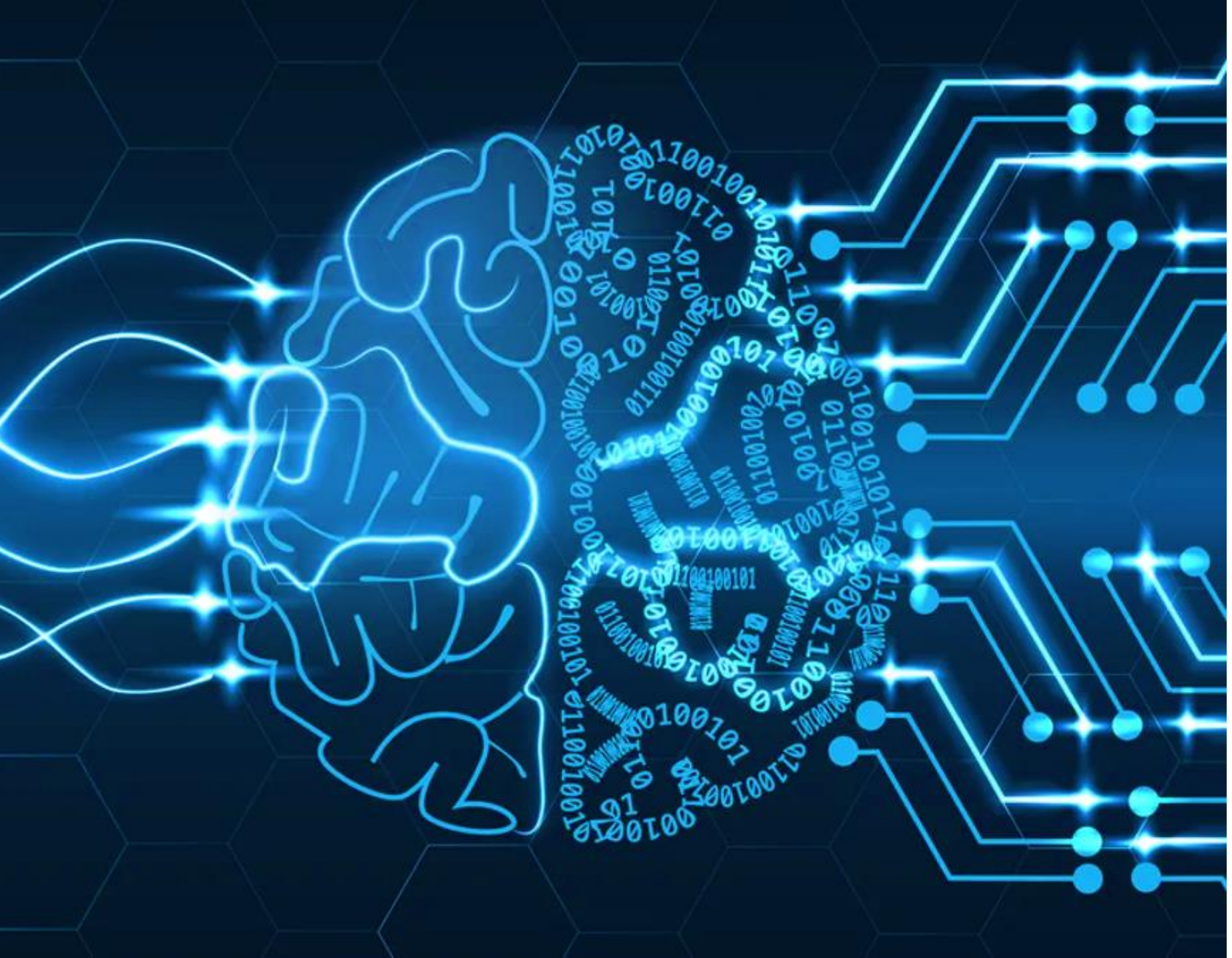


شبکه‌های عصبی مصنوعی

اردیبهشت ۱۴۰۲

دانشکده مهندسی صنایع

دانشگاه صنعتی شریف

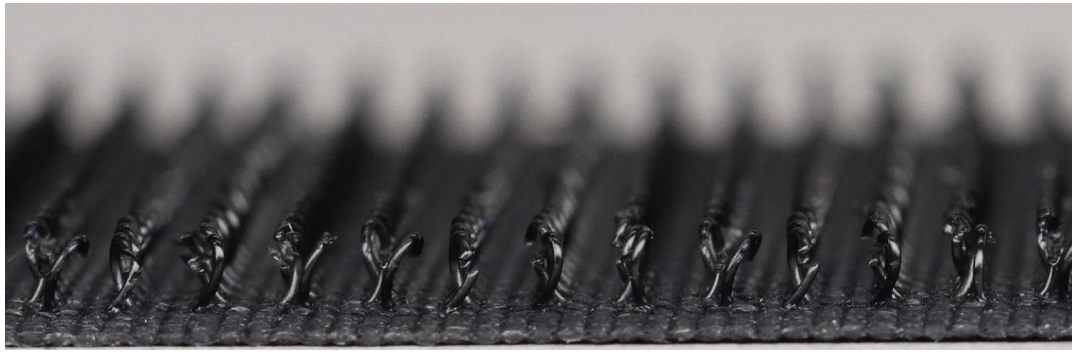
دکتر مهدی شریف‌زاده

سعیدرضا زواشکیانی


بسم الله الرحمن الرحیم

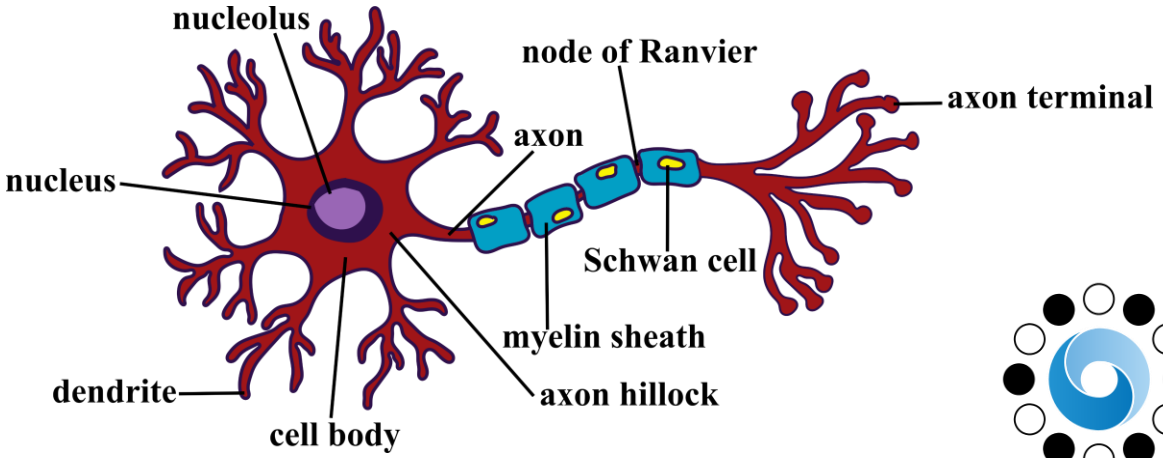
Motivation

Nature has inspired many inventions, like birds inspiring the idea of flying and burdock plants inspiring Velcro. Artificial neural networks (ANNs) are machine learning models inspired by the networks of biological neurons in the brain. While ANNs have become quite different from their biological counterparts, some argue that the biological analogy should be dropped to avoid restricting creativity to biologically plausible systems.



Motivation

ANNs are at the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple's Siri), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of Go (DeepMind's AlphaGo).



