10

Introduction to Deep Learning

Deep learning is currently the most successful machine learning approach, with notable successes in object recognition, speech and language understanding, self-driving cars, automated Go playing, etc. It is not easy to give a single definition to such a broad and influential field; nevertheless here is a recent definition by Chris Manning:[1]

[1] Source: `https://hai.stanford.edu/sites/default/files/2020-09/AI-Definitions-HAI.pdf`.

> *Deep Learning is the use of large multi-layer (artificial) neural networks that compute with continuous (real number) representations, a little like the hierarchically-organized neurons in human brains. It is currently the most successful ML approach, usable for all types of ML, with better generalization from small data and better scaling to big data and compute budgets.*

Deep learning does not represent a specific a model per se, but rather categorizes a group of models called (artificial) neural networks (NNs) (or *deep nets*) which involve several computational layers. Linear models studied in earlier chapters, such as logistic regression in Section 4.2, can be seen as special sub-cases involving only a single layer. The main difference, however, is that general deep nets employ nonlinearity in between each layer, which allows a much broader scale of expressivity. Also, the multiple layers in a neural net can be viewed as computing "intermediate representations" of the data, or "high level features" before arriving at its final answer. By contrast, a linear model works only with the data representation it was given.

Deep nets come in various types, including Feed-Forward NNs (FFNNs), Convolutional NNs (CNNs), Recurrent NNs (RNNs), Residual Nets, and Transformers. [2] Training uses a variant of *Gradient Descent*, and the gradient of the loss is computed using an algorithm called *backpropagation*.

[2] Interestingly, a technique called *Neural Architecture Search* uses deep learning to design custom deep learning architectures for a given task.

Due to the immense popularity of deep learning, a variety of software environments such as Tensorflow and PyTorch allow quick implementation of deep learning models. You will encounter them in the homework.

10.1 A Brief History

Neural networks are inspired by the biological processes present within the brain. The concept of an artificial neuron was first outlined by Warren MuCulloch and Walter Pitts in the 1940s. [3] Later in 1986, backpropagation was discovered and provided a way for efficiently applying gradient-based training methods to these models. [4] The basic frameworks for CNNs and modern training soon followed in the late 1980s. [5] However, by the 21st century deep learning had gone out of fashion. This changed in 2012, when Krizhevsky, Sutskever, and Hinton leveraged deep learning techniques through their *AlexNet* model and set new standards for performance on the ImageNet dataset. [6] Deep learning has since begun a resurgence throughout the last decade, boosted by some key factors:

[3] Paper: `https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf`.

[4] Paper: `https://www.nature.com/articles/323533a0`.

[5] Paper: `https://link.springer.com/article/10.1007/BF00344251`.

[6] Paper: `https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

- Hardware, such as GPU and TPU (Tensor Processing Unit, specifically developed for neural network machine learning) technology have made training faster.
- The development of novel neural network architecutres as well as better algorithms for training neural networks.
- A vast amount of data collection, boosted by the spread of the internet, have augmented the performance of NN models.
- Popular frameworks, such Tensorflow and PyTorch, have made it easier to prototype and deploy NN architectures.
- Commercial payoff has caused tech corporations to invest more financial resources.

Each of the reasons listed above have interfaced in a positively reinforcing cycle, causing the acceleration of this technology into the foreseeable future.

10.2 Anatomy of a Neural Network

10.2.1 Artificial Neuron

An *artificial neuron*, or a *node*, is the main component of a neural network. Artificial neurons were inspired by early work on neurons in animal brains, with the analogies in Table 10.1.

Formally, a node is a computational unit which receives m scalar inputs and outputs 1 scalar. This scalar output can be used as an input for a different neuron.

