Introduction to neural ODE

Della Bona Sarah, Dumez Erika

18 février 2021

1 Ordinary Differential Equations

1.1 A reminder on ODE

An ODE is a function that describes the changes of a function, u, through time. In this setting, time is a continuous variable and we don't know the real function, only its derivative.

Définition 1. Let $f : \Omega \subseteq \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}^N$.
A *first order ODE* takes of the form :

$$\partial_t u(t) = f(t, u(t)) \quad \textbf{(1)}$$

— A *solution* for (1) is a function $u : I \to \mathbb{R}^N$ where I is an interval of $\mathbb{R}$ such that :
  — u is derivable on I,
  — $\forall t \in I, f(t, u(t)) \in \Omega$,
  — $\forall t \in I, \partial_t u(t) = f(t, u(t))$
— An *initial condition* (IC) is a condition of the type :

$$u(t_0) = u_0$$

where $(t_0, u_0) \in \Omega$ is fixed.

A *Cauchy problem* is an ODE with IC

$$\begin{cases} \partial_t u(t) &= f(t, u(t)) \\ u(t_0) &= u_0 \end{cases}$$

Définition 2. A *k-order ODE* is of the form :

$$\partial_t^k v(t) = g(t, v(t), ..., \partial^{k-1} v(t))$$

where

$$\begin{aligned} v &: I \to \mathbb{R}^N \\ g &: \Theta \subseteq \mathbb{R} \times \mathbb{R}^N \times ... \times \mathbb{R}^N \to \mathbb{R}^N \end{aligned}$$

It is not always possible to explicitly find a solution to a Cauchy problem, but we can compute a finite number of points $u_i \in \mathbb{R}^N$ which are close to the real solution.

More precisely, let $T \in \mathbb{R}$
$\{0\}$ such that the solution u exists on $[t_0, t_0 + T]$ and let $n \in \mathbb{N}^{\geqslant 2}$. We are then looking for $(u_i)_{i=0}^n$ s.t.

$$u_i \approx u(t_i) \quad \text{where } t_0 < ... < t_n \in [t_0, t_0 + T]$$

Let $h_i := t_{i+1} - t_i$, it is called the *step*. To compute those u_i, we use *1-step methods* such as Euler's method.

1.2 Euler's method

Euler's method is similar to a Taylor development, the idea is to compute $u(t_{i+1})$ using the following formula : [1]

$$u(t_{i+1}) \approx u(t_i) + h.\partial u(t_i)$$

where

$$\partial u(t_i) = f(t_i, u(t_i)).$$

2 Neural networks

In a typical machine learning problem, you are given some input x and you want to predict an output y. A *neural network* can be used to solve such a problem. It consists of a series of layers. There are three types of layers :

— The *input* layer
— The *output* layer
— The *hidden* layers

Each layer consist of a certain number of neurons. We give an input to the neurons of a layer, they do some calculus (non-linear activation function for example) and they give an output. Therefore, layers can be seen has matrix operations.

The neurons of a layer are connected to the neurons of the next layer. Thanks to these connections, the output given by a neuron can be transmitted over the neural network. We begin by giving an input to the input layer, which transmits information to the first hidden layer (There isn't always an hidden layer in a neural network. In this case, the input layer is directly connected to the output layer). In turn, it transmit information to the next hidden layer and so on, until the output layer which give us the final output, the prediction.

With a loss function, we can then determined the accuracy of the neural network and make it more accurate by changing the parameters. We are trying to find the optimal parameters which minimize this loss function. In order to do that, we use gradient descent.

We can see a neural network as a composite of functions, a function for each layer. That is because each layer uses an activation function on its input. Each layer introduces a little bit of error that propagate through the network.

1. We consider that $\forall i \ \{0, ..., n\}, h_i = h$.

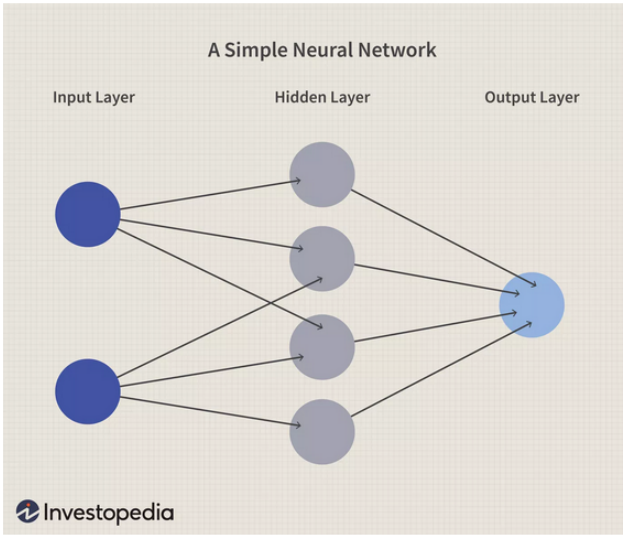


Image by Sabrina Jiang © Investopedia 2020

2.1 Example

Let's consider a neural network, with one hidden layer, that takes a 2-dimensional input $x = (x_1, x_2)$, and gives a 2-dimensional output $y = (y_1, y_2)$. We can represent this network with the following equations :

$$\begin{aligned} z_i &= \sum_{j=1}^{2} w_{ij}^{(1)} x_j + b_i^{(1)} \text{ pour } i = 1, 2 \\ h_i &= \sigma(z_i) \text{ pour } i = 1, 2 \\ y_k &= \sum_{i=1}^{2} w_{ki}^{(2)} h_i + b_k^{(2)} \text{ pour } k = 1, 2 \\ \mathcal{L} &= \frac{1}{2} \sum_{k=1}^{2} (y_k - t_k)^2 \end{aligned}$$

where $w^{(1)}$, $w^{(2)}$, $b^{(1)}$ and $b^{(2)}$ are parameters of the network, and $t = (t_1, t_2)$ is the value we want to approximate (the "real" output for x).

2.2 Back propagation

3 Residual neural network

A *residual neural network* is simply a regular network except that they have more connections. Indeed, not only do we feed the output of the previous layer to the next, but also the input of that layer. Thanks to that, residual networks have a better accuracy.

In these networks, the $k+1$th layer has the formula :

$$x_{k+1} = x_k + F(x_k)$$

where F is the function of the kth layer and its activation. This simple formula is a special case of the formula :

$$x_{k+1} = x_k + h.F(x_k),$$

which is the formula for the Euler method for solving ordinary differential equations (ODEs) when $h = 1$. It is with this observation that we can later introduce neural ODE.

4 Implicit Layers

There is two different ways to define a layer : *explicitly* or *implicitly*. When we define a layer explicitly, we specify the exact sequence of operations to do from the input to the output layer like in the example of the section 2.1.

However, when we add some functionality to the layers, it can become complex to define them explicitly. Instead, we should define them implicitly. In this case, we specify the condition we want the layer's output to satisfy.

In other words, lets assume that we have an input space $\mathcal{X}$ and an output space $\mathcal{Y}$. Then we define the explicit layers by a function $f : \mathcal{X} \to \mathcal{Y}$ such as $y = f(x)$ for some $x \in \mathcal{X}$. But we define an implicit layer by giving the condition that the function $g : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^n$ should satisfy (for example we can search for a y such that $g(x, y) = 0$).

A special case is a layer that uses an ODE solver. For each layer t we define the output $y(t)$ as the solution of the ODE :

$$\partial_t y(t) = f(t, y(t)), \qquad y(0) = y_0.$$

À retravailler ! ! !

4.1 Implicit function theorem

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a function.

We denote the derivative of f evaluated at a point $x \in \mathbb{R}^n$ as :

$$\partial f : \mathbb{R}^n \to \mathbb{R}^m$$

We can write the first-order Taylor's development for f at x as :

$$f(x + a) = f(x) + a.\partial f(x) + O(\|a\|^2)$$

where $a \in \mathbb{R}^n$ is a vector.

We also use the following notation :

$$\partial_0 f(x, y) = \frac{\partial f(x, y)}{\partial x}$$

$$\partial_1 f(x, y) = \frac{\partial f(x, y)}{\partial y}$$

Théorème 1. The implicit function theorem

Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ be a function and $a_0 \in \mathbb{R}^p, z_0 \in \mathbb{R}^n$ two vectors such that :

1. $f(a_0, z_0) = 0$;
2. f is continuously differentiable with a non-singular Jacobian, i.e. its determinant is non zero, $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z* : S_{a_0} \to S_{z_0}$ such that :
— $z_0 = z^*(a_0)$,
— $\forall a \in S_{a_0}, f(a, z^*(a)) = 0$,
— z^* is differentiable on S_{a_0}.

5 Neural ODE

5.1 Définition

(Here, let's consider a neural network as a continuous function with no grouping, no blocks. We normally have to decide how many layers we want in our neural network, and then we can build the network. With this idea, instead of specifying the number of layers, we need to specify the desired accuracy of the function, and it will learn how to train itself within that margin of error.)

In a residual neural network, the output for input x is a function $F(x, \Theta)$ where Θ represents the parameters of each layer.

But we want to extract all these individual layers and only have one "shared" layer.

insérer des dessins ! ! !

Any output of the layer of a residual network can be computed in the ODE network with the function :

$$F(z_t, t, \Theta)$$

with t being the number of that layer of the network minus one.