YouTube

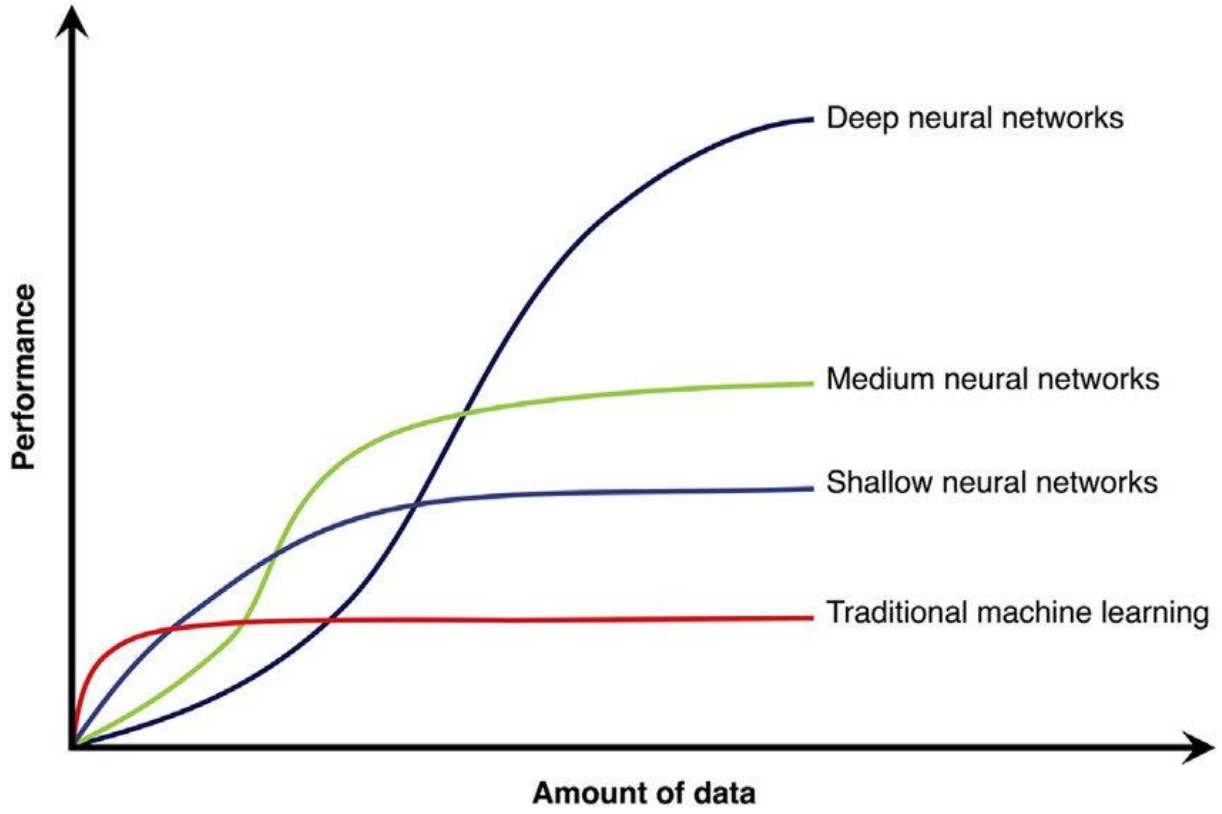
AlphaGo

History

- First introduced back in 1943 by neurophysiologist Warren McCulloch and mathematician Walter Pitts
- Entered a long winter in the 1960s
- New architectures invented and better training techniques developed in the early 1980s
- Replaced with other powerful ML techniques such as SVMs by the 1990s
- Will the wave of interest in ANNs die out now?

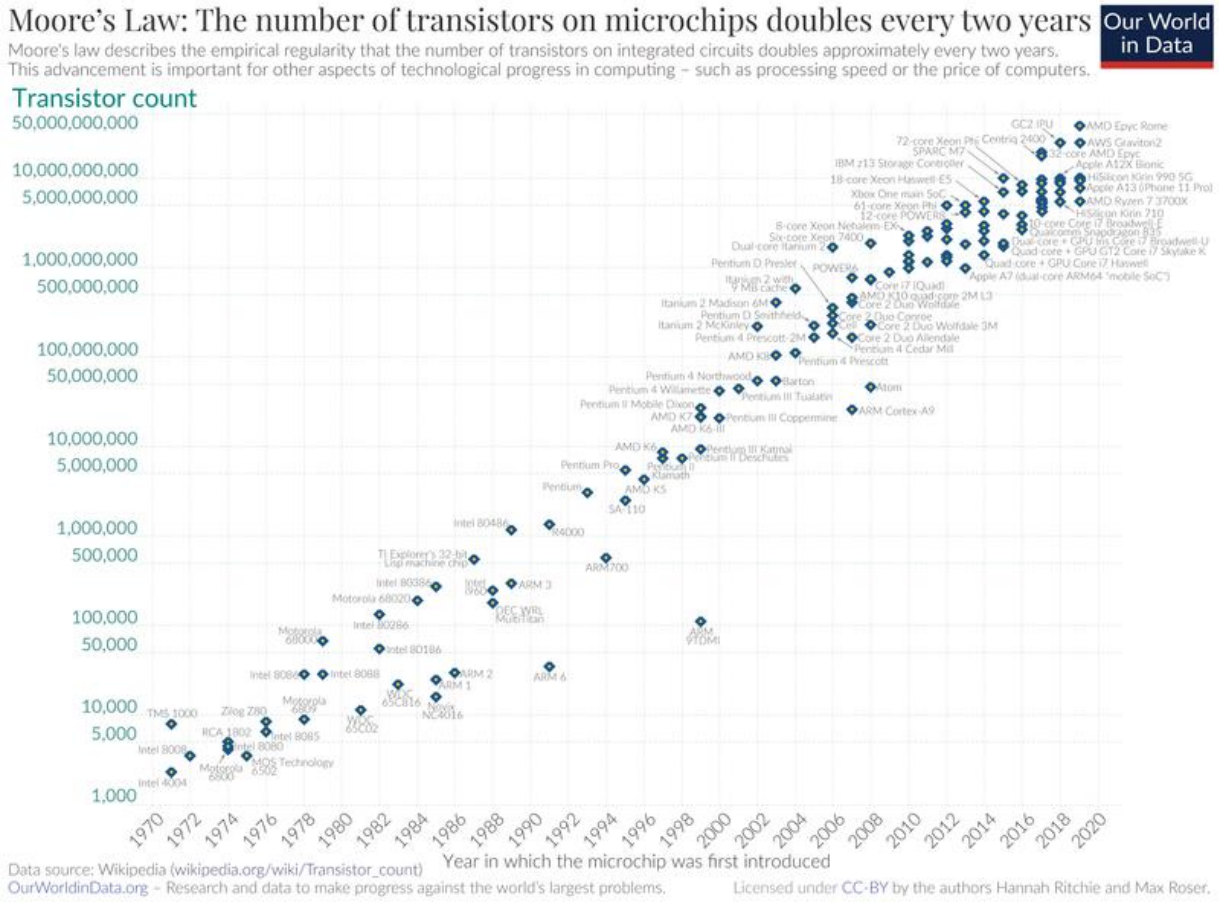
History

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.



History

- The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's law (the number of components in integrated circuits has doubled about every 2 years over the last 50 years), but also thanks to the gaming industry, which has stimulated the production of powerful GPU cards by the millions. Moreover, cloud platforms have made this power accessible to everyone.

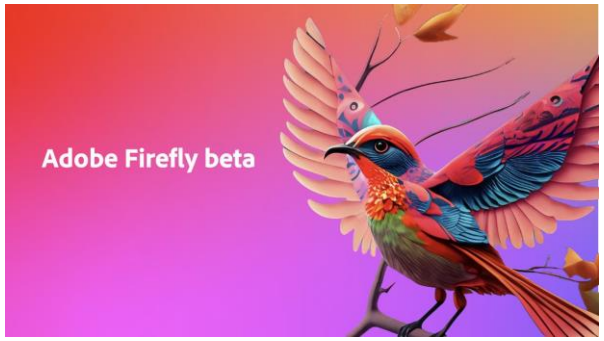


History

- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have had a huge positive impact.
- Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is rather rare in practice (and when it is the case, they are usually fairly close to the global optimum).

History

ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products.

