In the case of a non-zero $u(t)$ over the full time axis, there is continuous injection of a signal, and consequently the system can not reach an asymptotic state. It is forced to a continuously transient motion, because of the presence of the signal $u(t)$, which is then said to trigger a transient trajectory in the phase space of the delay dynamics. Such a transient trajectory will be later used for information processing according to a brain-inspired technique known as *reservoir computing*.

Driven systems are common for other applications. For example, in the case of a *system identification* [44], the system under analysis is driven by a sinusoidal signal $u(t)$. Another example is the optical information processing demonstrated in [45] for chaos-based encryption application. Information processing systems are driven, in fact, by an external information input signal $u(t)$. For instance, voice recognition systems are driven by a speech waveform $u(t)$, self-driving cars are driven by environment signals $\mathbf{u}(t)$ and so on. In the most general context, the utilization of such driven networks of nonlinear nodes for information processing defines *artificial neural networks*, a prominent subject in machine learning field.

1.2 Artificial neural networks

What I cannot create, I do not understand.
– Richard Feynman, Nobel Prize winner in Physics (1965)

Despite the fact that computers are ubiquitous today, their working principle is far from biological. The human brain, which consumes as little as 20 W of power [46], is able to recognize familiar faces in presence of noise (e.g. poor lighting conditions) in fractions of seconds. However, this task is tough even for the fastest modern computers. This capability poses one of the central problems of biologically inspired hardware: how can information be processed more efficiently.

Another significant problem is to understand the brain from biological view point. It is known that the human brain consists of hundred billions of neurons, which are estimated to have thousand trillions of interconnections. However, little is known about how does that biological network operate. Understanding of this intricate organization might be crucial for medicine and related fields. So-called constructive approach has a goal to approximate the brain, or at least a network of neurons. Building a system analogous to the brain could help to shed some light onto the organ, which is still treated mostly as a blackbox.

1.2.1 Approximation of neural behavior

Though the anatomy of neural cells is known (Fig. 1.2.1, upper), modeling collective cell dynamics is an open question. A biologically justified model of a neuron was introduced by Hodgkin and Huxley in 1952 [5] (Nobel Prize 1963). The model is a four-dimensional nonlinear system of ordinary differential equations (ODEs). However, computationally simulating a network of such neurons is extremely hard. Should one try to compute, for instance, the dynamics of 10^9 neurons (only 1% of estimated total neurons in the human brain), the model of Hodgkin-Huxley becomes impractical because of the amount required resources, such as processing power, memory, and storage. It is worth mentioning this computational problem represents one of the global challenges of natural sciences and is addressed by the Human Brain Project[3] (1 billion €).

A much simpler neuronal model was proposed by Izhikevich [47]. The model is a two-dimensional system of ODEs:

$$\begin{aligned} \dot{v} &= 0.04v^2 + 5v + 140 - u + I, \\ \dot{u} &= a(bv - u), \end{aligned} \tag{1.2.1}$$

where v and u are dimensionless variables, characterizing the membrane potential of the neuron and a membrane recovery variable, respectively. a and b are dimensionless parameters describing the timescale of the recovery variable u and the sensitivity of the variable u to the fluctuations of membrane potential v, respectively. External forcing I represents synaptic currents, making Eq. (1.2.1) a driven system (Section 1.1.5). Being significantly simpler compared to the original Hodgkin-Huxley model, system of equations (1.2.1) is still able to capture such typical neural regimes as *spiking*, *chattering*, and *bursting*. Thus, it can be used to model biologically-plausible neural behavior. However, even computation with this simplified model is still a complicated engineering task when dealing with billions of neurons and, therefore, is not yet feasible.

To achieve computational efficiency for biologically-inspired information processing, one may need to make a further step towards system simplification. Instead of individual neurons, modeling a collective behavior of neuronal network can be attempted. Here *artificial neural networks*, referred further in the text as neural networks (NNs), are coming into the play.

