Neural Networks as Universal Approximators Exploiting Mathematical Characteristics

Maximilian Reihn
Dep. Information Science
University of Konstanz
Konstanz, Germany
maximilian-martin.reihn@uni-konstanz.de

***Abstract*—This work begins by motivating the mathematical characteristics of continuity of functions induced by simple neural networks with one hidden layer. This continuity is important for later parts since continuity of the derivative implies differentiability of the original function. Then 'pseudo' step functions induced by neural networks are presented with the help of which we can build arbitrary blocks of functions making it possible to approximate any real continuous function. This in fact is a more intuitive proof of why neural networks lie dense in the set of continuous functions over $\mathbb{R}$. A brief introduction to differential equations will then be given and finally a method presented with which differential equations can be solved with the aid of well established machine learning algorithms.**

I. Motivation

A. Continuity

Most real world processes include some kind of continuity. For example every kind of physical movement is continuous if the object of observance stays the same. To begin the simplest version of continuity will be presented and it will be elaborated on how that is useful for our final goal.

B. Differential equations

Nearly every field of science makes use of differential equations, they are omnipresent in every optimisation and physics task. A whole theoretical world of mathematics is constructed around the problem of solving differential equations in the most efficient manner possible. Sometimes it is also possible to get an analytical solution but other than in academics or theoretical physics this is not an approach commonly made.

C. Aim

The goal of this paper is to explain and establish how commonly known machine learning algorithms and the mathematical characteristics of an one hidden layer neural network can be exploited such that most kinds of differential equations can be solved iteratively.

II. Continuous and non-continuous functions

This section will begin purely theoretical and then go from there to explain how the concept of continuity helps in the problem at hand, which is the original aim to solve differential equations.

A. Theory

The most common definition of continuous functions is $\epsilon-\delta$ continuous functions. We begin stating the definition

$$\forall \epsilon > 0 \quad \exists \delta > 0 \quad s.t.\ \forall x \in D \subset \mathbb{R} \ \text{ if } \ |x - \hat{x}| < \delta$$
$$then \quad |f(x) - f(\hat{x})| < \epsilon.$$

This is made more clear with a counter example. Consider the function

$$f(x) = \begin{cases} x & for \quad x < 0 \\ x + 1 & for \quad x \geq 0 \end{cases}$$

which is visualised below.

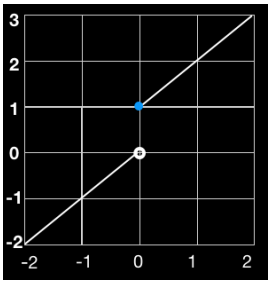

Fig. 1. Non-continuous function.

Now having seen the above definition, in this example consider $\hat{x} = 0$ and $x \nearrow 0$ then we will not be able to find a $\delta > 0$ for an $\epsilon < 1$ since the difference $|\hat{x} - x| > 1$ for all $x < 0$.

B. Neural Networks as functions

Here we will mainly consider neural networks of the form $[1, n, 1]$ but the same is applicable to networks of the form

$[m, n, k]$ for $m, n, k \in \mathbb{N}$. Mathematically they are represented as

$$N(x) = \sum_{i=1}^{n} \beta_i \, \sigma(\alpha_i^{(T)} x + \gamma_i),$$

the T as transpose is only needed if the networks input dimension $m > 1$ otherwise α_i, γ_i, x are just scalar values. For network outputs $k > 1$ this sum only represents the output of a single final layer neuron.
If the elementwise operator σ is chosen to be a continuous function, the whole function $N(x)$ will be continuous. However if the output is a prediction into classes and is defined as $y = argmax(N(x))$ then this is not continuous, but the single output neurons themselves are.
That is an important observation, neural networks with continuous activation functions are also continuous since the solution of differential equations must also be continuous. To conclude this, a neural network as described comes into question to be an approximative solution for a differential equation since it is continuous if the activation function is. Note that differentiability is also needed but again, if the activation function is differentiable p-times then the network is, too.

III. Approximation

A. Theory

This subsection will only briefly some up the purely mathematical paper of [1]. The analytical proof is made, that the set of functions called Σ lies dense in the continuous functions $\mathcal{C}([a, b])$. The subset Σ lies dense in $\mathcal{C}([a, b])$ if and only if

$$\forall \epsilon > 0, \; g \in \mathcal{C}([a, b]) \; \exists f \in \Sigma$$
$$s.t. \; max|f(x) - g(x)| < \epsilon \quad \forall x \in [a, b].$$

That means for arbitrary accuracy (maximal error ϵ) we can find a neural network which approximates any given function from $\mathcal{C}([a, b])$. This is shown more intuitive in the following sections.

