Introduction to neural ODE

Della Bona Sarah, Dumez Erika

June 3, 2021

What are ODE-nets?

ODE-nets are deep neural networks models using ordinary differential equations. Here, we will focus in particular on the mathematical aspects of these neural networks. We will give definitions and properties for different notions such as ordinary differential equations, regular and residual neural networks, implicit layers, ...

At the end, we'll conclude with the advantages and disadvantages of ODE-nets. The code use to make the examples can be found at `https://github.com/DumezErika/ProjetMachineLearning`.

First Order Ordinary Differential Equations

An *ordinary differential equation* (ODE) [9] is an equation that describes the changes of a function through time. The aim is to compute that function from the ODE which describes its derivative. In this setting, time is a continuous variable.

Definition

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

$$\frac{\partial u}{\partial t}(t) = f(t, u(t))$$

Solution of ODE's

Definition

A *solution* for this ODE is a function $u : I \in \mathbb{R} \to \mathbb{R}^N$, where I is an interval, such that

- u is differentiable on I,
- $\forall t \in I, (t, u(t)) \in \Omega$,
- $\forall t \in I, \frac{\partial u}{\partial t}(t) = f(t, u(t))$

Definition

An *initial condition* (IC) is a condition of the type

$$u(t_0) = u_0$$

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

Cauchy Problems

Definition

A *Cauchy problem* is an ODE with IC

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

Exemple : Let $\frac{\partial x}{\partial t}(t) = x(t)$ an ODE.
The solutions of this ODE are $\{x(t) = ae^t \mid a \in \mathbb{R}\}$. Indeed,

$$\forall a \in \mathbb{R}, \frac{\partial ae^t}{\partial t} = ae^t$$

If we add an initial condition $x(0) = 1$, we have a Cauchy problem and its solution is e^t, since $e^0 = 1$ and $\partial_t e^t = e^t$.

Lipschitz continuous functions

If we want to find the solution to an ODE, we need to know the conditions under which this ODE has a solution.
Thus, we define *Lipschitz continuous functions*.

Definition

Let (X, d_X) and (Y, d_Y) be two metric spaces.
A function $f : X \to Y$ is called *Lipschitz continuous* if

$$\exists K \geq 0,\ \forall x_1, x_2 \in X,\ d_Y(f(x_1), f(x_2)) \leq K d_X(x_1, x_2).$$

Picard-Lindelöf theorem

The notion of Lipschitz continuous functions is crucial for the Picard-Lindelöf theorem which gives conditions for the existence and uniqueness of a solution to an ODE.

Theorem

Consider the Cauchy problem

$$\frac{\partial u}{\partial t}(t) = f(t, u(t)), \quad u(t_0) = u_0.$$

Suppose f is uniformly Lipschitz continuous in u and continuous in t. Then for some value $T > 0$, there exists a unique solution $u(t)$ to the Cauchy problem on the interval $[t_0, t_0 + T]$.

One-step methods

Unfortunately, it is not always possible to explicitly find a solution to a Cauchy problem.
However, let $T > 0$ such that the solution u exists on $[t_0, t_0 + T]$ and let $n \geqslant 2$ be a natural. Let $t_0 < ... < t_n \in [t_0, t_0 + T]$ where $t_n = t_0 + T$. We can compute a finite number of points $(u_1, \ldots, u_n)$ such that:

$$\forall i \in \{0, \ldots, n\}, u_i \approx u(t_i).$$

To compute those points, we use *one-step methods* which compute the points u_{i+1} from the previous point u_i, the time t_i and the *step* $h_i := t_{i+1} - t_i$.

Euler's method is a one-step method with a constant step h.
It is similar to a Taylor development, the idea is to compute $u(t_{i+1})$ using the formula

$$u(t_{i+1}) \approx u(t_i) + h\frac{\partial u}{\partial t}(t_i) \tag{1}$$

where

$$\frac{\partial u}{\partial t}(t_i) = f(t_i, u(t_i)).$$

Machine Learning problem

In a typical machine learning problem [8], we have an output variable Y to p predictors $X_1, \ldots, X_p$, also called input variable, where $p \in \mathbb{N} \backslash \{0\}$.

- The inputs belongs to an input space $\mathcal{X}$ and usually $\mathcal{X} \subset \mathbb{R}^p$.
- The output belongs to a output space $\mathcal{Y}$. It depends on the problem, for example: if this is a regression problem, $\mathcal{Y} \subset \mathbb{R}$. But if we have a classification problem with K categories, $\mathcal{Y} = \{1, 2, \ldots, K\}$.

Machine Learning problem

Let's assume that there is some relationship between Y and $X = (X_1, \ldots, X_p)$, which can be written in the general form

$$Y = f(X) + \epsilon.$$

Here f is some fixed but unknown function, called *target function*, of $X_1, \ldots, X_p$ and ϵ is a random error term which is independent of X and has mean zero and finite variance.

The goal of machine learning is to estimate this function f as precisely as possible. To do that, we need a *data set* to learn. The data is a set of n points in $\mathcal{X} \times \mathcal{Y}$

$$\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}.$$

Machine Learning problem

Let x be a data point, then we can predict its output y using

$$\hat{y} = \hat{f}(x),$$

where $\hat{f}$ represents our estimate for f, and $\hat{y}$ represents the resulting prediction for y.