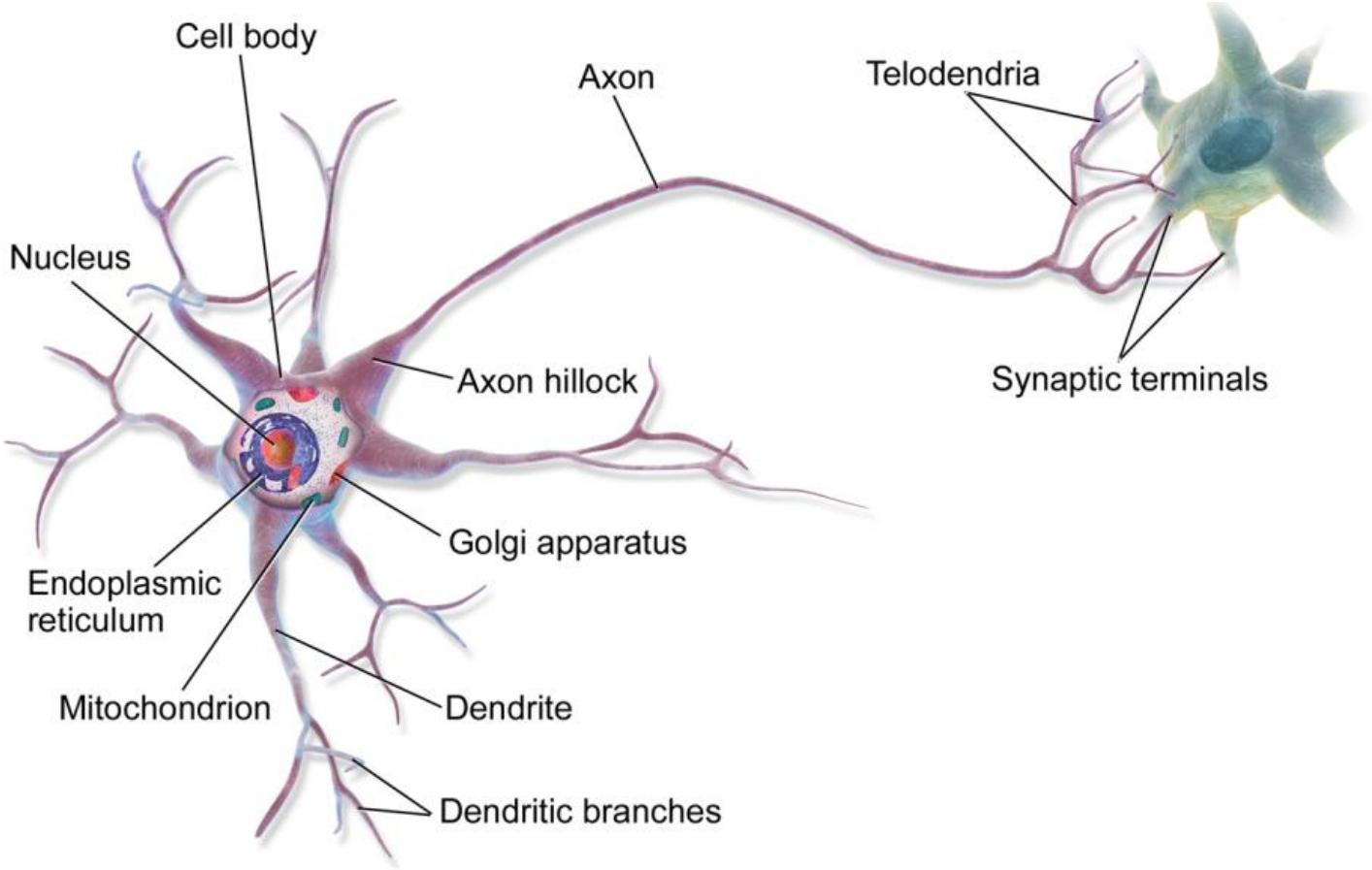


Biological Neurons

It’s composed of a cell body containing the **nucleus** and most of the cell’s complex components, many branching extensions called **dendrites,** plus one very long extension called the **axon**. The axon’s length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called **telodendria**, and at the tip of these branches are minuscule structures called **synaptic terminals** (or simply **synapses**), which are connected to the dendrites or cell bodies of other neurons. Biological neurons produce **short electrical impulses** called **action potentials** (APs, or just signals) which travel along the axons and make the synapses release chemical signals called **neurotransmitters**

Biological Neurons

When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing)



Biological Neurons

Thus, individual biological neurons seem to behave in a rather simple way, but they are organized in a **vast network of billions**, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of **fairly simple neurons**, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural networks (BNNs) is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in **consecutive layers,** especially in the cerebral cortex (i.e., the outer layer of your brain), as shown in Figure 10-2

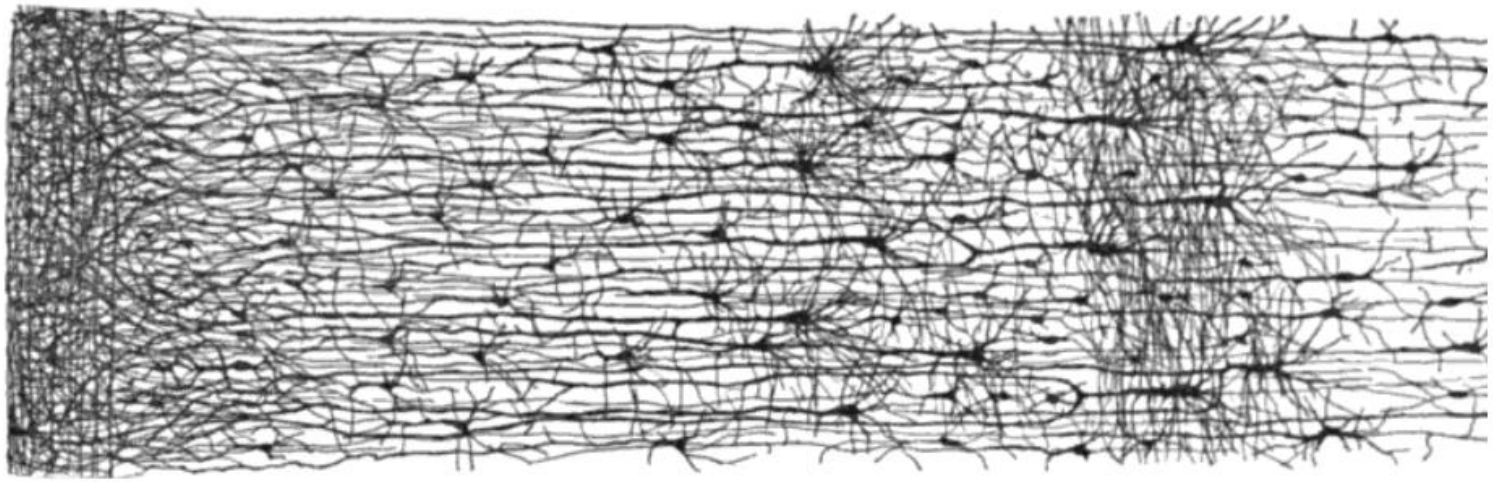


Figure 10-2. Multiple layers in a biological neural network (human cortex)[6]

Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an artificial neuron: it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active. In their paper, they showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want. To see how such a network works, let's build a few ANNs that perform various logical computations (see Figure 10-3), assuming that a neuron is activated when at least two of its inputs are active.

Logical Computations with Neurons

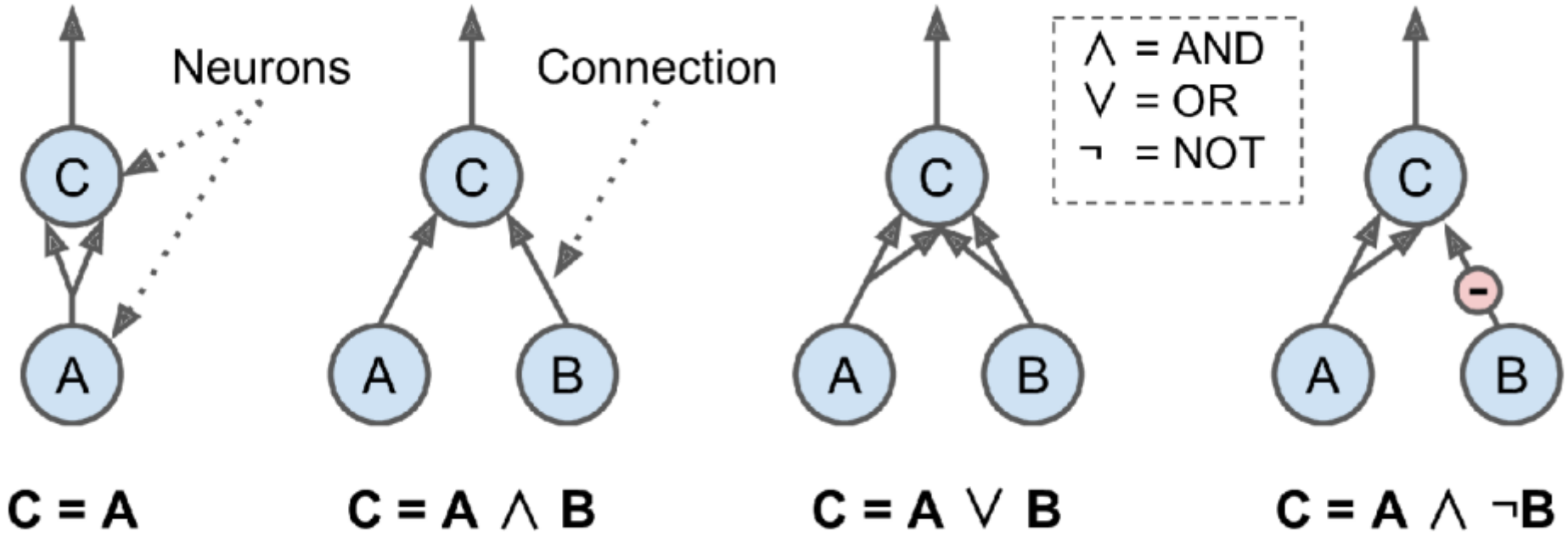


Figure 10-3. ANNs performing simple logical computations

Logical Computations with Neurons

Let’s see what these networks do:

- The first network on the left is the **identity function**: if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A); but if neuron A is off, then neuron C is off as well.
- The second network performs a **logical AND**: neuron C is activated only when both neurons A and B are activated (a single input signal is not enough to activate neuron C).

Logical Computations with Neurons

- The third network performs a **logical OR**: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and neuron B is off. If neuron A is active all the time, then you get a **logical NOT**: neuron C is active when neuron B is off, and vice versa.