B. Step function using sigmoid function

Now we use the sigmoid function as σ which is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$. Therefore the output of a neuron in the hidden layer is

$$\sigma(\alpha_i \, x + \gamma_i) = \frac{1}{1 + e^{-(\alpha_i \, x + \gamma_i)}}.$$

For $x = \frac{-\gamma}{\alpha}$ the output then is

$$\sigma(\frac{-\gamma}{\alpha}) = \frac{1}{1 + e^{-(\alpha_i \, \frac{-\gamma}{\alpha} + \gamma_i)}} = \frac{1}{2}.$$

Now we increase the magnitude of α, γ and can observe that the 'jump' from 0 to 1 is getting steeper. Parameters are set to $\gamma = -\alpha = 10,000$, therefore $\hat{x} = \frac{-(-10,000)}{10,000} = 1$. Now consider $\epsilon = 10^{-3}$, then $N_1(\hat{x}) = \frac{1}{1+e^{-10000\hat{x}-10000}} = \frac{1}{2}$, but $N_1(\hat{x} + \epsilon) = \frac{1}{1+e^{-10000(\hat{x}+\epsilon)-10000}} = 0.9995$ and $N_1(\hat{x} - \epsilon) = \frac{1}{1+e^{-10000(\hat{x}-\epsilon)-10000}} = 4.5397 \cdot 10^{-5}$. For visualisation the function is shown below.

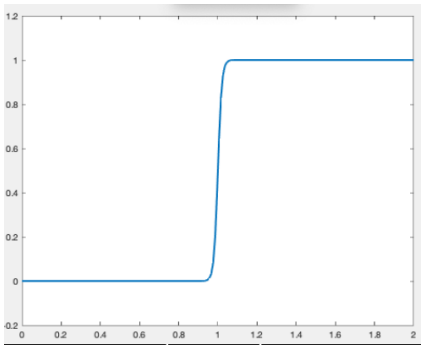

Fig. 2. $\alpha = 100$, $\gamma = -100$

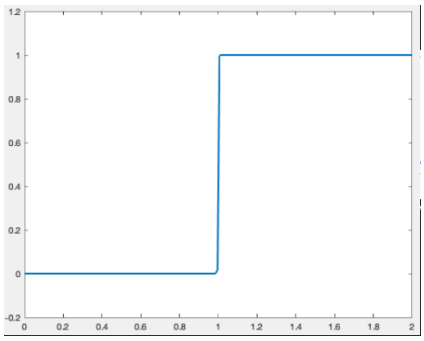

Fig. 3. $\alpha = 10,000$, $\gamma = -10,000$

Now those sharp steps can be put together with a pair of neurons and a 'block' function can be build as shown below.

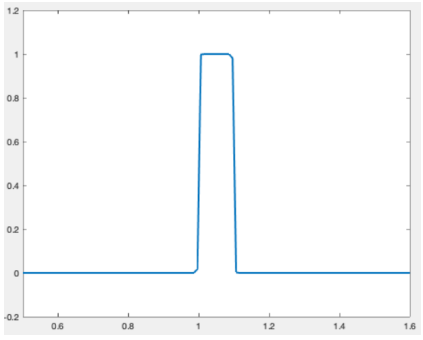

Fig. 4. Block function with $\alpha_1 = 1000, \gamma_1 = -1000$, $\frac{-\gamma_1}{\alpha_1} = 1.0$, $\beta_1 = 1$ and $\alpha_2 = 1000, \gamma_2 = -1100$, $\frac{-\gamma_2}{\alpha_2} = 1.1$, $\beta_2 = -1$

Now we are able to state a pseudo algorithm to approximate any given continuous function by using the above scheme.

Algorithm 1 Approximation via step functions

Require: function handle: f, interval: [a,b] and number of blocks: n (accuracy)

Return: function handle g of approximated function

$step = (b-a)/(n-1)$
$\alpha = 10,000$
$\gamma_1 = \gamma_2 = \beta = [None] \cdot n$
for i in range(n) **do**
  $height = f(a + i \cdot step + step/2)$
  $\beta[i] = height$
  $\gamma_1[i] = -(a + i \cdot step) \cdot 10,000$
  $\gamma_2[i] = -(a + (i+1) \cdot step) \cdot 10,000$
end for
return $g = \sum_{i=1}^{n} [\beta[i] \cdot \sigma(\alpha \cdot x + \gamma_1[i]) - \beta[i] \cdot \sigma(\alpha \cdot x + \gamma_2[i])]$

Note that this is to intuitively show why neural networks are able to approximate any continuous function. The same approximation could have been made by just training a neural network with x as input and $f(x)$ as training examples. Also the theoretical proof makes no statements about the size of the hidden layer n or how many neurons are needed for a certain accuracy. Results of this pseudo code are shown below. A similar technique is presented in [3] with some slight changes and different theoretical interpretation.