Consider the vector $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_m)$ of m inputs. A neuron internally maintains a trainable weight vector $\vec{\mathbf{w}} = (w_1, w_2, \ldots, w_m)$

Biological neuron	Artificial neuron
Dendrites	Input
Cell Nucleus / Soma	Node
Axon	Output
Synapse	Interconnections

Table 10.1: A comparison between biological neurons in the brain and artificial neurons in neural networks

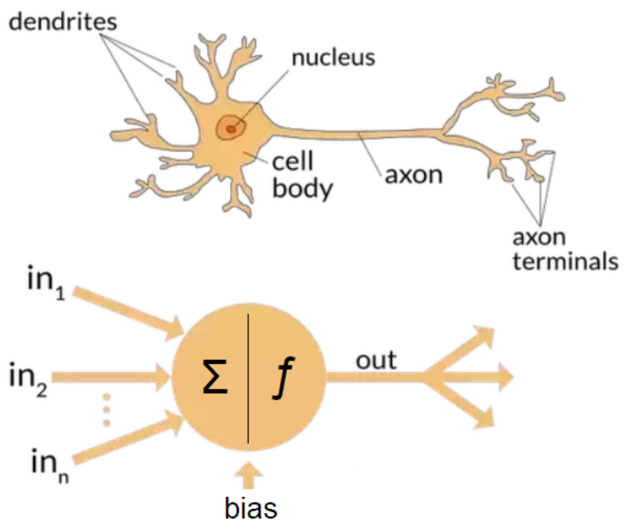

Figure 10.1: A comparison between a brain neuron and an artificial neuron.

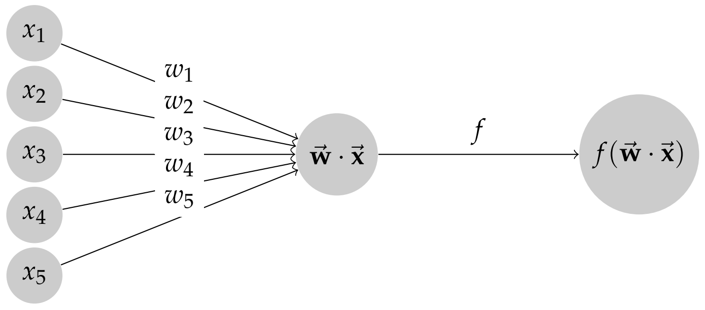
x_1
x_2
x_3
x_4
x_5
w_1
w_2
w_3
w_4
w_5
$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$
f
$f(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$

Figure 10.2: A sample artificial neuron.

and optionally a nonlinear activation function $f : \mathbb{R} \to \mathbb{R}$ and outputs the following value: [7]

$$y = f(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) \tag{10.1}$$

[7] If no activation function is chosen, we can assume that f is an identity function $f(z) = z$.

We can also add a scalar bias b before applying the activation function $f(z)$ in which case the output will look like the following: [8]

$$y = f(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b)$$

[8] If we introduce a dummy variable for the constant bias term as in Chapter 1 we can absorb the bias term into the equation in (10.1).

10.2.2 *Activation Functions*

An artificial neuron can choose its nonlinear activation function $f(z)$ from a variety of options. One such choice is the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{10.2}$$

Note that in this case, the neuron represents a logistic regression unit. [9] Another popular activation function is the hyperbolic tangent, which is similar to the sigmoid function:

[9] However, in this context the output is *not* considered to be a subjective probability as in the case of standard logistic regression.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{10.3}$$

In fact, we can rewrite the hyperbolic tangent in terms of sigmoid:

$$\tanh(z) = 2\sigma(2z) - 1 \qquad \text{(10.3 revisited)}$$

According to this expression, tanh function can be viewed as a rescaled sigmoid function. The key difference is: the range of $\sigma(z)$ is $(0, 1)$ and the range of $\tanh(z)$ is $(-1, 1)$.

Arguably the most commonly used activation function is the Rectified Linear Unit, or ReLU:

$$\text{ReLU}(z) = [z]_+ = \max\{z, 0\} \tag{10.4}$$

There are several benefits to the ReLU activation function. It is far cheaper to compute than the previous two alternatives and avoids the "vanishing gradient" problem. [10] With sigmoid and hyperbolic tangent activation functions, the vanishing gradient problem happens when $z = \vec{\mathbf{x}} \cdot \vec{\mathbf{w}}$ has high absolute values, but ReLU avoids this problem because the derivative is exactly 1 even for high values of z.

[10] The vanishing gradient problem refers to a situation where the derivative of a certain step is too close to 0, which can stall the gradient-based learning techniques common in deep learning.

