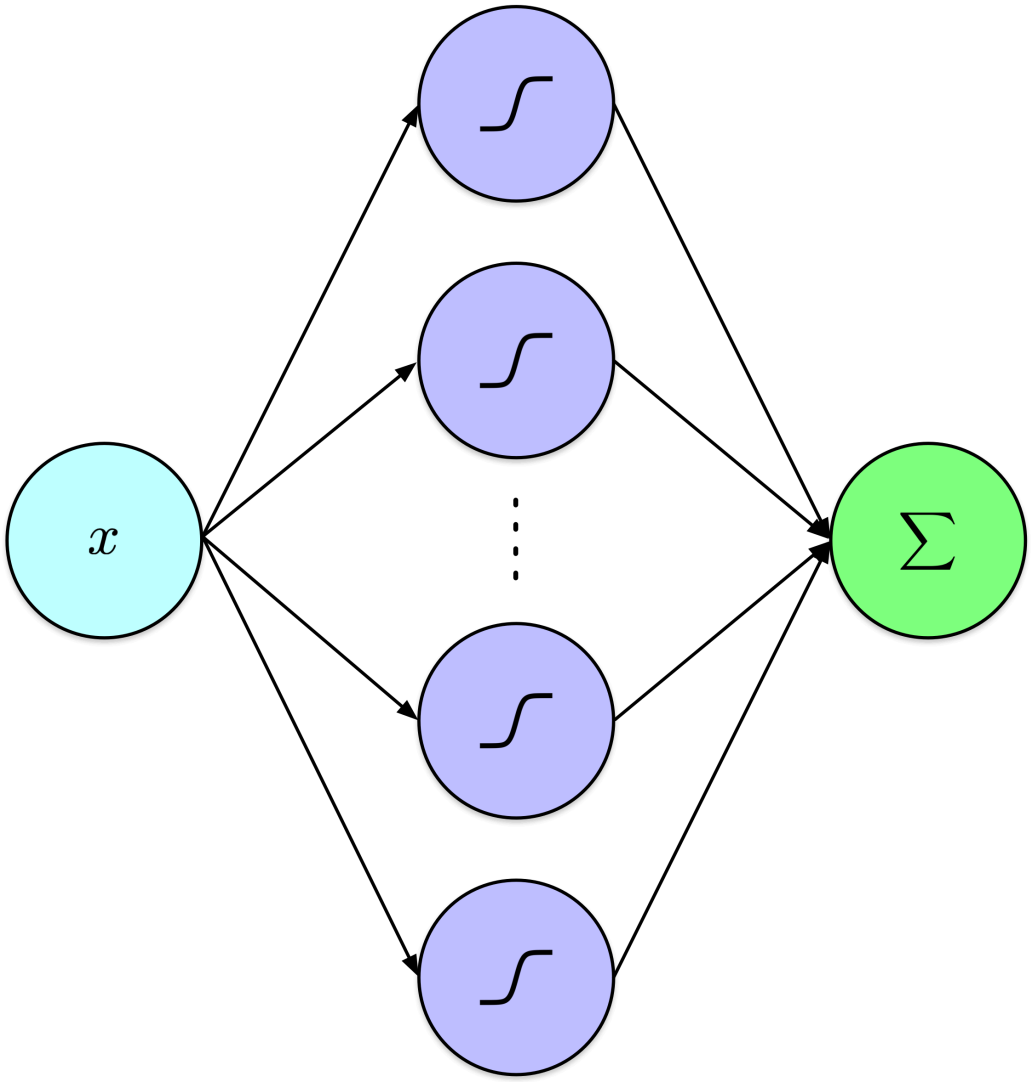
Cybenko (1989); Hornik, Stinchcombe, and White (1989)

In mathematical terms, a subset of $\mathbb{R}^n$ is considered **compact** if it is both **closed** and **bounded**.

- **Closed**: A set is closed if it contains all its boundary points. In other words, it includes its limit points or accumulation points.
- **Bounded**: A set is bounded if there exists a real number (M) such that the distance between any two points in the set is less than M.

In the context of the universal approximation theorem, compactness ensures that the function being approximated is defined on a finite and well-behaved region, which is crucial for the theoretical guarantees provided by the theorem.

