Using multilayer perceptrons to approximate the solution of partial differential equations

Andre Weiner

April 8, 2018

Abstract

This report compares several approaches using multilayer perceptrons to approximate the solution of a boundary layer problem. The models differ in the way how the neural network is used in the approximate solution and also in the learning strategies. Two learning strategies have been investigated: classical, supervised learning based on training data (the numerical problem solution) and residual based learning, where partial derivatives of the network with respect to the input parameters are computed. Using the pure network and data driven learning turned out to be the easiest, fasted, and most accurate attempt. The final model reaches a maximum deviation of about 4% on the validation set. The storage requirement of the final model is more than 60 times lower compared the numerical solution.

1 Introduction

The report explains based on an example how the multilayer perceptron (MLP) algorithm can be used to approximate the solution of partial differential equation (PDE). Differential equations and their accompanying boundary or initial conditions arise from a variety of mathematical models: weather forecast, the spread of an infectious disease, the growth of a tree, or even the anti-lock braking system in a car. Constructing an analytical solution for such models is unfortunately very hard or impossible, except for some simplified examples. The most common approach in practice is therefore to calculate approximate solutions by means of numerical methods (e.g. finite difference/element/volume method). A drawback is that the numerical solutions will be only known at discrete locations within the domain, defined by a so-called mesh, and not in a closed analytical form. If one wants to evaluate the solution at points in-between these discrete locations, search algorithms and interpolations schemes are needed. With an increasing number of mesh elements the evaluation of the numerical solution and also the storage requirements become very demanding. My aim is to create models based on numerical data that are fast and easy to evaluate (no search/interpolation, accurate gradient computation) and that require a minimum of storage. The MLP algorithm seems to be a suitable base to create such models, because of its flexibility (see universal approximation theorem[1]). There are basically two approaches how a MLP could learn an approximate solution of an initial/boundary value problem:

- Supervised learning: the numerical solution can be seen as set of feature/label pairs. The loss function is defined such that the difference between the MLP evaluated at the discrete mesh locations, and the numerical solution, calculated beforehand, becomes minimal.

[1] `https://en.wikipedia.org/wiki/Universal_approximation_theorem`

- Partial derivatives of the MLP: because the MLP is always a continuous, smooth function, its gradient with respect to the input features can be computed. The loss function is then defined such that the residual of the approximated partial derivatives forming the differential equation becomes minimal.

The two approaches are described in more details in section 4 and tested against a relatively simply boundary layer problem, described hereafter.

2 Problem description

The test problem is one that could be used to describe the dissolution of a bubble rising in a stagnant liquid. The gas is first transferred over the gas-liquid interface (it dissolves) and then transported along the interface due to the liquid motion. The dissolution happens at a rate proportional to the gradient of the dissolved gas at the interface. This transport process is usually called diffusion. Because of buoyancy forces due to the different densities of gas and liquid, the bubbles rises with a relatively high velocity (an air bubbles of $2mm$ size rises with approximately $30cm/s$). The strong relative motion between bubble and surrounding liquid quickly transports the dissolved gas away from the interface. This transport process is usually called advection. In the case of a rising bubble diffusion happens mainly normal and advection tangential to the interface. Since advection is typically much stronger than diffusion, the dissolved gas is distributed in long, thin structures around the interface, so-called boundary layers. Such a scenario may be described by a simplified set-up, sketched in figure 1.

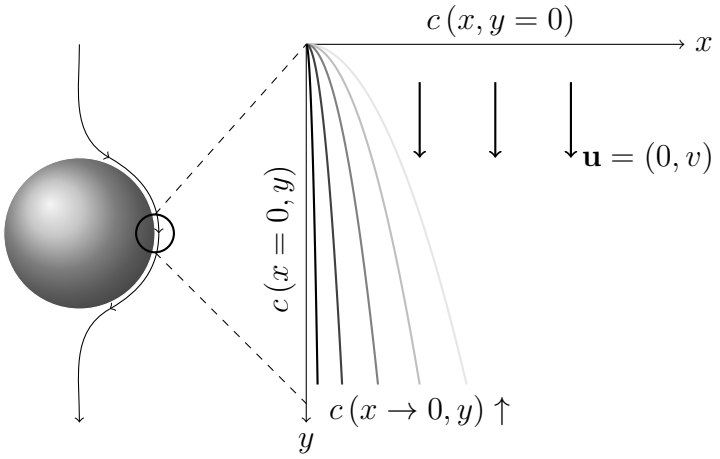

Figure 1: Sketch of the boundary value problem to model the dissolution of a rising bubble.

The transport of a diluted, dissolved gas c in a steady state scenario may be described by the following partial differential equation:

$$\underbrace{v\frac{\partial c}{\partial y}}_{\text{advection}} = \underbrace{D\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}\right)}_{\text{diffusion}} \quad . \tag{1}$$

An intuitive explanation of equation 1 is that at steady state advection with the constant velocity v and diffusion with a molecular diffusivity D are in equilibrium (hence '='). In this notation $\partial c/\partial y$ stands for the derivative of c with respect to coordinate y, $\partial^2 c/\partial x^2$ expresses the second derivative of c with respect to x, and so on. Equation 1 is considered in a domain $x \in [0, 1]$ and $y \in [0, 1]$. To complete the problem, sensible boundary conditions have to be defined. The original idea, which can be found in many textbooks on heat and mass transfer,

was to assume constant values of 1 and 0 at the boundaries $x = 0$ and $y = 0$, respectively. For such a case an analytical reference solution can be obtained. Unfortunately, the solution has a singularity in the point $(0, 0)$, because the boundary conditions in that point jump from 0 ($x \to 0$) to 1 ($y \to 0$). Singularities will always cause problems in numerical or machine learning algorithms, which is why I modified the original problem slightly.

$$c(x = 0, y) = \frac{2}{1 + \exp(-\beta y)} - 1 \quad \text{with} \quad \beta > 0, \qquad c(x = 1, y) = 0, \tag{2}$$

$$c(x, y = 0) = 0, \qquad \partial c/\partial y(x, y = 1) = 0. \tag{3}$$

The expression for $c(x = 0, y)$ is a modified sigmoid function that varies between 0 for $y = 0$ and 1 for $y \to \infty$. The constant β is an adjustable parameter which controls the steepness of the transition between minimum and maximum. As $\beta \to \infty$, the original problem is recovered. A less abstract, visual confirmation of this characteristics will be given in the next section.