Example 10.2.1. *Consider a vector* $\vec{\mathbf{x}} = (-2, -1, 0, 1, 2)$ *of inputs and a neuron with the weights* $\vec{\mathbf{w}} = (1, 1, 1, 1, 1)$*. If the activation function of this neuron is the sigmoid, then the output will be:*

$$y = \sigma(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) = \sigma(0) = \frac{1}{2}$$

If the activation is ReLU, it will output:

$$[\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}]_+ = [0]_+ = 0$$

Problem 10.2.2. *Consider a neuron with the weights* $\vec{\mathbf{w}} = (1, 1, 5, 1, 1)$ *and the ReLU activation function. What will the outputs* y_1 *and* y_2 *be for the inputs* $\vec{\mathbf{x}}_1 = (-2, -2, 0, 1, 2)$ *and* $\vec{\mathbf{x}}_2 = (2, -1, 0, 1, 2)$ *respectively?*

10.2.3 Neural Network

A neural network consists of nodes connected with directed edges, where each edge has a trainable parameter called its "weight" and each node has an activation function as well as associated parameter(s). There are designated *input* nodes and *output nodes*. The input nodes are given some input values, and the rest of the network then computes as follows: each node produces its output by taking the

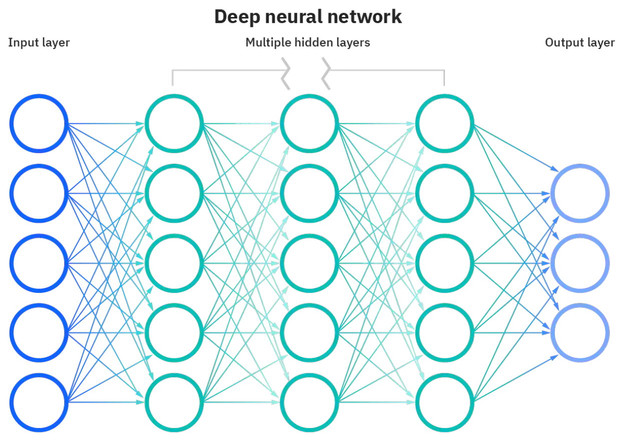

Figure 10.3: A sample neural network design. Each circle represents one artificial neuron. Two nodes being connected by an edge means that the output of the node on the left is being used as one of the inputs for the node on the right.

values produced by all nodes that have a directed edge to it. If the directed graph of connections is *acyclic* — which is the case in most popular architectures — this process of producing the values takes finite time and we end up with a unique value at each of the output nodes. [11] The term *hidden nodes* is used for nodes that are not input or output nodes.

[11] We will not study *Recurrent Neural Nets (RNNs)*, where the graph contains cycles. These used to be popular until a few years ago, and present special difficulties due to the presence of directed loops. For instance, can you come up with instances where the output is not well-defined?

10.3 Why Deep Learning?

Now that we are aware of the basic building blocks of neural networks, let's consider why we prefer these models over techniques explored in previous chapters. The key understanding is that the models previously discussed are fundamentally *linear* in nature. For instance, if we do binary classification, where the data point $\vec{\mathbf{x}}$ is mapped to a label based on $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$, then this corresponds to separating the points with label $+1$ from the points with label -1 via a linear hyperplane $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = 0$. But such models are not a good choice for datasets which are not linearly separable. Deep learning is inherently *nonlinear* and is able to do classification in many settings where linear classification cannot work.

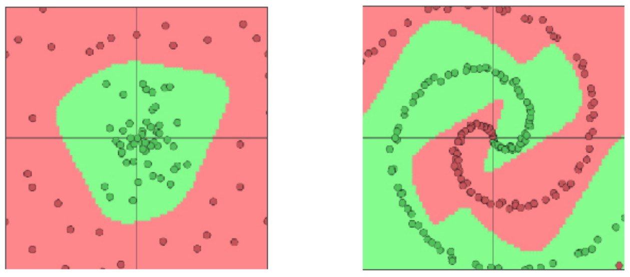

Figure 10.4: Some examples of datasets that are not linearly separable.

10.3.1 The XOR Problem

Consider the boolean function *XOR* with the truth table in Table 10.2.

x_1	x_2	**XOR**
0	0	0
0	1	1
1	0	1
1	1	0

Table 10.2: The truth table for the *XOR* Boolean function.

