Abstract

Differential programming is the combination of classical neural networks modules with algorithmic ones in an end-to-end differentiable model. These new models, that use automatic differentiation to calculate gradients, have new learning capabilities (reasoning, attention and memory). In this tutorial, aimed at researchers in nonlinear systems with prior knowledge of deep learning, we present this new programming paradigm, describe some of its new features such as attention mechanisms, and highlight the benefits they bring. Then, we analyse the uses and limitations of traditional deep learning models in the modeling and prediction of dynamical systems. Here, a dynamical system is meant to be a set of state variables that evolve in time under general internal and external interactions. Finally, we review the advantages and applications of differential programming to dynamical systems.

Keywords: Deep learning, differential programming, dynamical systems, attention, recurrent neural networks

1. Introduction

The increase in computing capabilities together with new deep learning models has led to great advances in several machine learning tasks \cite{1, 2, 3}.

Deep learning architectures such as Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), as well as the use of distributed representations in natural language processing, have allowed to take into account the symmetries and the structure of the problem to be solved.

However, a major criticism of deep learning remains, namely, that it only performs perception, mapping inputs to outputs \cite{4}.

A new direction to more general and flexible models is differential programming, that is, the combination of geometric modules (traditional neural networks) with more algorithmic modules in an end-to-end differentiable model. As a result, differential programming is a dynamic computational graph composed of differentiable functions that provides not only perception but also reasoning, attention and memory. To efficiently calculate derivatives, this approach uses automatic differentiation, an algorithmic technique similar to backpropagation and implemented in modern software packages such as PyTorch, Julia, etc.

To keep our exposition concise, this tutorial is aimed at researchers in nonlinear systems with prior knowledge of deep learning; see \cite{5} for an excellent introduction to the concepts and methods of deep learning. Therefore, this tutorial focuses right away on the limitations of traditional deep learning and the advantages of differential programming, with special attention to its application to dynamical systems. By a dynamical system we mean here and hereafter a set of state variables that evolve in time under the influence of internal and possibly external inputs.

Examples of differential programming techniques that have been successfully developed in recent years include

(i) attention mechanisms \cite{6}, which allow the model to automatically search and learn which parts of a source sequence are relevant to predict the target element,

(ii) self-attention,

(iii) end-to-end Memory Networks \cite{7}, and

(iv) Differentiable Neural Computers (DNCs) \cite{8}, which are neural networks (controllers) with an external read-write memory.

As expected, in recent years there has been a growing interest in applying deep learning techniques to dynamical systems. In this regard, RNNs and Long Short-Term Memories (LSTMs), specially designed for sequence modelling and temporal dependence, have been
successful in various applications to dynamical systems such as model identification and time series prediction [9, 10, 11].

The performance of these models (e.g., encoder-decoder networks), however, degrades rapidly as the length of the input sequence increases and they are not able to capture the dynamic (i.e., time-changing) interdependence between time steps. The combination of neural networks with new differentiable modules could overcome some of those problems and offer new opportunities and applications.

Among the potential applications of differentiable programming to dynamical systems let us mention

(i) attention mechanisms to select the relevant time steps and inputs,

(ii) memory networks to store historical data from dynamical systems and selectively use it for modelling and prediction, and

(iii) the use of differentiable components in scientific computing.

Despite some achievements, more work is still needed to verify the benefits of these models over traditional networks.

Thanks to software libraries that facilitate automatic differentiation, differentiable programming extends deep learning models with new capabilities (reasoning, memory, attention, etc.) and the models can be efficiently coded and implemented.

In the following sections of this tutorial we introduce differentiable programming and explain in detail why it is an extension of deep learning (Section 2). We describe some models based on this new approach such as attention mechanisms (Section 3.1), memory networks and differentiable neural computers (Section 3.2), and continuous learning (Section 3.3). Then we review the use of deep learning in dynamical systems and their limitations (Section 4.1). And, finally, we present the new opportunities that differentiable programming can bring to the modelling, simulation and prediction of dynamical systems (Section 4.2). The conclusions and outlook are summarized in Section 5.

2. From deep learning to differentiable programming

In recent years, we have seen major advances in the field of machine learning. The combination of deep neural networks with the computational capabilities of Graphics Processing Units (GPUs) [12] has improved the performance of several tasks (image recognition, machine translation, language modelling, time series prediction, game playing and more) [1, 2, 3]. Interestingly, deep learning models and architectures have evolved to take into account the structure of the problem to be resolved.

Deep learning is a part of machine learning that is based on neural networks and uses multiple layers, where each layer extracts higher level features from the input. RNNs are a special class of neural networks where outputs from previous steps are fed as inputs to the current step [13, 14]. This recurrence makes them appropriate for modelling dynamic processes and systems.

CNNs are neural networks that alternate convolutional and pooling layers to implement translational invariance [15]. They learn spatial hierarchies of features through backpropagation by using these building layers. CNNs are being applied successfully to computer vision and image processing [16].

Especially important is the use of distributed representations as inputs to natural language processing pipelines. With this technique, the words of the vocabulary are mapped to an element of a vector space with a much lower dimensionality [17, 18]. This word embedding is able to keep, in the learned vector space, some of the syntactic and semantic relationships presented in the original data.

Let us recall that, in a feedforward neural network (FNN) composed of multiple layers, the output (without the bias term) at layer $l$, see Figure 1, is defined as

$$\hat{x}^{l+1} = \sigma(W^l x^l),$$

(1)

$W^l$ being the weight matrix at layer $l$. $\sigma$ is the activation function and $\hat{x}^{l+1}$, the output vector at layer $l$ and the input vector at layer $l+1$. The weight matrices for the different layers are the parameters of the model.