3 Numerical solution and data sets

The numerical solution of the boundary layer problem described in the previous section was obtained using a standard finite volume method. Without going too much into detail, one can imagine that the non-linear function $c(x, y)$ is approximated by many small, linear segments. The smaller these segments are, the more accurate the approximate solution will be. A common approach is to compute the numerical solution on several meshes of different size, and to compare how much the solution changes. Two of the computed solutions are depicted in figure 2.

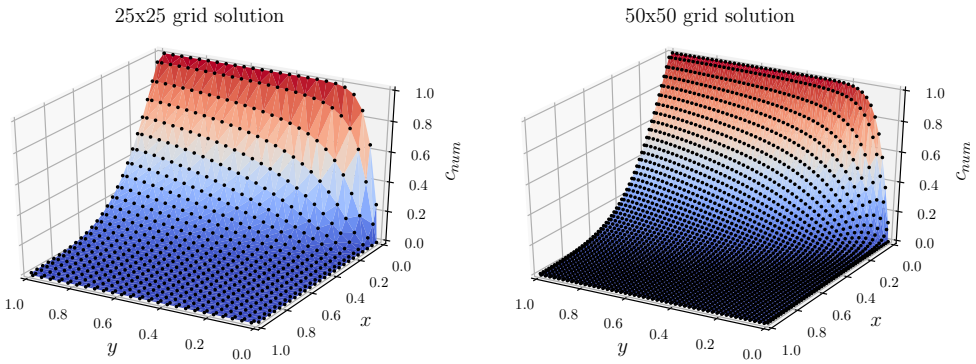

Figure 2: Numerical solution $c_{num}(x, y)$ for two mesh resolutions. The solution is only known for discrete locations marked as black dots.

In general for each mesh the numerical solution is known at different discrete locations. Instead of trying to interpolate between different meshes, it is more common to compare different integral quantities. I use here the average gradient normal to the interface

$$Sh = \frac{1}{y_{max} - y_{min}} \int_{y_{min}}^{y_{max}} \left. \frac{\partial c}{\partial x} \right|_{x=0} \mathrm{d}y, \tag{4}$$

and the average norm of the gradient in the entire domain

$$\varphi = \frac{1}{(x_{max} - x_{min})(y_{max} - y_{min})} \int_{x_{min}}^{x_{max}} \int_{y_{min}}^{y_{max}} \|\nabla c\| \, \mathrm{d}y \mathrm{d}x, \tag{5}$$

which are suitable to measure mesh dependency. The results in table 1 show that Sh is slightly more sensitive to the resolution than φ. Between the coarsest and the finest mesh Sh increases by about 5%. Instead, the difference between the two highest resolutions is much less than 1%. Therefore it is reasonable to assume that the computed solution is mesh independent.

	25x25	50x50	100x100	200x200	400x400
Sh	4.9901	5.1169	5.1766	5.2049	5.2185
φ	1.0024	1.0020	1.0026	1.0027	1.0027

Table 1: Global Sherwood number Sh and gradient norm φ for different mesh resolutions.

Based on this study I decided to create four data sets:

1. **baseline training data**: numerical solution computed on the 50x50 mesh, resulting in $2500 + 200$ feature-label pairs used for baseline model training.
2. **baseline validation data**: numerical solution computed on the 100x100 mesh, resulting in $10^4 + 400$ feature-label pairs used for baseline model validation.
3. **training data**: numerical solution computed on the 200x200 mesh, resulting in $4{\cdot}10^4{+}800$ feature-label pairs used for model training.
4. **validation data**: numerical solution computed on the 400x400 mesh, resulting in $1.6 \cdot 10^5 + 1600$ feature-label pairs used for model validation.

The first number in listing 3 indicates the discrete points within the domain, and the second number tells how many additional points are exactly located on the domain boundary. Having large data sets is beneficial to reach higher accuracy, but at the same time the computational effort per training epoch increases, which is why I used smaller data sets for preliminary tests with the baseline architecture. The numerical solution is stored in a simple row-column structure 2, which can be read using *Pandas* csv-reader.

x	y	c_{num}
0.1	0.1	0.33
0.1	0.2	0.41
...	...	...

Table 2: Example of the data format in which the numerical solution is stored.

4 Approximating the solution of a differential equation

4.1 Multilayer perceptron (MLP) algorithm

To compress the numerical solution into a closed function I used the basic multilayer perceptron algorithm as sketched in figure 3. The MLP network $N(x, y, \mathbf{p})$ has only two input perceptrons, one for each coordinate, and one output unit. Between input and output one or more hidden layers with a given number of perceptrons are used with adjustable weights and biases $\mathbf{p}$ in order to minimize some loss function $E(\mathbf{p})$. One restriction is placed on the network architecture: each hidden layer contains an equal number of units. All units employ a sigmoid function activation except for in- and output, where the activation is linear.