Let us first attempt to represent the *XOR* function with a single linear neuron. That is, consider a neuron that takes two inputs x_1, x_2 with weights w_1, w_2, a bias term b, and the following Heaveside step activation function: [12]

[12] This neuron is called a *linear perceptron*. It uses a nonlinear activation function, but the nonlinearity is strictly for the binary classification in the final step. The boundary of the classification is still linear.

$$g(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases} \tag{10.5}$$

Proposition 10.3.1. *There are no values of w_1, w_2, b such that the linear neuron defined by the values represent the XOR function.*

Proof. Assume to the contrary that there are such values. Let $\vec{\mathbf{x}}_1 = (0,0), \vec{\mathbf{x}}_2 = (0,1), \vec{\mathbf{x}}_3 = (1,0), \vec{\mathbf{x}}_4 = (1,1)$. Then we know that

$$\begin{aligned} g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_1 + b) = g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_4 + b) = 0 \\ g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_2 + b) = g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_3 + b) = 1 \end{aligned}$$

which implies that

$$\begin{aligned} \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_1 + b \leq 0, \quad \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_4 + b \leq 0 \\ \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_2 + b > 0, \quad \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_3 + b > 0 \end{aligned}$$

Now let $\vec{\mathbf{x}} = \left(\frac{1}{2}, \frac{1}{2}\right)$. Since we have $\vec{\mathbf{x}} = \frac{1}{2}\vec{\mathbf{x}}_1 + \frac{1}{2}\vec{\mathbf{x}}_4$, we should have

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b = \frac{1}{2} \cdot ((\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_1 + b) + (\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_4 + b)) \leq 0$$

since we are taking the average of two non-positive numbers. But at the same time, since $\vec{\mathbf{x}} = \frac{1}{2}\vec{\mathbf{x}}_2 + \frac{1}{2}\vec{\mathbf{x}}_3$, we should have

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b = \frac{1}{2} \cdot ((\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_2 + b) + (\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_3 + b)) > 0$$

since we are taking the average of two positive numbers. This leads to a contradiction. □

Problem 10.3.2. *Verify that the AND, OR Boolean functions can be represented by a single linear node.*

Figure 10.5 visualizes the truth table for *XOR* in the 2*D* plane. The two axes represent the two inputs x_1 and x_2; blue circles denote that $y = 1$; and white circles denote that $y = 0$. A single linear neuron can be understood as drawing a red line that can separate white points from blue points. Notice that it is possible to draw such a line for *AND* and *OR* functions, but the data points that characterize the *XOR* function are not linearly separable.

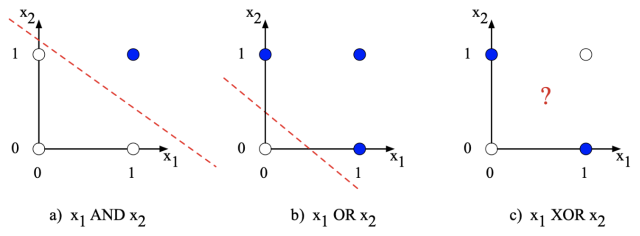

Figure 10.5: The data points that characterize the *XOR* function are not linearly separable.

Instead, we will leverage neural networks to solve this problem. Let us design an architecture with inputs x_1, x_2, a hidden layer with two nodes h_1, h_2, and a final output layer with one node y_1. We assign the *ReLU* activation function to the hidden nodes and define weights and biases as shown in Figure 10.6.

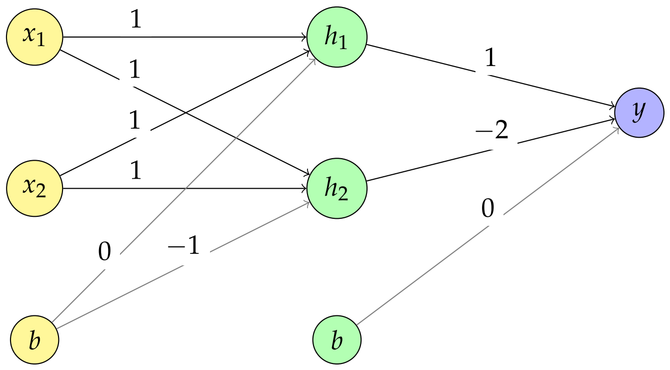

Figure 10.6: A sample neural network which computes the *XOR* of its inputs x_1 and x_2. The weights for inputs are shown by black arrows, while bias terms are shown by grey arrows.