Single Hidden Layer



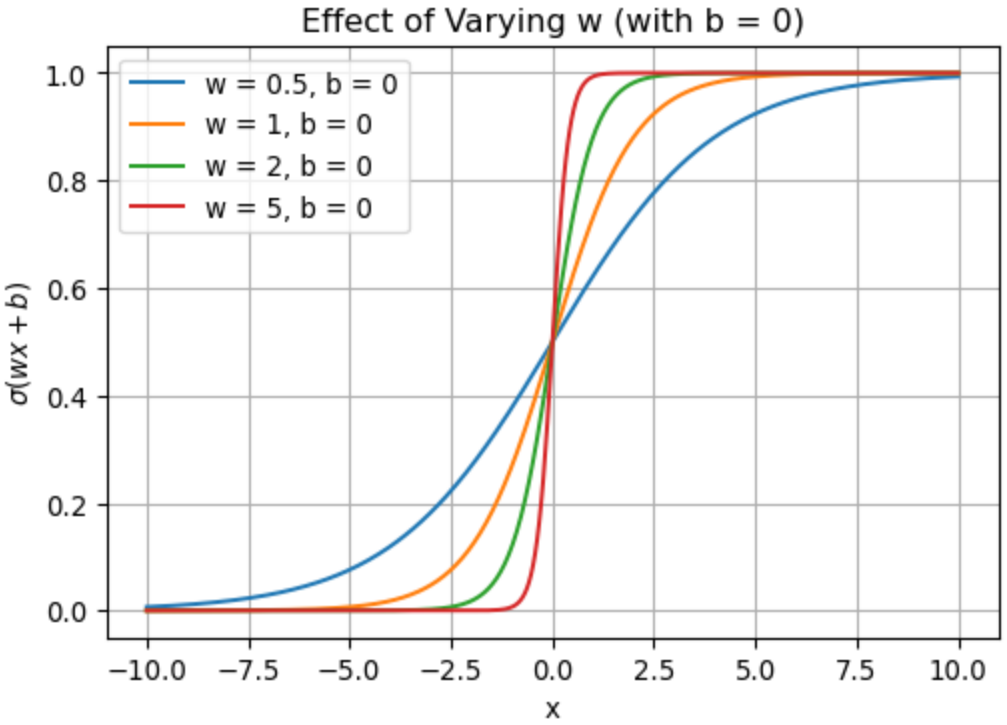
$$y = \sum_{i=1}^{N} \alpha_i \sigma(w_{1,i} x + b_i)$$

Notation adapted to follow that of Cybenko (1989).

Effect of Varying w

In [7]:
```
def logistic(x, w, b):
    """Compute the logistic function with parameters w and b."""
    return 1 / (1 + np.exp(-(w * x + b)))

# Define a range for x values.
x = np.linspace(-10, 10, 400)

# Plot 1: Varying w (steepness) with b fixed at 0.
plt.figure(figsize=(6,4))
```

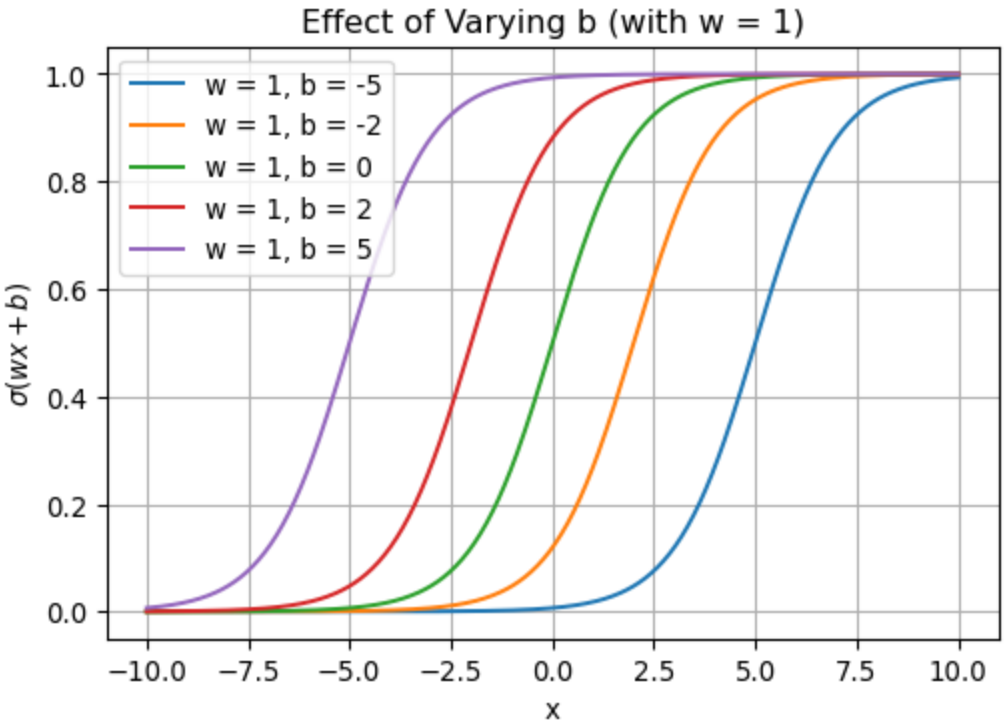
```
w_values = [0.5, 1, 2, 5]  # different steepness values
b = 0  # fixed bias

for w in w_values:
    plt.plot(x, logistic(x, w, b), label=f'w = {w}, b = {b}')
plt.title('Effect of Varying w (with b = 0)')
plt.xlabel('x')
plt.ylabel(r'$\sigma(wx+b)$')
plt.legend()
plt.grid(True)

plt.show()
```



Sigmoid activation function: $\sigma(wx + b)$.