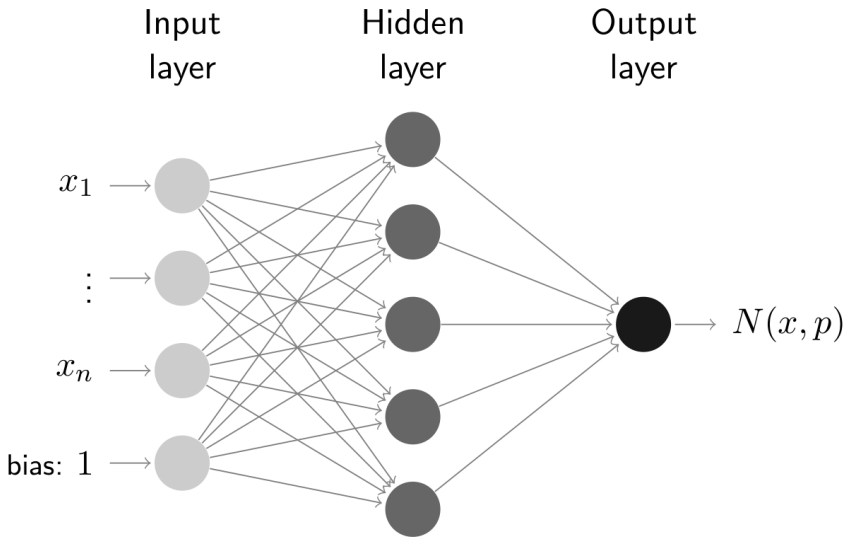

Figure 3: Conceptual sketch of a MLP with only one hidden layer, copied from [1].

There are several approaches of how to use the network to approximate the target function (the exact problem solution). Hence, the network output can have different meanings, which are sometimes more and sometimes less intuitive. The first approach presented hereafter, in contrast to classical supervised learning, does not even require training data, but only control points where derivatives of the network are evaluated. Each approach has pros and cons which are discussed in the next sections.

4.2 Trial function approach

The idea of the authors of [4] was to express the approximate solution as a trial function $c_t(x, y, \mathbf{p})$ which fulfils by its construction all initial and boundary conditions. The trial function contains the feedforward network $N(x, y, \mathbf{p})$ in such a way that the network parameters $\mathbf{p}$ can be adjusted to learn the solution $c(x, y)$ inside the domain $x, y \in (0, 1)$ without effecting boundary or initial conditions. This makes the minimization problem (finding the best network parameters) unconstrained, and therefore presumably easier to solve. The trial function used in the remaining part of the work is equation 6.

$$\begin{aligned} c_t\left(x, y, \mathbf{p}\right) = &\left(1 - x\right) c\left(x = 0, y\right) \\ &+ x\left(1 - x\right) y \left[N\left(x, y, \mathbf{p}\right) - N\left(x, y = 1, \mathbf{p}\right) - \frac{\partial N\left(x, y = 1, \mathbf{p}\right)}{\partial y}\right] \end{aligned} \tag{6}$$

It is relatively easy to see that $c_t\left(x = 0, y\right) = c(x = 0, y)$, $c_t(x = 1, y) = 0$, and $c_t(x, y = 0) = 0$. To prove that the gradient with respect to y vanishes at the boundary $y = 1$ one has to calculate $\partial c_t / \partial y$. A visual confirmation is given in figure 4.

Two possible ways come to mind for formulating a loss function which can be minimized to obtain optimal network parameters. The first one complies with classical supervised learning. The loss is defined as the sum of the squared difference between numerical solution c_{num} and trial function c_t over all grid points N_p.

$$E_{tr,data}(\mathbf{p}) = \sum_{i=1}^{N_p} \left[c_{num}(x_i, y_i) - c_t(x_i, y_i, \mathbf{p})\right]^2 \tag{7}$$

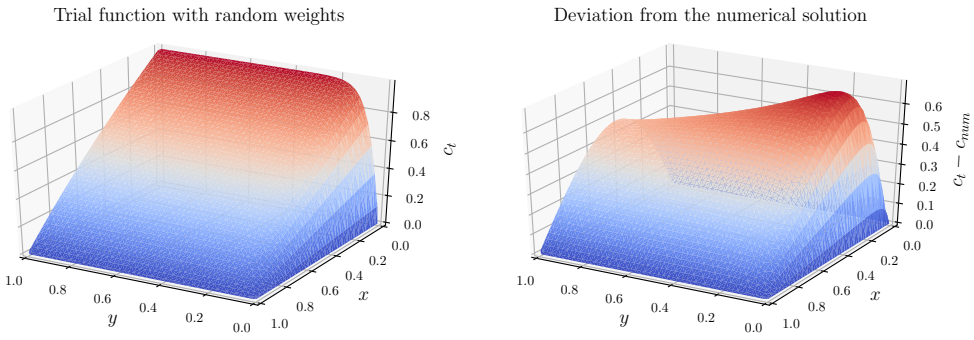

Figure 4: Trial function $c_t(x, y, \mathbf{p})$ with randomly initialized weights.

Another possibility is to evaluate the partial derivatives at certain control points, since equation 1 holds true for the entire domain. This approach was already used by many authors in the field of machine learning [1, 4, 5], but was almost never applied in the field of computational physics. The residual based loss function reads

$$E_{tr,res}(\mathbf{p}) = \sum_{i=1}^{N_c} \left[\left(v \frac{\partial c_t(x, y, \mathbf{p})}{\partial y} - D \left(\frac{\partial^2 c_t(x, y, \mathbf{p})}{\partial x^2} + \frac{\partial^2 c_t(x, y, \mathbf{p})}{\partial y^2} \right) \right)_{x_i, y_i} \right]^2 \tag{8}$$

where N_c is the number control points, and $(...)_{x_i,y_i}$ indicates that the term in parenthesis is evaluated at the point (x_i, y_i). Again, there are several ways how to evaluate the partial derivatives in equation 8: numerical, automatic, and symbolic differentiation. Numerical differentiation I found to be very unstable, which is why I excluded it soon after beginning this project. Automatic differentiation in principal works fine but is rather slow, because many higher order derivatives of the network with respect to the input parameters have to be calculated. On the other hand side, symbolic differentiation of a MLP with more than one hidden layer is too demanding and error prone to be done by hand. I implemented and tested a fully symbolic procedure for a simple three layer MLP, which ran and converged much faster than with automatic differentiation. Yet, to approximate non-linear functions effectively, deeper networks are necessary, and the symbolic calculation is not an option any longer. The compromise I finally made was to compute the derivatives of equation 6 by hand, without explicitly computing the network derivatives. One then gets explicit formulas of each term in the loss function containing higher order derivatives of the network, which are evaluated by automatic differentiation. In this way, each of the necessary network derivatives only has to be calculated once. Further information is given in section 4.6 and in the accompanying source code.