Figure 1.2.1 illustrates the idea behind the transition from a biological to a very simplified mathematical, formal model of the neuron. The biological neuron (Fig. 1.2.1, upper) has a cell body (soma) that receives electrical impulses from other neurons

[3] http://www.humanbrainproject.eu

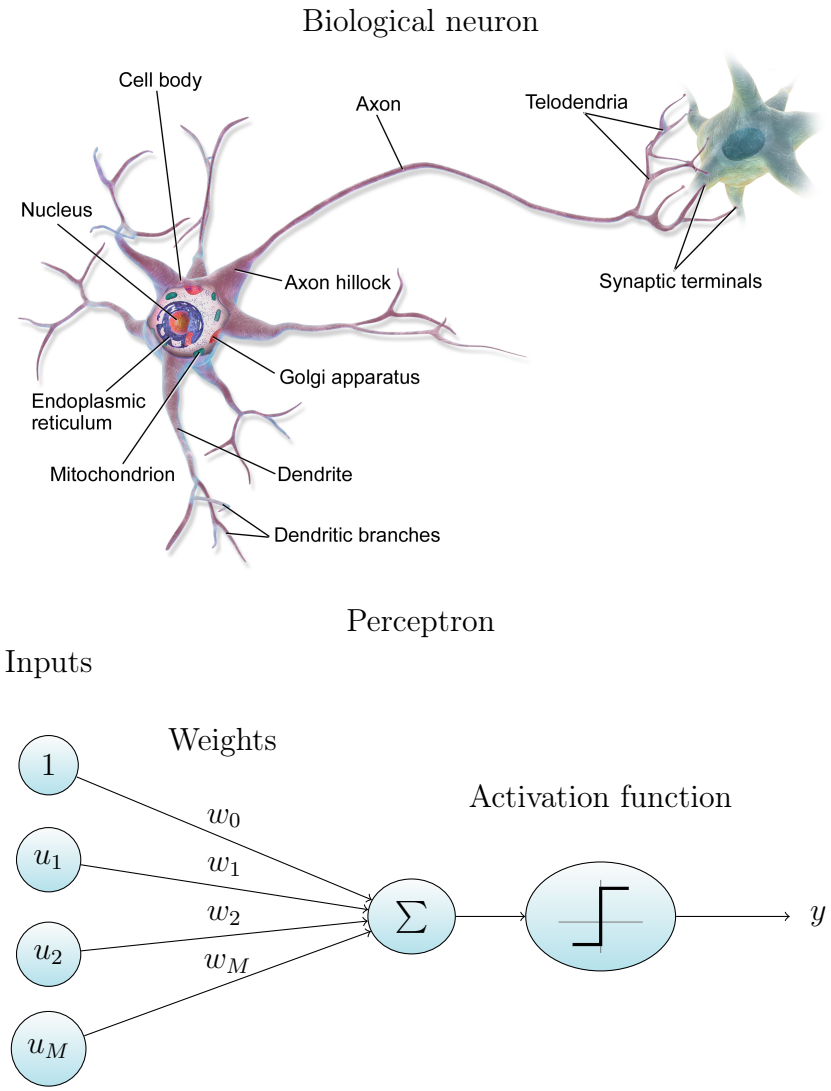

Figure 1.2.1: Simplification of the neuron's architecture for information processing. **Upper**: The biological neuron's model. Image source: [48]. **Lower**: *The perceptron*, a formal neuron's model. The input vector $\mathbf{u} = (1, u_1, u_2, \dots u_M)$ is weighted and nonlinearly transformed via the activation function, creating a scalar output value y.

through dendrites. The soma performs non-linear processing over the input signals. If the total input is higher than a certain threshold, then an output impulse is produced [49]. This impulse is then transmitted through an axon, the long projection of the neuronal cell that conducts electrical impulses away from the cell body. In contrast to the biological neuron, the formal neuron called *perceptron* [6] (Fig. 1.2.1, lower) captures only the essential of the neuron's features: input connections, nonlinear transformation, and the output. Similarly to the neuron, the perceptron performs a

thresholded transformation over the received information. As it can be seen from Fig. 1.2.1, perceptron is an extremely refined version of its biological counterpart and we stress that it is a purely mathematical concept used for the purpose of information processing.