Effect of Varying b

In [8]:
```
# Plot 2: Varying b (horizontal shift) with w fixed at 1.
plt.figure(figsize=(6,4))
w = 1  # fixed steepness
b_values = [-5, -2, 0, 2, 5]  # different bias values

for b in b_values:
    plt.plot(x, logistic(x, w, b), label=f'w = {w}, b = {b}')
plt.title('Effect of Varying b (with w = 1)')
plt.xlabel('x')
plt.ylabel(r'$\sigma(wx+b)$')
plt.legend()
plt.grid(True)

plt.show()
```

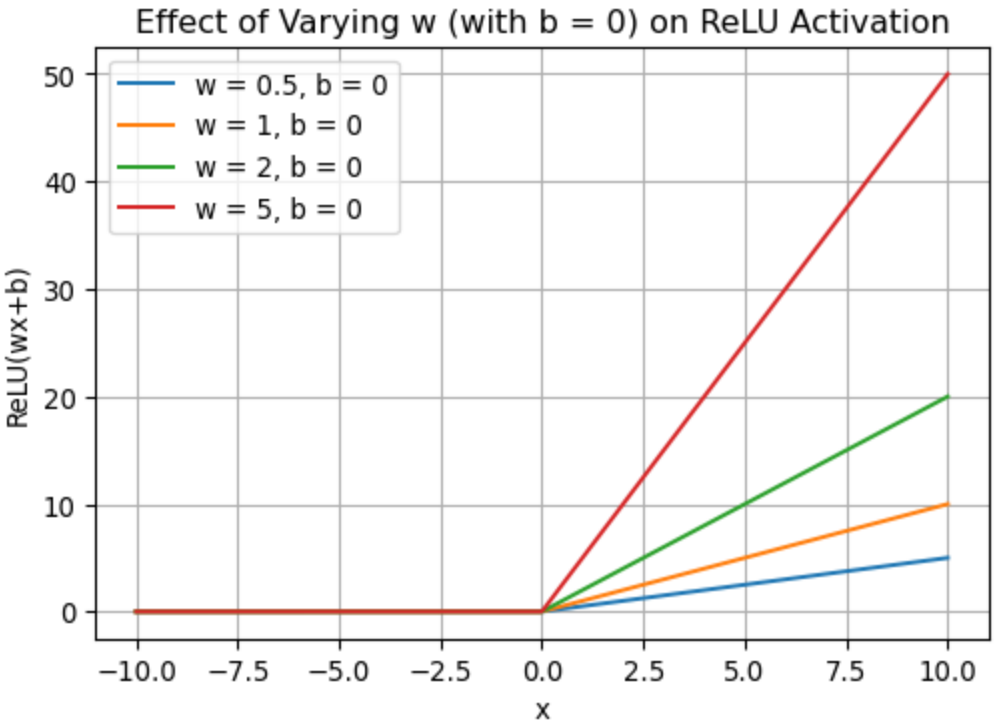


Sigmoid activation function: $\sigma(wx + b)$.

Effect of Varying w

In [9]:

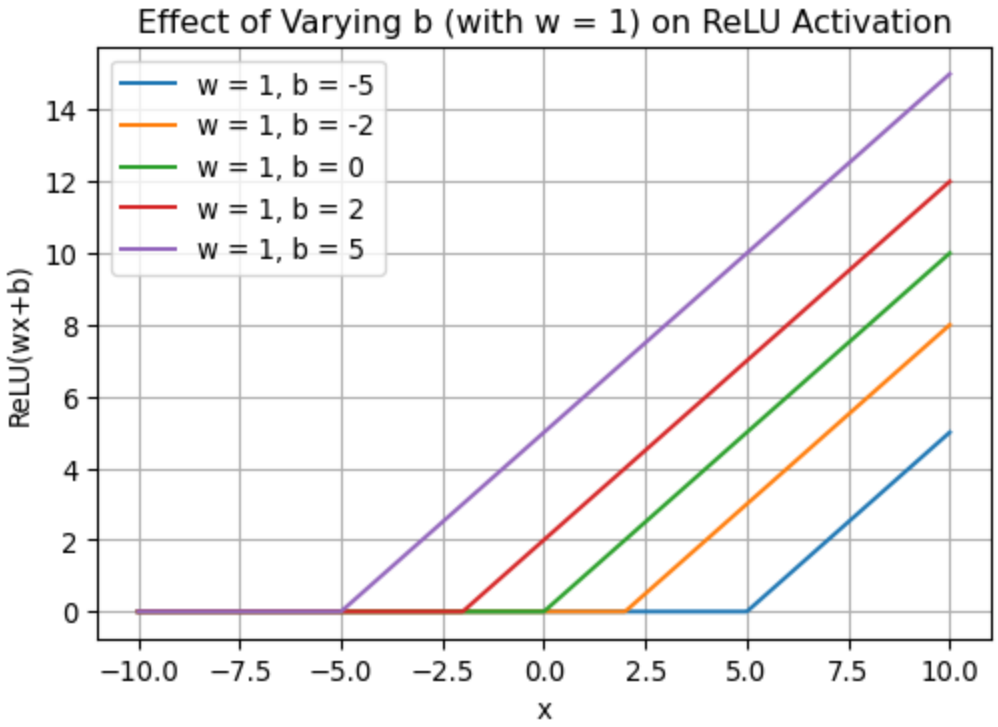
```python
def relu(x, w, b):
    """Compute the ReLU activation with parameters w and b."""
    return np.maximum(0, w * x + b)

# Define a range for x values.
x = np.linspace(-10, 10, 400)

# Plot 1: Varying w (scaling) with b fixed at 0.
plt.figure(figsize=(6,4))
w_values = [0.5, 1, 2, 5]  # different scaling values
b = 0  # fixed bias

for w in w_values:
    plt.plot(x, relu(x, w, b), label=f'w = {w}, b = {b}')
plt.title('Effect of Varying w (with b = 0) on ReLU Activation')
plt.xlabel('x')
plt.ylabel('ReLU(wx+b)')
plt.legend()
plt.grid(True)

plt.show()
```



ReLU activation function: `np.maximum(0, w * x + b)`.