Learning is the mechanism by which the parameters of a neural network are adapted to the environment in the training process. This is an optimization problem which has been addressed by using gradient-based methods, in which given a cost function $f : \mathbb{R}^n \to \mathbb{R}$, the algorithm finds local minima $w^* = \arg\min_w f(w)$ updating each layer parameter $w_{ij}$ with the rule $w_{ij} := w_{ij} - \eta \nabla_{wij} f(w)$, $\eta > 0$ being the learning rate.

In addition to regarding neural networks as universal approximators, there is no sound theoretical explanation for a good performance of deep learning. Several theoretical frameworks have been proposed:

(i) As pointed out in [19], the class of functions of practical interest can be approximated with exponentially fewer parameters than the generic ones.
Symmetry, locality and compositionality properties make it possible to have simpler neural networks.

(ii) From the point of view of information theory [20], an explanation has been put forward based on how much information each layer of the neural network retains and how this information varies with the training and testing process.

Although deep learning can implicitly implement logical reasoning [21], it has limitations that make it difficult to achieve more general intelligence [4]. Among these limitations, we can highlight the following:

(i) It only performs perception, representing a mapping between inputs and outputs.
(ii) It follows a hybrid model where synaptic weights perform both processing and memory tasks but don’t have an explicit external memory.
(iii) It does not carry out conscious and sequential reasoning, a process that is based on perception and memory through attention.

A path to a more general intelligence, as we will see below, is the combination of geometric modules with more algorithmic modules in an end-to-end differentiable model. This approach, called differentiable programming, adds new parametrizable and differentiable components to traditional neural networks.

Differentiable programming, a broad term, is defined in [22] as a programming model (model of how a computer program is executed), trainable with gradient descent, where neural networks are truly functional blocks with data-dependent branches and recursion.

Here, and for the purposes of this tutorial, we define differentiable programming as a programming model with the following characteristics:

(i) Programs are directed acyclic graphs.
(ii) Graph nodes are mathematical functions or variables and the edges correspond to the flow of intermediate values between the nodes.
(iii) \( n \) is the number of nodes and \( l \) the number of input variables of the graph, with \( 1 \leq l < n \). \( v_i \) for \( i \in [1, ..., n] \) is the variable associated with node \( i \).
(iv) \( E \) is the set of edges in the graph. For each \( (i, j) \in E \) we have \( i < j \), therefore the graph is topologically ordered.
(v) \( f_i \) for \( i \in [(l+1), ..., n] \) is the differentiable function computed by node \( i \) in the graph. \( \alpha_i \) for \( i \in [(l + 1), ..., n] \) contains all input values for node \( i \).
(vi) The forward algorithm or pass, given input variables \( v_1, ..., v_l \) calculates \( v_i = f_i(\alpha_i) \) for \( i = [(l + 1), ..., n] \).
(vii) The graph is dynamically constructed and composed of parametrizable functions that are differentiable and whose parameters are learned from data.

Then, neural networks are just a class of these differentiable programs composed of classical blocks (feed-forward, recurrent neural networks, etc.) and new ones such as differentiable branching, attention, memories, etc.

Differentiable programming can be seen as a continuation of the deep learning end-to-end architectures that have replaced, for example, the traditional linguistic components in natural language processing [23,24]. To efficiently calculate the derivatives in a gradient descent, this approach uses automatic differentiation, an algorithmic technique similar but more general than backpropagation.

Automatic differentiation, in its reverse mode and in contrast to manual, symbolic and numerical differentiation, computes the derivatives in a two-step process [25,26]. As described in [25] and rearranging the indexes of the previous definition, a function \( f : R^l \rightarrow R^m \) is constructed with intermediate variables \( v_i \) such that:

(i) variables \( v_{1:n} = x_i, i = 1, ..., n \) are the inputs variables.
(ii) variables \( v_i, i = 1, ..., l \) are the intermediate variables.
(iii) \( v_{m-I} = v_{l-1}, i = m-1, ..., 0 \) are the output variables.

In a first step, similar to the forward pass described before, the computational graph is built populating intermediate variables \( v_i \) and recording the dependencies. In a second step, called the backward pass, derivatives are calculated by propagating for the output \( y_j \) being
considered, the adjoints $v_{y_{ij}} = \frac{\partial y_{ij}}{\partial v_{ii}}$ from the output to the inputs.

The reverse mode is more efficient to evaluate for functions with a large number of inputs (parameters) and a small number of outputs. When $f : \mathbb{R}^{n} \to \mathbb{R}$, as is the case in machine learning with $n$ very large and $f$ the cost function, only one pass of the reverse mode is necessary to compute the gradient $\nabla f = (\frac{\partial y}{\partial x_1}, ..., \frac{\partial y}{\partial x_n})$.

In the last years, deep learning frameworks such as PyTorch have been developed that provide reverse-mode automatic differentiation [27]. The define-by-run philosophy of PyTorch, whose execution dynamically constructs the computational graph, facilitates the development of general differentiable programs.

Differentiable programming is an evolution of classical (traditional) software programming where, as shown in Table 1:

(i) Instead of specifying explicit instructions to the computer, an objective is set and an optimizable architecture is defined which allows to search in a subset of possible programs.

(ii) The program is defined by the input-output data and not predefined by the user.