In the figure ?, the output of the kth layer is :

$$z_k = f(z_{k-1}, k-1) + z_{k-1} = F(z_{k-1}, k-1, \Theta).$$

We can then view z as a function of t. For example,

$$z(1) = f(x, 0) + x.$$

We can also write F as a continuous function of t, so that F is no longer a function of z. However, we need to give it the initial value of z, which is $z(0) = x$ (the input).

Let's now consider that the value given by $F(z(t), t, \Theta)$ is the derivative of $z(t)$. Thus, instead of doing $z(t) = F(z(t), t, \Theta)$, we put the model on the derivative, $z'(x) = F(z(t), t, \Theta)$. We obtain the following ODE :

$$\partial_t z(t) = F(z(t), t, \Theta)$$

where the initial condition for "time" 0 is $z(0) = x$.

To get the output we will have to solve this ODE.

The final condition at a certain "time" (the number of layers) will be the desired output of the neural network.

5.2 Forward pass

With the above, we have that the output of the residual neural network is given by

$$F(z(n), n, \Theta)$$

where n is the number of layers. In the ODE neural network, instead of having multiple individual layers, the entire network is one continuous block of computation. This means that we do not need to specify the number of layers beforehand.

We observe that in this case, the layer is defined implicitly by the ODE with initial condition :

$$\partial_t z(t) = F(z(t), t, \Theta).$$

But how do we find the solution to this ODE, i.e. the output ? We can simply use an ODE Solver, like Euler method or Runge-Kutta for example. In the case of the Euler method, the result is equivalent to a residual neural network, as we saw in section 3.

ici du code ! ! !

5.3 Backward pass : the Adjoint method

Now that we know how to calculate the output from the input and the parameter θ, we need a method to find the optimal θ that minimize the loss function.

In regular neural networks, we usually use the gradient descent. However in our case, it is more difficult because we used an ODE solver in the forward pass which is some sort of black box. This is why we are introducing the adjoint method. This method computes the gradient by solving a second ODE backwards and is applicable to all ODE solvers.

Let L be a loss function.

Then the error for an input $z(t_0)$ is given by :

$$L(z(t_1)) = L(z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta)dt) = L(ODESolve(z(t_0), f, t_0, t_1, \theta))$$

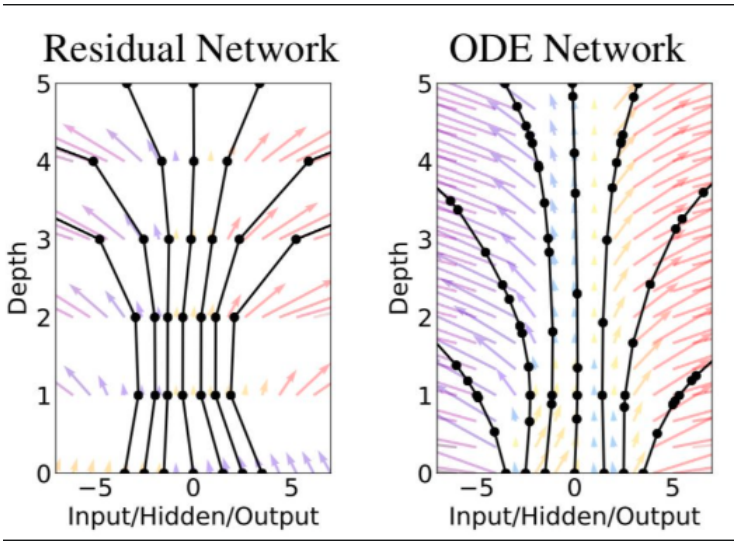
To optimise L, we need gradients with respect to θ. We have :

5.4 Advantages and disadvantages

In regular neural networks, we consider discrete, individual and independent layers. They are just a block of operations and the neural network consists of those blocks. However, ODE network can be seen as continuous functions. Instead of having separate layers, the entire network is one continuous block of computation. This leads to many advantages but also some disadvantage :

— The most benefit is that ODENet has more accurate results for time series predictions. Regular neural network have discrete layers, which means they expect the intervals for these time series data sets to be fixed. Therefore, they are bad at predicting output for time series data that is irregular.

— They have a faster testing time than regular networks, but a slower training time. Thus it's perfect for low power edge computing. There is a trade-off between precision and speed.
— We can use ordinary differential equations solvers instead of gradient descent. These solvers have a hundred plus years of theory behind them.
— Lastly, there's a constant memory cost, instead of increasing the cost linearly with each layer in a regular network.
— Regular neural networks can be evaluated with a fixed amount of computation, and are typically faster to train. In this case, we don't have to choose an error tolerance for a solver.



voir def vector field
voir probleme du vanishing gradient dans les nn regulier
recherche the Universal Approximation Theorem states that, for enough layers or enough parameters, $ML(x)$ can approximate any nonlinear function sufficiently close (subject to some constraints).