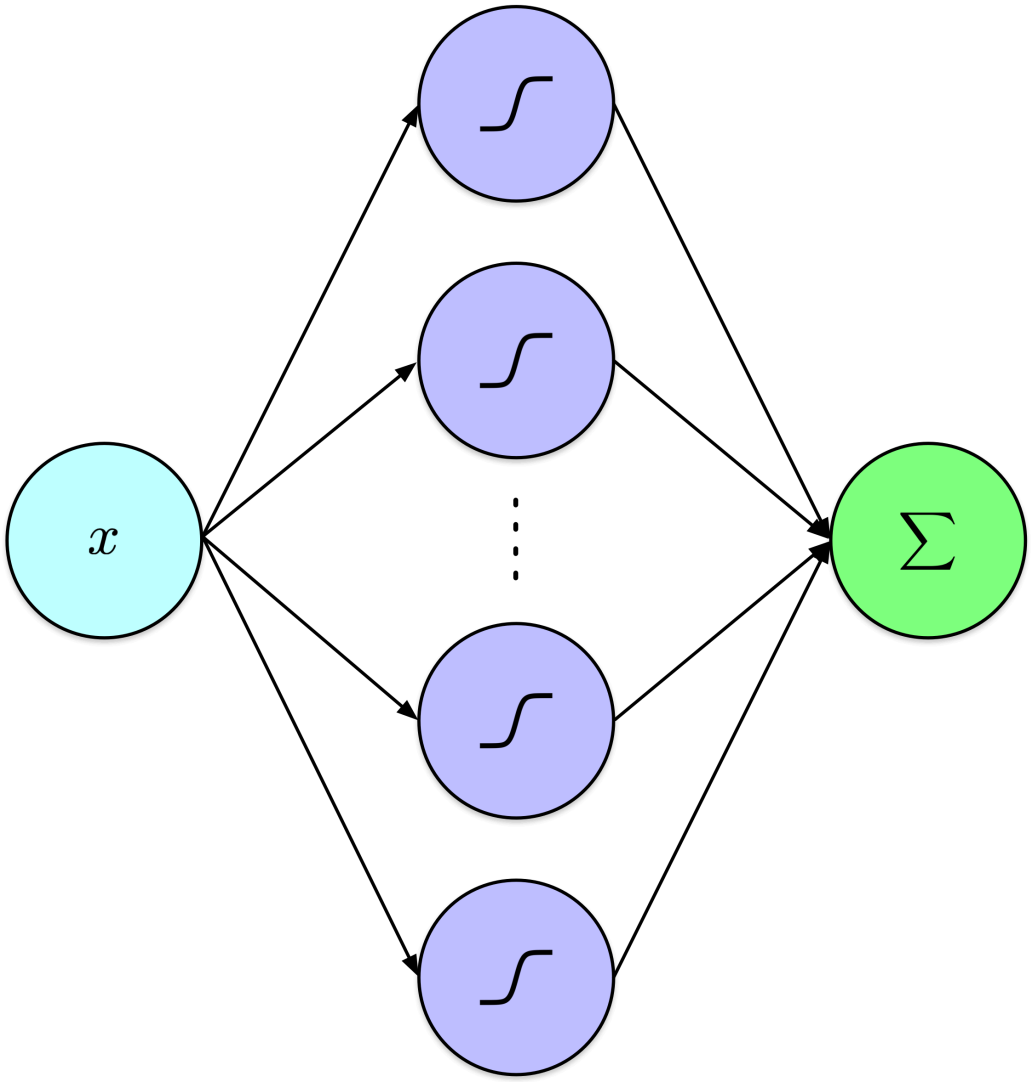
Effect of Varying b

In [10]:
```
# Plot 2: Varying b (horizontal shift) with w fixed at 1.
plt.figure(figsize=(6,4))
w = 1  # fixed scaling
b_values = [-5, -2, 0, 2, 5]  # different bias values

for b in b_values:
    plt.plot(x, relu(x, w, b), label=f'w = {w}, b = {b}')
plt.title('Effect of Varying b (with w = 1) on ReLU Activation')
plt.xlabel('x')
plt.ylabel('ReLU(wx+b)')
plt.legend()
plt.grid(True)

plt.show()
```



ReLU activation function: `np.maximum(0, w * x + b)`.

Single Hidden Layer



$$y = \sum_{i=1}^{N} \alpha_i \sigma(w_{1,i} x + b_i)$$

See also: Chapter 4: A visual proof that neural nets can compute any function, Neural Networks and Deep Learning by Michael Nielsen.

