Introduction to neural ODE

Della Bona Sarah, Dumez Erika

12 mars 2021

1 Introduction

In this document, we introduce ODE-nets, which are deep neural networks models using ordinary differential equations. We focus in particular on the mathematical aspects of these neural networks. In order to do this, we give the significant definitions and properties of different notions, like ordinary differential equations, regular and residual neural networks, implicit layers, ... At the end of the document, the advantages and disadvantages of ODE-nets are presented.

2 Ordinary Differential Equations

2.1 A reminder on ordinary differential equations

An ordinary differential equation, noted "ODE", is an equation that describes the changes of a function through time. In this setting, time is a continuous variable. The aim is to compute the function from an ODE which describes its derivative.

Notation 1. Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a function.
We denote the derivative of f as :

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

We also write, for $x \in \mathbb{R}^n$:

$$\partial_t f(x) = \frac{\partial f(x)}{\partial t}$$

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

$$\partial_t u(t) = f(t, u(t))$$

— A *solution* for this ODE 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 :

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

Definition 2. A *k-order ODE* takes the form :

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

where

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

2.2 A simple example

Let $\partial_t x(t) = x(t)$ an ODE. The solutions of the ODE are given by :

$$a.e^t \text{ where } a \in \mathbb{R}.$$

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

2.3 Existence and uniqueness of a solution

If we want to calculate a function from an ODE, we need to know the conditions under which this ODE has a solution. Thus, we define what is a function Lipschitz continuous. This notion is crucial for the following theorem giving the conditions for the existence and uniqueness of a solution to an ODE.

Definition 3. Let d_X and d_Y be the metrics on the sets X and Y respectively. Let (X, d_X) and (Y, d_Y) be two metric spaces.
A function $f : X \to Y$ is called *Lipschitz continuous* if

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

Theorem 1. Picard-Lindelöf theorem
Consider the Cauchy problem :

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0.$$

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

2.4 Euler's method

Unfortunately, as we can see from the previous theorem, it is not always possible to explicitly find a solution to a Cauchy problem. However, we can compute a finite number of points $u_i \in \mathbb{R}^N$ which are close to the real solution and thus, approximate the real function.

More precisely, let $T \in \mathbb{R}\backslash\{0\}$ 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 $(u_i)_{i=0}^n$ such that :

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

To compute those points, we use *1-step methods*. These methods compute de next point 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 1-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 following formula :

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

where

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

Definition 4. Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a function.

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.

3 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 as matrix operations.

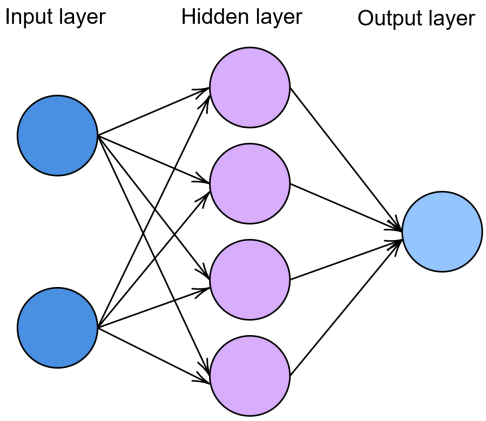
The neurons of each layer are connected to those of the next layer. Thanks to these connections, the output given by a neuron can be transmitted through the neural network.

We begin by giving an input to the input layer, which transmits information to the first hidden layer [1]. In turn, it transmit information to the next layer and so on, until the output layer gives us the final output, the *prediction*.

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.

1. There isn't always an hidden layer in a neural network

With a loss function, we can determine the accuracy of the neural network. Each layer introduces a little bit of error that propagate through the network. We are trying to find the optimal parameters which minimize this loss function. In order to do that, we use gradient descent.



3.1 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 :

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

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).

3.2 Back propagation

To find the parameters that minimize the loss function, we need to determine the differential of the loss function with respect to the parameters. This process is called *backpropagation*.