To be more explicit, the neural network is defined by the following three neurons:

$$\begin{aligned} h_1 &= ReLU(x_1 + x_2) \\ h_2 &= ReLU(x_1 + x_2 - 1) \\ y_1 &= ReLU(h_1 - 2h_2) \end{aligned}$$

Problem 10.3.3. *Verify that the model in Figure 10.6 represents the XOR function by constructing a truth table.*

The main difference between the single linear neuron approach and the neural network for the *XOR* function is that the network now has two layers of neurons. If we only focus on the final layer of the neural network, we expect the boundary of the binary classification to be linear to the values of h_1, h_2. However, the values of h_1, h_2 are *not* linear to the input values x_1, x_2 because the hidden nodes utilize a *nonlinear* activation function. Hence the boundary of the classification is also *not* linear to the input values x_1, x_2. The nonlinear activation function transforms the input space into a space where the XOR operation is *linearly separable*. As shown in Figure 10.7, the h space is quite clearly linearly separable in contrast to the original x space.

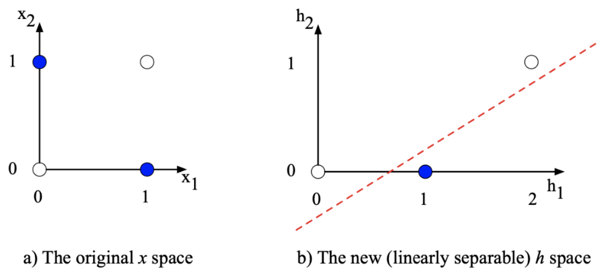

Figure 10.7: Unlike the x space, after applying the nonlinear *ReLU* activation function, the mapped h space is linearly separable.

10.4 Multi-class Classification

Neural networks, like multi-class regression in Chapter 4, can be used for classification tasks where the number of possible labels is larger than 2. Real-life scenarios include hand-written digit recognition on the MNIST dataset, where the model designer could use ten different classes to correspond to each possible digit. Another possible example is an speech recognition language model, where the model is trained to distinguish between sounds of $|V|$ vocabularies.

It turns out that such functionality can be added by simply including as many output neurons as desired classes in the output layer. Then, the values of the output neurons will be converted into a probability distribution $\Pr[y = k]$ over the number of classes.

10.4.1 Softmax Function

Just as in Chapter 4, we use the softmax function for the purpose of the multi-class classification. See Chapter 19 for the definition of the softmax function.

Example 10.4.1. *Say* $\vec{\mathbf{o}} = (3, 0, 1)$ *are the values of the output neurons of a neural network before applying the activation function. If we decide to apply the softmax function as the activation function, the final outputs of the* *network will be* $softmax(\vec{\mathbf{o}}) \simeq (0.84, 0.04, 0.11)$. *If the network was trying to solve a multi-class classification task, we can understand that the given input is most likely to be of class* 1, *with probability* 0.84 *according to the model.*

One notable property of the softmax function is that the output of the function is the same if all coordinates of the input vector is shifted by the same amount; that is $softmax(\vec{\mathbf{z}}) = softmax(\vec{\mathbf{z}} + c \cdot \vec{\mathbf{1}})$ for any $c \in \mathbb{R}$, where $\vec{\mathbf{1}} = (1, 1, \ldots, 1)$ is a vector of all ones.

Example 10.4.2. *Consider two vectors* $\vec{\mathbf{z}}_1 = (5, 2, 3)$ *and* $\vec{\mathbf{z}}_2 = (3, 0, 1)$. *Then* $softmax(\vec{\mathbf{z}}_1) = softmax(\vec{\mathbf{z}}_2)$ *because* $\vec{\mathbf{z}}_2 = \vec{\mathbf{z}}_1 - (2, 2, 2)$.

Problem 10.4.3. *Prove the property that* $softmax(\vec{\mathbf{z}}) = softmax(\vec{\mathbf{z}} + c \cdot \vec{\mathbf{1}})$ *for any* $c \in \mathbb{R}$. *(Hint: multiply both the numerator and the denominator of* $softmax(\vec{\mathbf{z}})_k$ *by* $\exp(c)$.*)*