Demonstration with code

In [11]:
```
# Defining the function to be approximated

def f(x):
  return 2 * x**3 + 4 * x**2 - 5 * x + 1

# Generating a dataset, x in [-4,2), f(x) as above

X = 6 * np.random.rand(1000, 1) - 4
```

```
y = f(X.flatten())
```

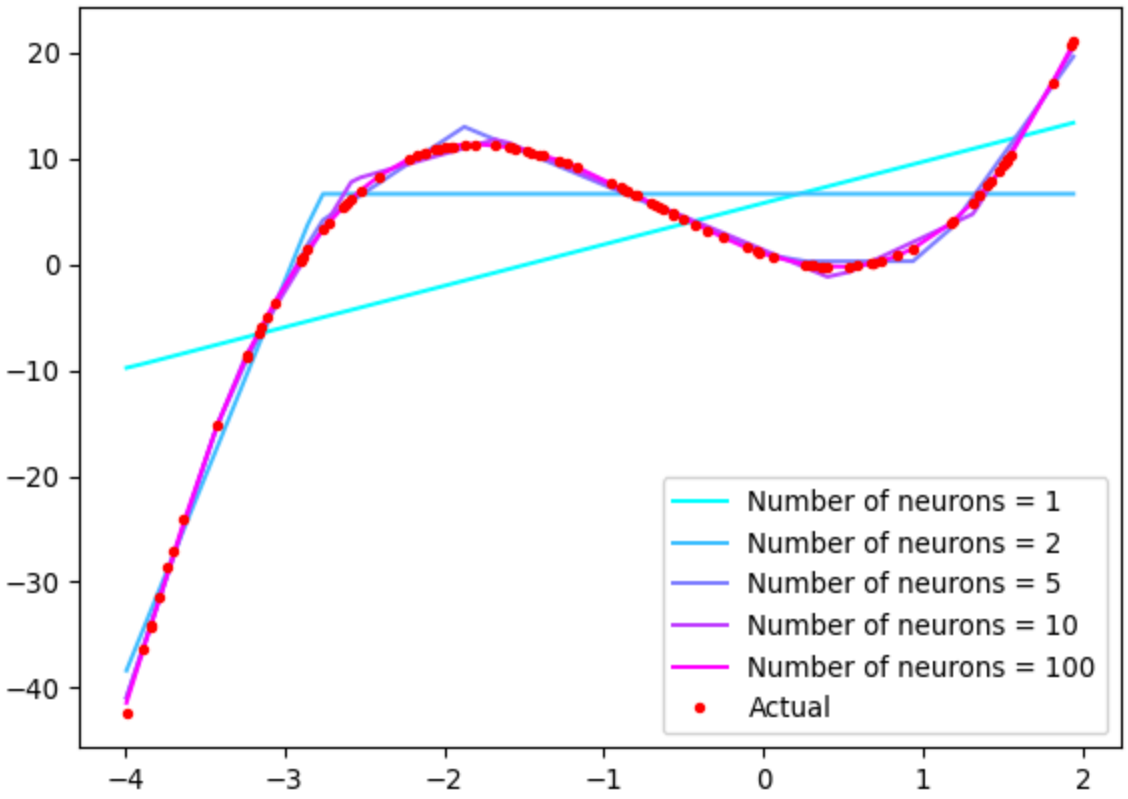
Increasing the number of neurons

In [12]:
```
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split

X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.1, r

models = []

sizes = [1, 2, 5, 10, 100]

for i, n in enumerate(sizes):

  models.append(MLPRegressor(hidden_layer_sizes=[n], max_iter=5000, random_s

  models[i].fit(X_train, y_train)
```

MLPRegressor is a multi-layer perceptron regressor from sklearn. Its default activation function is `relu`.

Increasing the number of neurons

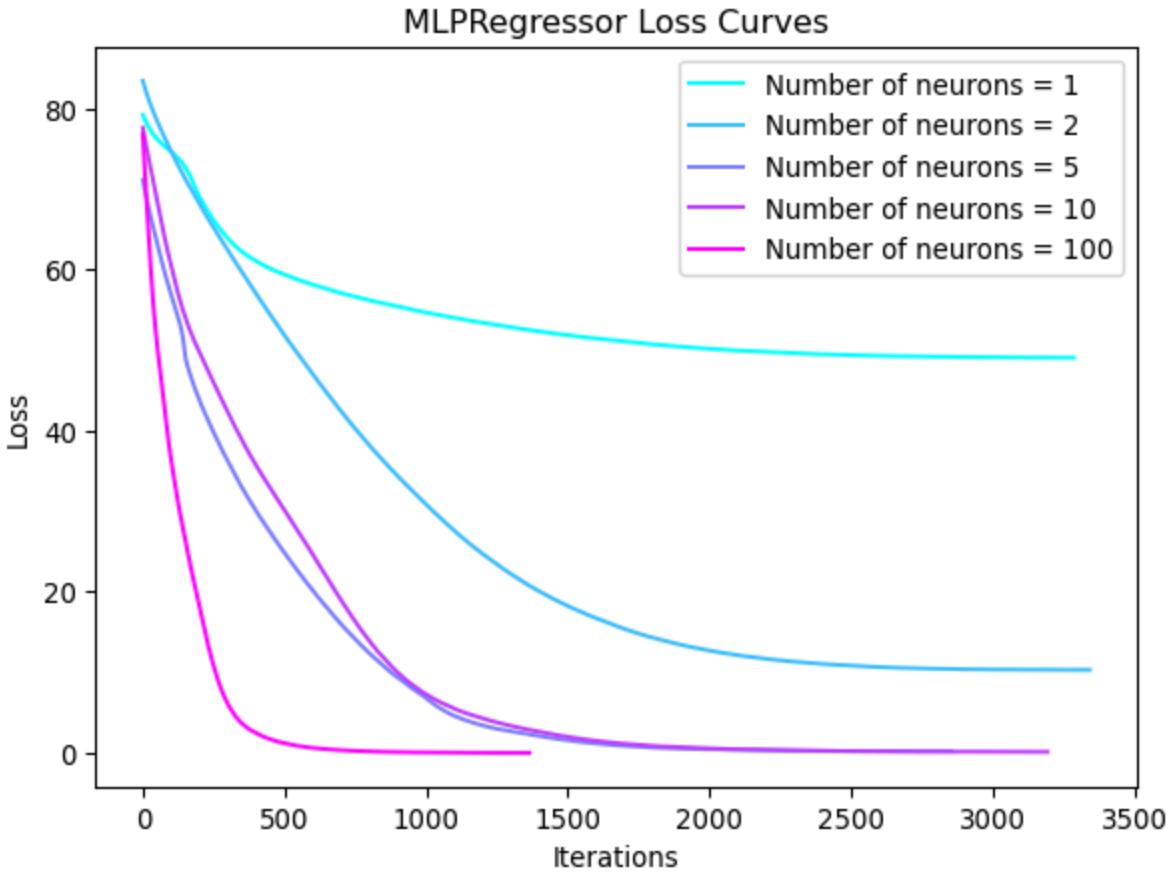
In [13]:
```
# Create a colormap
colors = plt.colormaps['cool'].resampled(len(sizes))

X_valid = np.sort(X_valid,axis=0)

for i, n in enumerate(sizes):

  y_pred = models[i].predict(X_valid)

  plt.plot(X_valid, y_pred, "-", color=colors(i), label="Number of neurons =

y_true = f(X_valid)
plt.plot(X_valid, y_true, "r.", label='Actual')

plt.legend()
plt.show()
```