Logical Computations with Neurons

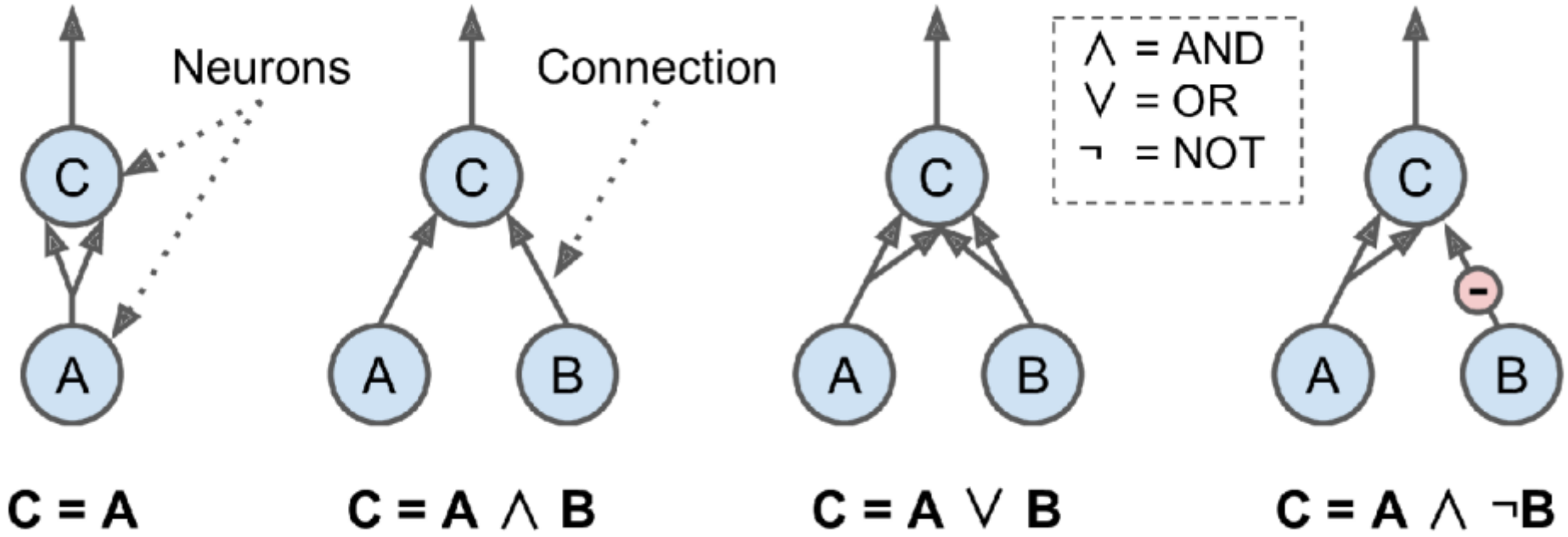


Figure 10-3. ANNs performing simple logical computations

The Perceptron

The Perceptron is one of the simplest ANN architectures, invented in 1957 **by Frank Rosenblatt.** It is based on a slightly different artificial neuron (see Figure 10-4) called a **threshold logic unit** (TLU), or sometimes a **linear threshold unit** (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU computes a weighted sum of its inputs ($z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \boldsymbol{x^T w}$), then applies a step function to that sum and outputs the result: $h_{\boldsymbol{w}}(\boldsymbol{x}) = step(z)$, where $z = \boldsymbol{x^T w}$.

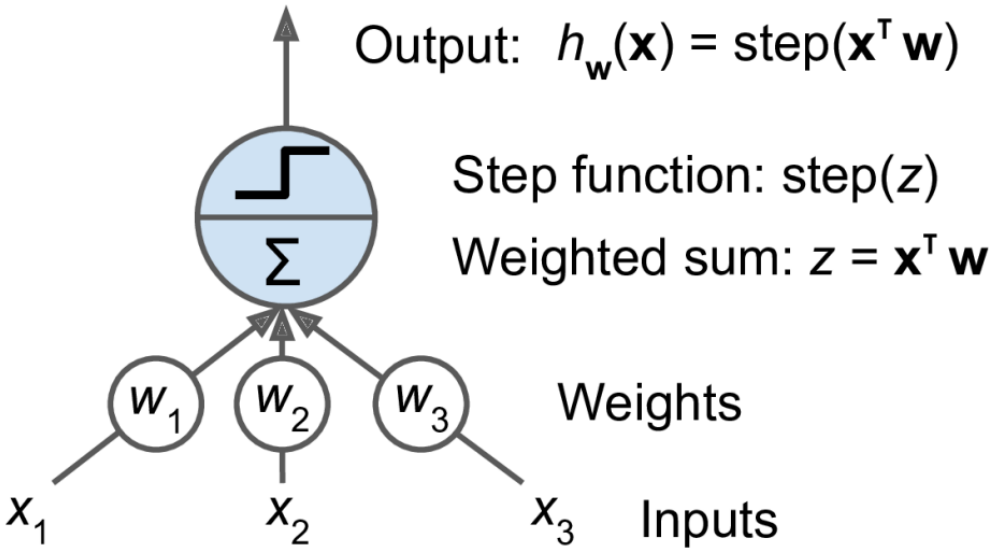


Figure 10-4. Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a step function

The Perceptron

The most common step function used in **Perceptrons** is the **Heaviside step function** (see Equation 10-1). Sometimes the sign function is used instead.

Equation 10-1. Common step functions used in Perceptrons (assuming threshold = 0)

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

The Perceptron

A **single TLU** can be used for simple linear binary classification. It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class (just like a Logistic Regression or linear SVM classifier). You could, for example, use a single TLU to classify iris flowers based on petal length and width (also adding an extra bias feature $x_0 = 1$, just like we did in previous chapters). Training a TLU in this case means finding the right values for w_0 , w_1 , and w_2 (the training algorithm is discussed shortly).

The Perceptron

A **Perceptron** is simply composed of a single layer of TLUs, with each TLU connected to all the inputs. When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), the layer is called a **fully connected layer**, or a **dense layer**. The **inputs of the Perceptron** are fed to special passthrough neurons called **input neurons**: they output whatever input they are fed. All the input neurons form the input layer. Moreover, an extra bias feature is generally added ($x_0 = 1$): it is typically represented using a special type of neuron called a bias neuron, which outputs 1 all the time. A Perceptron with two inputs and three outputs is represented in Figure 10-5. This Perceptron can classify instances simultaneously into three different binary classes, which makes it **a multioutput classifier**.

The Perceptron

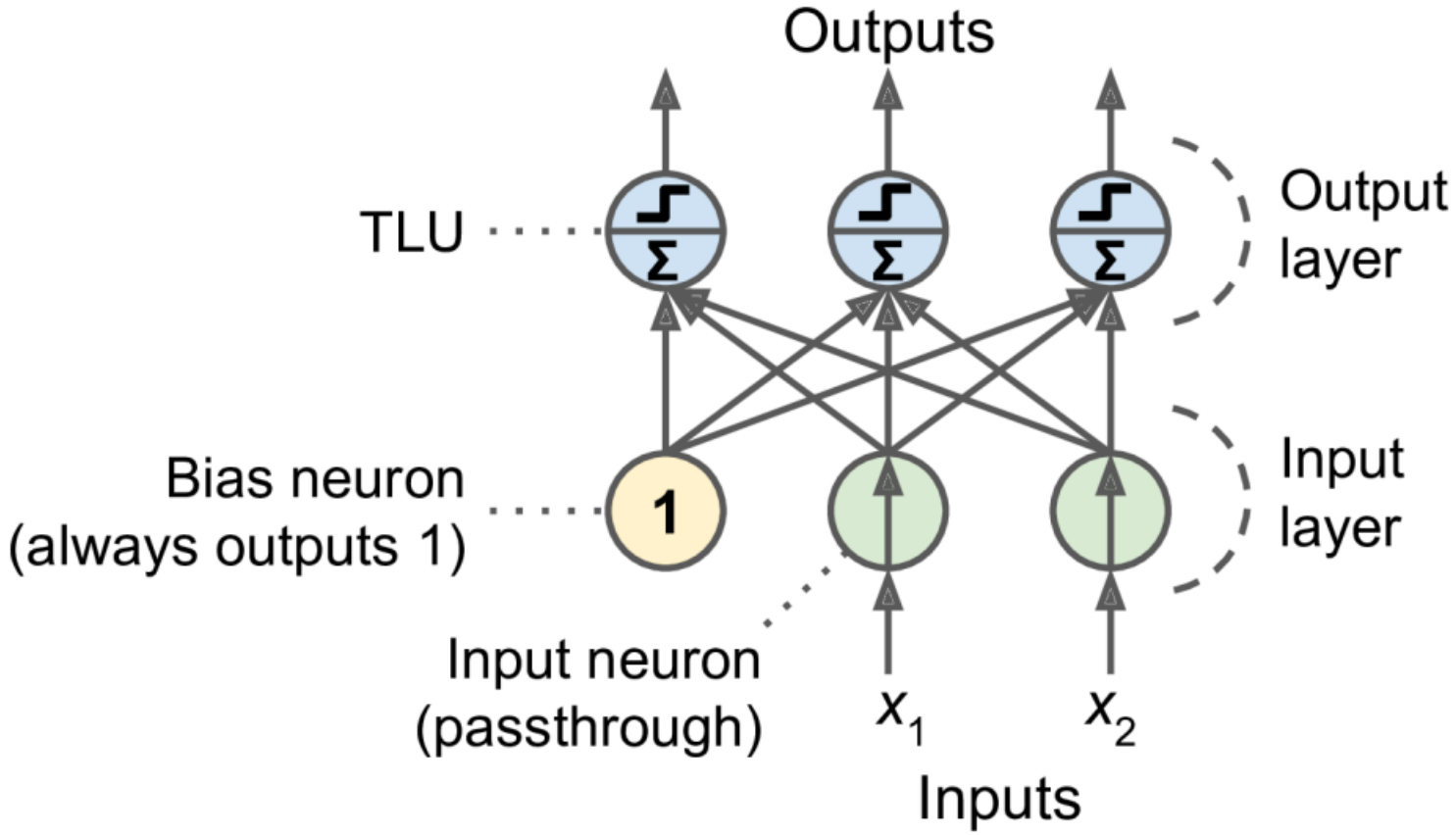


Figure 10-5. Architecture of a Perceptron with two input neurons, one bias neuron, and three output neurons

The Perceptron

Equation 10-2 makes it possible to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

Equation 10-2. Computing the outputs of a fully connected layer

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

In this equation:

- As always, X represents the matrix of input features. It has one row per instance and one column per feature.
- The weight matrix W contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector b contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron
- The function ϕ is called the **activation function**: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).