A single-layer perceptron is mathematically described as:

$$y = f\left(\sum_{i=0}^{M} w_i u_i\right), \tag{1.2.2}$$

where $\mathbf{u} = (1, u_1, u_2, \dots u_M)$ is an input vector (Fig. 1.2.1, lower), $\mathbf{w} = (w_0, w_{1,} w_{2,} \dots w_M)$ is a weights row-vector, and f is a nonlinear *activation function*, e.g. step function, sigmoid, etc. Note, expression $\sum_{i=0}^{M} w_i u_i$ is the inner product between the vectors $\mathbf{u}$ and $\mathbf{w}$, where the first term $w_0 \cdot 1$ in Eq. (1.2.2) is the input bias. Thus, using a dot product notation $(\cdot)$ for vectors, Eq. (1.2.2) can be rewritten as:

$$y = f\left(\mathbf{w} \cdot \mathbf{u}\right). \tag{1.2.3}$$

Comparing Eq. (1.2.3) with physical models of the neuron, such as Eq. (1.2.1), we point out that there are no time-dependent variables involved in the model anymore. Thus, perceptron is a mathematical concept, a memoryless function, which maps the input vector $\mathbf{x}$ into the output y. It is necessary to highlight that even though there is a nonlinear activation function, the perceptron acts as a *linear* separator, a separation hyperplane in a n-dimensional phase space.

1.2.2 Feedforward neural networks

Not every problem is linear. In Fig. 1.2.2, there is an example of inputs which cannot be separated using a straight line, i.e. by a hyperplane in a 2-dimensional space. The input vectors $\mathbf{u} = (u_1, u_2)$ and the respective target results y of binary operation, known as XOR (eXclusive OR), are given by Table 1.1.

Therefore, linear separators such as perceptron cannot solve the XOR problem. To solve it, an additional layer of artificial neurons has to be introduced (Fig. 1.2.3). Now, the computation has to be done in two stages. In the first stage, the hidden layer $\mathbf{x}$ is a vector calculated as:

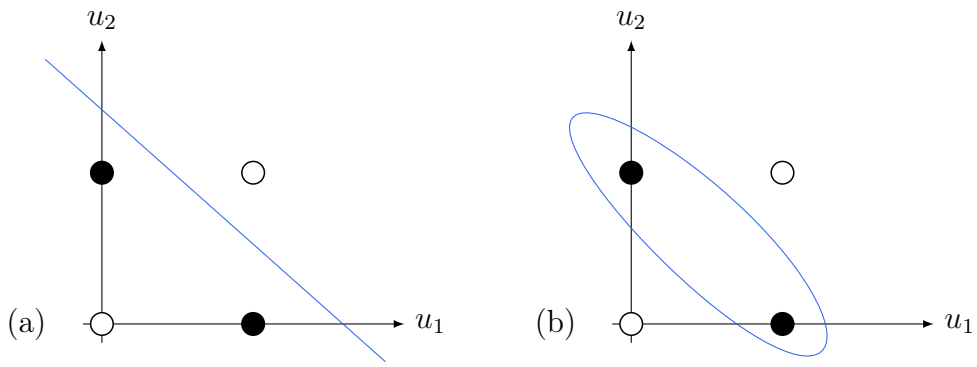

Figure 1.2.2: (a) The XOR operator cannot be resolved by a separation with a straight line (a hyperplane in 2D space). (b) Only transition to a higher-dimensional space, which is equivalent to drawing a curve in 2D space, may allow separation between classes. Coding: filled dots – 1, hollow dots – 0.

u_1	u_2	$\mathrm{XOR}(u_1, u_2)$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.1: The XOR binary function

$$\mathbf{x} = f_1\left(W^I\mathbf{u}\right), \tag{1.2.4}$$

where $\mathbf{u} = (u_1, \dots u_M)$ and $\mathbf{x} = (x_1, \dots x_N)$ are vectors. Matrix $W^I \in \mathbb{R}^{N\times(M+1)}$ is the input weights matrix, thus the hidden layer consists of multiple perceptrons given by row-vectors of W^I. The result of matrix-vector product $W^I\mathbf{u}$ is a vector, which is transformed by the perceptron's transfer function f_1. The function f_1 is applied element-wise, i.e. $f_1(\mathbf{x}) = (f(x_1), \dots f(x_N))$. Note that if $N = 1$, Eq. (1.2.4) is reduced to Eq. (1.2.3). In the second and final stage, the output of the network is calculated as a perceptron which receives the values from the hidden layer:

$$\mathbf{y} = f_2\left(W^R\mathbf{x}\right), \tag{1.2.5}$$

where $W^R \in \mathbb{R}^{K\times(N+1)}$ is the readout matrix, K is the output dimension (in Fig. 1.2.3 $K = 1$), and f_2 is another transfer function, which is often the identity. For brevity, we will assume $M := M' + 1$ and $N := N' + 1$, thus omitting bias inputs when describing $W^I \in \mathbb{R}^{N\times M}$ and $W^R \in \mathbb{R}^{K\times N}$, respectively.