To determine the precision of an estimation $\hat{f}$, we use a *loss function*. Some example of loss functions are

- Square error loss: $\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$;
- Absolute error loss: $\mathcal{L}(y, \hat{y}) = |y - \hat{y}|$;
- Zero-one loss: $\mathcal{L}(y, \hat{y}) = \mathbb{1}_{\{(y,\hat{y})|y \neq \hat{y}\}}(y, \hat{y})$.

Neural Network

Definition

A *neural network* [1] can be used to solve a machine learning 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 x to the neurons of a layer, they do some calculus and give an output z.

An *activation function* is then applied to this output and obtain a value h before transmitting it to the next layer thanks to the connections between the neurons of each layer.

Examples

Some example of activation function are:

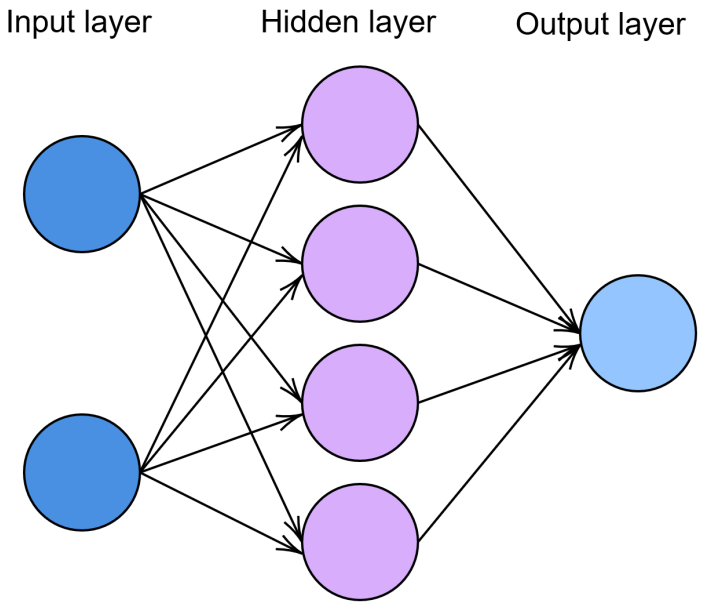
- Hyperbolic tangent ($tanh$);
- Sign;
- Sigmoid;
- ReLu.

The simplest example of a neural network layer is

$$h = \sigma(wx + b)$$

where σ is an activation function, w is a weight matrix and b a bias vector.

We begin by giving an input to the input layer, which transmits information to the first hidden layer. In turn, it transmit information to the next layer and so on, until the output layer gives us the final output, the *prediction*.



The goal is to minimize the error for every input. To do that, we need to find the optimal parameters for the network which minimize the loss function.

Back propagation

Let θ be the parameters of the network. We want to find θ which minimize the loss function in order to have the error as small as possible.

We know that if the partial derivative of a function is 0 at a certain point, then this point is a local extremum. Therefore, we need to determine the partial derivative of the loss function with respect to the parameters, $\frac{\partial L}{\partial \theta}$.

Backpropagation [5] is the process used to compute this derivative. It works by computing the gradient of the loss function with respect to each parameter by the chain rule, computing the gradient one layer at a time, iterating backward from the final layer to avoid redundant calculations of intermediate terms in the chain rule.

Example

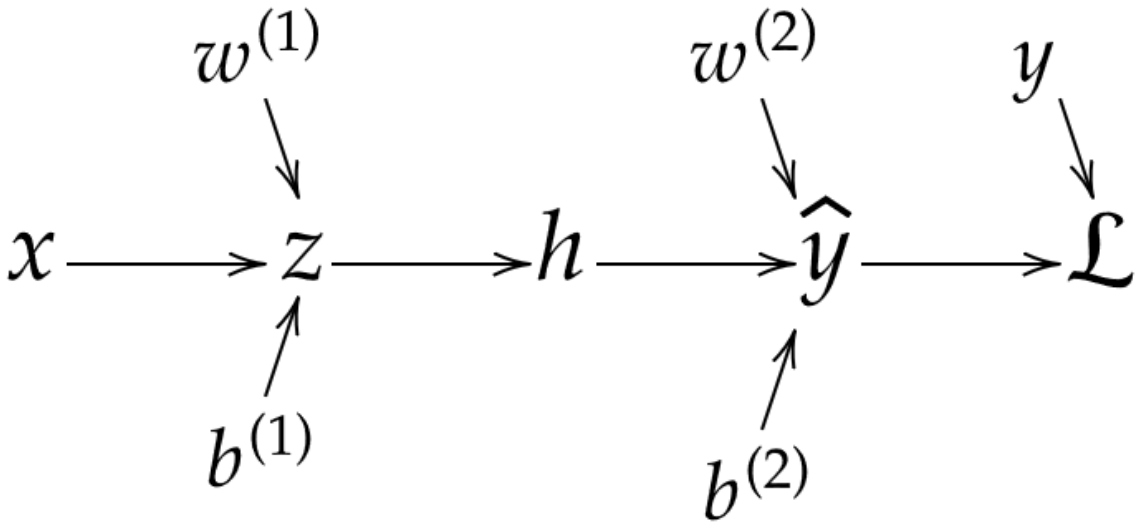
Let's consider a neural network with one hidden layer that takes a two-dimensional input $x = (x_1, x_2)$ and gives a 2-dimensional output $\hat{y} = (\hat{y_1}, \hat{y_2})$. We can represent this network with the following equations:

$$\begin{aligned} z &= w^{(1)}x + b^{(1)} \\ h &= \sigma(z) \\ \hat{y} &= w^{(2)}h + b^{(2)} \\ \mathcal{L} &= \frac{1}{2}\|\hat{y} - y\|_2^2 \end{aligned}$$

where $w^{(1)}, w^{(2)} \in \mathbb{R}^2 \times \mathbb{R}^2$ and $b^{(1)}, b^{(2)} \in \mathbb{R}^2$ are parameters of the network and σ is an activation function.

Example

We can now use the backpropagation algorithm to easily compute $\frac{\partial \mathcal{L}}{\partial w^{(1)}}, \frac{\partial \mathcal{L}}{\partial w^{(2)}}, \frac{\partial \mathcal{L}}{\partial b^{(1)}}, \frac{\partial \mathcal{L}}{\partial b^{(2)}}$, the partial derivatives of the loss function with regards to the parameters.



Example

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathcal{L}} &= 1 \\
\frac{\partial \mathcal{L}}{\partial \hat{y}} &= \frac{\partial \mathcal{L}}{\partial \mathcal{L}} (\hat{y} - y) \\
\frac{\partial \mathcal{L}}{\partial w^{(2)}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} h^T \\
\frac{\partial \mathcal{L}}{\partial b^{(2)}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}}
\end{aligned}
\qquad
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial h} &= \left(w^{(2)}\right)^T \frac{\partial \mathcal{L}}{\partial \hat{y}} \\
\frac{\partial \mathcal{L}}{\partial z} &= \frac{\partial \mathcal{L}}{\partial h} \circ \sigma'(z) \\
\frac{\partial \mathcal{L}}{\partial w^{(1)}} &= \frac{\partial \mathcal{L}}{\partial z} x^T \\
\frac{\partial \mathcal{L}}{\partial b^{(1)}} &= \frac{\partial \mathcal{L}}{\partial z}
\end{aligned}
$$

Gradient Descent

Gradient descent [8] is a process used to find a local minimum of a differentiable function.
It works as follow: at each step of the process, we take a step in the opposite direction of the gradient of the function at the current point.

More formally, if we have a function $f : \mathbb{R}^n \to \mathbb{R}$, $n > 1$, differentiable and a point $x_0 \in \mathbb{R}^n$, we have that if

$$x_{n+1} = x_n - \gamma_n \nabla f(x_n), n \geq 0$$

for $\gamma_n \in \mathbb{R}^+$ small enough, then $f(x_n) \geq f(x_{n+1})$.

Gradient descent

We get a sequence $x_0, x_1, \ldots$ that converges to the desired local minimum under some conditions, such that

$$f(x_0) \geq f(x_1) \geq \ldots.$$

If the function f is convex, all local minima are also global minima, so the gradient descent can converge to the global minimum.

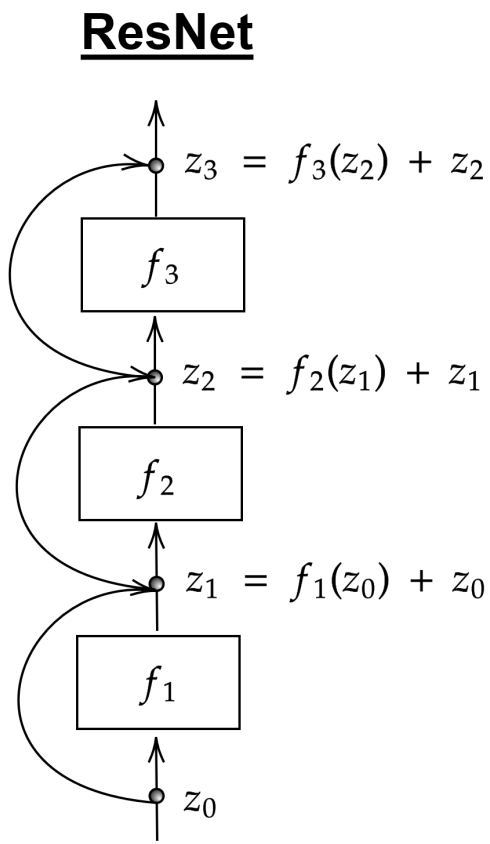
Vanishing and exploding gradient

- The gradient descent algorihtm updates each weight using $\frac{\partial \mathcal{L}}{\partial w}$;
- If this gradient is too small, it will prevent the weight from changing. In this case, the neural network will not learn.
- For example, if we use *tanh*, since this function has gradient in the range $]0, 1[$ and backpropagation computes gradients by the chain rule, we multiply several of these small numbers which leads the gradient to decrease exponentially.

The exploding gradient problem is the opposite, it happens when the derivatives take on larger values.

Residual neural network

A *residual neural network* [4], also called ResNet, is a regular neural network which has more connections. Indeed, a layer receives as input the outputs of the previous layer and its inputs.



In these networks, the output of the $k+1$th layer is given by $x_{k+1} = x_k + f_k(x_k)$, where f_k is the function of the kth layer and its activation.

ResNets and Euler

We can see that this simple formula is a special case of the formula

$$x_{k+1} = x_k + hf_k(x_k),$$

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

Solution for vanishing gradient

With these additional connections, we can avoid the problems of the *vanishing gradient* and the *exploding gradient* and thus have a better accuracy.

Indeed, by introducing short paths which can carry a gradient over the entire extent of very deep networks, we add information from the previous layer which makes these activations larger. Thus, they will prevent these activations from becoming exponentially small.

Example

We can implement a simple ResNet to approximate the function

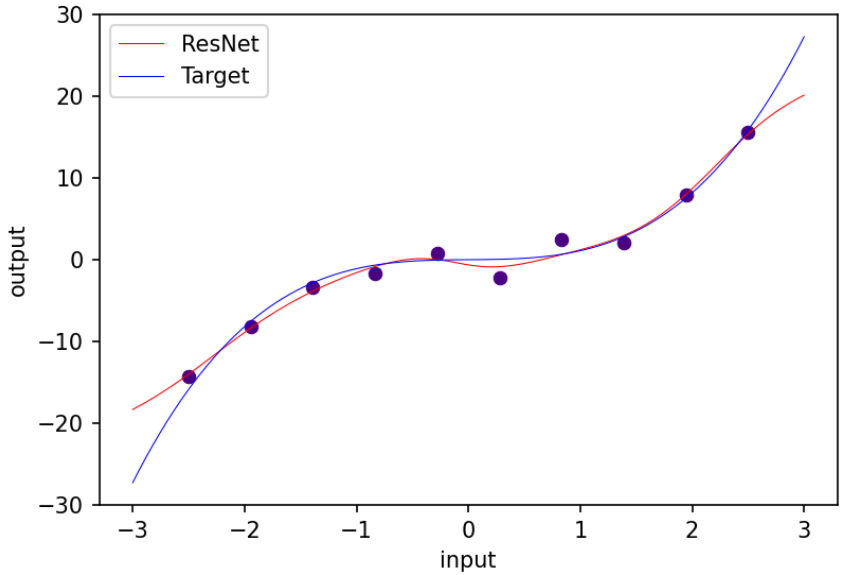
$$h(x) = x^3 + 0.1x.$$

To do that, we generate 10 points between -2.5 and 2.5. Their associated output comes from the function

$$h(x) + \varepsilon,$$

where ε is a noise variable with mean 0 and standard deviation 1.

We train a ResNet with 3 layers, the hidden one having 20 neurons. After 1000 iterations.



The green points represent the data used for the training, the blue line is the function we want to approximate and the red line is the function represented by the ResNet. The out-of-sample error for the points used to trace the line is 4.6735477.

Explicit and implicit layers

There is two different ways to define a layer : *explicitly* or *implicitly* [7]. When we define a layer explicitly, we specify the exact sequence of operations to do from the input to the output layer.

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

Definition

An *explicit layer* is defined by a function $f : \mathcal{X} \to \mathcal{Y}$. For an implicit layer, we give a condition that a function $g : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^n$ should satisfy. For example we can search for a y such that $g(x, y) = 0$.

Neural ODE

In a residual neural network, the output for an input x is a composition of functions. We want to extract all these individual layers to only have one "shared" layer.

Definition

A *neural ODE network* (or ODE-Net) [2, 7, 3, 4] takes a simple layer as a building block. This “base layer” is going to specify the dynamics of an ODE.

ODE-Net enable us to replace layers of neural networks with a continuous-depth model. This means that we do not need to specify the number of layers beforehand.

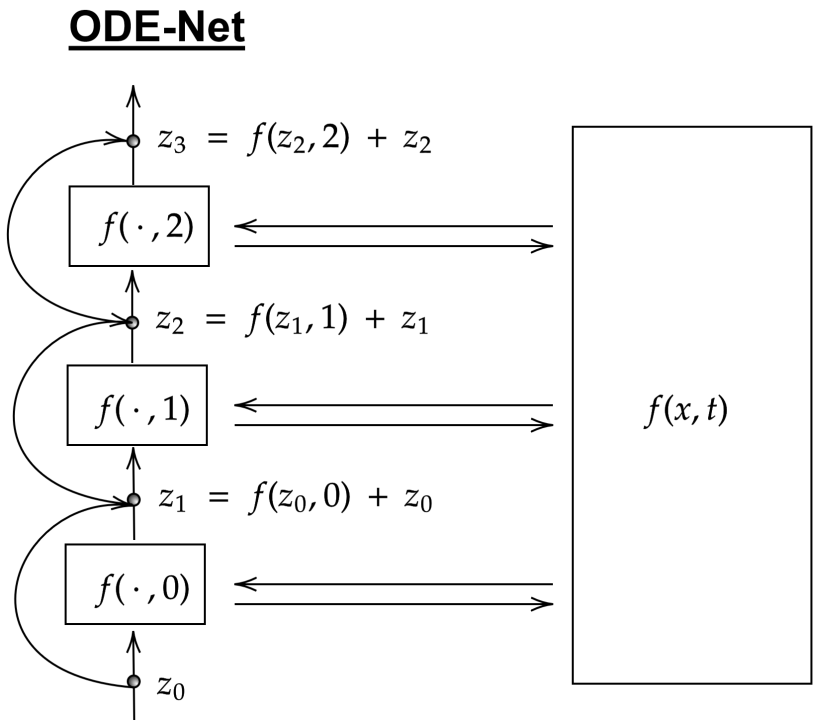
Comparison with ResNets

Let us return to ResNets to give intuition behind this definition.
We know that any output of the k^{th} layer of a residual network can be computed with the function

$$F(z_t, t; \theta) = f(z_t, t) + z_t$$

where $t = k - 1$.

Thus, in the ResNet, the output for the input $z_0 = x$ is a composition of the functions $F(z_t, t; \theta)$ where θ represents the parameters of the layers.



We can then view z as a function of t. For example, $z(1) = f(x, 0) + x$.

With that, we can write $F(z_t, t, \theta) = F(z(t), t, \theta)$. However, we need to give it the initial value of z, which is $z(t_0) = x$.

We saw that in ResNets, the outputs of each layer are the solutions of an ODE using Euler's method (cf Section 3).
The ODE from which it is a solution is $\frac{\partial z}{\partial t}(t) = f(z(t), t; \theta)$.

But we want to use a more precise method and then use a more complex ODE solver such as linear multistep methods. With what we've just shown, it is possible !
If we consider that the value given by $f(z(t), t, \theta)$ is the derivative of $z(t)$, we obtain the following Cauchy problem:

$$\begin{cases} \frac{\partial z}{\partial t}(t) = f(z(t), t; \theta) \\ z(t_0) = x \end{cases} \tag{2}$$

Forward pass

The layer in an ODE-Net is implicit. The output $z(t_N)$ of an ODE-Net with the input $z(t_0)$ is defined by the Cauchy problem (2), which depends on the parameters $z(t_0), t_0, t_N, \theta$.

But how do we solve this problem?

We can use an ODE Solver with the parameters given above.
To be able to use an ODE solver we have to make sure that the function satisfies the hypotheses in the theorem of existence and uniqueness (cf Section 1).

Backward pass

Now that we know how to calculate the output of an ODE-Net, we need a method to find the optimal parameters that minimize the loss function.

- In regular neural networks, we usually use the gradient descent
- For ODE-Nets, 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* [2]. This method computes the gradient by solving a second ODE backwards and is applicable to all ODE solvers.

Adjoint method

Let $\mathcal{L} : \mathbb{R}^p \to \mathbb{R}$ be a loss function.
To minimize this loss function $\mathcal{L}$, we need gradients with respect to the parameters $z(t_0), t_0, t_N, \theta$. To achieve that, we can determine how the gradient of the loss depends on the hidden state $z(t)$ for each t, which is

$$a(t) = \frac{\partial \mathcal{L}}{\partial z(t)} \tag{3}$$

This quantity is called the **adjoint**. We would like to determine its dynamics, so we need to compute its derivative with respect to t.

Adjoint method

With a continuous hidden state, we can write the transformation after an ε change in time as :

$$z(t+\varepsilon) = \int_t^{t+\varepsilon} f(z(t), t, \theta)dt + z(t). \tag{4}$$

Let $G : \varepsilon \mapsto z(t+\varepsilon)$. We can apply the Chain rule and we have

$$\frac{\partial \mathcal{L}}{\partial z(t)} = \frac{\partial \mathcal{L}}{\partial z(t+\varepsilon)} \frac{\partial z(t+\varepsilon)}{\partial z(t)}.$$

In other words

$$a(t) = a(t+\varepsilon)\frac{\partial G(\varepsilon)}{\partial z(t)}. \tag{5}$$

Adjoint method

$$
\begin{aligned}
\frac{\partial a}{\partial t}(t) &= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t)}{\varepsilon} \text{ by definition of the derivative.} \\
&= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t+\varepsilon)\frac{\partial G(\varepsilon)}{\partial z(t)}}{\varepsilon} \text{ by (5).} \\
&= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t+\varepsilon)\frac{\partial z(t) + \varepsilon f(z(t),t,\theta) + O(\varepsilon^2)}{\partial z(t)}}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t+\varepsilon)(\mathbb{1} + \varepsilon \frac{\partial f(z(t),t,\theta)}{\partial z(t)} + O(\varepsilon^2))}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{-\varepsilon a(t+\varepsilon)\frac{\partial f(z(t),t,\theta)}{\partial z(t)} + O(\varepsilon^2)}{\varepsilon} \\
&= -a(t)\frac{\partial f(z(t),t,\theta)}{\partial z(t)}
\end{aligned}
$$

Adjoint method

We now have the dynamics of $a(t)$

$$\frac{\partial a(t)}{\partial t} = -a(t)\frac{\partial f(z(t), t, \theta)}{\partial z(t)} \tag{6}$$

As we are searching for $a(t_0) = \frac{\partial L}{\partial z(t_0)}$, we need to solve an ODE for the adjoint backwards in time because the value for $a(t_N)$ is already known. The constraint on the last time point, which is simply the gradient of the loss with respect to $z(t_N)$,

$$a(t_N) = \frac{\partial \mathcal{L}}{\partial z(t_N)},$$

has to be specified to the ODE solver.

Adjoint method

Then, the gradients with respect to the hidden state can be calculated at any time, including the initial value.
We have

$$\begin{aligned} a(t_0) &= a(t_N) + \int_{t_N}^{t_0} \frac{\partial a(t)}{\partial t} dt \\ &= a(t_N) - \int_{t_N}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} dt \text{ par (6).} \end{aligned}$$

If we want to compute the gradient with respect to the parameters θ, we have to evaluate another integral, which depends on both $z(t)$ and $a(t)$,

$$\frac{\partial L}{\partial \theta} = - \int_{t_N}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt. \tag{7}$$

Adjoint method

To avoid computing each ODE on its own, we can do all of them at the same time. To do that we can generalize the ODE to

$$\frac{\partial}{\partial t}\begin{bmatrix} z \\ \theta \\ t \end{bmatrix}(t) = f_{aug}([z(t), \theta, t]) := \begin{bmatrix} f([z(t), \theta, t]) \\ 0 \\ 1 \end{bmatrix},$$

$$a_{aug}(t) := \begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix}(t), \; a(t) = \frac{\partial \mathcal{L}}{\partial z(t)}, \; a_\theta(t) = \frac{\partial \mathcal{L}}{\partial \theta(t)}, \; a_t(t) := \frac{\partial \mathcal{L}}{\partial t(t)}.$$

Adjoint method

The jacobian of f has the form

$$\frac{\partial f_{aug}}{\partial [z(t), \theta, t]}([z(t), \theta, t]) = \begin{bmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}(t)$$

where each $\mathbf{0}$ is a matrix of zeros with the corresponding dimensions. We can inject a_{aug} in (6) and we get

$$\begin{aligned} \frac{\partial a_{aug}(t)}{\partial t} &= -[a(t)\ a_\theta(t)\ a_t(t)] \frac{\partial f_{aug}}{\partial [z(t), \theta, t]}([z(t), \theta, t]) \\ &= -\left[a\frac{\partial f}{\partial z}\ a\frac{\partial f}{\partial \theta}\ a\frac{\partial f}{\partial t}\right](t). \end{aligned}$$

Adjoint method

We can see that the first component, $-a(t)\frac{\partial f(z(t),t,\theta)}{\partial z(t)}$, is the adjoint differential equation that we calculated previously in (6).

The total gradient with respect to the parameters is given by integrating the second component, $-a(t)\frac{\partial f(z(t),t,\theta)}{\partial \theta(t)}$ over the full interval and by setting $a_\theta(t_N) = \mathbf{0}$.

We obtain

$$\frac{\partial \mathcal{L}}{\partial \theta} = a_\theta(t_0) = -\int_{t_N}^{t_0} a(t)\frac{\partial f(z(t),t,\theta)}{\partial \theta}dt.$$

Adjoint method

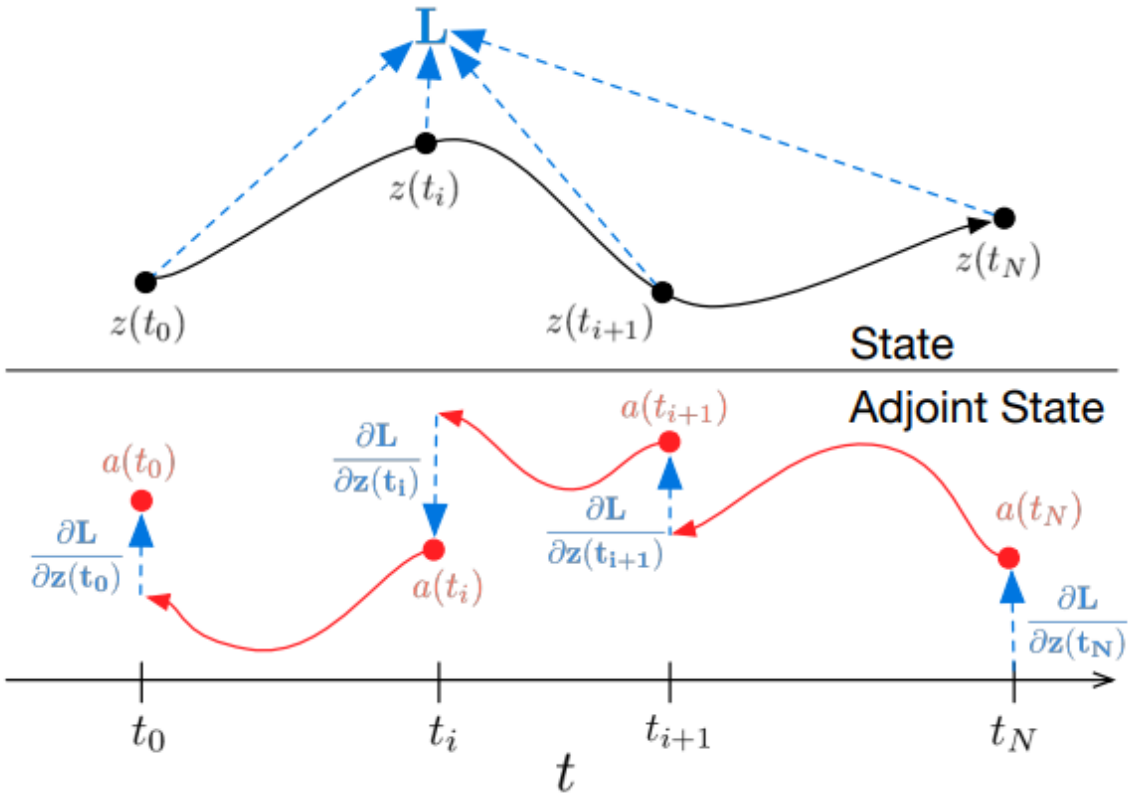
We can also get gradients with respect to t_0 and t_N by integrating the last component, $-a(t)\frac{\partial f(z(t),t,\theta)}{\partial t(t)}$, and by the Chain rule respectively.
We have

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial t_0} &= a_t(t_0) = a_t(t_N) - \int_{t_N}^{t_0} a(t)\frac{\partial f(z(t),t,\theta)}{\partial t}dt;\\ \frac{\partial \mathcal{L}}{\partial t_N} &= \frac{\partial \mathcal{L}}{\partial z(t_N)}\frac{\partial z(t_N)}{\partial t_N} = a(t_N)f(z(t_N),t_N,\theta).\end{aligned}$$

With this generalized method, we have gradients for all possible inputs to a Cauchy problem solver.
In the development above, we assumed that the loss function L depends only on the last time point t_N.

Adjoint method

We can represent this process with this figure. As we can see, during the forward pass, the loss function is evaluated a each time to be able to determine the smallest error.

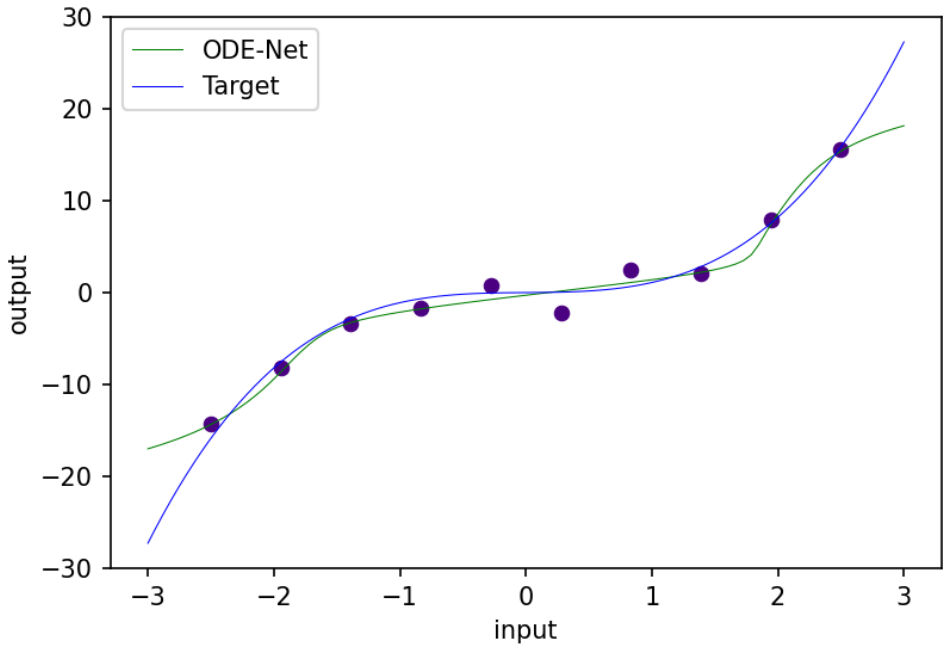


Example

We have the function

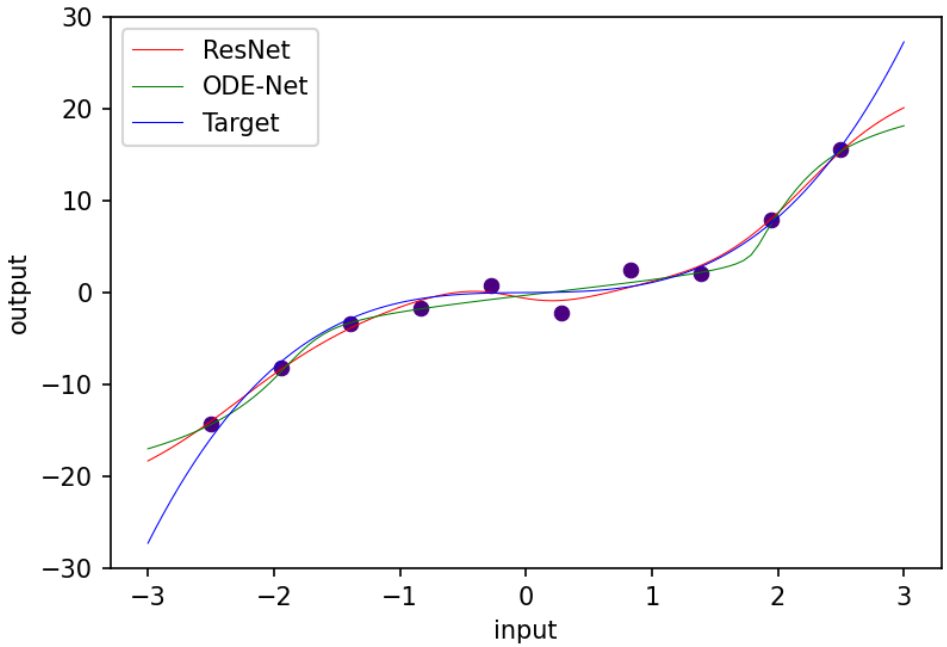
$$h(x) = x^3 + 0.1x$$

that we wish to approximate.
The dynamics of the ODE-Net is specified by a layer of size 20. After 1000 iteration, we get the function given in this figure.



Example

We can compare these results with those we had for the ResNet, we can see that the ResNets is slightly better with these parameters.



Advantages

- *Continuous time series predictions*
  The biggest advantage of ODE-Nets is that they have 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 whereas ODE-Net have a continuous layer which means we can evaluate the hidden states at every point t in time. Therefore, regular neural networks are bad at predicting output for time series data that is irregular.

Advantages

- *ODE solvers*
  We can use ordinary differential equations solvers instead of gradient descent. These solvers have more than a hundred years of theory behind them which is a great advantage against gradient descent.

Advantages

- *Robustness* [10]
  After experimenting, it was proved that ODE-Net are very robust against perturbed data compared to regular neural network. Two experiments were conducted: in the first one they trained an ODE-Net and a convolutional neural network[1] on real images without perturbations. They tested these models on the original images and the ODE-Net outperformed the CNN. In the second experiment, they trained these networks on the original and perturbed images. Again, the ODE-Net was much better.

[1] A neural network that is usually good with images.

Advantages

- *Constant memory cost*
