Machine Learning Engineer Nanodegree

Capstone Proposal

Andre Weiner
December 21st, 2017

Proposal

Domain Background

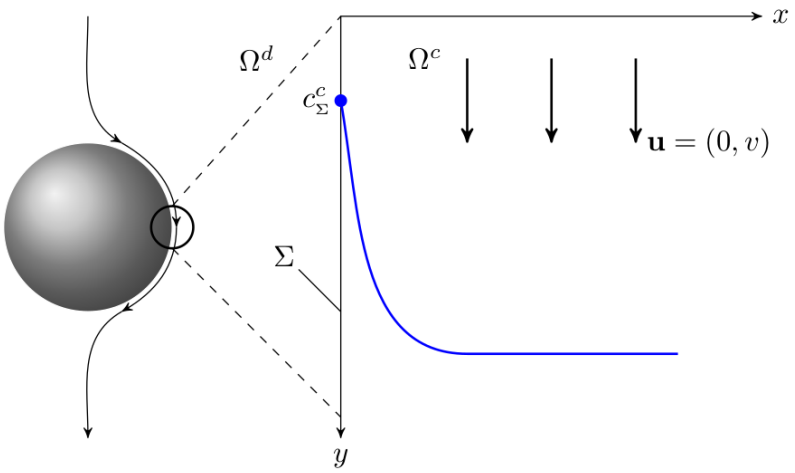
My aim in the project is to learn the solution of a partial differential equation using feed-forward neural networks. Weather forecast, the spread of diseases, the growth of trees, or the anti-lock braking system in cars: many mathematical models to describe physical, chemical, or biological phenomena lead to differential equations accompanied by boundary and initial conditions. To construct an analytical solution for such a boundary or initial value problem is unfortunately extremely hard, except for some simplified examples. The most common approach is therefore to calculate approximate solutions by means of numerical methods (e.g. finite element/difference/volume methods). Numerical methods are computationally expensive and the approximate solution is only known at several control points in the domain, but not in a closed form (like a function). Many algorithms, e.g. optimization, require the derivative of the solution with respect to its input variables. Such derivatives or integrals can not be easily obtained using numerical methods. Multilayer perceptrons are an alternative way to generate an approximate solution for differential equations. Because of their excellent approximation properties they can learn any non-linear function, provided that the network has sufficient neurons (universal approximation theorem). One idea is to generate training data for the neural network based on numerical methods, or, in a test problem, from a preexisting analytical solution. The trained network could then be used for an advanced analysis of the solution, and also to compress the massive amount of data that numerical methods usually produce without

loosing valuable information. One only has to store the network weights instead of the computational grid and the numerical solution in each grid point. But there is also another neural network based approach which entirely omits a numerical solution. During the *introduction to deeplearning* section of the course, I understood that back propagation algorithms rely on calculating partial derivatives of the network with respect to the weights. So I thought it should be also possible to calculate the derivative with respect to the input variables and to replace the corresponding terms in the differential equation such that the numerical solution can be avoided entirely. After a quick literature review, I found plenty of research based on this idea, dating back as far as 1998 [1]. I also found a code example [3], which demonstrates the basic steps for some of the example problems in [1,2].

[1] I. E. Lagaris, A. Likas, D. I. Fotiadis: *Artificial Neural Networks for Solving Ordinary and Partial Differential Equations*, IEEE Transactions on Neural Networks, Vol. 9, 1998
[2] M. M. Chiaramonte, M. Kiener: *Solving Differential Equations using Neural Networks*, Online Resource, accessed Dec 20*th*, 2017
[3] A. Honchar: *Using Neural Networks for solving Differential Equations*, Online Resource, accessed Dec 20*th*, 2017

Problem Statement

The specific problem I want to look at is the following boundary layer problem. It describes in a simplified way how a layer of dissolved gas is forming around a bubble rising in a stagnant fluid.



The setup leads to the following partial differential equation and boundary/initial conditions for

a substance c (the dissolved gas):

$$v\partial_y c = D\partial_{xx} c$$

$$c(x=0, y>0) = c_\Sigma; \quad c(x>0, y=0) = c_\infty; \quad c(x\to\infty, y>0) = c_\infty$$

with the velocity v, and the molecular diffusivity D. The symbol ∂_i preceding the function c denotes the partial derivative of c with respect to i (i being x or y in the example). A repeated lower index denotes a higher order derivative, e.g. ∂_{xx} stands for the second derivative of c with respect to x. A solution in terms of the error function can be obtained by techniques like Laplace transformation or similarity analysis:

$$c(x,y) = c_\Sigma + (c_\infty - c_\Sigma)\,\mathrm{Erf}\left(\frac{x}{\sqrt{4Dy/v}}\right)$$

Datasets and Inputs

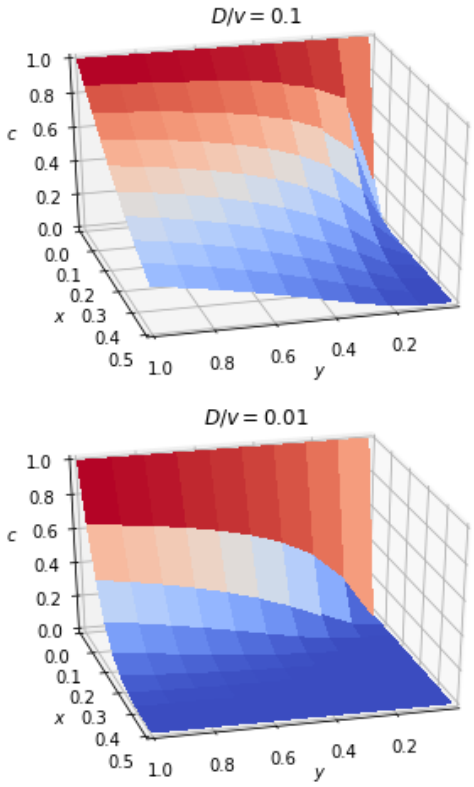
For the first approach I prepared a Jupyter-notebook to create datasets from the analytical solution given in the previous section. This emulates the data that would be given as output from a numerical simulation. I chose this approach to focus more on the machine learning part rather than the numerics. The Python script I used contains mainly four functions:

- *analytical_solution*: computes the analytical solution for a set of points (x, y) and the ratio of molecular diffusivity and velocity D/v
- *write_solution*: writes points (x, y) and the corresponding concentration value to the file *numerical_solution.csv*
- *read_solution*: reads a file formatted as *numerical_solution.csv*
- *visualize_data*: creates a 3D plot of the concentration field

The *numerical_solution.csv* currently has the following format:

x-position	y-position	concentration
0.2	0.4	3.245
...		

As a first test I created a dataset for 10×10 collocation points. The file *numerical_solution.csv* contains therefore 100 input vectors (x, y) with their corresponding labels c. Yet, this is only a dataset created for the initial training. To make the model more complex and the solution more flexible, the ratio D/v can be added to the input vector: $(x, y, D/v)$. The goal would then be to predict the concentration field for any given ration D/v in the prescribed domain defined by the points (x, y). Because the Error function is highly non-linear, a rather small change of D/v can create a significantly different solution, as the following two figures demonstrate.



The second procedure does not require any explicit input or training data. Instead, control points, sometimes called collocation points, are placed in the solution domain. The sum of the squared error in all control points is then minimized.

Solution Statement

In a differential equation a function is only known as a combination of the function itself and its derivatives with respect to the independent variables. In a numerical method the solution domain is split into many smaller sub-domains (a mesh or grid) for which the derivatives are

replaced by finite approximations. The more of such sub-domaines are used, the better the numerical approximation will be. But for the additional accuracy one has to pay with computational effort and storage.

A multilayer perceptron is nothing but a weighted sum of all inputs, which is then given as input for some non-linearity, e.g a logistic function. The network equation can be arbitrarily often differentiated, because the logistic function contains the exponential function. The result will be some combination of the weights and the non-linear function.

For the first approach the network weights will be adopted by backpropagation such that the numerical solution in the grid points can be reproduced as good as possible. The network is so to say the approximation of an approximation. In the second approach the problem solution is constructed from so-called trial functions, which fulfill initial and boundary conditions, and the neural network. The mainly first and second order derivatives of the original differential equation can then be calculated and replaced, forming an unconstrained optimization problem. Practically this means to calculate the weights such that the deviation from the exact solution becomes minimal. In the capstone report, I will explain the procedure more thoroughly using mathematical notation.

Benchmark Model

The benchmark model will be a very simple neural network consisting of only three layers (input layer, hidden layer, activation/output layer), with 5 nodes in the hidden layer and a logistic function as activation. This is the model used in reference [1].

Evaluation Metrics

There are several sensible metrics to evaluate the network performance:

- the sum of the squared error (the difference between reference and network solution squared, evaluated at several control points in the domain, and summed up)
- the prediction time (how long does it take to make a certain number of predictions)
- the storage (how much storage is required to save the network information, e.g. the weights)
- the sum of the squared error of the first derivative with respect to x

Only the sum of the squared error will be used to train the neural network. The other quantities are secondary metrics that I am personally interested in, especially the accuracy of the approximated first derivative.

Project Design

There are some adjustable parameters which can be used to optimize the network once there is a basic model:

- number of neurons
- number of hidden layers
- number of control points
- placement of control points
- activation functions

I have found literature reporting on the network performance using different numbers of neurons (see Rudd et al.: *A constrained backpropagation approach for the adaptive solution of partial differential equations*). Also the number of control points was varied slightly in reference [1]. The authors of [1] also suggest to vary the location of the control points in future works. This is a common strategy in numerical methods: the grid points are mainly positioned where the solution is changing the most. I believe this strategy could also work for networks, but it has not been investigated yet. A network with more than one hidden layer has not been studied in the literature I reviewed. The intended workflow is therefore the following:

1. create a baseline solution using the network described in the benchmark model section
2. derive and report the necessary equations needed to train a network with more than one hidden layer
3. vary and report the influence of the number of neurons, layers, and control points
4. try a (leaky) rectified linear unit as activation function
5. instead of uniformly distributing the control points, place them close to where the function is highly non-linear, e.g. close to $x = 0$

One of my goals is to describe and visualize the entire approach, which may seems complicated on the first sight, as simple as possible such that any student or researcher with basic analysis and machine learning knowledge can understand and replicate the idea.