4.3 Custom function approach

The trial function introduced in the previous section is rather complicated and contains the network and even derivatives with respect to the inputs several times, and hence requires several network passes for a single prediction. A much simpler function, which does not fulfil by default all boundary conditions, is the following:

$$c_t\left(x, y, \mathbf{p}\right) = c\left(x = 0, y\right) \left[1 - \mathrm{Erf}\left(\frac{x|N(x, y, \mathbf{p})|}{\sqrt{4Dy/v}} \right) \right], \tag{9}$$

where Erf is the error function. Function 9 is inspired by the analytical solution of a similar, simplified version of the problem. It has some useful properties that are interesting for later use cases: because the absolute value of the network output is taken, the argument of the error

function will be always between zero and one for $x, y > 0$, and therefore c_t will also be always between zero and one (as the exact solution must be). This can be verified visually in figure 5.

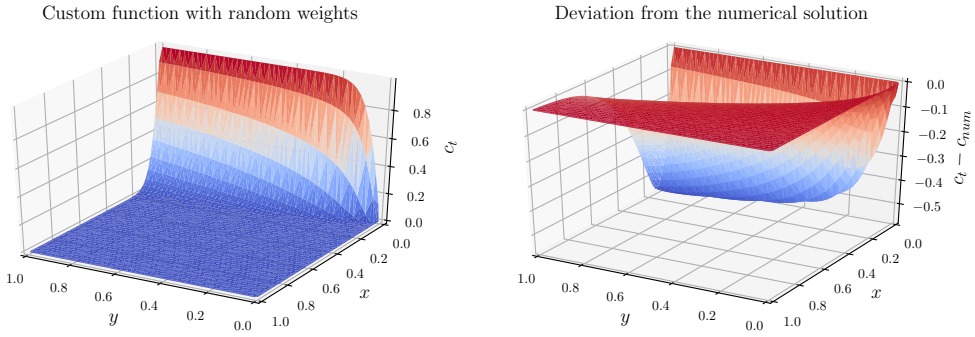

Figure 5: Custom function $c_t(x, y, \mathbf{p})$ with randomly initialized weights.

The loss function is defined as before by simply comparing numerical solution and custom function.

$$E_{cust}(\mathbf{p}) = \sum_{i=1}^{N_p} \left[c_{num}(x_i, y_i) - c_t(x_i, y_i, \mathbf{p})\right]^2 \tag{10}$$

4.4 Pure network approach

The easiest approach is to use a pure MLP without any particular measures for boundary conditions.

$$c_t\left(x, y, \mathbf{p}\right) = N(x, y, \mathbf{p}) \tag{11}$$

Such an approach is more general than the ones introduced before but also requires a stronger adaptation by the MLP itself, because no guidance regarding the general shape of the function is prescribed. One way to put more emphasis during the network training on boundary points is to introduce a penalty term $f(x, y)$ in the loss function

$$E_{net,p}(\mathbf{p}) = \sum_{i=1}^{N_p} \left[\{c_{num}(x_i, y_i) - c_t(x_i, y_i, \mathbf{p})\} \cdot f(x_i, y_i)\right]^2 \tag{12}$$

with the penalty function being, for example,

$$f(x, y) = \begin{cases} p & \text{if } x = 0 \text{ or } y = 0 \\ 1 & \text{otherwise} \end{cases} \tag{13}$$

where p is some real value larger than one.

4.5 Solving the minimization problem

In order to minimize the loss functions introduced in the previous sections I tried plain backpropagation, the BFGS[2] algorithm, and the recently presented ADAM algorithm [3]. Pure backpropagation in all cases converged really slowly or got stuck in local minima. For the BFGS algorithm I used the SciPy[3] implementation. Other authors report that they got good

[2] Broyden-Fletcher-Goldfarb-Shanno

[3] `https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html`

results using the quasi Newton BFGS algorithm for similar problems, e.g. learning the solution of a PDE [1, 4]. The version I tested converged too often to local minima, which might be a problem of the implementation or the different loss function compared to previous studies. The best working algorithm, which was also easy to implement, was ADAM, which is why I used it for all tests reported hereafter. The loss function derivatives with respect to the weights and also the network derivatives with respect to the input parameters were computed using Autograd[4].

4.6 Notes on the implementation

Core of the Python code are two classes contained in *pdeUtils.py*:

- **IBVProblem** (**I**nitial **B**oundary **V**alue **P**roblem): implements trial function, loss function, initialization, training, prediction, and error norms.
- **SimpleMLP**: implements MLP architecture, weight initialization, activation, and read/write operations for weights.

The *IBVProblem* class is abstract, because it has two abstract methods which have to be implemented in every derived class: *loss_function_kernel* and *trial_function*. The following classes implement loss, trial, and complementary functions described in the previous sections:

- **TrialFunctionDataBasedLearning** - section 4.2
- **TrialFunctionResidualBasedLearning** - section 4.2
- **CustomFunctionLearning** - section 4.3
- **PureNetworkLearning** - section 4.4

The source code also contains a manual implementation of the error function, since it is not yet present in the Autograd library. As a reference and also to test a larger range of network architectures I implemented a similar MLP using Keras and Tensorflow. Furthermore, a function to visualize the trial function and its deviation from the numerical data as surface plots is implemented in *visuals.py*.

All tests comparing different approaches, implementations, and architectures are contained in two Jupyter notebooks:

- capstone_baseline.ipynb - tests with the baseline model (architecture)
- network_arch_test.ipynb - different network architectures (only Keras implementation)

I also want to mention that the blog post and the accompanying Git repository by A. Honchar [2] were really helpful to get started with the implementation of my project.