For the previous example, we have :

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathcal{L}} &= 1 \\
\frac{\partial \mathcal{L}}{\partial y_k} &= \frac{\partial \mathcal{L}}{\partial \mathcal{L}}.(y_k - t_k) \\
\frac{\partial \mathcal{L}}{\partial w_{ki}^{(2)}} &= \frac{\partial \mathcal{L}}{\partial y_k}.h_i \\
\frac{\partial \mathcal{L}}{\partial b_k^{(2)}} &= \frac{\partial \mathcal{L}}{\partial y_k} \\
\frac{\partial \mathcal{L}}{\partial h_i} &= \sum_{k=1}^{2} \frac{\partial \mathcal{L}}{\partial y_k}.w_{ki}^{(2)} \\
\frac{\partial \mathcal{L}}{\partial z_i} &= \frac{\partial \mathcal{L}}{\partial h_i}\sigma'(z_i) \\
\frac{\partial \mathcal{L}}{\partial w_{ij}^{(1)}} &= \frac{\partial \mathcal{L}}{\partial z_i}.x_j \\
\frac{\partial \mathcal{L}}{\partial b_i^{(1)}} &= \frac{\partial \mathcal{L}}{\partial z_i}
\end{aligned}$$

4 Residual neural network

We still want to approximate a function. A deep neural network really close the the neural ODE network is the *residual neural network*, also called ResNet. It is simply a regular neural network except that it has more connections. Not only do we feed the output of the previous layer to the next, but also the input of that layer.

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

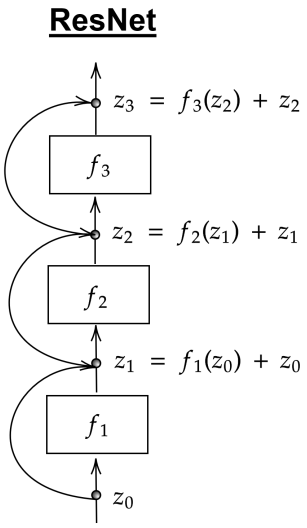
$$x_{k+1} = x_k + F(x_k)$$

where F is the function of the kth layer and its activation.

We can see that 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 ODEs when $h = 1$. It is with this observation that we can later introduce neural ODE networks.



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

The vanishing gradient problem is encountered when using gradient descent in the backpropagation. Each of the neural network's weights receives an update proportional to the partial derivative of the loss function with respect to the current weight in each iteration of training.

The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. Given that these partial derivatives are computed with the chain rule, this can easily occur, because you keep on multiplying small numbers. The deeper is the neural network, the more likely this problem can occur. In the worst case, this may completely stop the neural network from further training.

When the derivatives take on larger values, the exploding gradient problem risks to be encountered.

Residual networks avoid the problem of vanishing gradient by introducing short paths which can carry a gradient over the entire extent of very deep networks. This is because adding the information from the previous layer will make these activations larger, so to some extent, they will prevent these activations from becoming exponentially small.

5 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 3.1.

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.

Formally, let's assume that we have an input space $\mathcal{X}$ and an output space $\mathcal{Y}$. 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 $g(x, y) = 0$.

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 :

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

5.1 Implicit function theorem

Sometimes, variables can not be defined by a function but are rather defined by an equation. In this case, the *implicit function theorem* can be used. It says that if a function f is sufficiently regular in the neighborhood of a point, then there exists a function φ at least as regular as f such that locally, the graph of f and the graph of φ are the same.

Theorem 2. 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, $\partial_z f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

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}.

We could use the theorem to compute the derivatives for the backpropagation, but in the following we will use a simpler derivation based on ResNet with the adjoint method.

6 Neural ODE

6.1 Définition

In a residual neural network, the output for an input x is a function $F(x, \theta)$ where θ represents the parameters of the layers.

We want to extract all these individual layers to only have one "shared" layer.

Similar to a residual network, a *neural ODE network* (or ODE-Net) takes a simple layer as a building block. This “base layer” is going to specify the dynamics of an ODE.

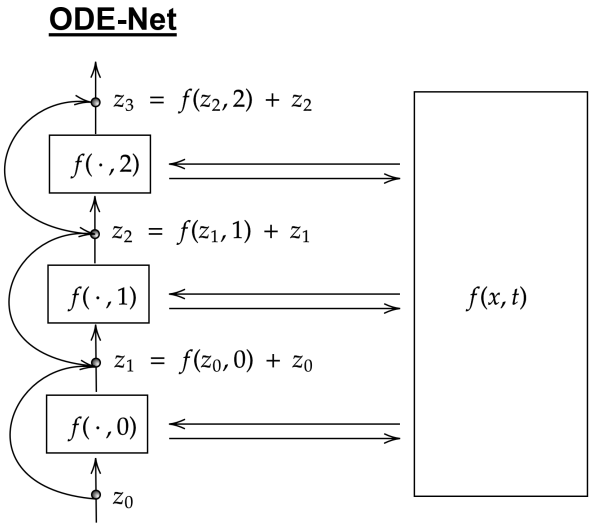
Neural ODE enable us to replace layers of neural networks with a continuous-depth model.