(iii) The algorithmic elements of the program have to be differentiable, say, by converting them into differentiable blocks.

| Classical                     | Differentiable                     |
|-------------------------------|-----------------------------------|
| Sequence of instructions      | Sequence of diff. primitives      |
| Fixed architecture            | Optimizable architecture           |
| User defined                  | Data defined                       |
| Imperative programming        | Declarative programming            |
| Intuitive                     | Abstract                           |

Table 1: Differentiable vs classical programming.

RNNs, for example, are an evolution of feedforward networks because they are classical neural networks inside a for-loop (a control flow statement for iteration) which allows the neural network to be executed repeatedly with recurrence. However, this for-loop is a predefined feature of the model. Differentiable programming allows to dynamically constructs the graph and vary the length of the loop. Then, the ideal situation would be to augment the neural network with programming primitives (for-loops, if branches, while statements, external memories, logical modules, etc.) that are not predefined by the user but are parametrizable by the training data.

The trouble is that many of these programming primitives are not differentiable and need to be converted into optimizable modules. For instance, if the condition $a$ of an "if" primitive (e.g., if $a$ is satisfied do $y(x)$, otherwise do $z(x)$) is to be learned, it can be the output of a neural network (linear transformation and a sigmoid function) and the conditional primitive will transform into a weighted combination of both branches $ay(x) + (1 - a)z(x)$. Similarly, in an attention module, different weights that are learned with the model are assigned to give different influence to each part of the input. Figure 2 shows the computational graph of a conditional branching.

The process of extending deep learning with differentiable primitives would consist of the following steps:

(i) Select a new function that improves the classical input-output transformation of deep learning, e.g. attention, continuous learning, memories, etc.

(ii) Convert this function into a directed acyclic graph, a sequence of parametrizable and differentiable functions. For example, Figure 2 shows this sequence of operations used in attention for differentiable branching.

(iii) Integrate this new function into the base model.

In this way, using differentiable programming we can combine traditional perception modules (CNN, RNN, FNN) with additional algorithmic modules that provide reasoning, abstraction and memory [28]. In the following section we describe, by following this process, some examples of this approach that have been developed in recent years.
3. Differentiable learning and reasoning

3.1. Differentiable attention

One of the aforementioned limitations of deep learning models is that they do not perform conscious and sequential reasoning, a process that is based on perception and memory through attention.

Reasoning is the process of consciously establishing and verifying facts combining attention with new or existing information. An attention mechanism allows the brain to focus on one part of the input or memory (image, text, etc), giving less attention to others.

Attention mechanisms have provided and will provide a paradigm shift in machine learning. From traditional large-scale vector transformations to a more conscious process that focuses only on a set of elements, e.g. decomposing a problem into a sequence of attention based reasoning operations [29].

![Figure 3: Attention diagram.](image)

One way to make this attention process differentiable is to make it a convex combination of the input or memory, where all the steps are differentiable and the combination weights are parametrizable.

As in [30], this differentiable attention process is described as mapping a query and a set of key-value pairs to an output:

$$\text{att}(q, s) = \sum_{i=1}^{T} \alpha_i(q, k_i)V_i,$$

where, as seen in figure [3] k, and V, are the key and the value vectors from the source/memory s, and q is the query vector. \(\alpha_i(q, k_i)\) is the similarity function between the query and the corresponding key and is calculated by applying the softmax function:

$$\text{Softmax}(z_i) = \frac{\exp(z_i)}{\sum_i \exp(z_i)}$$

to the score function \(score(q, k_i)\):

$$\alpha_i = \frac{\exp(score(q, k_i))}{\sum_{i=1}^{T} \exp(score(q, k_i))}$$

(4)

The score function can be computed using a feedforward neural network:

$$score(q, k_i) = Z_a \tanh(W_a[q, k_i])$$

(5)

as proposed in [6], where \(Z_a\) and \(W_a\) are matrices to be jointly learned with the rest of the model and \([q, k_i]\) is a linear function or concatenation of \(q\) and \(k_i\). Also, in [31] the authors use a cosine similarity measure for content-based attention, namely,

$$score(q, k_i) = \cos((q, k_i))$$

(6)

where \((q, k_i)\) denotes the angle between \(q\) and \(k_i\).

Then, differentiable attention can be seen as a sequential process of reasoning in which the task (query) is guided by a set of elements of the input source (or memory) using attention.

The attention process can focus on:

(i) Temporal dimensions, e.g. different time steps of a sequence.
(ii) Spatial dimensions, e.g. different regions of an image.
(iii) Different elements of a memory.
(iv) Different features or dimensions of an input vector, etc.

Depending on where the process is initiated, we have:

(i) Top-down attention, initiated by the current task.
(ii) Bottom-up, initiated spontaneously by the source or memory.

3.1.1. Attention mechanisms in seq2seq models

RNNs (see Figure [4]) are a basic component of modern deep learning architectures, especially of encoder-decoder networks. The following equations define the time evolution of an RNN:

$$h_t = f^h(W^h x_t + W^{hh} h_{t-1}),$$

(7)

$$y_t = f^o(W^{ho} h_t),$$

(8)

\(W^h, W^{hh}\) and \(W^{ho}\) being weight matrices. \(f^h\) and \(f^o\) are the hidden and output activation functions while \(x_t\), \(h_t\) and \(y_t\) are the network input, hidden state and output.

An evolution of RNNs are LSTMs [32], an RNN structure with gated units, i.e. regulators. LSTM are composed of a cell, an input gate, an output gate and a forget gate, and allow gradients to flow unchanged. The
memory cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.