In the example above, I retained only 10% of the data as the test set because the function being approximated is straightforward and noise-free. This decision was made to ensure that the true curve does not overshadow the other results.

Increasing the number of neurons

In [14]:
```
for i, n in enumerate(sizes):

  plt.plot(models[i].loss_curve_, "-", color=colors(i), label="Number of neu

plt.title('MLPRegressor Loss Curves')
plt.xlabel('Iterations')
plt.ylabel('Loss')

plt.legend()
plt.show()
```



As expected, increasing neuron count reduces loss.

Universal Approximation

https://youtu.be/CqOfi41LfDw

This video effectively conveys the underlying intuition of the universal approximation theorem. (18m 53s)

The video effectively elucidates key concepts (terminology) in neural networks, including nodes, layers, weights, and activation functions. It demonstrates the process of summing activation outputs from a preceding layer, akin to the aggregation of curves. Additionally, the video illustrates how scaling an output by a weight not only alters the amplitude of a curve but also inverts its orientation when the weight is negative. Moreover, it clearly depicts the function of bias terms in vertically shifting the curve, contingent on the sign of the bias.

Let’s code

Frameworks

PyTorch and TensorFlow are the leading platforms for deep learning.

- PyTorch has gained considerable traction in the **research community**. Initially developed by **Meta AI**, it is now part of the Linux Foundation.
- TensorFlow, created by **Google**, is widely adopted in **industry** for deploying models in production environments.

Official PyTorch Documentary: Powering the AI Revolution

Keras

Keras is a high-level API designed to build, train, evaluate, and execute models across various backends, including PyTorch, TensorFlow, and JAX, Google's high-performance platform.

Keras is powerful enough for most projects.

As highlighted in previous Quotes of the Day, François Chollet, a Google engineer, is the originator and one of the primary developers of the Keras project.

Fashion-MNIST dataset

"Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of **60,000 examples** and a test set of **10,000 examples**. Each example is a **28x28 grayscale image**, associated with a label from **10 classes**."

Attribution: Géron (2022) – 10_neural_nets_with_keras.ipynb

Loading

In [16]:
```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist

(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

Setting aside 5000 examples as a validation set.

Exploration

In [17]:
```
X_train.shape
```

```
(55000, 28, 28)
```

...

In [18]:
```
X_train.dtype
```

```
dtype('uint8')
```

...

Transforming the pixel intensities from integers in the range 0 to 255 to floats in the range 0 to 1.

In [19]:
```
X_train = X_train / 255.0
X_valid = X_valid / 255.0
```

What are these images anyway!

In [20]:
```
plt.figure(figsize=(2, 2))
plt.imshow(X_train[0], cmap="binary")
plt.axis('off')
plt.show()
```

...

In [21]:
```
y_train
```

```
array([9, 0, 0, ..., 9, 0, 2], shape=(55000,), dtype=uint8)
```

...

Since the labels are integers, 0 to 9. Class names will become handy.

In [22]:
```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

First 40 images

In [23]:
```
n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
```

```
for row in range(n_rows):
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(X_train[index], cmap="binary", interpolation="nearest")
        plt.axis('off')
        plt.title(class_names[y_train[index]])
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```

Creating a model

In [24]:
```
tf.random.set_seed(42)

model = tf.keras.Sequential()

model.add(tf.keras.layers.InputLayer(shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