[4] `https://github.com/HIPS/autograd`

5 Results

5.1 Baseline model

5.1.1 Results after 100 training epochs

The baseline model is defined as a pure MLP with only one hidden layer containing five units (perceptrons). As a first test evaluate the general performance of the different approaches I run 100 training epochs and compared execution time and residuals 3. Because there is no meaning in comparing the absolute values of the losses, instead I checked how much the loss was reduced compared to the initial loss. All networks were initialized using the same seed value in the random number generator.

approach	$E_{tr,data}$	$E_{tr,res}$	E_{net}	$E_{net,p=2}$	E_{cust}
time in *min*	13.37	104.50	2.98	2.90	5.24
normalized time	4.61	36.0	1.03	1.00	1.81
$(E_{max} - E_{min})/E_{max}$	0.66	0.14	0.98	0.97	0.98

Table 3: Elapsed time and relative drop of E after 100 training epochs on the 50x50 training set. The residual based method used a comparable number of control points to evaluate the partial derivatives. The minimum time was used for normalization.

Both approaches based on a trial function, figures 6 and 7, lead to similar results, but the data based approach is roughly 9 times faster. Also the adaptation of the network parameters seems to be very slow for the residual driven approach. Using a trial function requires more passes through the network and is therefore much slower than all other variants.

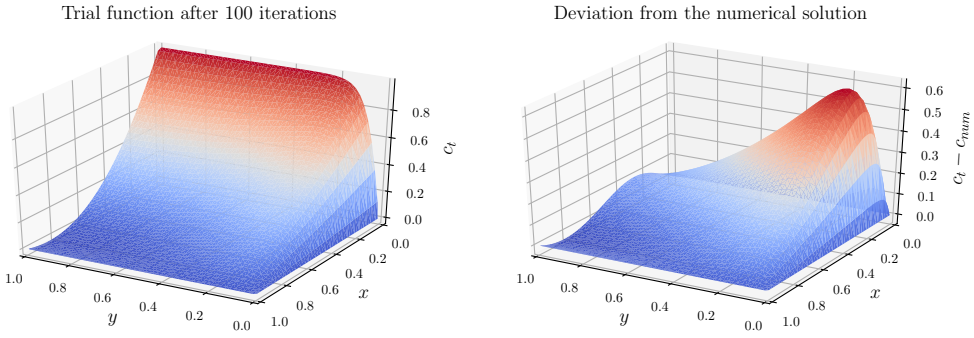

Figure 6: Trial function after 100 iterations using the 50x50 training set.

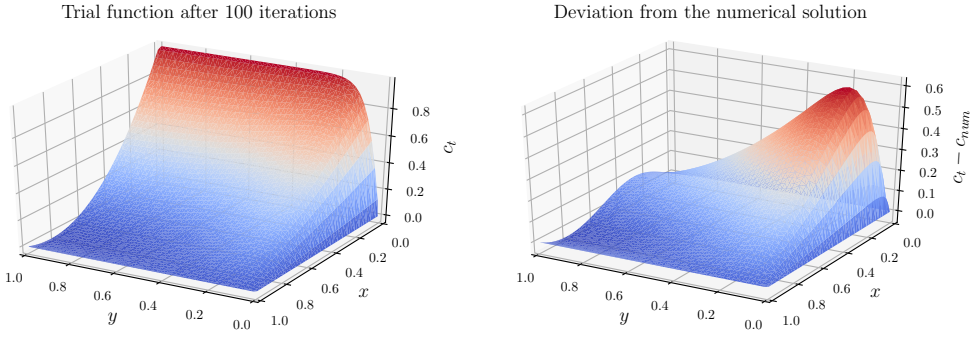

Figure 7: Trial function after 100 iterations using the residuals at 2400 control points.

The custom function in plot 8 has a maximum deviation of about 10% after training compared to the numerical solution. Computational effort and run time are somewhere between the trial function and the pure network approach.

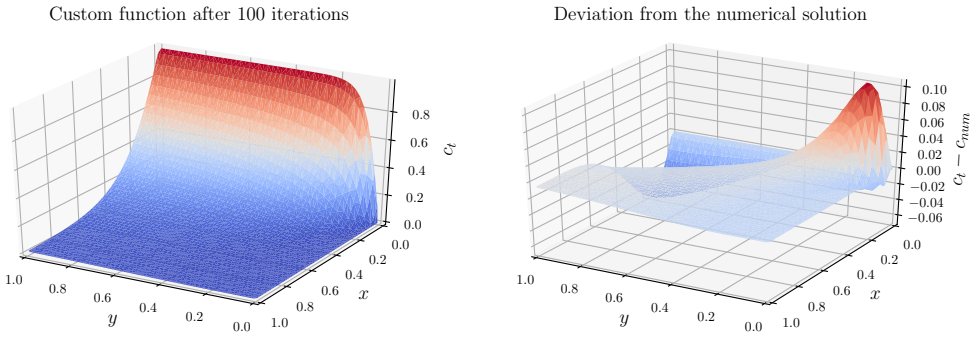

Figure 8: Custom function after 100 iterations using the 50x50 training set.

The pure network approach was tested with (figure 10) and without (figure 9) penalty term. After 100 iterations the network output does not have much in common with the numerical data. This is of course, because no prior shape information is given. A positive aspect is that the approach is the fasted one and that the adaptation of the parameters happens quickly (the loss reduces quickly). The additional penalty term seems to accelerate the learning process further. The maximum and average deviation after 100 iterations is already significantly lower.

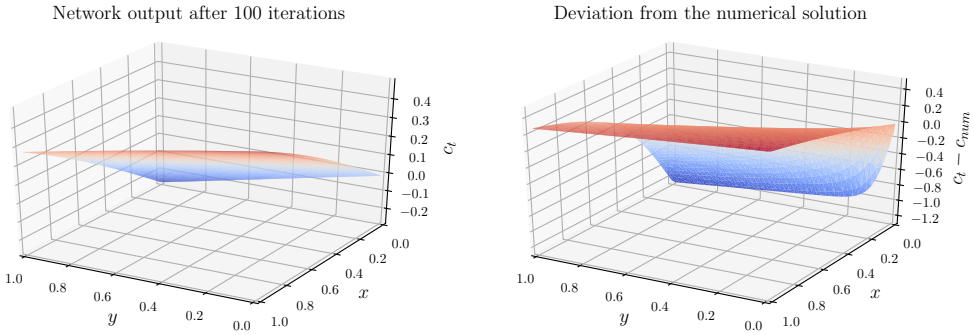

Figure 9: Network output after 100 iterations using the 50x50 training set.