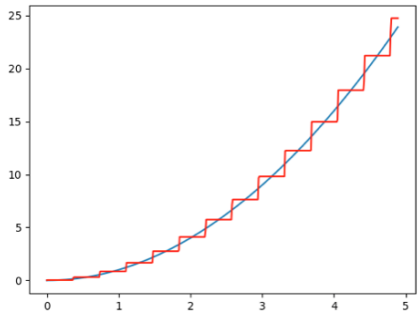

Fig. 5. $n = 20$, $f(x) = x^2$ on $[0, 5]$

C. Machine learning ?

This method is not a real machine learning task since there is no optimisation/learning phase involved. This is a deterministic scheme. To sum up we have now established that neural networks may well approximate any given continuous function and therefore come into question if one wants a approximative solution for an differential equation.

IV. Differential equations

Those equations state the relationship of functions to their derivatives. Their solution is a function itself which fulfills the relationship demanded in the equation. In general most physical and real world problems can be stated as differential equations, from simple mechanical processes up to super complex quantum physics which usually is expressed within partial differential equations.

For example let $x(t)$ be the time-distance relationship of an object, then one can describe the the free fall with $x^{(2)}(t) = -g$. An analytical solution to this equation would be

$$x(t) = -\frac{g}{2}(t - t_0)^2 + v_0\,(t - t_0) + x_0$$

for initial value conditions $x(t_0) = x_0$ and $x^{(1)}(t_0) = v(t_0) = v_0$.

V. Combining theory and machine learning

We want to now step by step fulfill the goal stated in I-C. A similar approach with different theoretical prerequisites and a different aim was taken in [2]. To explain we take an example at hand. Consider the following differential equation:

$$f^{(1)}(x) = -\frac{1}{5} f(x) \cdot e^{\frac{-x}{5}} \cdot \cos(x)$$

with initial value condition $f(0) = 0$ to be solved on the interval $x \in [0, 2]$.

Now we introduce a trial solution $\hat{f}$. This trial solution is dependent on a neural network and the goal is to manipulate the parameters of that network such that the trial solution approximates the differential equation. The trial solution's general form is

$$\hat{f}(x) = B(x) + G(x, N(x)).$$

The function $B(x)$ is to be set such that $\hat{f}$ fulfills the initial value condition, while $G(x, N(x))$ should be 0 at the initial value. In our case that would be $B(x) = 0$ is a constant function and $G(x, N(x)) = xN(x)$. We can easily see that $\hat{f}(0) = 0 + 0 \cdot N(x) = 0$ and since $\hat{f}(0) = 0$ the initial value condition is fulfilled.

However this does not make any statement for the rest of the interval we want the differential equation to be approximated since the parameters of the network have not been changed yet and are initialized randomly. To change the parameters such that the function approximates the solution of the differential equation given we need a loss function. Usually a loss function depends on training examples and the output of the network given current parameters. For example quadratic loss would be

$$\sum_{i=1}^{n} (N(x_i) - y_i)^2$$

for given training example input X with correct output (labels) Y consisitng of n elements.

For the goal of solving the equation we discretise the interval $[0, 2]$ in equidistant points, that means for example $X = \{0, 0.05, 0.1, 0.15, ..., 1.95, 2\}$. Now we can state the loss function which is

$$L(X) = \sum_{i=1}^{n} (\hat{f}^{(1)}(x_i) - (-\frac{1}{5} \hat{f}(x_i) \cdot e^{\frac{-x_i}{5}} \cdot cos(x_i))^2$$

where $\hat{f}^{(1)}$ is the derivative of the trial solution w.r.t. input x. We are able to derive the trial solution w.r.t. x by

$$\frac{\delta}{\delta x} \hat{f}(x) = \frac{\delta}{\delta x} x\, N(x) = N(x) + x\, \frac{\delta}{\delta x} N(x) = \hat{f}^{(1)}(x).$$

In fact this is just

$$\frac{\delta}{\delta x} N(x) = \sum_{i=1}^{n} \alpha_i \beta_i \sigma^{(1)}(\alpha_i\, x + \gamma_i)$$

. Therefore the whole loss function for our example is

$$\begin{aligned} L(X) &= \sum_{i=1}^{n} (\hat{f}^{(1)}(x_i) - (-\frac{1}{5}\hat{f}(x_i) \cdot e^{\frac{-x_i}{5}} \cdot cos(x_i))^2 \\ &= \sum_{i=1}^{n} (N(x_i) + x_i \, \frac{\delta}{\delta x} N(x_i) - \\ &(-\frac{1}{5} x_i \, N(x_i) \cdot e^{\frac{-x_i}{5}} \cdot cos(x_i)))^2. \end{aligned}$$

We have now established a loss function which only needs the discretised interval X as input in order for it to measure performance. It is clear that if $L(X) = 0$, then our trial solution is the same at all discretised values as the analytical solution.
Now we can use the known backpropangation algorithm ([4]) on L and just derive w.r.t. parameters and change them accordingly.
For this ADAM was used with 15,000 iterations and step size 0.001. Below you can see the result of the above example. To better compare the performance we compare it to the analytical solution of the differential equation, which is

$$f(x) = e^{-\frac{x}{5}} \cos(x).$$

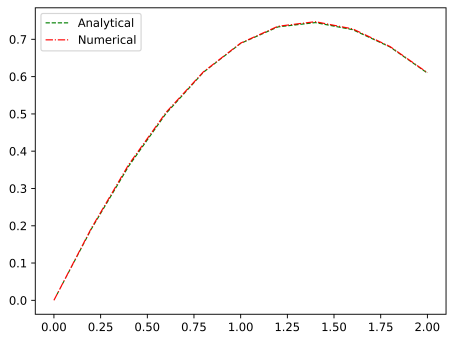

Fig. 6. Both analytical and approximative solution plotted on the interval the equation was solved

To further discuss the results, next the absolute difference is given. That is $f(x) - \hat{f}(x)$, where f again is the analytical solution.

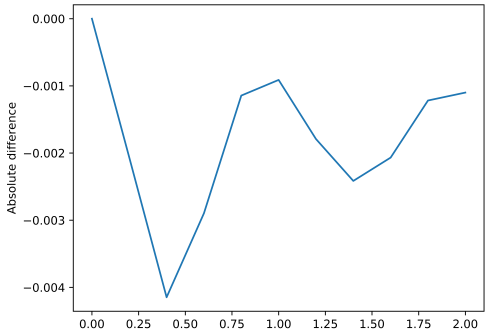

Fig. 7. Absolute error of the method compared to the analytical solution

Considering that the implementation was rather naive, the error is reasonably low. Another advantage of this method is that the solution is continuous, that means it can be evaluated at any point on the interval as opposed to other methods to solve differential equations like the popular Runge-Kutta method ([5]).

VI. Conclusion

To sum up we have presented a strong method which allows us to solve a differential equation numerically. In theory this method could be expanded to higher order differential equations since $N(x)$ is differentiable just as many times as the activation function is, therefore the loss function needs to be changed but no further changes would be needed. The same goes for partial differential equations, for example functions mapping $\mathbb{R}^2$ to $\mathbb{R}$ would for example need a trial solution which includes a neural network of size $[2, n, 1]$ and the loss function gets more complex but again no further changes are needed. A disadvantage of this method is that the loss function needs to be derived analytically that means the trial solution needs to be derived analytically w.r.t. input x and then implemented. We have not found any libraries which are able to easily do that so a changed loss function and update rules need to be implemented for every given differential equation which makes it rather unsuitable if one wants to quickly solve complex differential equations. On the other hand the solution is continuous, so once solved we can evaluate at any arbitrary point inside (theoretically also outside of the interval) without solving again which would for example be the case with Runge-Kutta.

References

[1] M. Stinchcombe and H. White: "Multilayer Feedforward Networks are Universal Approximators", 1988
[2] M. Baymani, A. Kerayechian, S. Effati: "Artificial Neural Networks Approach for Solving Stokes Problem", Applied Mathematics 2010
[3] M. Nielsen: "Neural Networks and Deep Learning", 2019
[4] D. E. Rumelhart, G. E. Hinton, R. J. Williams: "Learning representations by back-propagating errors", NATURE, 323 (1986), pp. 533– 536.
[5] C. Runge: "Über die numerische Auflösung von Differentialgleichungen, Mathematische Annalen" 46 (1895), p. 167–178.