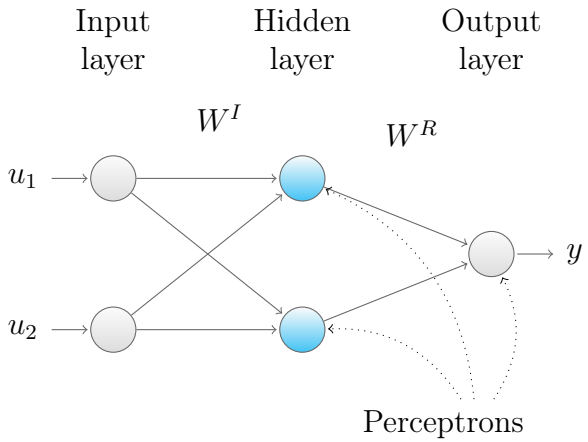

Figure 1.2.3: A feedforward neural network (FNN) solving the XOR problem. In this example, the hidden layer consists of two perceptrons $\mathbf{x} = (x_1, x_2)$, each of which receives two variable inputs u_1 and u_2, plus a constant input (not marked). The output layer y is a perceptron with two inputs x_1 and x_2 coming from the previous (hidden) layer, plus a constant input (not marked). The connection weights are given by the matrices W^I and W^R. Note that the network is not unique, i.e. there exist many possible configurations of W^I and W^R solving the same problem.

The computational process within a NN can be well illustrated by solving the XOR problem. Let the input and the readout matrices be

$$W^I = \begin{pmatrix} -10 & 20 & 20 \\ 30 & -20 & -20 \end{pmatrix}, W^R = \begin{pmatrix} -3 & 2 & 2 \end{pmatrix}, \tag{1.2.6}$$

and let the transfer functions be the following:

$$f_1 = \begin{cases} 1, & if \quad x > 0, \\ 0, & if \quad x \leq 0, \end{cases} \qquad f_2(x) = x. \tag{1.2.7}$$

Here f_2 is the identity, thus, the answer is a linear combination of the hidden layer's values. Consider an input $u_1 = 1$, $u_2 = 0$. Then, the hidden layer is computed as:

$$\begin{aligned} x = & \quad f \begin{pmatrix} -10 + 20 \cdot u_1 + 20 \cdot u_2 \\ 30 - 20 \cdot u_1 - 20 \cdot u_2 \end{pmatrix} \\ = & \quad f \begin{pmatrix} -10 + 20 \cdot 1 + 20 \cdot 0 \\ 30 - 20 \cdot 1 - 20 \cdot 0 \end{pmatrix} = \begin{pmatrix} f(10) \\ f(10) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \end{aligned} \tag{1.2.8}$$

An answer is obtained, using the state of the hidden layer $\mathbf{x}$:

$$y = -3 + 2 \cdot x_1 + 2 \cdot x_2 = -3 + 2 \cdot 1 + 2 \cdot 1 = 1. \tag{1.2.9}$$

The other combinations of the input $\mathbf{u} = (u_1, u_2)$ can be verified in a similar fashion.

Artificial neural networks given by equations (1.2.4) and (1.2.5) are called *feedforward neural networks* (FNNs). FNNs with a single hidden layer of neurons have been proven [8, 50] to be able to approximate an arbitrary continuous function y_{target} on compact subsets of Euclidean space.
Theorem. *If $y_{target}(\mathbf{u}) \in C(I^M)$, where I^M is the M-dimensional hypercube $[0,1]^M$, $C(I^M)$ is the space of continuous functions, and f is a nonconstant, bounded, and monotonically-increasing continuous function, such that*

$$y(\mathbf{u}) = W^R f\left(W^I \mathbf{u}\right).$$

Then for each $\varepsilon > 0$ and $\mathbf{u} \in I^M$, there exist $W^I \in \mathbb{R}^{N \times M}$ and $W^R \in \mathbb{R}^{K \times N}$, such that $\|y(\mathbf{u}) - y_{target}(\mathbf{u})\| < \varepsilon$.

The *universal approximation theorem* [8] stated above not only provides formal requirements for FNNs but also implies that FNNs are a powerful mean of computation. Consisting of multiple perceptrons, on the other hand, FNNs can be regarded as a semblance of collective behavior in biological neuronal systems. The transition from biologically-justified to biologically-inspired architecture was motivated by computational efficiency to fit the existing technology.

1.2.3 Recurrent neural networks

FNN architecture provides a one-directional information processing flow[4]. Recurrent neural networks (RNNs), on the other hand, represent a class of NNs, the connection topology of which allows cycles. Such cycles are not present in FNNs, and RNNs can therefore be regarded as their extension. A schematic illustration is given in Fig. 1.2.4.

In RNNs, the hidden layer in the middle is typically referred to as the *recurrent layer*. As illustrated by the non-exclusively forward-directed connections in Fig. 1.2.4, the recurrent layer has internal connections which include recurrent loops. Because of the

[4]Hence the name, *feedforward* neural networks.

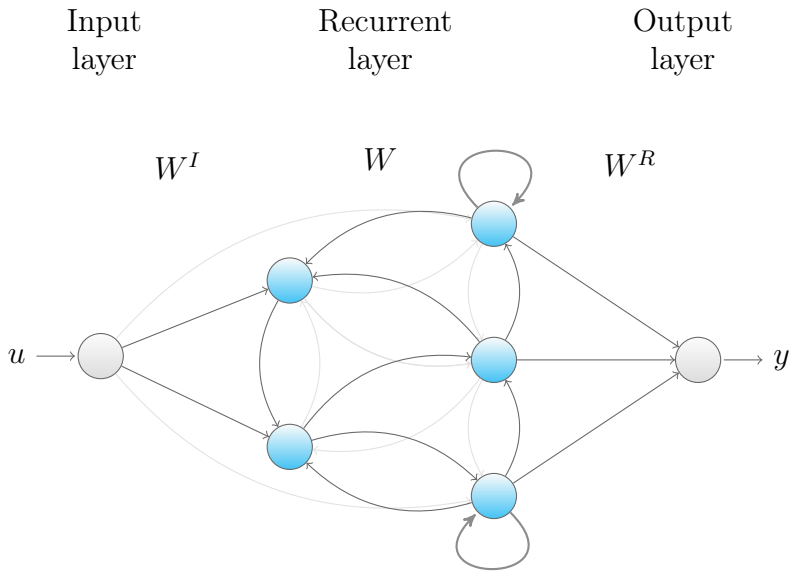

Figure 1.2.4: Schematic of information processing flow in a recurrent neural network (RNN). The recurrent layer, in the middle, retains the information and, therefore, the output of RNN depends on the previous inputs.

presence of recurrent loops, RNNs provide a distant analogy with the biological brain [46].

Mathematically, RNNs can be expressed as:

$$\mathbf{x}(n) = f\left(W^I\mathbf{u}(n) + W\mathbf{x}(n-1)\right), \ n = 1, \ldots, T, \tag{1.2.10}$$

where matrix $W^I \in \mathbb{R}^{N\times M}$ is the input map, $W \in \mathbb{R}^{N\times N}$ is the map of the previous state of the recurrent layer, and $\mathbf{x}(n)$ is the *internal state* of a network at discrete time n. The result of computation $\mathbf{y}(n)$ is obtained as

$$\mathbf{y}(n) = W^R\mathbf{x}(n), \tag{1.2.11}$$

where matrix $W^R \in \mathbb{R}^{K\times N}$ is a readout map. As it can be seen from Eq. (1.2.10), the internal state of the RNN $\mathbf{x}(n)$ depends not only on current inputs, but also on the previous inputs. Such dependence on the previous inputs is the reason why RNNs are more complex objects comparing to FNNs, where information flows unidirectionally from input to output.

While FNNs are universal function approximators, RNNs can be regarded as *algorithms*. The main difference is that functions (or maps) do not have memory (they

are *stateless*) while algorithms do. It has been shown, RNNs are universal approximators of dynamical systems [51]. Moreover, it has been shown that they are Turing equivalent [52]. Thus, RNNs constitute a more powerful class of NNs than FNNs. RNNs can be applied for such problems as system identification, inverse system identification, filtering, prediction, pattern classification, associative memory, data compression [53].

1.2.4 Neural networks training

1.2.4.1 Supervised vs unsupervised learning

The most striking idea behind feedforward NNs is a construction of computing network that can be reused for a wide range of computational problems. Tasks such as spam[5] detection, handwriting recognition, face recognition, time series forecasting and many others, can be regarded as an unknown function y_{target} approximation problems. For instance, in the case of spam detection, the function $y_{\text{target}}(\mathbf{u})$ has to return one of two values, true or false, depending if the argument $\mathbf{u}$ is a spam or not. In this example, $\mathbf{u}$ is a vector of characters representing an email message. In the case of handwriting recognition, the function $y_{\text{target}}(\mathbf{u})$ has to return a character (or a string of characters) depending on its input $\mathbf{u}$, which is a matrix of pixels representing an image. In the case of face recognition, $y_{\text{target}}(\mathbf{u})$ has to identify a face $\mathbf{u}$ in a database of known faces. In the last case, of time series forecasting, $y_{\text{target}}(\mathbf{u})$ has to predict the forthcoming (future) element(s) of time series $\mathbf{u}$.

Behind the training of the listed above examples, (a) there is a set of inputs $\{\mathbf{u}^{(m)}, m = 1, \ldots, N_{\text{total}}\}$, (b) there is a set of corresponding known outputs $\{y_{\text{target}}(\mathbf{u}^{(m)}), m = 1, \ldots, N_{\text{total}}\}$, and (c) the goal is to find a function $y(\mathbf{u})$, e.g. implemented by a NN. The function $y(\mathbf{u})$ is required to minimize a certain error measure $\|y(\mathbf{u}) - y_{\text{target}}(\mathbf{u})\|$. The tasks listed above, therefore, belong to a *supervised learning* class of problems. The goal of supervised training method is known as *generalization* of known outcomes. The present thesis is concentrated around problems solved with the help of supervised learning.

However, there are problems that do not have a predefined solution. Consider, for instance, a dataset of arbitrary data, in which there are hidden relations, e.g. clusters, to be uncovered. If no prior information is given about the number of clusters, their structure or size, then the system is trained with an *unsupervised learning* method. Kohonen networks [9] are an example of such self-organized NNs.

[5] Irrelevant or inappropriate email messages sent to a large number of recipients.

There also exists an intermediate approach known as a *reinforced learning* [54]. During the reinforcement learning process no explicit teacher signal is given. However, the trained system receives a reward upon successful completion of given task or penalty as a consequence of fail. For instance, reinforced learning is often used to teach intelligent agents (robots) solving motion problems, such as walking or object manipulations. There are no explicit instructions, how to balance the robot's body or position the limbs. However, upon completion of each training round, the agent receives a certain reward or "pleasure" function, depending on how close it was to the completion of given goal.

1.2.4.2 FNN training

Backpropagation is one of the most popular supervised learning methods used in FNNs training. The objective of the method is to find network weights W^I and W^R such that the summed square error is minimized

$$E^R = \frac{1}{2N_{\text{total}}} \sum_{m=1}^{N_{\text{total}}} \left\| y(\mathbf{u}^{(m)}) - y_{target}(\mathbf{u}^{(m)}) \right\|^2 .$$

Before the training, weights in W^I and W^R are initialized randomly, typically with small values. The backpropagation is performed iteratively. In each iteration, a *forward pass* given by formulas (1.2.4) and (1.2.5) is computed for each training sample. Then, during the *backward pass* weights are updated from the output to the input layer. This procedure allows the error to be incrementally decreased along the direction of the error gradient ∇E with respect to weights, i.e.

$$\nabla E = \sum_{m=1}^{N_{\text{total}}} \frac{\partial E^l(m)}{\partial w_{ij}^l} \tag{1.2.12}$$

according to the rule

$$\text{new } w_{ij}^l = w_{ij}^l - \alpha \nabla E, \tag{1.2.13}$$

where α is a (small) learning rate, l is the updated layer's index. This way, new readout weights $w_{ij}^R \in W^R,\ i = 1, \ldots, K,\ j = 1, \ldots, N$ and consequently new input weights $w_{ij}^I \in W^I,\ i = 1, \ldots, N,\ j = 1, \ldots, M$ are computed. The forward and backward passes are repeated again until a stop condition, e.g. training convergence. For more practical details about backpropagation algorithm see e.g. this tutorial [55].

Multiple modifications of the backpropagation technique exist, e.g. *incremental learning* where the weights are changed after each training sample individually, modifications with adaptive learning rate α, etc. The algorithm can also be extended to NNs with multiple hidden layers. However, there are several limitations of the backpropagation method family. One of the most important limitations is, like all gradient-descent methods, it is not guaranteed to find a global error minimum. Another major limitation is that training is often slowly converging. Lasting multiple iterations, it may require substantial computation resources to train a neural network.

1.2.4.3 RNN training

While RNNs have an algorithmic advantage over FNNs[6], their training is not as straightforward using error backpropagation. In addition to problems of local optima and slow convergence, the learning process is also driven through bifurcations of the dynamical system [53]. That may slow down or even disrupt the learning process.

Training an RNN is similar to training a FNN with a large number of hidden layers as we will explain in the following paragraphs. Fig. 1.2.5 (a) shows an example of an RNN. The input u_t together with the recurrent feedback x_t (blue arrow), is transformed by a nonlinear map f. Altogether, the input u_t, the feedback x_t, and the nonlinear map f constitute a recurrent layer producing the internal network state x_{t+1}. The resulting state x_{t+1} is the input to the next layer, where the readout map g finally returns the RNN's answer y_{t+1}.

By unfolding the RNN in time, one is able to obtain a FNN representation (Fig. 1.2.5 (b)). Unfolding for n steps is achieved by n replications of the recurrent layer. That results in a FNN with n virtual layers that also has n distinct inputs $u_t, u_{t+1}, \ldots, u_{t+n-1}$. Therefore, the inner state x of the initial RNN is dependent on all the past inputs. That illustrates how RNNs implicitly store the information about the history of all the previous inputs.

To train such unfolded network, a gradient descent method can be used. This technique is called back-propagation through time (BPTT). During backpropagation training, gradients either increase or decrease at each time step. A consequence of the highly increased network's depth, is the creation of so-called *exploding* and, in other cases, so-called *vanishing* gradients [12]. Thus, BPTT becomes a non-trivial task requiring experimentation. Additional tricks (e.g. teacher forcing) are often required to achieve

[6]Here we have to remind that RNNs have the internal state (memory), while FNNs do not. Therefore, RNNs are equivalent to algorithms, while FNNs are memoryless maps. The presence of memory is a tremendous difference that makes RNNs closer to the real brain. Consequently, FNNs are a conceptually less complex class of NNs.

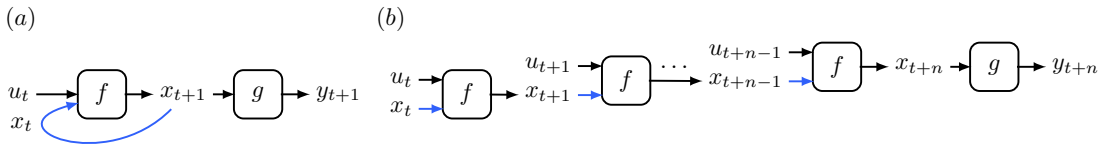

Figure 1.2.5: (a) A simple RNN where f is a nonlinear activation function and g is a readout map. (b) The same network, but unfolded in time for n time steps, which transforms it into an FNN with many layers.

convergence, see [53] for details. Finally, the BPTT procedure drastically increases computational costs for RNNs training, compared to FNNs with the same number of real layers.

Artificial neural networks have demonstrated potential in solving such difficult problems as object recognition [56], playing Go [57] and video games [58], skin cancer diagnosis [59] and others. However, all those algorithms are mostly implemented on general-purpose devices such as CPUs and GPUs[7]. On the other hand, implementation in dedicated hardware promises higher processing speeds and a much better energy efficiency. Because of inherit complex dynamics, physical systems realizing nonlinear delay dynamics are a viable candidate to implement neural networks directly in hardware [60, 61].

1.3 Applications of nonlinear delay dynamics

The diversity of dynamical behavior found in delay systems can be appreciated by the diversity of dynamical states found in their bifurcation diagram (see Fig. 1.3.1). Once translated into a discrete representation of dynamical networks, they share a strong analogy to dynamics widely exploited with artificial neural networks in the field of machine learning. All three regimes (fixed point, limit cycles, and chaotic regime) exhibited by DDEs found an application.

Chaotic regime can be used to hide an information signal in chaotic carrier. This idea is the basis of secure chaos communication where the broadband chaotic waveform replaces a classical sinusoidal carrier, bringing a ciphering (steganography) functionality to the transmitted information. Synchronization between chaos systems is the requirement for encryption/decryption of the signal [62]. Therefore, an identical pair of systems is required. Implementations of a delayed-feedback oscillator provide a major

[7] Central processing units and graphical processing units.