`model.summary()`

In [25]:
```
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Par
flatten_1 (Flatten)	(None, 784)	
dense_3 (Dense)	(None, 300)	235
dense_4 (Dense)	(None, 100)	30
dense_5 (Dense)	(None, 10)	1

Total params: 266,610 (1.02 MB)
Trainable params: 266,610 (1.02 MB)
Non-trainable params: 0 (0.00 B)

As observed, `dense_3` has $235,500$ parameters, while $784 \times 300 = 235,200$.

Could you explain the origin of the additional parameters?

Similarly, `dense_3` has $30,100$ parameters, while $300 \times 100 = 30,000$.

Can you explain why?

Creating a model (alternative)

In [26]:
```
# extra code – clear the session to reset the name counters
tf.keras.backend.clear_session()
tf.random.set_seed(42)
```

In [27]:
```
model = tf.keras.Sequential([
    tf.keras.Input(shape=(28, 28)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

`model.summary()`

In [28]:
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Par
flatten (Flatten)	(None, 784)	
dense (Dense)	(None, 300)	235
dense_1 (Dense)	(None, 100)	30
dense_2 (Dense)	(None, 10)	1

Total params: 266,610 (1.02 MB)
Trainable params: 266,610 (1.02 MB)
Non-trainable params: 0 (0.00 B)

Compiling the model

In [29]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])
```

`sparse_categorical_crossentropy` is the appropriate function for a multiclass classification problem (more later).

The method `compile` allows to set the loss function, as well as other parameters. Keras then prepares the model for training.

Training the model

In [30]:
```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

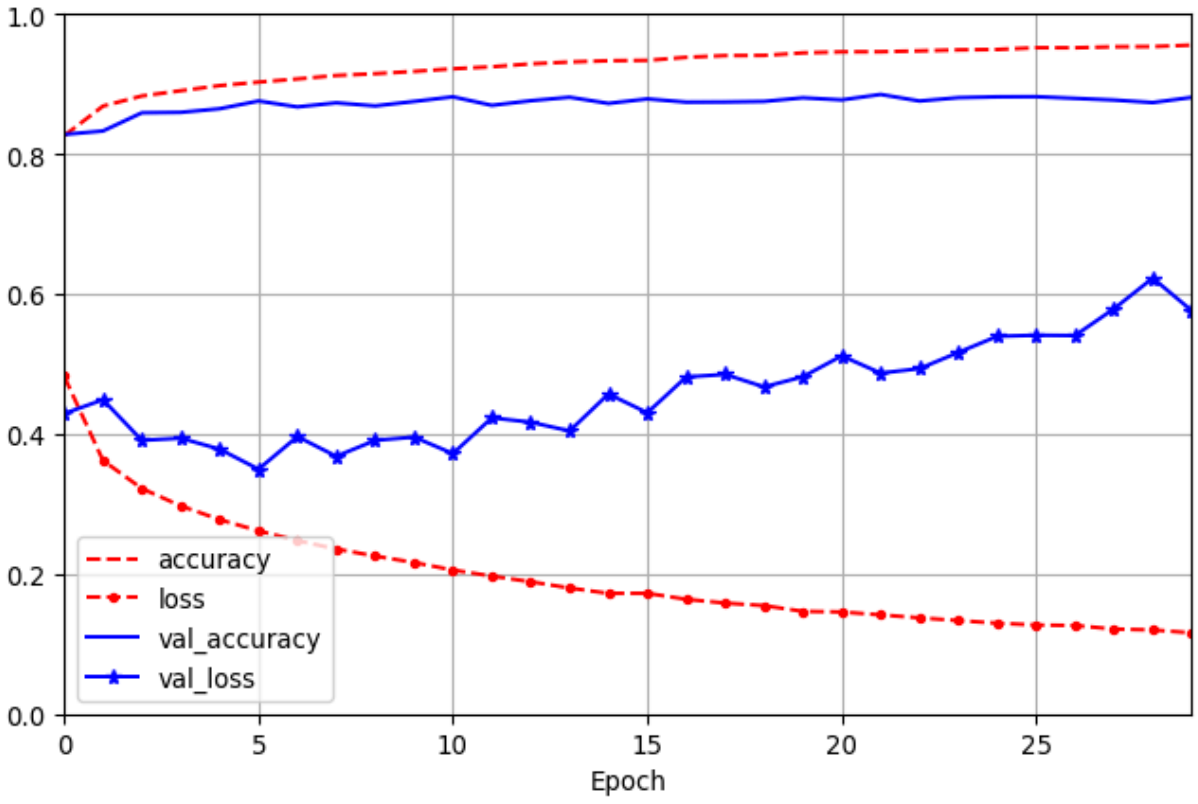
The model is provided with both a taining set and a validation set. At each step, the model will report its performance on both sets. This will also allow to visualize the accuracy and loss curves on both sets (more later).

When calling the `fit` method in Keras (or similar frameworks), each step corresponds to the evaluation of a mini-batch. A mini-batch is a subset of the training data, and during each step, the model updates its weights based on the error calculated from this mini-batch.

An epoch is defined as one complete pass through the entire training dataset. During an epoch, the model processes multiple mini-batches until it has seen all the training data once. This process is repeated for a specified number of epochs to optimize the model's performance.