The Perceptron

So, how is a Perceptron trained? The Perceptron training algorithm proposed by Rosenblatt was largely inspired by **Hebb's rule.** In his 1949 book ***The Organization of Behavior*** (Wiley), Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. Siegrid Löwel later summarized Hebb's idea in the catchy phrase, "Cells that fire together, wire together"; that is, the connection weight between two neurons tends to increase when they fire simultaneously. This rule later became known as Hebb's rule (or Hebbian learning).

The Perceptron

- Perceptrons are trained using a variant of this rule that takes into account the **error** made by the network when it makes a prediction; the Perceptron learning rule reinforces connections that help reduce the error. More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction. The rule is shown in Equation 10-3.

Equation 10-3. Perceptron learning rule (weight update)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta \left(y_j - \hat{y}_j \right) x_i$$

The Perceptron

- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

Equation 10-3. Perceptron learning rule (weight update)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

The Perceptron

The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers). However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution. This is called the **Perceptron convergence theorem**.

The Perceptron

Scikit-Learn provides a Perceptron class that implements a single-TLU network. It can be used pretty much as you would expect—for example, on the iris dataset (introduced in Chapter 4):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)  # Iris setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

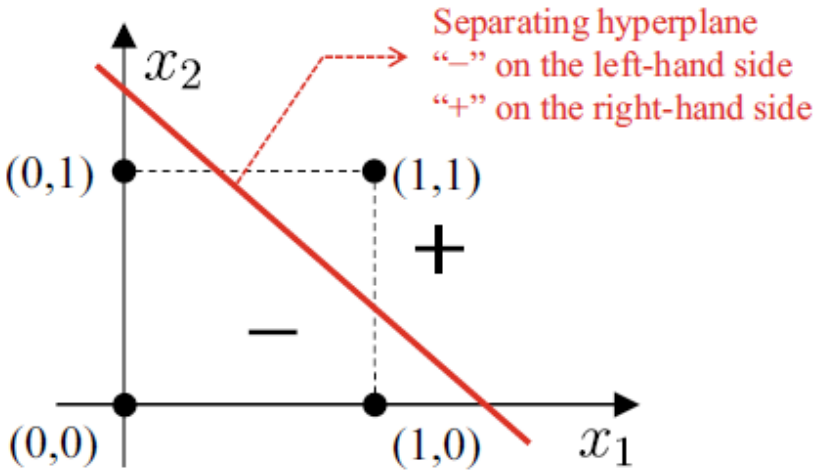
The Perceptron

You may have noticed that the **Perceptron learning algorithm** strongly resembles **Stochastic Gradient Descent**. In fact, Scikit-Learn's Perceptron class is equivalent to using an SGDClassifier with the following hyperparameters: loss="perceptron", learning_rate="constant", eta0=1 (the learning rate), and penalty=None (no regularization). Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they make predictions based on a hard threshold. This is one reason to prefer Logistic Regression over Perceptrons.

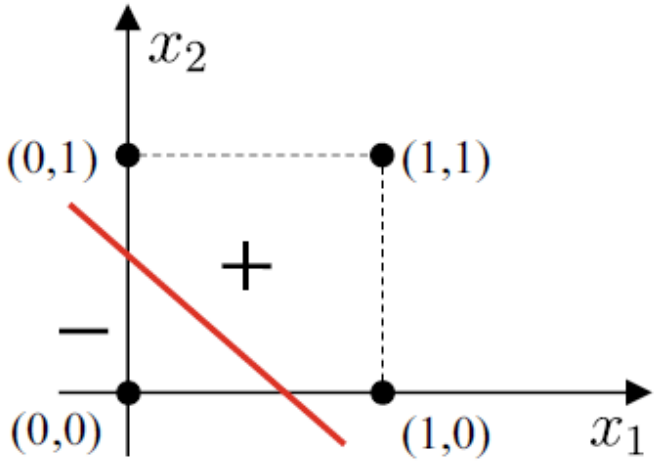
The Perceptron

In their 1969 monograph Perceptrons, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of Perceptrons—in particular, the fact that they are incapable of solving some trivial problems (e.g., the Exclusive OR (XOR) classification problem; see the left side of Figure 10-6). This is true of any other linear classification model (such as Logistic Regression classifiers), but researchers had expected much more from Perceptrons, and some were so disappointed that they dropped neural networks altogether in favor of higher-level problems such as logic, problem solving, and search.

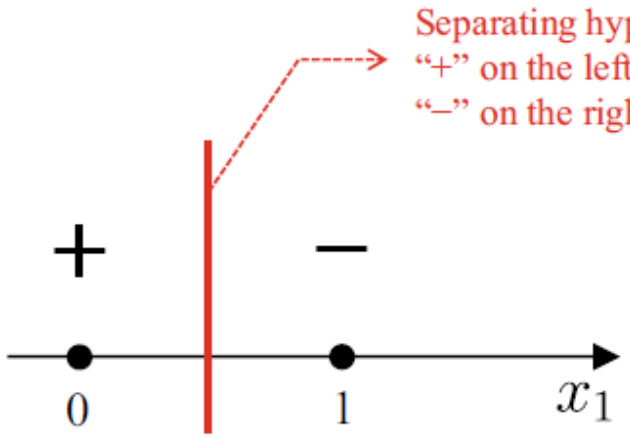
The Perceptron



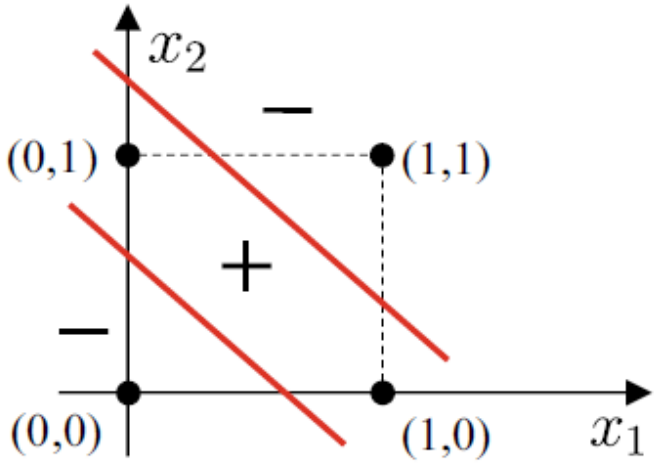
(a) "AND" ($x_1 \wedge x_2$).



(b) "OR" ($x_1 \vee x_2$).



(c) "NOT" ($\neg x_1$).



(d) "XOR" ($x_1 \oplus x_2$).

"AND", "OR", and "NOT"are linearly separable problems. "XOR" is a nonlinearly separable problem

The Perceptron

It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons. The resulting ANN is called a **Multilayer Perceptron (MLP).** An MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the right side of Figure 10-6: with inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1. All connections have a weight equal to 1, except the four connections where the weight is shown. Try verifying that this network indeed solves the XOR problem!