  Lastly, there's a constant memory cost, instead of increasing the cost linearly with each layer in a regular network. In ODE-Net, we know the state at every time t. Because of that, we can always reconstruct the entire trajectory of an ODE forwards and backwards in time only by knowing this point. This means that ODE-Nets can be trained with a memory cost constant in the number of evaluations of f. There is a trade-off between memory cost and computational time: ResNets are faster but use more memory and ODE-Net are slower but use less memory.

Disadvantages

- *Slower training time*
  ODE-Net have a slower training time. Indeed, during training, the dynamics we want to learn tend to become expensive to solve since the network becomes deeper. However, 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.
  There is then a trade-off between accuracy and computational time: if we choose a small error tolerance, then the computational time be bigger.

Disadvantages

- *More Hyperparameters*
  In ODE-Nets we need to choose a solver and its error tolerance, which induces more choices to find the parameters which works better.

Disadvantages

- *Restriction on activation functions*
  To ensure that the ODE has a solution we have to make sure the dynamics are uniformly continuous Lipschitz (q.v. Theorem 1). This is why we mostly use *tanh* as an activation function.


Stanford cs229: Machine learning | autumn 2018, lecture 11, 12, 13.
https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU.


Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud.
Neural ordinary differential equations.
https://arxiv.org/pdf/1806.07366.pdf.


Ayan Das.
Neural ordinary differential equation (neural ode).
https://ayandas.me/blog-tut/2020/03/20/neural-ode.html.


Andriy Drozdyuk.
Neural odes introduction.
https://www.youtube.com/watch?v=uPd0B0WhH5w.
Carleton University.


Roger Grosse.
Csc321 lecture 6: Backpropagation.
http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/slides/lec6.pdf.


Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani.
An introduction to statistical learning with applications in r.
Springer.


Zico Kolter, David Duvenaud, and Matt Johnson.
Deep implicit layers - neural odes, deep equilibirum models, and beyond.
https://implicit-layers-tutorial.org/.


Souhaib Ben Taieb.
Machine learning.
Université de MONS, 2019-2020.


Christophe Troestler.
Introduction à l'analyse numérique.
Université de MONS, 2018-2019.


Hanshu Yan, Jiawei Du, Vincent Tan, and Jiashi Feng.
On robustness of neural ordinary differential equations.
https://openreview.net/forum?id=B1e9Y2NYvS.


Amer Zayegh and Nizar Al Bassam.
Neural network principles and applications.
https://www.researchgate.net/publication/329264107_Neural_Network_Principles_and_Applications.