Visualization

In [31]:
```
import pandas as pd

pd.DataFrame(history.history).plot(
    figsize=(8, 5), xlim=[0, 29], ylim=[0, 1], grid=True, xlabel="Epoch",
    style=["r--", "r--.", "b-", "b-*"])
plt.legend(loc="lower left")  # extra code
plt.show()
```



Evaluating the model on our test

In [32]:
```
model.evaluate(X_test, y_test)
```

```
[121.29401397705078, 0.8600999712944031]
```

Making predictions

In [33]:
```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```

```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)

...
```

In [34]:
```
y_pred = y_proba.argmax(axis=-1).astype(int)
y_pred
```

...

In [35]:
```
y_new = y_test[:3]
y_new
```

As can be seen, the predictions are unambiguous, with only one class per prediction exhibiting a high value.

Predicted vs Observed

In [36]:
```
plt.figure(figsize=(7.2, 2.4))
for index, image in enumerate(X_new):
    plt.subplot(1, 3, index + 1)
    plt.imshow(image, cmap="binary", interpolation="nearest")
    plt.axis('off')
    plt.title(class_names[y_test[index]])
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```

Test Set Performance

In [38]:
```
from sklearn.metrics import classification_report

y_proba = model.predict(X_test)
y_pred = y_proba.argmax(axis=-1).astype(int)
```

Test Set Performance

In [39]:
```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.82      0.80      0.81      1000
           1       0.96      0.98      0.97      1000
           2       0.86      0.63      0.73      1000
           3       0.91      0.81      0.86      1000
           4       0.64      0.91      0.75      1000
           5       0.99      0.94      0.96      1000
           6       0.68      0.64      0.66      1000
           7       0.94      0.94      0.94      1000
           8       0.96      0.98      0.97      1000
           9       0.92      0.96      0.94      1000

    accuracy                           0.86     10000
   macro avg       0.87      0.86      0.86     10000
weighted avg       0.87      0.86      0.86     10000
```

Prologue

Summary

- **Introduction to Neural Networks and Connectionism**
  - Shift from symbolic AI to connectionist approaches in artificial intelligence.
  - Inspiration from biological neural networks and the human brain's structure.
- **Computations with Neurodes and Threshold Logic Units**
  - Early models of neurons (neurodes) capable of performing logical operations (AND, OR, NOT).
  - Limitations of simple perceptrons in solving non-linearly separable problems like XOR.
- **Multilayer Perceptrons (MLPs) and Feedforward Neural Networks (FNNs)**
  - Overcoming perceptron limitations by introducing hidden layers.
  - Structure and information flow in feedforward neural networks.
  - Explanation of forward pass computations in neural networks.
- **Activation Functions in Neural Networks**
  - Importance of nonlinear activation functions (sigmoid, tanh, ReLU) for enabling learning of complex patterns.
  - Role of activation functions in backpropagation and gradient descent optimization.
  - Universal Approximation Theorem and its implications for neural networks.
- **Deep Learning Frameworks**
  - Overview of PyTorch and TensorFlow as leading platforms for deep learning.
  - Introduction to Keras as a high-level API for building and training neural networks.
  - Discussion on the suitability of different frameworks for research and industry applications.

- **Hands-On Implementation with Keras**
  - Loading and exploring the Fashion-MNIST dataset.
  - Building a neural network model using Keras' Sequential API.
  - Compiling the model with appropriate loss functions and optimizers for multiclass classification.
  - Training the model and visualizing training and validation metrics over epochs.
  - Evaluating model performance on test data and interpreting results.
- **Making Predictions and Interpreting Results**
  - Using the trained model to make predictions on new data.
  - Visualizing predictions alongside actual images and labels.
  - Understanding the output probabilities and class assignments in the context of the dataset.

3Blue1Brown on Deep Learning

- But what is a Neural Network?
  - youtu.be/aircAruvnKk
  - 19 minutes
- Gradient descent, how neural networks learn?
  - youtu.be/IHZwWFHWa-w
  - 21 minutes
- What is backpropagation really doing?
  - youtu.be/Ilg3gGewQ5U
  - 14 minutes
- Backpropagation calculus
  - youtu.be/Ilg3gGewQ5U

Next lecture

- We will discuss the training algorithm for artificial neural networks.

References

Cybenko, George V. 1989. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals and Systems* 2: 303–14. https://api.semanticscholar.org/CorpusID:3958369.

Géron, Aurélien. 2022. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 3rd ed. O'Reilly Media, Inc.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. Adaptive Computation and Machine Learning. MIT Press.

https://dblp.org/rec/books/daglib/0040158.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. 1989. “Multilayer Feedforward Networks Are Universal Approximators.” *Neural Networks* 2 (5): 359–66. https://doi.org/https://doi.org/10.1016/0893-6080(89)90020-8.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. “Deep Learning.” *Nature* 521 (7553): 436–44. https://doi.org/10.1038/nature14539.

LeNail, Alexander. 2019. “NN-SVG: Publication-Ready Neural Network Architecture Schematics.” *Journal of Open Source Software* 4 (33): 747. https://doi.org/10.21105/joss.00747.

McCulloch, Warren S, and Walter Pitts. 1943. “A logical calculus of the ideas immanent in nervous activity.” *The Bulletin of Mathematical Biophysics* 5 (4): 115–33. https://doi.org/10.1007/bf02478259.

Minsky, Marvin, and Seymour Papert. 1969. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press.

Rosenblatt, F. 1958. “The perceptron: A probabilistic model for information storage and organization in the brain.” *Psychological Review* 65 (6): 386–408. https://doi.org/10.1037/h0042519.

Russell, Stuart, and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson. http://aima.cs.berkeley.edu/.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EE**CS**)

University of Ottawa