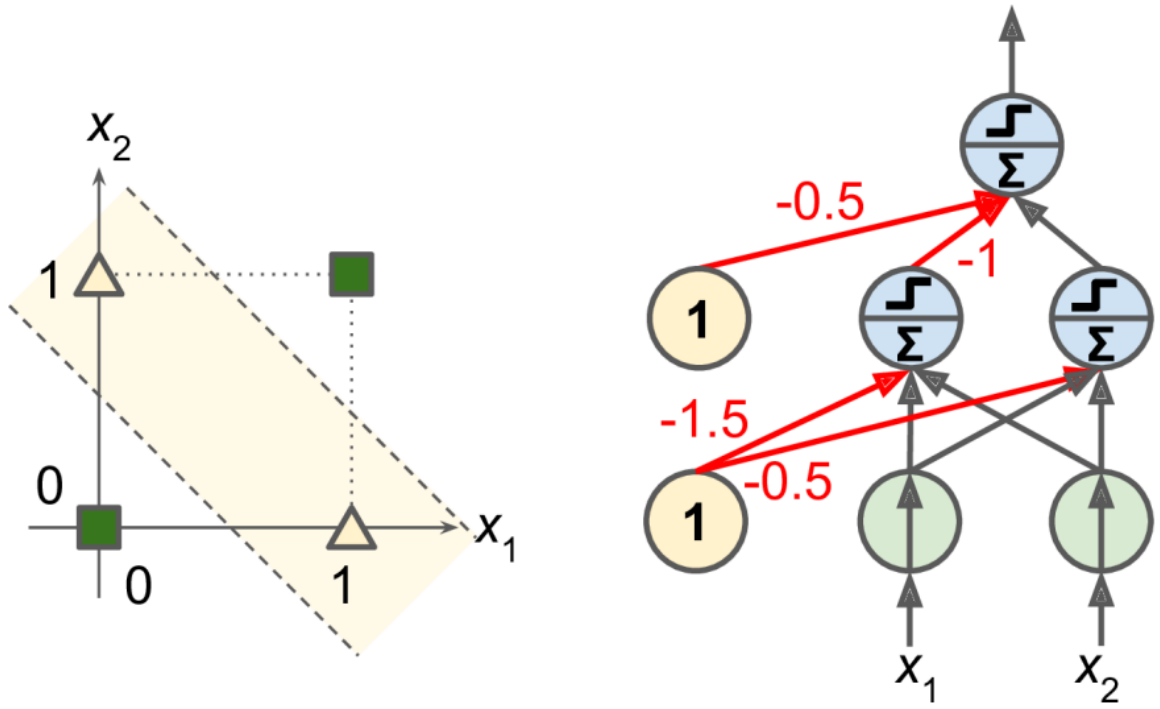
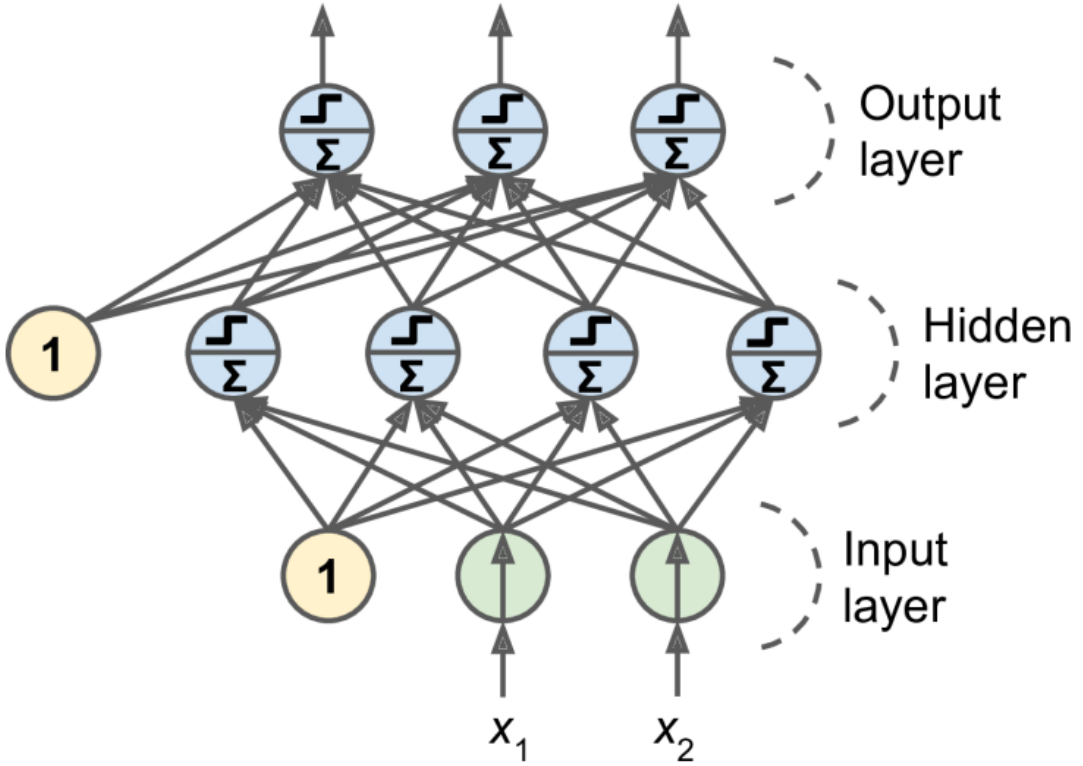


Figure 10-6. XOR classification problem and an MLP that solves it

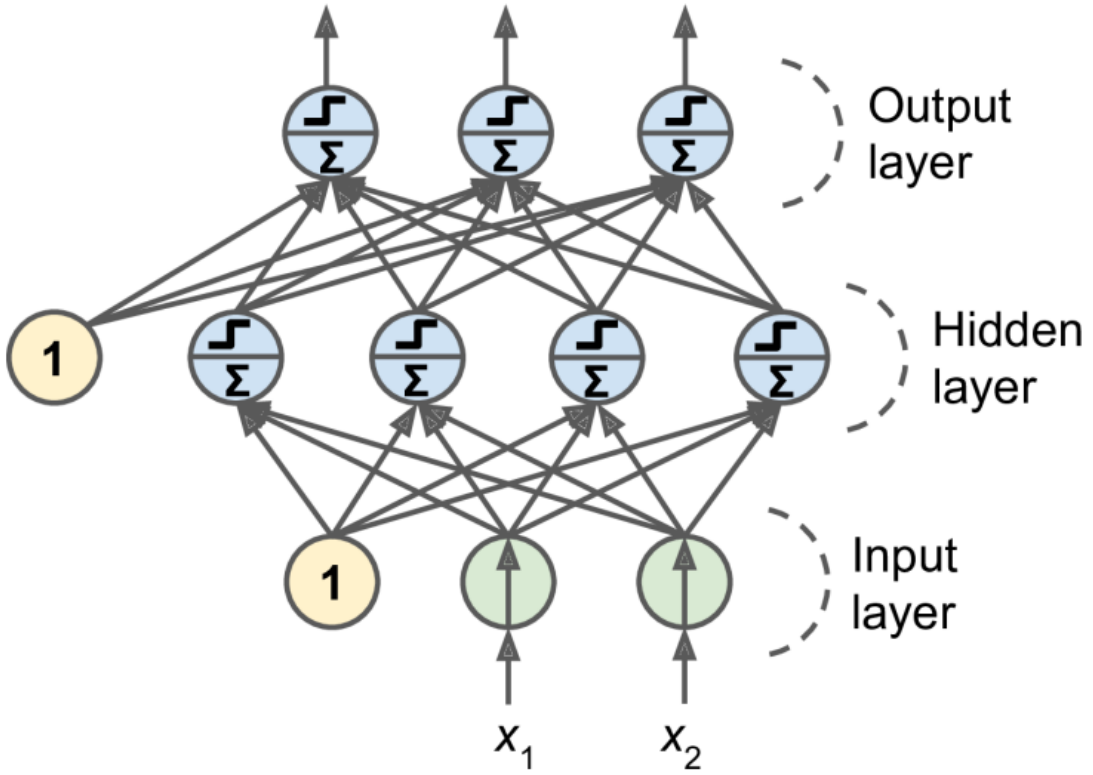
The Multilayer Perceptron and Backpropagation

An MLP is composed of one (passthrough) **input layer**, one or more layers of TLUs, called **hidden layers**, and one final layer of TLUs called the **output layer** (see Figure 10-7). The layers close to the input layer are usually called the lower layers, and the ones close to the outputs are usually called the upper layers. Every layer except the output layer includes a **bias neuron** and is **fully connected** to the next layer.



The Multilayer Perceptron and Backpropagation

When an ANN contains a deep stack of hidden layers, it is called a **deep neural network (DNN)**. The field of Deep Learning studies DNNs, and more generally models containing deep stacks of computations. Even so, many people talk about Deep Learning whenever neural networks are involved (even shallow ones).



The Multilayer Perceptron and Backpropagation

For many years researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper that introduced the **backpropagation training algorithm**, which is still used today. In short, it is **Gradient Descent** (introduced in Chapter 4) using an efficient technique for computing the gradients automatically: in just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter. In other words, it can find out how each connection weight and each bias term should be tweaked in order to reduce the error. Once it has these gradients, it just performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.

The Multilayer Perceptron and Backpropagation

Let's run through this algorithm in a bit more detail:

- It handles one mini-batch at a time (for example, containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an **epoch.**
- Each **mini-batch** is passed to the network's input layer, which sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch). The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the forward pass: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.

The Multilayer Perceptron and Backpropagation

- Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).
- Then it computes how much each output connection contributed to the error. This is done analytically by applying **the chain rule** (perhaps the most fundamental rule in calculus), which makes this step fast and precise.

The Multilayer Perceptron and Backpropagation

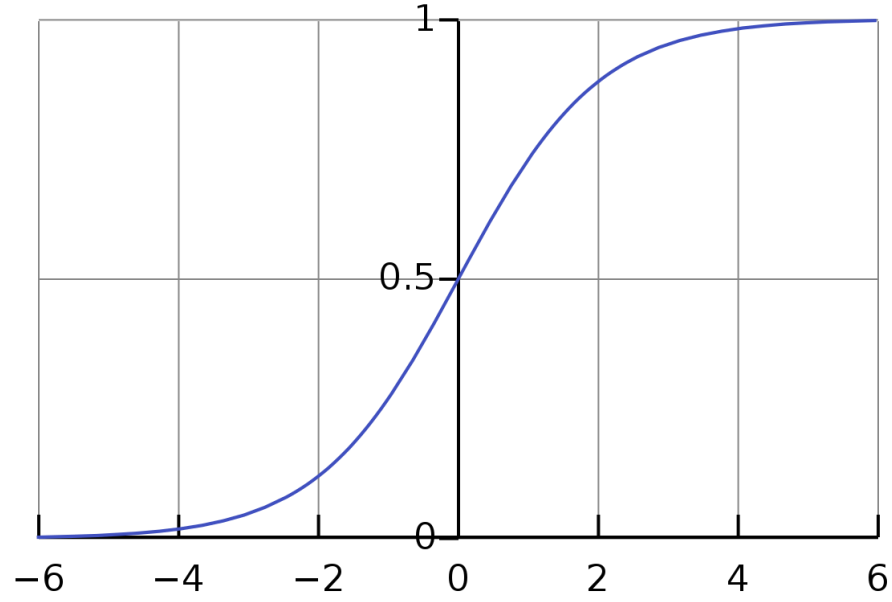
- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until the algorithm reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network (hence the name of the algorithm).
- Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed.

The Multilayer Perceptron and Backpropagation

This algorithm is so important that it’s worth summarizing it again: for each training instance, the backpropagation algorithm first makes a prediction (forward pass) and measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally tweaks the connection weights to reduce the error (Gradient Descent step).