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.

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

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

with t being the layer's number minus one.



In this 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 write $F(z_t, t, \theta) = F(z(t), t, \theta)$. However, we need to give it the initial value of z, which is $z(0) = x$ (the input).

We saw that in ResNets, the outputs of each layer are the solutions of an ODE using Euler's method. But here we want to use a more precise method and then use a more complex ODE solver. 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 can put the model on the derivative, $z'(t) = F(z(t), t, \theta)$. We obtain the following ODE :

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

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

6.2 Forward pass

The output $z(t)$ of a layer in an ODE-Net is defined implicitly by the ODE with initial condition :

$$\partial_t z(t) = F(z(t), t, \theta), \quad z(0) = x,$$

where x is the input.

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 4.

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. For example, if the activation function used in the network is ReLu, we can't apply the theorem since it is not derivable in 0.

6.3 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 :

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

To minimize the loss function L, we need gradients with respect to θ. To achieve that, we first need to determine how the gradient of the loss depends on the hidden state $z(t)$ for each t, which is $\frac{\partial L}{\partial z(t)}$. This quantity is called the *adjoint* and is noted $a(t)$. We would like to determine its dynamics, so we need to compute its derivative with respect to t. [2]

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

$$z(t+\varepsilon) = \int_{t}^{t+\varepsilon} f(z(t), t, \theta)dt + z(t)$$

Let $G : \varepsilon \mapsto z(t+\varepsilon)$.

We can apply the Chain rule and we have :

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

In other words :

$$a(t) = a(t+\varepsilon)\frac{\partial G(\varepsilon)}{\partial z(t)} \quad (1)$$

2. Here we see the vectors as row vectors.

We have :

$$\begin{aligned}
\frac{\partial a(t)}{\partial t} &= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t)}{\varepsilon} \text{ by definition.} \\
&= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t+\varepsilon)\frac{\partial G(\varepsilon)}{\partial z(t)}}{\varepsilon} \text{ by (1).} \\
&= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t+\varepsilon)\frac{\partial z(t) + \varepsilon f(z(t),t,\theta) + \mathcal{O}(\varepsilon^2)}{\partial z(t)}}{\varepsilon} \text{ by Taylor's development of G in 0.} \\
&= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t+\varepsilon)(\mathbb{1} + \varepsilon\frac{\partial f(z(t),t,\theta)}{\partial z(t)} + \mathcal{O}(\varepsilon^2))}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{-\varepsilon a(t+\varepsilon)\frac{\partial f(z(t),t,\theta)}{\partial z(t)} + \mathcal{O}(\varepsilon^2)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} -a(t+\varepsilon)\frac{\partial f(z(t),t,\theta)}{\partial z(t)} + \mathcal{O}(\varepsilon) \\
&= -a(t)\frac{\partial f(z(t),t,\theta)}{\partial z(t)}
\end{aligned}$$

We need to solve an ODE for the adjoint backwards in time. The constraint on the last time point, which is simply the gradient of the loss with respect to this point, has to be specified :

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

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

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

Hence, the dynamics of the adjoint are given by another ODE :

$$\frac{\partial a(t)}{\partial t} = -a(t)^T \frac{\partial f(z(t),t,\theta)}{\partial z(t)} \quad (2)$$

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

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

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 :

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

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

The jacobian of f has the form :

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

where each $\mathbf{0}$ is a matrix of zeros with the corresponding dimensions.

We can use a_{aug} in (2) and we get :

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

We can see that the first element is the adjoint differential equation that we calculated previously. The total gradient with respect to the parameters is given by integrating the second element over the full interval and by setting $a_\theta(t_N) = \mathbf{0}$. We obtain :

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

We can also get gradients with respect to t_0 and t_N by integrating the last element and by the Chain rule respectively.

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

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. If function L depends also on intermediate time points $t_1, t_2, \ldots, t_{N-1}$, we can repeat the adjoint step for each of the intervals $[t_{N-1}, t_N], [t_{N-2}, t_{N-1}], \ldots, [t_0, t_1]$ in the backward order and sum up the obtained gradients.

In practice, most ODE solvers have the option to output the state $z(t)$ at multiple times. When the loss depends on these intermediate states, the reverse-mode derivative must be broken into a sequence of separate solves, one between each consecutive pair of output times. At each observation, the adjoint must be adjusted in the direction of the corresponding partial derivative $\frac{\partial L}{\partial z(t_i)}$.

6.4 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.