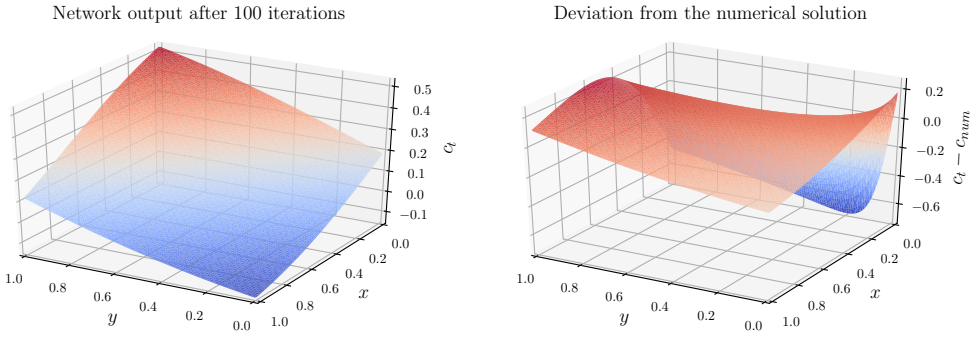

Figure 10: Network output after 100 iterations using the 50x50 training set with a penalty factor of 2 for points on the boundaries.

5.1.2 Results with adaptive learning

Using the full data set for model training leads to large run times that are net very practical. It is known that the learning process can be accelerated significantly by computing the loss only on a smaller subset of the data. This leads to more frequent updates of the network parameters and also helps in some cases to overcome local minima. A negative aspect is that sometimes, when the network is close to the minimum, training on a subset prevents further loss reduction, because the approximation of the loss function is worse than on the full data set. I implemented a hybrid strategy, which is very similar to stochastic or mini-batch training. The algorithm works as follows:

1. select random subset of the training data using every N_{tr}th point
2. compute loss function gradient and update weights using ADAM
3. compute loss function on subset and compare to prescribed tolerance
   (a) if loss is below tolerance: increase N_{tr} and go to 1.
   (b) if loss is above tolerance: go to 2.

A comparison of adaptive and non-adaptive training after 100 and 1000 training epochs is given in table 4. Adaptive training reduces the execution time by at least on order of magnitude and even leads to lower error norms. Two error norms are used for comparison, one expressing the integral error and one expressing the absolute deviation.

$$L_2 = \frac{1}{N_p} \sum_{i=1}^{N_p} \left[c_{num}(x_i, y_i) - c_t(x_i, y_i) \right]^2 \tag{14}$$

$$L_{max} = \max \left(|c_{num}(x_i, y_i) - c_t(x_i, y_i)| \right) \quad \forall \; x_i, y_i \tag{15}$$

	L_2 norm	L_{max} norm	training time in min
non-adaptive, 100 iter.	$4.9071 \cdot 10^{-4}$	0.1118	5.24
adaptive, 100 iter.	$4.1564 \cdot 10^{-4}$	0.1044	0.02
non-adaptive, 1000 iter.	$1.8735 \cdot 10^{-4}$	0.0851	53.79
adaptive, 1000 iter.	$1.3314 \cdot 10^{-4}$	0.0536	2.62

Table 4: Comparison of error norms and training time with adaptive and full training set size.

5.1.3 Final evaluation of the baseline model

For the final evaluation of the baseline model I only used the custom function approach, pure network learning, and a Keras implementation of pure network learning. The trial function approach was dropped because of to high computational costs with little return in terms of accuracy. One can understand why the trial function approach does not work very well by asking the following question: How would a network output $N(x, y, \mathbf{p})$ have to look like in order to get perfect agreement with the numerical solution? This could in theory be done by reordering equation 6 to get $N(x, y, \mathbf{p})$ as a function of $c(x, y)$. Practically it is simpler to ask, how the network dependent term of equation 6 has to look like:

$$[...] = \frac{c_t\left(x, y, \mathbf{p}\right) - \left(1 - x\right) c\left(x = 0, y\right)}{x\left(1 - x\right) y}. \tag{16}$$

Equation 16 makes immediately clear, that the network output must be very large if $x \to 0$ or $y \to 0$ in order the correct the internal field (the denominator tends to zero and therefore the term tends to infinity). Because the output of neural networks is always bounded, it is very hard to modify the internal field close to the boundaries. Other authors referenced before only investigate problems where the boundary values are zero or do not change a lot. Therefore the stated problem is much less evident.

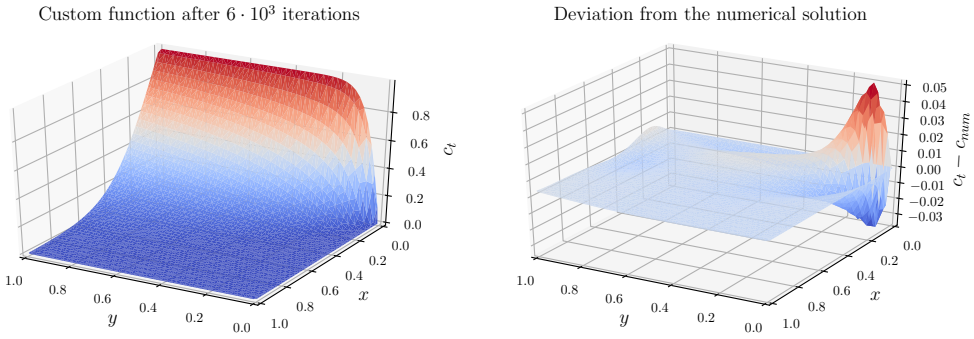

Figure 11: Custom function after 6000 iterations using the 50x50 training set.

The custom function scores the best among the three variants tested in this section (see figure 11). A very promising maximum deviation of about 5% is reached, and the custom function is visually not distinguishable from the numerical training data. Unfortunately, the custom function approach suffers from the same problem as the trial function, although it is much less evident because the function is already very similar to the exact solution, especially close to the boundaries. If an even higher accuracy needs to be obtained, also this approach will be limited.

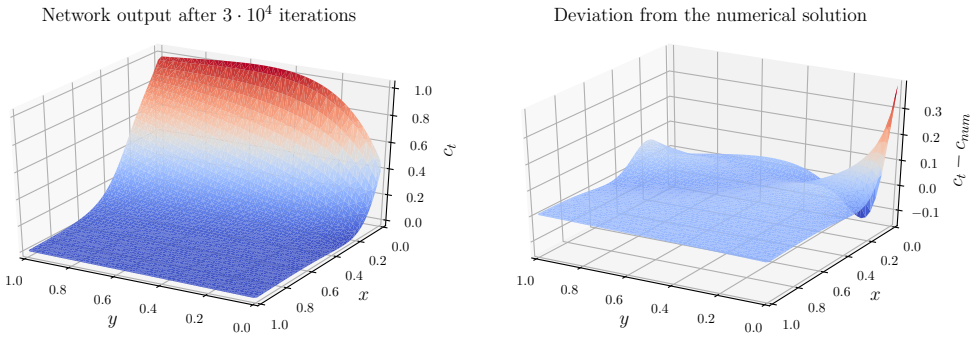

Figure 12: Network output after 10^4 iterations using the 50x50 training set and a penalty factor of 2 for points on the boundaries.

In the two versions of pure network learning the final shape looks already similar to the reference solution. The Keras network reaches a respectable maximum deviation below 10%. Note that Keras network has one additional parameter, because also the output unit has a bias term. This should, however, not make much of a difference in the final result. Another difference is the applied loss function. The Keras network uses the mean squared error whereas I used the sum of squared errors in the Python implementation. A different loss function will presumably lead to a different error distribution in the approximation. However, given a network with sufficient adjustable parameters, both variants should converge. It is notable that there is a zone around the origin $(0, 0)$ where the error is much larger than in the rest of the domain. This is probably because the function is the most non-linear in that region, in the sense that the gradients with respect to the input parameters are the largest.

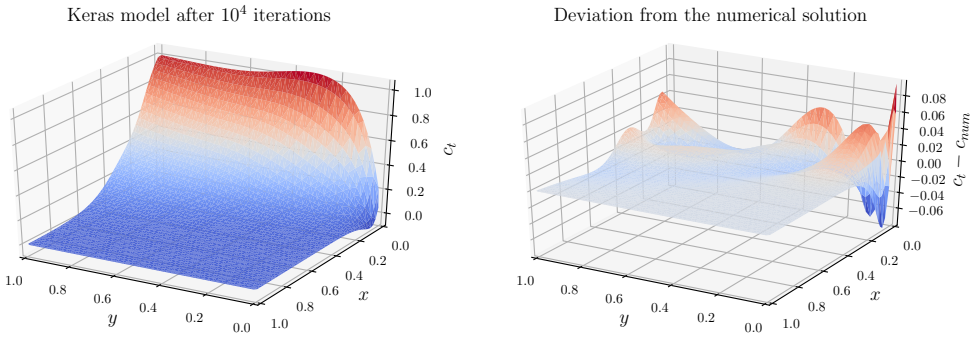

Figure 13: Pure network learning (Keras) after 10^4 iterations using the 50x50 training set.

	L_2 norm	L_{max} norm	training time in *min*
custom function, $6 \cdot 10^3$ iter.	$5.2032 \cdot 10^{-5}$	0.0469	14.42
Python MLP, $3 \cdot 10^4$ iter.	$7.3110 \cdot 10^{-3}$	0.9997	12.12
Keras MLP, $1 \cdot 10^4$ iter.	$1.0535 \cdot 10^{-4}$	0.1120	1.62

Table 5: Final error norms using different approaches and the baseline MLP architecture.

Because the Keras implementation using the Tensorflow backend is significantly faster than my Python script, I only used Keras for further tests of the network architecture presented in the next section.

5.2 Varying the network architecture

One factor that makes classical numerical methods relatively reliable and widely spread is the formal convergence order they have. The convergence order expresses, how much the error will decrease when the number of grid points is increased. For example, the method I used to obtain the reference solution is said to be of second order, meaning that when the spacing between the grid points is reduced by one half, the error will be reduced quadratically, that is one fourth. At the present time there is no such convergence theory for deep MLPs as far as I know, so there is no reason why a neural network should behave like classical numerical methods. In spite of this fact, it is important to know how the approximate solution behaves when the network architecture changes. That is why I tested 25 different architectures, varying the number of hidden layers and units per layer. Error norms computed on the 400x400 validation set are compiled in table 6.

Unfortunately, there are no clear tendencies regarding a possible correlation between the obtained accuracy and the network architecture. Comparing the changes per row against that per column of table 6, one can infer that using more layers sometimes leads to very strong changes in that the L_2 norm improves by orders of magnitude. When more perceptrons per layer are used, the changes are smaller but more predictable. This observations are already widely known, and the non-linear changes that arise from multiple hidden layers are one of the reasons for the success of deep learning. However, the deeper the network and the more free parameters, the harder it is to train. The network reaching the lowest L_{max} norm is the one with 3 hidden layers and 16 units per layer. This appears to be random and emphasises that there is probably potential to reach much higher accuracies with an advanced concept for training. Regarding the maximum deviation, with only 1% relative error the final model displayed in figure 14 scores about eight times better than the baseline model (the L_{max} norm on the 400x400 validation that is higher than on the 50x50 set used for plotting), and the overall result is probably good enough for some applications employing the approximate MLP

perceptrons → hidden layers ↓	2	4	8	16	32
1	$1.4276 \cdot 10^{-3}$ $5.7955 \cdot 10^{-1}$	$2.5719 \cdot 10^{-4}$ $1.9803 \cdot 10^{-1}$	$2.6775 \cdot 10^{-5}$ $6.2283 \cdot 10^{-2}$	$3.8700 \cdot 10^{-5}$ $6.6099 \cdot 10^{-2}$	$2.5414 \cdot 10^{-5}$ $5.3022 \cdot 10^{-2}$
2	$1.3533 \cdot 10^{-4}$ $1.9199 \cdot 10^{-1}$	$7.1601 \cdot 10^{-6}$ $7.4219 \cdot 10^{-2}$	$7.6634 \cdot 10^{-7}$ $7.7017 \cdot 10^{-3}$	$1.8481 \cdot 10^{-6}$ $2.4212 \cdot 10^{-2}$	$7.5154 \cdot 10^{-7}$ $6.4094 \cdot 10^{-3}$
3	$3.3069 \cdot 10^{-4}$ $3.4576 \cdot 10^{-1}$	$1.0910 \cdot 10^{-5}$ $9.6037 \cdot 10^{-2}$	$7.2944 \cdot 10^{-7}$ $1.1591 \cdot 10^{-2}$	$3.6242 \cdot 10^{-7}$ $\mathbf{4.7274 \cdot 10^{-3}}$	$3.1572 \cdot 10^{-7}$ $4.8715 \cdot 10^{-3}$
4	$3.6614 \cdot 10^{-4}$ $3.6090 \cdot 10^{-1}$	$6.0905 \cdot 10^{-6}$ $7.7769 \cdot 10^{-2}$	$1.0211 \cdot 10^{-6}$ $1.0682 \cdot 10^{-2}$	$1.2035 \cdot 10^{-6}$ $1.0516 \cdot 10^{-2}$	$3.5662 \cdot 10^{-7}$ $5.0301 \cdot 10^{-3}$
5	$3.5957 \cdot 10^{-4}$ $3.6053 \cdot 10^{-1}$	$8.4874 \cdot 10^{-6}$ $9.2533 \cdot 10^{-2}$	$5.5004 \cdot 10^{-7}$ $1.1405 \cdot 10^{-2}$	$9.1060 \cdot 10^{-7}$ $8.0269 \cdot 10^{-3}$	$1.8220 \cdot 10^{-6}$ $1.9991 \cdot 10^{-2}$

Table 6: L_2 and L_{max} norm for different numbers of perceptrons per hidden layer and hidden layers after 1000 training epochs. The lowest L_{max} norm is highlighted.

solution. It is also noticeable that the final model has only 609 free parameters, while the discrete training set contains more than $4 \cdot 10^4$ values (compression factor of approximately 67). When the training procedure is further improved the data reduction factor could be even higher. For example, the network with two hidden layers and eight perceptrons per layer only has 105 free parameters and reaches a similar performance as the final model displayed in figure 14 (389-fold reduction). I also varied the activation function, for example hyperbolic tangents or linear rectified units, but the sigmoid activation performed always better in the tested parameter range. Another points that needs close attention is that the loss never drops below 10^{-7}. Even though I used double precision floats, round-off errors may have compromised the solution at some unknown point.

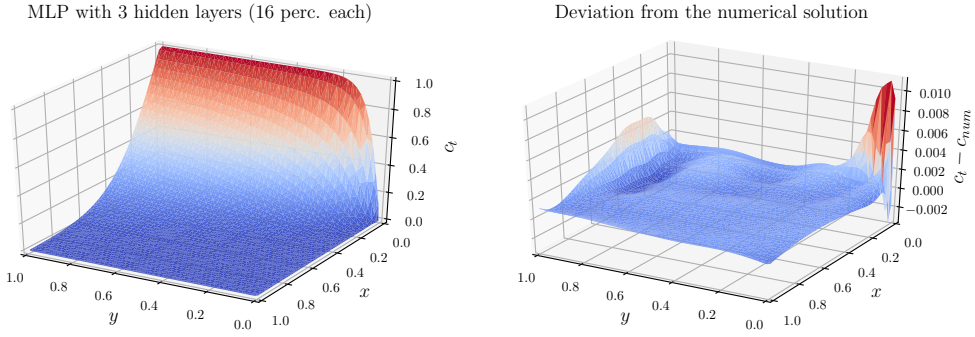

Figure 14: Pure network learning with the lowest L_{max} norm after 1000 training epochs.

6 Conclusion

In this project I tried to use MLPs with different architectures to learn a continuous function that approximates the solution of initial/boundary values problems. The training was either based on the numerical problem solution or on residuals calculated from partial derivatives of some trial function containing the neural network. I have to admit that the task was much more challenging than I expected. The residual based approach turned out to be too slow and unreliable when applied to boundary values problems that contain non-zero boundary-values.

Using the pure network was the easiest and surprisingly also the most effective method in terms of training time and final accuracy. For future studies there are a couple of attempts and tests that could further improve the learning and the applicability:

- different hidden layer techniques like convolution or memory cells
- loss function based on the partial derivatives computed using the training data
- increased parameter space, e.g. a variable ratio D/v
- coupled systems of PDEs, e.g. for reactive boundary layer problems

References

[1] M. M. Chiaramonte and M. Kiener. Solving differential equations using neural networks. http://cs229.stanford.edu/proj2013/. Accessed: 2018-03-15.

[2] A. Honchar. Neural networks for solving differential equations. goo.gl/ktfq9k. Accessed: 2018-03-15.

[3] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[4] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, Sep 1998.

[5] K. Rudd, G. Di Muro, and S. Ferrari. A constrained backpropagation approach for the adaptive solution of partial differential equations. *IEEE Transactions on Neural Networks and Learning Systems*, 25(3):571–584, 2014.