The Multilayer Perceptron and Backpropagation

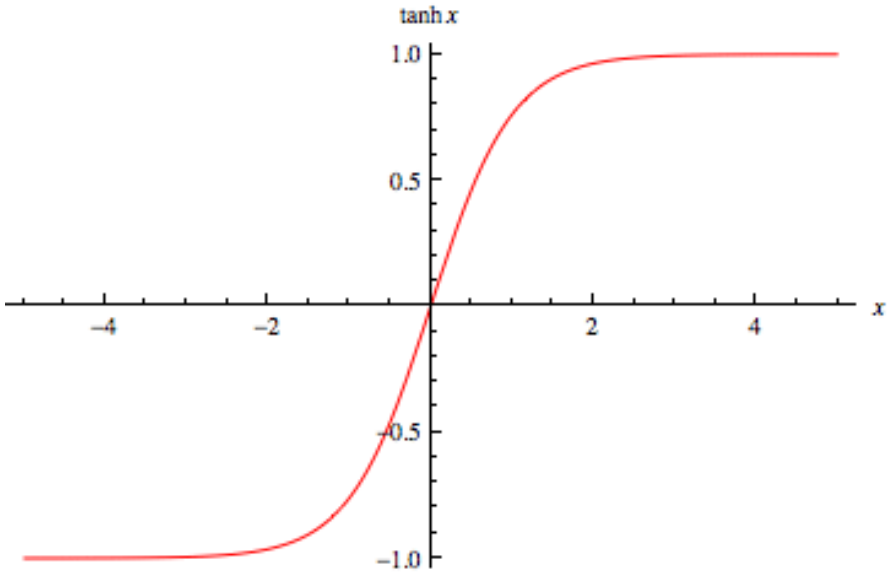
In order for this algorithm to work properly, its authors made a key change to the MLP's architecture: they replaced the step function with the logistic (sigmoid) function, $\sigma(z) = 1/(1 + \exp(-z))$. This was essential because the step function contains only flat segments, so there is no gradient to work with (Gradient Descent cannot move on a flat surface), while the logistic function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step. In fact, the backpropagation algorithm works well with many other activation functions, not just the logistic function



The Multilayer Perceptron and Backpropagation

- The hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$

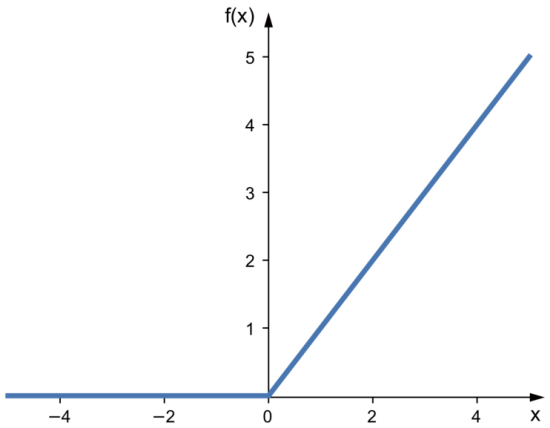
  Just like the logistic function, this activation function is S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic function). That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.



The Multilayer Perceptron and Backpropagation

- The Rectified Linear Unit function: $ReLU(z) = \max(0, z)$

  The ReLU function is continuous but unfortunately not differentiable at z = 0 (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for z < 0. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default. Most importantly, the fact that it does not have a maximum output value helps reduce some issues during Gradient Descent (we will come back to this in Chapter 11).



The Multilayer Perceptron and Backpropagation

These popular activation functions and their derivatives are represented in Figure 10-8. But wait! Why do we need activation functions in the first place? Well, if you chain several linear transformations, all you get is a linear transformation. For example, if $f(x) = 2x + 3$ and $g(x) = 5x - 1$, then chaining these two linear functions gives you another linear function: $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$. So, if you don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that. Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

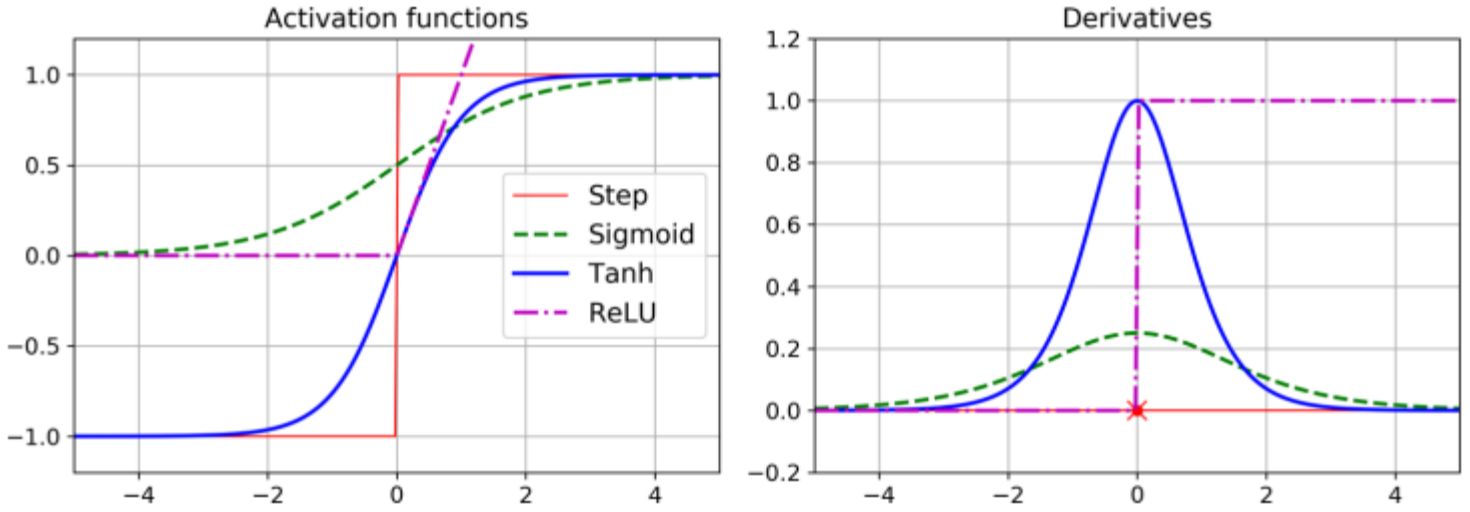
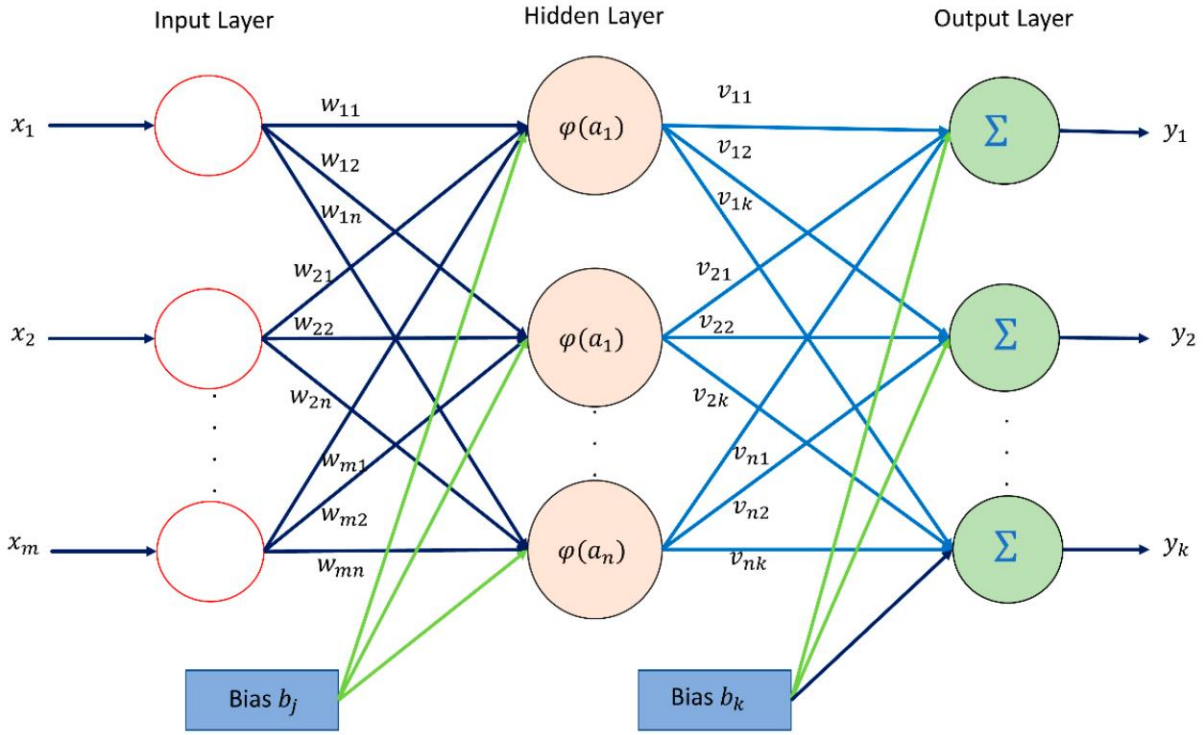


Figure 10-8. Activation functions and their derivatives

The Multilayer Perceptron and Backpropagation

OK! You know where neural nets came from, what their architecture is, and how to compute their outputs. You've also learned about the backpropagation algorithm. But what exactly can you do with them?