An encoder-decoder network maps an input sequence to a target one with both sequences of arbitrary length \[2\]. They have applications ranging from machine translation to time series prediction.

More specifically, this mechanism uses an RNN (or any of its variants, an LSTM or a GRU, Gated Recurrent Unit) to map the input sequence to a fixed-length vector, and another RNN (or any of its variants) to decode the target sequence from that vector (see Figure 5). Such a seq2seq model features normally an architecture composed of:

(i) An encoder which, given an input sequence \(X = (x_1, x_2, ..., x_T)\) with \(x_t \in \mathbb{R}^n\), maps \(x_t\) to

\[
    h_t = f_1(h_{t-1}, x_t),
\]

where \(h_t \in \mathbb{R}^m\) is the hidden state of the encoder at time \(t\), \(m\) is the size of the hidden state and \(f_1\) is an RNN (or any of its variants).

(ii) A decoder, where \(s_t\) is the hidden state and whose initial state \(s_0\) is initialized with the last hidden state of the encoder \(h_T\). It generates the output sequence \(Y = (y_1, y_2, ..., y_T)\), \(y_t \in \mathbb{R}^o\) (the dimension \(o\) depending on the task), with

\[
    y_t = f_2(s_{t-1}, y_{t-1}),
\]

where \(f_2\) is an RNN (or any of its variants) with an additional softmax layer.

Because the encoder compresses all the information of the input sequence in a fixed-length vector (the final hidden state \(h_T\)), the decoder possibly does not take into account the first elements of the input sequence. The use of this fixed-length vector is a limitation to improve the performance of the encoder-decoder networks. Moreover, the performance of encoder-decoder networks degrades rapidly as the length of the input sequence increases \[33\]. This occurs in applications such as machine translation and time series prediction, where it is necessary to model long time dependencies.

The key to solve this problem is to use an attention mechanism. In \[6\] an extension of the basic encoder-decoder architecture was proposed by allowing the model to automatically search and learn which parts of a source sequence are relevant to predict the target element. Instead of encoding the input sequence in a fixed-length vector, it generates a sequence of vectors, choosing the most appropriate subset of these vectors during the decoding process.

With the attention mechanism, the encoder is a bidirectional RNN \[34\] with a forward hidden state \(\overrightarrow{h}_i = f_1(\overrightarrow{h}_{i-1}, x_i)\) and a backward one \(\overleftarrow{h}_i = f_1(\overleftarrow{h}_{i+1}, x_i)\). The encoder state is represented as a simple concatenation of the two states,

\[
    h_i = [\overrightarrow{h}_i; \overleftarrow{h}_i],
\]

with \(i = 1, ..., T\). The encoder state includes both the preceding and following elements of the sequence, thus capturing information from neighbouring inputs.

The decoder has an output

\[
    y_t = f_2(s_{t-1}, y_{t-1}, c_t)
\]

for \(t = 1, ..., T'\). \(f_2\) is an RNN with an additional softmax layer, and the input is a concatenation of \(y_{t-1}\) with the context vector \(c_t\), which is a sum of hidden states of the input sequence weighted by alignment scores:
\[ c_t = \sum_{i=1}^{T} \alpha_t h_i. \] (13)

Similar to equation (4), the weight \( \alpha_t \) of each state \( h_i \) is calculated by
\[
\alpha_t = \frac{\exp(\text{score}(s_{t-1}, h_i))}{\sum_{i'=1}^{T} \exp(\text{score}(s_{t-1}, h_{i'}))}.
\] (14)

In this attention mechanism, the query is the state \( s_{t-1} \) and the key and the value are the hidden states \( h_i \). The score measures how well the input at position \( i \) and the output at position \( t \) match. \( \alpha_t \) are the weights that implement the attention mechanism, defining how much of each input hidden state should be considered when deciding the next state \( s_t \) and generating the output \( y_t \) (see Figure 6).

As we have described previously, the score function can be parametrized using different alignment models such as feedforward networks and the cosine similarity. An example of a matrix of alignment scores can be seen in Figure 7. This matrix provides interpretability to the model since it allows to know which part (time-step) of the input is more important to the output.

3.2. Other attention mechanisms and differentiable neural computers

A variant of the attention mechanism is self-attention, in which the attention component relates different positions of a single sequence in order to compute a representation of the sequence. In this way, the keys, values and queries come from the same source. The mechanism can connect distant elements of the sequence more directly than using RNNs [35].

Another variant of attention are end-to-end memory networks [7], which we describe in Section 4.2.2 and are neural networks with a recurrent attention model over an external memory. The model, trained end-to-end, outputs an answer based on a set of inputs \( x_1, x_2, \ldots, x_n \) stored in a memory and a query.

Traditional computers are based on the von Neumann architecture which has two basic components: the CPU (Central Processing Unit), which carries out the program instructions, and the memory, which is accessed by the CPU to perform write/read operations. In contrast, neural networks follow a hybrid model where synaptic weights perform both processing and memory tasks.

Neural networks and deep learning models are good at mapping inputs to outputs but are limited in their ability to use facts from previous events and store useful information. Differentiable Neural Computers (DNCs) [8] try to overcome these shortcomings by combining neural networks with an external read-write memory.

As described in [8], a DNC is a neural network, called the controller (playing the role of a differentiable CPU), with an external memory, an \( N \times W \) matrix. The DNC uses differentiable attention mechanisms to define distributions (weightings) over the N rows and learn the importance each row has in a read or write operation.

To select the most appropriate memory components during read/write operations, a weighted sum \( w(i) \) is used over the memory locations \( i = 1, \ldots, N \). The attention mechanism is used in three different ways:

(i) Access content (read or write) based on similarity.
(ii) Time ordered access (temporal links) to recover the sequences in the order in which they were written.
(iii) Dynamic memory allocation, where the DNC assigns and releases memory based on usage percentage.

At each time step, the DNC gets an input vector and emits an output vector that is a function of the combination of the input vector and the memories selected.

DNCs, by combining the following characteristics, have very promising applications in complex tasks that require both perception and reasoning:

(i) The classical perception capability of neural networks.
(ii) Read and write capabilities based on content similarity and learned by the model.
(iii) The use of previous knowledge to plan and reason.
(iv) End-to-end differentiability of the model.
(v) Implementation using software packages with automatic differentiation libraries such as PyTorch, Tensorflow or similar.

3.3. Meta-plasticity and continuous learning

The combination of geometric modules (classical neural networks) with algorithmic ones adds new learning capabilities to deep learning models. In the previous sections we have seen that one way to improve the learning process is by focusing on certain elements of the input or a memory and making this attention differentiable.

Another natural way to improve the process of learning is to incorporate differentiable primitives that add flexibility and adaptability. A source of inspiration is neuromodulators, which furnish the traditional synaptic transmission with new computational and processing capabilities [36].

Unlike the continuous learning capabilities of animal brains, which allow animals to adapt quickly to the experience, in neural networks, once the training is completed, the parameters are fixed and the network stops learning. To solve this issue, in [37] a differentiable plasticity component is attached to the network that helps previously-trained networks adapt to ongoing experience.

The process to introduce the differentiable plastic component in the network is as follows. The activation $y_j$ of neuron $j$ has a conventional fixed weight $w_{ij}$ and a plastic component $\alpha_{ij}H_i(t)$, where $\alpha_{ij}$ is a structural parameter tuned during the training period and $H_i(t)$ a plastic component automatically updated as a function of ongoing inputs and outputs. The equations for the activation of $y_j$ with learning rate $\eta$, as described in [37], are:

$$y_j = \tanh\left(\sum_{\text{inputs}} (w_{ij} + \alpha_{ij}H_i(t))y_j\right), \quad (15)$$

$$H_i(t + 1) = \eta y_j + (1 - \eta)H_i(t). \quad (16)$$

Then, during the initial training period, $w_{ij}$ and $\alpha_{ij}$ are trained using gradient descent and after this period, the model keeps learning from ongoing experience.

4. Dynamical systems and differentiable programming

4.1. Modeling dynamical systems with neural networks

Dynamical systems deal with time-evolutionary processes and their corresponding systems of equations. At any given time, a dynamical system has a state that can be represented by a point in a state space (manifold). The evolutionary process of the dynamical system describes what future states follow from the current state. This process can be deterministic, if its entire future is uniquely determined by its current state, or non-deterministic otherwise [38] (e.g., a random dynamical system [39]). Furthermore, it can be a continuous-time process, represented by differential equations or, as in this paper, a discrete-time process, represented by difference equations or maps. Thus,

$$h_t = f(h_{t-1}; \theta) \quad (17)$$

for autonomous discrete-time deterministic dynamical systems with parameters $\theta$, and

$$h_t = f(h_{t-1}, x_t; \theta) \quad (18)$$

for non-autonomous discrete-time deterministic dynamical systems driven by an external input $x_t$.

Dynamical systems have important applications in physics, chemistry, economics, engineering, biology and medicine [40]. They are relevant even in day-to-day phenomena with great social impact such as tsunami warning, earth temperature analysis and financial markets prediction.

Dynamical systems that contain a very large number of variables interacting with each other in non-trivial ways are sometimes called complex (dynamical) systems [41]. Their behaviour is intrinsically difficult to model due to the dependencies and interactions between their parts and they have emergence properties arising from these interactions such as adaptation, evolution, learning, etc.

Here we consider discrete-time, deterministic and non-autonomous (i.e., the time evolution depending also
on exogenous variables) dynamical systems as well as the more general complex systems. Specifically, the dynamical systems of interest range from systems of difference equations with multiple time delays to systems with a dynamic (i.e., time-changing) interdependence between time steps. Notice that the former ones may be rewritten as higher dimensional systems with time delay 1.

On the other hand, in recent years deep learning models have been very successful in performing various tasks such as image recognition, machine translation, game playing, etc. When the amount of training data is sufficient and the distribution that generates the real data is the same as the distribution of the training data, these models perform extremely well and approximate the input-output relation.

In view of the importance of dynamical systems for modeling physical, biological and social phenomena, there is a growing interest in applying deep learning techniques to dynamical systems. This can be done in different contexts, such as:

(i) Modeling dynamical systems with known structure and equations but non-analytical or complex solutions [42].

(ii) Modeling dynamical systems without knowledge of the underlying governing equations [43, 44]. In this regard, let us mention that commercial initiatives are emerging that combine large amounts of meteorological data with deep learning models to improve weather predictions.

(iii) Modeling dynamical systems with partial or noisy data [45].

A key aspect in modelling dynamical systems is temporal dependence. There are two ways to introduce it into a neural network [46]:

(i) A classical feedforward neural network with time delayed states in the inputs but perhaps with an unnecessary increase in the number of parameters.

(ii) A recurrent neural network (RNN) which, as shown in Equations (7) and (8), has a temporal recurrence that makes it appropriate for modelling discrete dynamical systems of the form given in Equations (17) and (18).

Thus, RNNs, specially designed for sequence modelling [47], seem the ideal candidates to model, analyze and predict dynamical systems in the broad sense used in this tutorial. The temporal recurrence of RNNs, theoretically, allows to model and identify dynamical systems described with equations with any temporal dependence.

To learn chaotic dynamics, recurrent radial basis function (RBF) networks [48] and evolutionary algorithms that generate RNNs have been proposed [49]. "Nonlinear Autoregressive model with exogenous input" (NARX) [50] and boosted RNNs [51] have been applied to predict chaotic time series.

However, a difficulty with RNNs is the vanishing gradient problem [52]. RNNs are trained by unfolding them into deep feedforward networks, creating a new layer for each time step of the input sequence. When backpropagation computes the gradient by the chain rule, this gradient vanishes as the number of time-steps increases. As a result, for long input-output sequences, as depicted in Figure 8, RNNs have trouble modelling long-term dependencies, that is, relationships between elements that are separated by large periods of time.

To overcome this problem, LSTMs were proposed. LSTMs have an advantage over basic RNNs due to their relative insensitivity to temporal delays and, therefore, are appropriate for modeling and making predictions based on time series whenever there exist temporary dependencies of unknown duration. With the appropriate number of hidden units and activation functions [10], LSTMs can model and identify any non-linear dynamical system of the form:

\[ h_t = f(x_t, ..., x_{t-T}, h_{t-1}, ..., h_{t-T}) \]  
\[ y_t = g(h_t) \]

where \( f \) and \( g \) are the state and output functions while \( x_t, h_t \) and \( y_t \) are the system input, state and output.

LSTMs have succeeded in various applications to dynamical systems such as model identification and time series prediction [9, 10, 11].
An also remarkable application of the LSTM has been machine translation \cite{2,53}, using the encoder-decoder architecture described in Section 3.1.1.

However, as we have seen, the decoder possibly does not take into account the first elements of the input sequence because the encoder compresses all the information of the input sequence in a fixed-length vector. Then, the performance of encoder-decoder networks degrades rapidly as the length of input sequence increases and this can be a problem in time series analysis, where predictions are based upon a long segment of the series.

Furthermore, as depicted in Figure 9, a complex dynamic may feature interdependencies between time steps that vary with time. In this situation, the equation that defines the temporal evolution may change at each \( t \in 1, ..., T \). For these dynamical systems, adding an attention module like the one described in Equation \ref{eq:13} can help model such time-changing interdependencies.

![Figure 9: Temporal interdependencies in a dynamical system.](image)

### 4.2. Improving dynamical systems with differentiable programming

Deep learning models together with graphic processors and large amounts of data have improved the modeling of dynamical systems but this has some limitations such as those mentioned in the previous section. The combination of neural networks with new differentiable algorithmic modules is expected to overcome some of those shortcomings and offer new opportunities and applications.

In the next three subsections we illustrate with examples the kind of applications of differentiable programming to dynamical systems we have in mind, namely: implementations of attention mechanisms, memory networks, scientific simulations and modeling in physics.

#### 4.2.1. Attention mechanisms in dynamical systems

In the previous sections we have described the attention mechanism, which allows a task to be guided by a set of elements of the input or memory source. When applying this mechanism to dynamical systems modeling or prediction, it is necessary to decide the following aspects:

(i) In which phase or phases of the model should the attention mechanism be introduced?
(ii) What dimension is the mechanism going to focus on? Temporal, spatial, etc.
(iii) What parts of the system will correspond to the query, the key and the value?

One option, which is also quite illustrative, is to use a dual-stage attention, an encoder with input attention and a decoder with temporal attention, as pointed out in \cite{54}.

Here we describe this option, in which the first stage extracts the relevant input features and the second selects the relevant time steps of the model. In many dynamical systems there are long term dependencies between time steps and these dependencies can be dynamic, i.e., time-changing. In these cases, attention mechanisms learn to focus on the most relevant parts of the system input or state.

\( X = (x_1, x_2, ..., x_T) \) with \( x_t \in \mathbb{R}^n \) represents the input sequence. \( T \) is the length of the time interval and \( n \) the number of input features or dimensions. At each time step \( t \), \( x_t = (x_t^1, x_t^2, ..., x_t^n) \).

**Encoder with input attention**

The encoder, given an input sequence \( X \), maps \( u_t \) to

\[
\hat{h}_i = f_i(h_{i-1}, u_t),
\]

where \( h_i \in \mathbb{R}^m \) is the hidden state of the encoder at time \( t \), \( m \) is the size of the hidden state and \( f_i \) is an RNN (or any of its variants). \( x_t \) is replaced by \( u_t \), which adaptively selects the relevant input features with

\[
u_t = (\alpha_t^1 x_t^1, \alpha_t^2 x_t^2, ..., \alpha_t^n x_t^n).
\]

\( \alpha_t^k \) is the attention weight measuring the importance of the \( k \) input feature at time \( t \) and is computed by

\[
\alpha_t^k = \frac{\exp(score(h_{i-1}, x_t^k))}{\sum_{l=1}^n \exp(score(h_{i-1}, x_t^l))},
\]

where \( x_t^k = (x_t^1, x_t^2, ..., x_t^n) \) is the \( k \) input feature series and the score function can be computed using a feed-forward neural network, a cosine similarity measure or other similarity functions.

Then, this first attention stage extracts the relevant input features, as seen in Figure 10, with the corresponding query, keys and values.
Decoder with temporal attention

Similar to the attention decoder described in Section 3.1.1, the decoder has an output

$$y_t = f_2(s_{t-1}, y_{t-1}, e_t)$$

for $$t = 1, \ldots, T'$$. $f_2$ is an RNN (or any of its variants) with an additional linear or softmax layer, and the input is a concatenation of $y_{t-1}$ with the context vector $c_t$, which is a sum of hidden states of the input sequence weighted by alignment scores:

$$c_t = \sum_{i=1}^{T} \beta_{it} h_i.$$  \hspace{1cm} (25)

The weight $\beta_{it}$ of each state $h_i$ is computed using the similarity function, $\text{score}(s_{t-1}, h_i)$, and applying a softmax function, as described in Section 3.1.1.

This second attention stage selects the relevant time steps, as shown in Figure 11 with the corresponding query, keys and values.

Further remarks

In [54], the authors define this dual-stage attention RNN and show that the model outperforms a classical model in time series prediction.

In [55], a comparison is made between LSTMs and attention mechanisms for financial time series forecasting. It is shown there that an LSTM with attention perform better than stand-alone LSTMs.

A temporal attention layer is used in [56] to select relevant information and to provide model interpretability, an essential feature to understand deep learning models. Interpretability is further studied in detail in [57], concluding that attention weights partially reflect the impact of the input elements on model prediction.

Despite the theoretical advantages and some achievements, further studies are needed to verify the benefits of the attention mechanism over traditional networks.

4.2.2. Memory networks

Memory networks allow long-term dependencies in sequential data to be learned thanks to an external memory component. Instead of taking into account only the most recent states, memory networks consider the entire list of entries or states.

Here we define one possible application of memory networks to dynamical systems, following an approach based on [71]. We are given a time series of historical data $n_1, \ldots, n_T$ with $n_i \in \mathbb{R}^n$ and the input series $x_1, \ldots, x_T$ with $x_t \in \mathbb{R}^n$ the current input, which is the query.

The set $\{n_i\}$ are converted into memory vectors $\{m_i\}$ and output vectors $\{c_i\}$ of dimension $d$. The query $x_t$ is also transformed to obtain an internal state $u_t$, of dimension $d$. These transformations correspond to a linear transformation: $A n_i = m_i, B n_i = c_i, C x_t = u_t$, being $A, B, C$ parameterizable matrices.

A match between $u_t$ and each memory vector $m_i$ is computed by taking the inner product followed by a softmax function:

$$p_i = \text{softmax}(u_t^T m_i).$$

The final vector from the memory, $o_t$, is a weighted sum over the transformed inputs $\{c_i\}$:

$$o_t = \sum_j p_i c_i.$$  \hspace{1cm} (27)

To generate the final prediction $y_t$, a linear layer is applied to the sum of the output vector $o_t$ and the transformed input $u_t$ and to the previous output $y_{t-1}$:

\[ \text{Figure 10: Diagram of the input attention mechanism.} \]

\[ \text{Figure 11: Diagram of the input attention mechanism.} \]
This model is differentiable end-to-end by learning the matrices (the final matrices $W^3$ and the three transformation matrices $A$, $B$ and $C$) to minimize the prediction error.

In [58] the authors propose a similar model based on memory networks with a memory component, three encoders and an autoregressive component for multivariate time-series forecasting. Compared to non-memory RNN models, their model is better at modeling and capturing long-term dependencies and, moreover, it is interpretable.

Taking advantage of the highlighted capabilities of Differentiable Neural Computers (DNCs), an enhanced DNC for electroencephalogram (EEG) data analysis is proposed in [59]. By replacing the LSTM network controller with a recurrent convolutional network, the potential of DNCs in EEG signal processing is convincingly demonstrated.

4.2.3. Scientific simulation and physical modeling

Scientific modeling, as pointed out in [60], has traditionally employed three approaches:

(i) Direct modeling, if the exact function that relates input and output is known.

(ii) Using a machine learning model. As we have mentioned, neural networks are universal approximators.

(iii) Using a differential equation if some structure of the problem is known. For example, if the rate of change of the unknown function is a function of the physical variables.

Machine learning models have to learn the input-output transformation from scratch and need a lot of data. One way to make them more efficient is to combine them with a differentiable component suited to a specific problem. This component allows specific prior knowledge to be incorporated into deep learning models and can be a differentiable physical model or a differentiable ODE (ordinary differential equation) solver.

(i) Differentiable physical models.

Differentiable plasticity, as described in Section 3.3, can be applied to deep learning models of dynamical systems in order to help them adapt to ongoing data and experience. As done in [37], the plasticity component described in Equations 15 and 15 can be introduced in some layers of the deep learning architecture.

In this way, the model can continuously learn because the plastic component is updated by neural activity.

DiffTaichi, a differentiable programming language for building differentiable physical simulations, is proposed in [62], integrating a neural network controller with a physical simulation module. A differentiable physics engine is presented in [63]. The system simulates rigid body dynamics and can be integrated in an end-to-end differentiable deep learning model for learning the physical parameters.

(ii) Differentiable ODE solvers.

As described in [60], an ODE can be embedded into a deep learning model. For example, the Euler method takes in the derivative function and the initial values and outputs the approximated solution. The derivative function could be a neural network. This solver is differentiable and can be integrated into a larger model that can be optimized using gradient descent.

In [61] a differentiable model of a trebuchet is described. In a classical trebuchet model, the parameters (the mass of the counterweight and the angle of release) are fed into an ODE solver that calculates the distance, which is compared with the target distance. In the extended model, a neural network is introduced. The network takes two inputs, the target distance and the current wind speed, and outputs the trebuchet parameters, which are fed into the simulator to calculate the distance. This distance is compared with the target distance and the error is back-propagated through the entire model to optimize the parameters of the network. Then, the neural network is optimized so that the model can achieve any target distance. Using this extended model is faster than optimizing only the trebuchet. This type of applications shows how combining differentiable ODE solvers and deep learning models allows to incorporate previous structure to the problem and makes the learning process more efficient.

We may conclude that combining scientific computing and differentiable components will open new avenues in the coming years.

5. Conclusions and future directions

Differentiable programming is the use of new differentiable components beyond classical neural networks. This generalization of deep learning allows to have data
parametrizable architectures instead of pre-fixed ones and new learning capabilities such as reasoning, attention and memory.

The first models created under this new paradigm, such as attention mechanisms, differentiable neural computers and memory networks, are already having a great impact on natural language processing.

These new models and differentiable programming are also beginning to improve machine learning applications to dynamical systems. As we have seen, these models improve the capabilities of RNNs and LSTMs in identification, modeling and prediction of dynamical systems. They even add a necessary feature in machine learning such as interpretability.

However, this is an emerging field and further research is needed in several directions. To mention a few:

(i) More comparative studies between attention mechanisms and LSTMs in predicting dynamical systems.

(ii) Use of self-attention and its possible applications to dynamical systems.

(iii) As with RNNs, a theoretical analysis (e.g., in the framework of dynamical systems) of attention and memory networks.

(iv) Clear guidelines so that scientists without advanced knowledge of machine learning can use new differentiable models in computational simulations.

Acknowledgments. This work was financially supported by the Spanish Ministry of Science, Innovation and Universities, grant MTM2016-74921-P (AEI/FEDER, EU).

References

[1] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (2015) 436–44. doi:10.1038/nature14539

[2] I. Sutskever, O. Vinyals, Q. V. Le, Sequence to sequence learning with neural networks, in: NIPS, 2014.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. R. Baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. F. C. Hui, L. Sifre, G. van den Driessche, T. Graepel, D. Hassabis, Mastering the game of go without human knowledge, Nature 550 (2017) 354–359.

[4] G. Marcus, Deep learning: A critical appraisal, ArXiv 1801.00631.

[5] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016. http://www.deeplearningbook.org

[6] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, ArXiv 1409.

[7] S. Sukhbaatar, A. Szlam, J. Weston, R. Fergus, End-to-end memory networks, in: NIPS, 2015.

[8] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwinska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, D. Hassabis, Hybrid computing using a neural network with dynamic external memory, Nature 538 (2016) 471–476.

[9] Z. Wang, D. Xiao, F. Fang, R. Govindan, C. Pain, Y. Guo, Model identification of reduced order fluid dynamics systems using deep learning, International Journal for Numerical Methods in Fluids 86. doi:10.1002/fld.4416

[10] Y. Wang, A new concept using lstm neural networks for dynamic system identification, 2017, pp. 5324–5329. doi:10.23919/ACC.2017.7963782

[11] Y. Li, H. Cao, Prediction for tourism flow based on lstm neural network, Procedia Computer Science 129 (2018) 277–283. doi:10.1016/j.procs.2018.03.076

[12] O. Yadan, K. Adams, Y. Taigman, M. Ranzato, Multi-gpu training of convnets, CoRR abs/1312.5853.

[13] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, J. Schmidhuber, A novel connectionist system for unconstrained handwriting recognition, IEEE Transactions on Pattern Analysis and Machine Intelligence 31 (2009) 855–868.

[14] A. Sherstinsky, Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network, ArXiv abs/1808.03314.

[15] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (1998) 2278 – 2324. doi:10.1109/5.726791

[16] R. Yamashita, M. Nishio, R. K. G. Do, K. Togashi, Convolutional neural networks: an overview and application in radiology, in: Insights into imaging, 2018.

[17] Y. Bengio, R. Ducharme, P. Vincent, C. Janvin, A neural probabilistic language model J. Mach. Learn. Res. 3 (2003) 1137–1155. URL http://dl.acm.org/citation.cfm?id=944919.944966

[18] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, NIPS’13, Curran Associates Inc., USA, 2013. pp. 3111–3119. URL http://dl.acm.org/citation.cfm?id=2999792.2999995

[19] H. W. Lin, M. Tegmark, Why does deep and cheap learning work so well?, Journal of Statistical Physics doi:10.1007/s10955-017-1836-5

[20] R. Shwartz-Ziv, N. Tishby, Opening the black box of deep neural networks, ArXiv abs/1808.07980.

[21] F. Wang, Backpropagation with continuation callbacks : Differentiable programming, CoRR abs/1808.03314.

[22] Y. Goldberg, Neural network methods for natural language processing, Synthesis Lectures on Human Language Technologies 10 (2017) 1–309. doi:10.23919/S00782EZED1901Y201703HLT037

[23] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, J. M. Siskind, Automatic differentiation in machine learning: a survey, Journal of Machine Learning Research 18 (153) (2018) 1–43.

[24] Y. Bengio, R. Ducharme, P. Vincent, C. Janvin, A neural probabilistic language model J. Mach. Learn. Res. 3 (2003) 1137–1155. URL http://dl.acm.org/citation.cfm?id=944919.944966

[25] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, J. M. Siskind, Automatic differentiation in machine learning: a survey, Journal of Machine Learning Research 18 (153) (2018) 1–43.