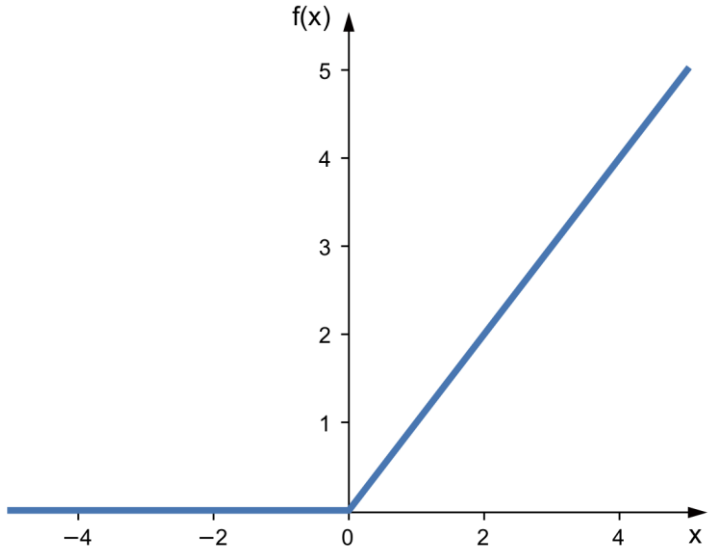


Regression MLPs

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value. For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So, you end up with four output neurons.

Regression MLPs

In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values. If you want to guarantee that the output will always be positive, then you can use the ReLU activation function in the output layer.

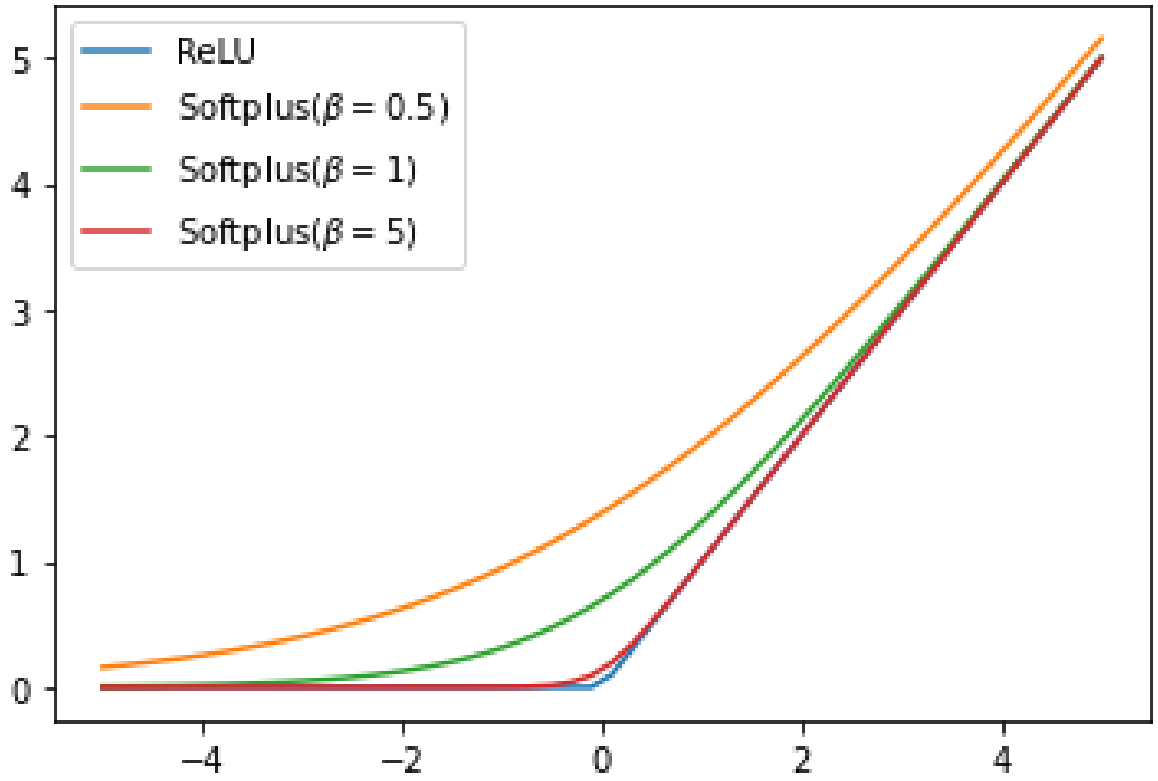


Regression MLPs

Alternatively, you can use the **softplus activation function,** which is a smooth variant of ReLU:

$$softplus(z) = \frac{1}{\beta}\log(1 + \exp(\beta z)).$$

It is close to 0 when z is negative, and close to z when z is positive. Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and then scale the labels to the appropriate range: 0 to 1 for the logistic function and –1 to 1 for the hyperbolic tangent.



Regression MLPs

The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the Huber loss, which is a combination of both

Table 10-1. Typical regression MLP architecture

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLPs

MLPs can also easily handle **multilabel binary classification** tasks (see Chapter 3). For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need two output neurons, both using the logistic activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have nonurgent ham, urgent ham, nonurgent spam, and perhaps even urgent spam (although that would probably be an error).

Classification MLPs

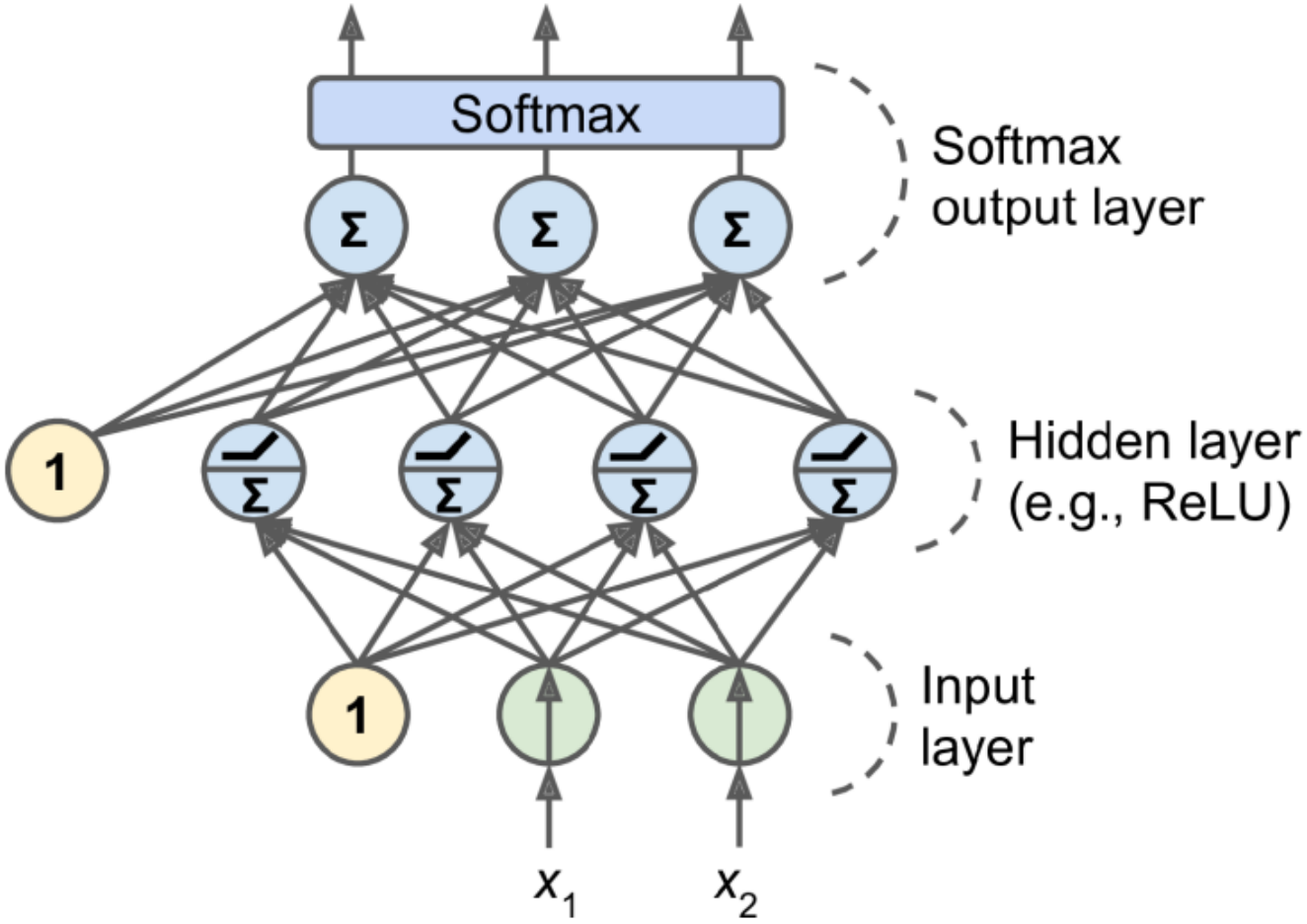


Figure 10-9. A modern MLP (including ReLU and softmax) for classification

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the **softmax activation function** for the whole output layer (see Figure 10-9). The softmax function (introduced in Chapter 4) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive). This is called **multiclass classification.**

Classification MLPs

Regarding the loss function, since we are predicting probability distributions, the **cross-entropy loss** (also called the log loss, see Chapter 4) is generally a good choice. Table 10-2 summarizes the typical architecture of a classification MLP

Table 10-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy