Solving Differential Equations with Machine Learning: Applications of Neural Networks and Genetic Algorithms

Sara Pernille Jensen & Håkon Olav Torvik

FYS-STK4155 – Applied Data Analysis and Machine Learning
Autumn 2021
Department of Physics
University of Oslo

December 17, 2021

Abstract: Machine learning algorithms have many applications, and were in this paper used to solve partial differential equations and non-linear differential equations. The methods used were a feed forward neural network and a genetic algorithm inspired by darwinian evolution. While the neural network gave good results when solving the diffusion equation, it was slow and considerably slower than the explicit solver tested. The genetic algorithm was able to find the solution to simple differential equations, but was unable to find it for the diffusion equation. Fundamental flaws were found in the algorithm, and ways to fix these will be discussed. Finally, the neural network was used to solve a non-linear differential equation where the solutions were eigenvectors of a symmetric matrix. This can be used to solve differential equations reformulatable as eigenvalue problems.

Contents

1 Introduction

The theoretical basis for machine learning and artificial intelligence was developed many decades ago, with the term *machine learning* being coined in 1959. However, it is only in the last decade or so where deep learning have seen widespread use, making complex data analysis possible. Advances in computing power is one of the main reasons for this, but also refining of the statistical methods have been important in yielding good results. Today, neural networks form an integral part of many algorithms, and is often able to solve both new problems and old problems more efficiently. In previous work [2] [3], feed forward neural networks were successfully used to fit both continuous data and fo classification. Here, the aim will be use machine learning to solve a partial differential equation.

Partial differential equations with many independent variables are notoriously hard to solve analytically. For example, the Navier-Stokes equations, describing the motion of viscous fluids, is a famous equation with no known exact solution. Numerous finite difference methods exist for solving such difficult equations numerically, and in recent years, different machine learning algorithms have also been employed for this purpose. In this paper, the one-dimensional diffusion equation will be attempted solved using a feed forward neural network and a genetic algorithm. For comparison of accuracy and efficiency, it will also be solved using the more traditional forward Euler scheme. The analytic solution will also be given.

Further, Yi et. al. showed that the eigenvalues of a symmetric matrix can be described by a differential equation, solvable by a neural network [7]. Inspired by their work, a similar method was here used, displaying an application of neural network-differential equation solvers.

First, the theory behind the forward Euler method, neural networks and genetic algorithms will be outlined, as well as the methods for using these to solve partial differential equations. The method for using neural network for finding the eigenvalues of symmetric matrices will also be given. The three methods were then used to solve the diffusion equation, the results of which will be presented and compared. Both the explicit scheme and the neural network gave satisfactory results, whereas substantial flaws were uncovered in the genetic algorithm, which will be discussed.

2 Theory

In its most general form, a PDE can be written as:

$$f(\mathbf{x}, g(\mathbf{x}), g'(\mathbf{x}), g''(\mathbf{x}), \ldots, g^{(n)}(\mathbf{x})) = 0, \tag{2.1}$$

where $\mathbf{x} = x_1, x_2, \ldots x_m$ are the independent variables, and $g(\mathbf{x})$ the solution of the PDE, with the derivatives containing a term for each of the independent variables.

The temperature gradient $u(x,t)$ in a rod of length $L = 1$ can be described by the one-dimensional diffusion equation, a partial differential equation with two independent variables, on the form

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, \tag{2.2}$$

where x is the length along the rod, and t is time.

The boundary and initial conditions are

$$\begin{aligned} u(0,t) &= 0, \\ u(L,t) &= 0, \\ u(x,0) &= sin(\pi x). \end{aligned} \tag{2.3}$$

It is easily verifiable that the specific analytic solution is

$$u(x,t) = e^{-\pi^2 t}\sin(\pi x). \tag{2.4}$$

2.1 Explicit Schemes

Traditionally, differential equations have been solved using numerical finite difference methods, either explicit or implicit. Here, the results obtained using the different machine learning algorithms was compared with those obtained using the explicit forward Euler method. First, discretise the variables as follows.

$$\begin{aligned} x &\to x_i = i\Delta x \\ t &\to t_j = j\Delta t \\ u(x,t) &\to u_i^j \end{aligned}$$

such that the lower index of u refers to the spatial step and the upper index to the temporal step.

The necessary derivatives are given by

$$\begin{aligned} \frac{\partial^2 u_i^j}{\partial x^2} &= \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{\Delta x^2}, \\ \frac{\partial u_i^j}{\partial t} &= \frac{u_i^{j+1} - u_i^j}{\Delta t}. \end{aligned}$$

Combining the two and solving for u_i^{j+1} gives

$$u_i^{j+1} = r(u_{i+1}^j - 2u_i^j + u_{i-1}^j) + u_i^j, \tag{2.5}$$

where

$$r = \frac{\Delta t}{\Delta x^2}.$$

Thus, given all the spatial steps at the current time-step, all spatial steps at future time-steps can be found by iteratively solving the above equation. Finally, the step sizes Δx and Δt for the explicit scheme are related by the stability criterion

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}. \tag{2.6}$$

2.2 Neural Network

A feed forward neural network consist of L layers, each with a set number of nodes. The layers are connected by weights and biases P, such that each node takes the value of a weighted sum of the nodes in the previous layer. An activation function is used to regularise the value each node can take. Some input is placed in the input layer, from where it is fed forward through the network, until it reaches the output layer. This is a prediction of the input data, which can be the function value of a continuous function at a certain point, or a discrete classification label, like whether a tumour is malignant or not. Under supervised learning, this prediction is compared with a true value to find the error. The back-propagation algorithm then calculates the gradients of the weights and biases which will minimise the error, and update them accordingly. Over several iterations, the network can become very accurate, and successfully predict results from new data. A full description of the architecture and inner workings of a feed forward neural network, along with discussion of strengths and weaknesses associated with such a method, can be found in [2] and [3].

2.2.1 Cost function

In order to determine the error, a cost function must be defined. In the case of regression problems, this could simply be the mean of the error in all points, squared. For classification, cross entropy is used.

Looking at equation (2.1), f^2 is then a natural choice of the cost function when optimising the solution of a PDE. When weighted by the total number of points, this is essentially the mean square error of the differential equation. The solution will be given in the output layer of the neural network, $g(\mathbf{x}) = N(\mathbf{x}, P)$, where N is the output-layer when $\mathbf{x}$ is fed through the weights and biases P. Then, through E iterations, the weights and biases P are updated using the back-propagation algorithm, as described in [2] and [3].

2.2.2 Trial function

In order to take care of the boundary and initial conditions, a trial-function

$$g_t(\mathbf{x}) = b(\mathbf{x}) + h(\mathbf{x}, N(\mathbf{x}, P))$$

is used rather than the actual solution given by the output layer. Here $b(\mathbf{x})$ ensures that the trial function gives correct initial and boundary values, while $h(\mathbf{x})$ ensures that $N(\mathbf{x})$ has no contribution at these points. While the neural network could have been used dirently, this would require many more training iterations before it converges to the solution.

2.3 Genetic Algorithms

Genetic algorithms form a family of machine learning algorithms inspired by the process of natural selection. The use of evolutionary systems to develop computational methods to solve optimisation problems stems back to the 1950s and 60s. Genetic algorithms were originally developed by John Holland and his students and colleagues at the University of Michigan in the 60s and 70s, and his main theoretical framework as presented in his 1975 book *Adaptation in Natural and Artificial Systems* [1] is still in use today.

The main idea behind genetic algorithms is to take inspiration from the unsupervised Darwinian evolution of populations of living organisms over generations and create an analogous machine learning algorithm. Before exploring the specific algorithm, it will therefore be useful to take a look at some of the main underlying principles of evolution.

There are three main principles of Darwinian evolution which are usually considered the necessary and sufficient conditions for natural selection to occur. These are a) phenotypic variation, b) differential fitness, and c) heritability [5]. Phenotypic variation is the population level property of there being variation in the phenotypic traits of the individuals in that population, and not just genotypic variation. Genotype is the complete set of genes of an individual, while phenotype is the observable traits of an individual. Differential fitness is the property that survival and reproduction rates vary between the individuals in a population, and heritability the property that some traits are passed onto offspring from their parents through their genes.

Over the generations, the individuals in the population gradually become better adapted to the environment in which they live as a result of cumulative selection. This requires that not only the final change is favourable, but the adaptive shift at each step must have provided an improvement in order for it to have been selected for. This requirement is well represented by the idea of "fitness landscapes", introduced by Sewall Wright in the 1930's [6]. These landscapes, also known as adaptive landscapes or evolutionary landscapes, provide a visualisation of the link between the genotype or phenotype and the individual's adaptive fitness or success. The adaptive success of the individual is represented by the height of its position in the landscape, and individuals with similar genotypes and phenotypes are expected to be located nearby each other in the landscape. These landscapes can be multidimensional, although the exact parameters along the other axes will be complex functions of the environment, and are rarely studied in detail. Instead, such landscapes provide a useful metaphor and tool for visualising the process of evolutionary optimisation, where the population as a whole is expected to move towards higher points in the landscape over time. Furthermore, it makes it easy to understand how populations can end up at local minima with suboptimal traits which evolution struggles to ”improve”, since moving to the true optimum would require an initial descent in the landscape, which cannot happen through cumulative selection. Random mutations can in some cases help to prevent this, but not always.

Applying this to machine learning, one starts with a “population” of “chromosomes”, each of which is a candidate solution, usually randomly initiated. Each chromosome is encoded as a list of genes following a set of encoding and decoding rules determined by the algorithms. Each gene then encodes an element of the candidate solution. For each generation, the “fitness” of each chromosome is evaluated based on some metric which is summarised in the cost function. The fitness of the individuals then determines their chance of reproduction. A new generation of chromosomes is then generated through reproduction (selection and crossover) and mutation, where the fittest chromosomes from the past generation are most likely to have their genes passed on. A range of different algorithms can be used for these two steps. Over the generations, it is hoped that the fitness of the candidate solutions will increase until a sufficiently good model is reached, giving an optimisation problem. The concept of fitness landscapes thus apply equally well to such machine learning algorithms as to the evolutionary process. There will be always be a significant risk of getting stuck at local maxima in the landscape, where the evolutionary process stagnates at a sub-optimal solution. Getting stuck in such local extrema is an ever-present problem in all of machine learning, not just for genetic algorithms. However, genetic algorithms stands out in that the mutation operator helps at getting out of these stagnation points, increasing the chances of reaching the global maxima in the fitness landscape.

Finally, a crucial premise for this algorithm to work is the so-called the building-block hypothesis, stating that the optimal solution is a combination of smaller elements [5]. At least partly independently of each other, these building blocks must provide a selective advantage to the individual in order to be selected for. Over the generations, cumulative selection ensures that the best building blocks spread in the population, until all the best building blocks have been combined in a single individual chromosome to form the optimal solution.

2.3.1 Differential Equations and Grammatical Evolution

Genetic algorithms have been found to be useful in various optimisation problems. As discussed above, it is possible to reformulate differential equations as optimisation problems. The use of genetic algorithms in solving such problems is a fairly recent development. It can be done in various ways, but in the present paper the focus shall be on a method based on *grammatical evolution*, first presented by Tsoulos and Lagaris in their 2006 paper *Genetic Programming and Evolvable Machines* [4]. Here, each chromosome is an analytic expression encoded by the genes through an encoding scheme. This scheme is given by the grammar of the algorithm. If the solution can be expressed in closed-form, there is thus hope of retrieving the true solution to the problem. If not, an approximate solution can be found. The algorithm and grammar here used is heavily inspired by the one used by Tsoulos and Lagaris.

2.4 Eigenvalue problems

While linear differential equations are not always trivial to solve, many techniques can be used to get an analytic solution to many families of PDEs. They are also very simple to solve numerically through explicit schemes. Non-linear differential equations are completely different beasts, however. While possible to solve, more effort is needed to implement in an explicit scheme. This is where the power of machine learning comes in, as they only need an expression to evaluate, which is easy to set up. This is one of the main reasons why differential equation solving neural networks and genetic algorithms are interesting.

Zhang Yi and Yan Fu showed in [7] a non-linear differential equation that will give an eigenvector of a symmetric $n \times n$ matrix A. It will here be stated without proof, and the reader is referred to [7] for further details. A random initial vector $x(0)$, if $x(0)$ is not orthogonal to one of the eigenvectors of A, a solution to

$$\dot{x}(t) = x(0)^T x(0) A x(t) - x(t)^T A x(t) x(t) \tag{2.7}$$

for $t > 0$, will, for large t, converge to an eigenvector of A, denoted ν. The corresponding eigenvalue can then be found as

$$\lambda = \frac{\nu^T A \nu}{\nu^T \nu}. \tag{2.8}$$

The neural network will be able to apprixmate the solution to $x(t)$, while the genetic algorithms find an expression that give solutions close to the analytical solution.

The reason this is interesting is that many differential equations can be reformulated as a eigenvalue problem, with a symmetric matrix. Thus,to solve a differential equation, the eigenvectors and eigenvalues of a symmetric matrix represent the solution.

3 Methods

The code where the algorithms are implemented can be found on the GitHub for this paper.

3.1 Neural networks

There are many hyper-parameters governing the efficiency and accuracy of a Neural Network. The PDE-solver network implemented for this paper has all the same features as the one made for regression and classification problems, described and extensively studied in

[3], so will not be detailed here. This includes a learning rate η, an L_2-regularisation term λ and a momentum-parameter γ. Further, the activation functions studied were sigmoid, tanh, and parametrised ReLU (PReLU). PReLU is a generalised form of ReLU and Leaky ReLU, taking a parameter α setting the slope of the negative activation. The weights were initialised using normalised Xavier-initialisation in the case of sigmoid and tanh activation functions, while He-initialisation was used for PReLU.

While the best method for determining the best parameters for a neural network is searching the entire hyperparameter-space, this is very time-consuming, and requires spending a lot of time on some parameter values giving quite bad results. Therefore, a manual grid-search was performed, optimising one parameter at a time. Setting all parameters to some reasonable value such that a result is obtained, they were in turn tweaked one by one until an optimal value was found for the set of other parameters, while training for a fixed number of epochs. A repeat of the process was done to re-optimise the previously set parameters. This method will not ensure the globally optimal parameter set, but will quickly lead to reasonably good results.

Specifically, the optimal value of the L_2-regularisation parameter λ was expected to be zero. Both the learning rate η and momentum γ should be as large as possible, without causing severe numerical problems, like over- or under-flow, in order to ensure as quick convergence as possible.

One thing discussed in [3] that could give better results was a different learning schedule, i.e. the way the learning rate changes during training. It is often beneficial to decrease the learning rate, as it allows for the convergence into a minimum. An experiment with 5 different functions was set up. These include a constant η, a linearly decreasing η, a parabola, a cosine decreasing from 1, and a cosine increasing from 1. The specific expressions were as follows, in the same order as above

$$\begin{aligned}\eta(e) &= \eta_0,\\ \eta(e) &= \eta_0(1-e),\\ \eta(e) &= \eta_0(0.3e^4 - e^3 + 0.6e + 0.9), \qquad (3.1)\\ \eta(e) &= \eta_0(2-\cos(e)),\\ \eta(e) &= \eta_0\cos(e),\end{aligned}$$

where e is the current epoch, divided by total number of epochs. The constant learning rate is the 'null hypothesis' to improve from. The linearly decreasing one was found to give good results in [3]. The parabola is meant to increase slowly at first, where there is a high risk of getting sudden spikes causing the program to crash, and then have higher learning rate when this risk is lower, before decreasing again, allowing for convergence into a local minimum. The constants were choses to give this behaviour. The increasing cosine is counterintuitively increasing the learning rate with time, included to see how this affects the results, while the normal cosine has it decreasing. Many more could have been tested, but these give a nice range of behaviours to test. The functions are shown in the top plot in Figure 1.

3.1.1 Solving the differential equation

When solving the differential equation (2.2), both the explicit scheme (2.5), and the neural network were used. The results were compared to the analytical solution (2.4). For the explicit scheme, both $\Delta x = 0.1$ and $\Delta x = 0.01$ were used, with Δt determined by the stability criterion (2.6). The same stability criterion is not applicable to the neural network, so one is more free to choose Δx and Δt. The neural network is considerably more computationally heavy, so using too many points was very slow. Further, by changing the step sizes, the optimal parameters will also change. Therefore, the neural network was only used for the first set of step sizes.

The trial function used to solve the differential equation (2.2) with initial and boundary conditions (2.3), satisfying the conditions set in 2.2.2 was

$$u_t(x,t,N) = \sin(\pi x)[1 + tN(x,t,P)]. \qquad (3.2)$$

3.2 Genetic Algorithm

The outline of the algorithm used is as follows. A population-class is defined containing a list of C chromosomes with randomly generated sequences of G genes. For each generation, the genetic sequence of each chromosome is decoded and its corresponding analytic expression is generated. The fitness value of each chromosome is then calculated, and the list of chromosomes ordered according to their fitness. The E fittest individuals are then passed onto the next generation unaltered, whereas the remaining new individuals are generated according to some crossover scheme, before mutations are applied to some, or all, of these. This process is repeated until the maximum number of generations is reached, or until the fitness criteria is met by the fittest chromosome. In this case, C was set to 1000, G to 50.

3.2.1 Grammar

Each chromosome contains 50 genes, where each gene is an integer value in the range [0, 255]. A chromosome is interpreted by reading the genes sequentially and constructing an analytic expression using the grammar as given in Table 1. The meaning of a given gene thus depends on the type expected when it is interpreted. To decode it, the modulus of the gene with the number of options in that type is taken. E.g., if an expression `<expr>` is expected, the gene 17 will be interpreted as 17 mod 6 = 5, giving "t". Note that the decoding algorithm used always interprets the first gene as an expression `<expr>`. Furthermore, to prevent the candidate solutions from being too simple, the first gene in the chromosomes was always set to 0 or 2, encoding `<expr><op><expr>` and `<func>`, respectively.

Table 1. Encoding scheme defining the grammar of the algorithm, showing how the genes are sequentially read to generate an analytic expression.

Type	Options	Index
`<expr>`	`(<expr><op><expr>)`	[0]
	`(<expr>)`	[1]
	`<func>`	[2]
	`<digit>`	[3]
	`<x>`	[4]
	`<t>`	[5]
`<op>`	`+`	[0]
	`-`	[1]
	`*`	[2]
	`/`	[3]
`<func>`	`sin(<expr>)`	[0]
	`cos(<expr>)`	[1]
	`exp(<expr>)`	[2]
	`x**(<expr>)`	[3]
	`t**(<expr>)`	[4]
`<digit>`	0	[0]
	1	[1]
	2	[2]
	3	[3]
	4	[4]
	5	[5]
	6	[6]
	7	[7]
	8	[8]
	9	[9]
	π	[10]

For reference, the analytic solution to the problem studied, (2.4), can be encoded as: [0, 2, 2, 0, 3, 0, 1, 0, 3, 10, 2, 0, 3, 10, 2, 5, 2, 2, 0, 0, 3, 10, 2, 3].

When encoding the chromosome to an expression, it is read until the encoding scheme reaches an end, even if there still are more genes to be read. These are then trash-genes, not contributing to the phenotype of that indvidual. During crossover or mutation, these genes can become used.

3.2.2 Fitness evaluation

The fitness of an individual is a weighed sum of its deviance from the differential equation and from the boundary conditions. Only the equations for partial differential equations will be included here, but these are easily simplified to other types of differential equations.

The ranges and number of collocation points, N, for each variable x and t must be given. 10 points were used for both variables. The fitness is calculated by adding the squared residuals of the collocation points for the differential equations and the squared error at the boundaries. Since the squared residual will always be a positive number, and the optimal model will have a residual of 0, to keep with the metaphor from the fitness landscape of the fittest individual having the highest fitness score, the fitness was defined as the negative of the squared residuals. Thus, the true solution has a fitness of 0. Each candidate solution is labelled $M_i(x,t)$ for $x \in [x_0, x_1]$ and $t \in [t_0, t_1]$, both with N points.

Using f as defined in eq. (2.1), the contribution to the fitness from the partial differential equation is given by

$$D(M_i) = -\frac{1}{N^2}\sum_{k,l=1}^{N} f^2.$$

In the case of the diffusion equation,

$$D(M_i) = \\ -\frac{1}{N^2}\sum_{k,l=1}^{N}\left(\frac{\partial^2 M_i(x_k,t_l)}{\partial x^2} - \frac{\partial M_i(x_k,t_l)}{\partial t}\right)^2.$$

The factor of $1/N^2$ is not strictly necessary, but it normalises the fitness to make solutions obtained with different number of collocation points comparable.

The contribution to the fitness from the boundary and intial conditions u is given by

$$B_1(M_i) = -\frac{1}{N}\sum_{l=1}^{N}\left(M_i(x_0,t_l) - u(x_0,t_l)\right)^2,$$
$$B_2(M_i) = -\frac{1}{N}\sum_{l=1}^{N}(M_i(x_1,t_l) - u(x_1,t_l))^2,$$
$$B_3(M_i) = -\frac{1}{N}\sum_{k=1}^{N}(M_i(x_k,t_0) - u(x_k,t_0))^2,$$
$$B_4(M_i) = -\frac{1}{N}\sum_{k=1}^{N}(M_i(x_k,t_1) - u(x_k,t_1))^2.$$

In the given problem, these are

$$B_1(M_i) = -\frac{1}{N}\sum_{l=1}^{N}\left(M_i(0,t_l)\right)^2,$$
$$B_2(M_i) = -\frac{1}{N}\sum_{l=1}^{N}(M_i(L,t_l))^2,$$
$$B_3(M_i) = -\frac{1}{N}\sum_{k=1}^{N}(M_i(x_k,0) - sin(\pi x_k))^2,$$
$$B_4(M_i) = 0.$$

The total fitness is then

$$F(M_i) = D(M_i) + \lambda[B_1(M_i) + B_2(M_i) + B_3(M_i) + B_4(M_i)].$$

Where the parameter λ is a penalising parameter which can be changed to tune the penalising contribution from the boundary conditions compared to the differential equation. This is useful if one factor contributes much more to the fitness than the other.

3.2.3 Reproduction

There exists a range of different algorithms for creating the next generation of chromosomes from the current generation. The main elements of the algorithm for reproduction are the selection, crossover and mutation schemes used.

First, elitism was used before any of the other reproduction operations. This involves a predetermined number E of the fittest individuals being passed onto the next generation unaltered (without mutations), whilst still being used for reproduction. This ensures that the highest fitness value never decreases between generations.

The selection scheme determines what chromosomes from the current generation are chosen as parents. Two different methods were here investigated; probabilistic selection and tournament selection. Assume the current generation contains C individuals and C-E new chromosomes are to be created. For probabilistic selection, a list containing *2(C-E)* integers distributed according to half a normal distribution centred at 0 with standard deviation *0.2C* is generated. These numbers are then iterated over in pairs, and are used as indices to choose which two individuals from the sorted list of chromosomes are to be chosen as parents. This ensures that the fittest individuals have a much higher chance of reproduction, but genotypic and phenotypic variance is still ensured by allowing the less fit individuals to reproduce some of the time. Tournament selection involves picking K random integers in the range [0, C] and picking out the lowest two. These are then used as the indices to pick out the two parents from the sorted list of chromosomes. Note that both methods allow the same chromosome to be chosen as both parents, in which case the child will always be a perfect copy of the parent(s) before mutation. How the new chromosomes are generated from the two parents is then determined by the crossover scheme.

The crossover scheme determines how new chromosomes are created from the past generation. Two different methods were investigated here. First, one can create a new genetic sequence which is a perfect mix of the genes of the two parents. Call this method "mixing". *0.5G* different integers are picked from the range [0, G], and these are used as the indices of the genes used from one parent, and the remaining genes are taken from the other parent at the loci not picked by the indices. Note that the loci of the genes in the sequence are always preserved. This is the method most analogous to how reproduction takes place in nature. Alternatively, a random index in the range [0, G] is chosen, and this is used as the crossover point, such that the genes before that point is taken from one parent and the genes after that point from the other parent. The two halves are then combined to create a new genetic sequence. Call this method "swapping".

After the crossover operation, a mutation operation is applied. Here, one must first decide what percentage of the new chromosomes are to have their genes mutated, and secondly how many of the genes are to be changed. The mutation operation itself works by randomly picking a given number of indices in the range [1, G] (the first gene was always constrained to be 0 or 2) to determine the loci of the genes to be changed. The genes at the chosen loci are then replaced by random integers in the range [0, 255].

3.3 Finding eigenvalues of symmetric matrices

A symmetric 6×6 matrix A was generated from a matrix Q with elements $q_{ij} \sim \mathcal{N}(0, 1)$, by

$$A = \frac{Q^T + Q}{2}. \tag{3.3}$$

A vector $x(0)$ was generated similarly. The neural network was then used to solve (2.7). No trial function was needed in this case, as there are no initial or boundary conditions to consider. When evaluating the differential equation, this is done for $t = t_{max}$, which is set to a sufficiently high value, where $t_{max} = 100$ was found to be sufficient.

A standard linear algebra library was used to find the eigenvalues and vectors of the matrix A, for comparison with the value found by the neural network. When comparing these, the log of the relative error was calculated as

$$\Delta_{rel} = \log_{10}\left(\frac{y_c - y_t}{y_t}\right), \tag{3.4}$$

where y_c is our computed value, and y_t the 'true' found by the library.

4 Results and Discussion

4.1 Neural network and explicit scheme

For $\Delta x = 0.1$, $\Delta t = \Delta x^2/2$ and $T = 1.5$, and a constant learning rate $\eta(t) = \eta_0$, the parameters that were found to give the lowest cost after 500 epochs, where $\eta_0 = 0.002$, $\lambda = 0$, $\gamma = 0.95$, using tanh as activation function and with 4 hidden layers, the first with 10 nodes, the next two with 20, and the last with 10.

With these parameters, the learning schedules in (3.1) were applied, now running for 5000 epochs. In the top

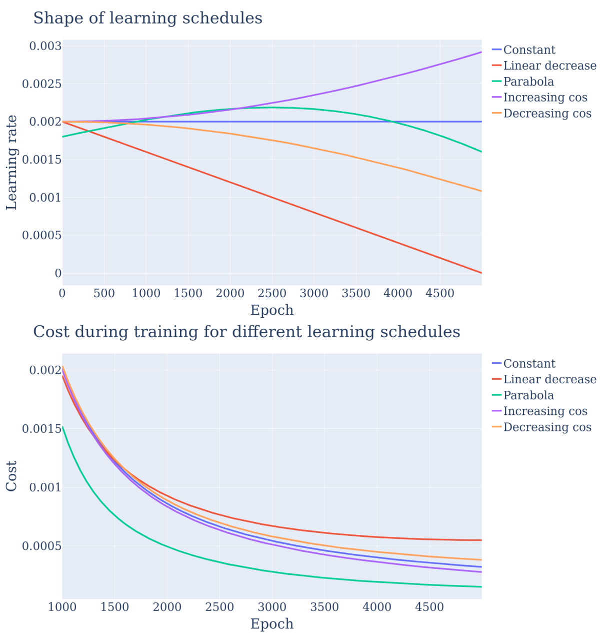

Figure 1. Top: Shape of learning schedules in (3.1) over 5000 epochs. Bottom: cost function as function of epoch for the last 4000 epochs of the simulation.

figure of Figure 1, the value of each learning rate is shown as function of epoch. In the bottom, the value of the cost function is shown, starting at epoch 1000, until the end of the simulation. This was to better see the the differences when the cost gets low. All learning schedules give a cost which initially decrease sharply, slowly converging towards the end of the simulation. The linear decrease is by far the worst of the 5 methods, and the parabola the best. Interestingly, increasing the learning rate towards the end also gave better results than keeping it constant.

None of these learning schedules takes into account the history of learning. An adaptive learning rate could change according to what would give faster convergence in the moment. Implementing this is left for future work on the matter.

Now, the solution from the neural network trained above was compared to the explicit solver with both $\Delta x = 0.1$ and $\Delta x = 0.01$, and the analytical solution. 3 points in time was chosen, and the 4 solutions was plotted as function of x. These are shown in Figure 2. At all times the solution from the explicit solver with $\Delta x = 0.01$ was indistinguishable from the analytic solution. The other explicit solver gave values that are too low, with the relative error increasing with time, which is as expected. At the time, the neural solver was more accurate than one of the explicit solvers, with a low relative error. However, at the next time step, it was worse, predicting too high values. At $T = 1.5$, the analytical solution was nearly flat, as seen from the magnitude of the curve. Both the explicit solvers gave results as expected. The neural network, however, did not, being too high, and not symmetric. While the absolute error is small, the relative error is very high.

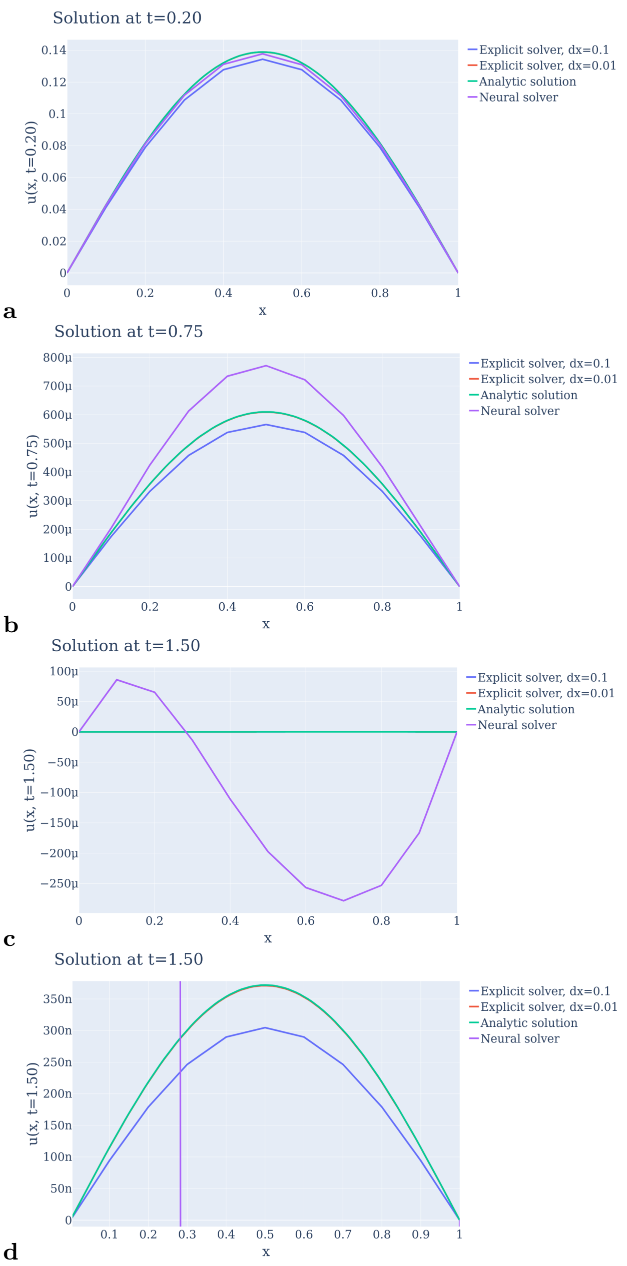

Figure 2. Solution of the differential equation, with explicit solvers with 2 step sizes, the neural network, and the analytical solution. **a**: Solution at $T = 0.2$. **b**: Solution at $T = 0.75$. **c**: Solution at $T = 1.5$. **d**: Solution at $T = 1.5$, zoomed in on the scale of the analytical solution.

While possible to solve this PDE with a neural network, this is more complex, less accurate and more inefficient than an explicit numerical solution. While the explicit solver with the smallest step size finished in 2 seconds, 40 minutes was needed to train the neural network to get these results, and then the higher step size was used, so using the same step size would have been non-viable. For more complex, non-linear PDEs, the neural network solver might be a better tool than explicit schemes, though this was not the case here.

4.2 Genetic Algorithm

4.2.1 Verification

First, in order to test the algorithm, some of the results presented in [4] were reproduced, namely two ordinary differential equations (ODEs) and one partial differential equation (PDE). The following two ordinary differential equations were tested:

$$\begin{aligned}\frac{dy}{dx} - \frac{2x-y}{x} &= 0,\\ y(0.1) &= 20.1,\\ x &\in [0.1, 1],\end{aligned}$$

with the analytic solution $y(x) = x + 2/x$.

$$\begin{aligned}\frac{d^2y}{dx^2} + 100y &= 0,\\ y(0) &= 0,\\ \frac{dy(0)}{dx} &= 10,\end{aligned} \tag{4.1}$$

with the analytic solution $y(x) = \sin(10x)$.

The partial differential equation tested was:

$$\begin{aligned}\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + 2u(x,y) &= 0,\\ y(0.1) &= 20.1,\\ x &\in [0.1, 1],\end{aligned} \tag{4.2}$$

with the analytic solution $u(x, y) = \sin(x)\cos(y)$.

In all three cases the number of chromosomes in the population was set to 1000, and the maximum number of generations to 1000 (same as was used by Tsoulos and Lagaris). The size of the genome was set to 50 genes. The elite size was set to 10 % of the population, and these individuals were passed straight onto the next generation without mutations. The remaining 90 % of the new generation were generated through reproduction, using tournament selection with 5 contestants with swapping as crossover method, as was used by Tsoulos and Lagaris [4]. Of these, half the chromosomes experienced mutations, and 10 randomly selected genes were then mutated. λ was set to 10, and 10 collocation points were used. All three analytic solutions were found after between 19 and 112 generations. Running the same programs several times, however, showed significant variance in the number of generations needed, so the stochastic element of the algorithm is clearly significant.

4.2.2 Diffusion Equation

The genetic algorithm was then used to attempt to solve the diffusion equation using two different combinations of selection and crossover methods. First, tournament selection was used in combination with swapping (denote this "tournament"). Secondly, probabilistic selection was combined with perfect mixing of the genes (denote this "mixing"). The elite size was kept at 10 %, and the mutation rate the same as before. The maximum number of generations was reduced to 500 due to time constraints. λ was set to 1, and 10 collocation points were used. In no case was the exact analytical solution found. To study the improvement over the generations, three different variables were used. The most important is the fitness value of the top chromosome. Additionally, the average fitness value of the top 10 % and 70% of the chromosomes was calculated, this in order to study the increased fitness of the population as a whole, i.e. how the fitter chromosomes become more dominant in the population. It is important to note, however, that it is in no way necessary for the average fitness of the population to be high to get a good result, since only the optimal solution will be of interest in the end.

A comparison of the two methods for the top chromosome, top 10 % and the top 70 % is shown in Figure 3. As can be seen, there are no major differences in the results obtained. The fitness of the top chromosome increases step-wise, whereas there is a gradual increase in the fitness of the top 10 %. This shows that the best candidate solutions spread in the population through reproduction, and gradually take up a larger proportion of the population. Finally, there is no development to be spotted in the average fitness of the top 70 %, which fluctuate significantly. This shows that more generations are needed before the fittest chromosomes spread that far, and even then the effects of random mutations between the generations might be sufficient to make enough of the chromosomes sufficiently unfit to prevent the average fitness from stabilising.

It was then attempted to independently change some of the other variables, namely the size of the elite group and the mutation rate. First, the size of the elite group was reduced to 5 %. Secondly, the size of the elite group was set back to 10 %, but the mutation scheme was changed. Instead of applying mutations to 10 genes in half of the new chromosomes, 3 genes were changed in all of the new chromosomes. This resulted in three different methods, and the fitness of the best chromosome over the generations for the different methods using mixing and tournament selection are shown in Figure 4. As can be seen, the differences are small, though the method using a smaller elite gives the best results both for tournament selection and mixing. However, no definite conclusions can be drawn from this single instance, for given the strong stochastic element of the algorithm, large samples would be needed to determine a definite trend. This was unfortunately not possible due to time constraints.

4.2.3 Critical Evaluation

In sum, the genetic algorithm employed successfully reproduces the results from [4], where fairly simple an-

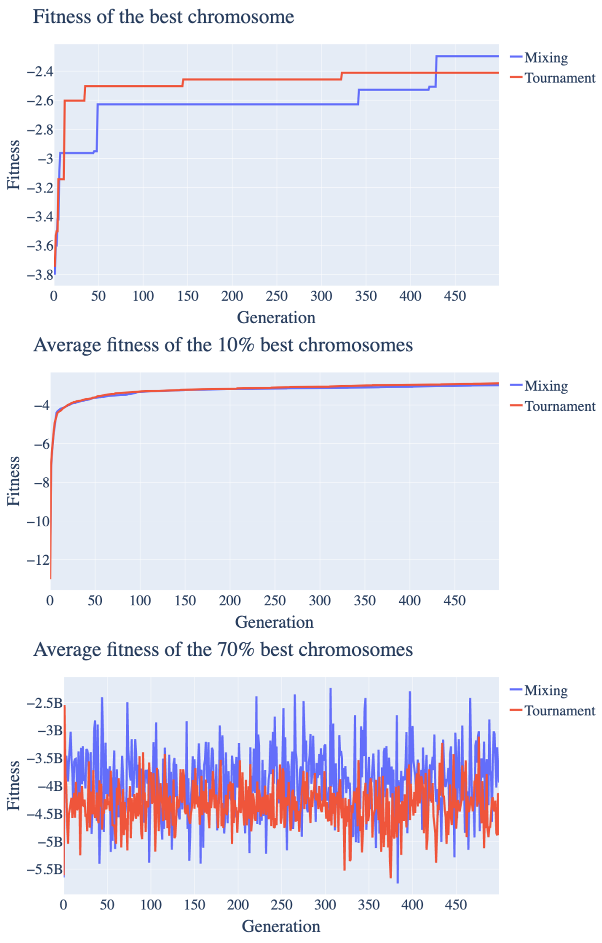

Figure 3. Three plots showing the fitness of the top chromosome, the average fitness of the 10 % best chromosomes and the 70 % best chromosomes respectively, using mixing and tournament selection.

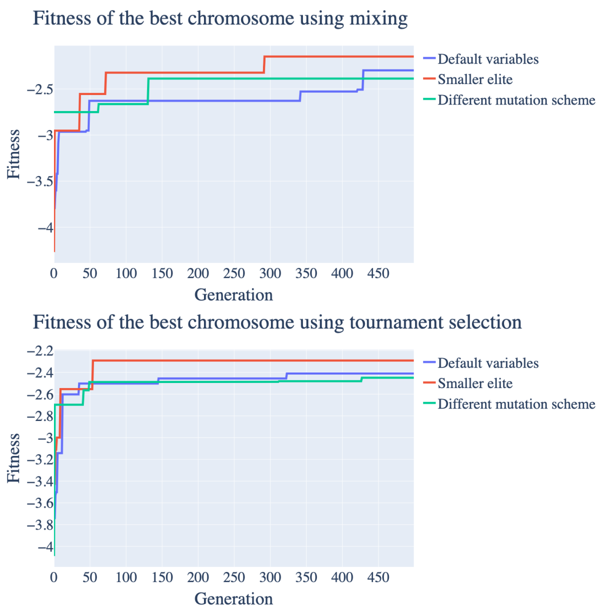

Figure 4. Plots of the fitness of the top gene as function of generations using slightly different variables, using mixing (upper plot) and tournament selection (lower plot).

alytic expressions are found which fulfil the differential equation and the boundary conditions. However, it is far from clear that these are found through cumulative selection, and not just by chance. Looking at the plots of the fitness of the fittest chromosome over the generations, it is clear that this tends to increase step-wise, rather than through a slow process of cumulative selection, which is the process the genetic algorithms are aiming to recreate. This should therefore be a source of worry. In fact, when analysing the employed algorithm of grammatical evolution, it is not surprising that there are few signs of cumulative selection in the results, for it is hard to see how the algorithm would be able to recreate this feature of evolution. To understand why, one must go back to the building block hypothesis and the notion of fitness landscapes. These ideas express two necessary conditions for evolution to take place which also apply to genetic algorithms. Firstly, as stated by the building-block hypothesis, the optimal solution must be built up from independent contributions from different building blocks, as encoded in the genes. Secondly, the fitness landscape expresses the requirement that evolution must be cumulative, such that small changes to the genotype (e.g. changing a few genes) does not normally lead to dramatic changes in the phenotype (here the resulting function), but instead makes it possible to slowly move towards local maxima. With the above grammar and algorithm used, however, neither requirement is met.

To understand why, it might be useful to look at an example of two sets of genes, [0, 2, 0, 4, 2, 5, 7, 3, 6, 8, 9, 2] and [0, 0, 0, 4, 2, 5, 7, 3, 6, 8, 9, 2]. The only difference in the genotype is the second gene, but whereas the first leads to the expression $sin(x)t$, the second gives the expression $\frac{xt}{6}+2$. Thus, changing a single gene completely changes the expression generated, making gradual improvements to the expression by changing individual genes near impossible.

One of the main problems with the grammar used, and how it becomes disanalogous with the process of natural selection, is in the dependence on the preceding genes in the interpretation of the current gene. In other words, the genes found at the different loci (positions in the genetic sequence) do not have any meaning independent of the genes found at the other loci. This breaks with the underlying assumption justifying the selection and crossover operations, where it is assumed that individual genes contribute to the fitness of the individual and are therefore "good" or "bad", such that mixing the genes of two fit individuals will give another fit individual. However, when this operation is performed using the above decoding scheme, the meaning of individual genes is completely changed by the change in the preceding genes, so the justification for the entire crossover-scheme breaks down. Thus, the structure of the analytical expressions is rarely preserved during crossover, especially not when the genes are mixed perfectly. Consequently, the above algorithm seems to mostly work through brute force by guessing a very high number of candidate solutions and preserving the best ones from each generation.

With 1000 chromosomes and 1000 generations, it is far from surprising the the correct solution eventually pops up by chance, provided this is not too complicated. This means that generating new genes at random instead of using a selection process should give similar results, as long at the "elite" is still passed on unaltered over the generations. Note that this is only expected to give similar results for the top chromosome - completely replacing the remaining 90 % of the population every generation will likely give a worse average fitness.

To test this assumption, a third random selection scheme was introduced. Here, after passing on the elite, consisting of the 10 % best chromosomes, the remaining 90 % were generated using the same random method as was used in the first generation when the chromosomes were first initialised. A comparison of the three methods is show in Figure 5. As can be seen, randomly selecting new genes at each generation gives similar results to the other methods tested. This might be an indication that the two former methods also work through fairly random processes, for since small changes in the genotype often lead to complete alterations in the phenotype (the analytic expression), mixing and tournament selection with mutations practically approaches randomly changing the phenotypes completely at each generation. To verify this further, similar plots were made for the second ODE (eq. (4.1)) and the PDE (eq. (4.2)) tested above, and the results are shown in Figure 6. The results further support the assumption, where in this case randomly changing the genes at each generation actually gives better results.

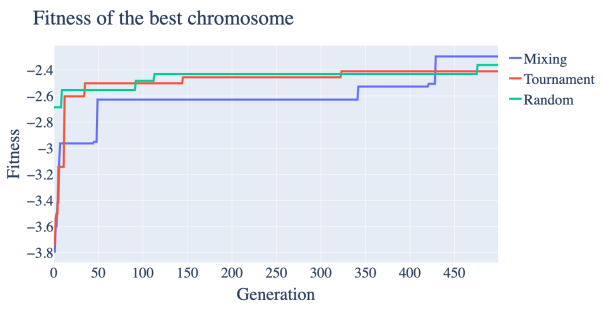

Figure 5. Plot of the fitness of the top gene as function of generations for the three different selection and crossover methods.

One might wonder why the analytical solutions are found for the ODEs and the PDE tested, but not for the diffusion equation. This is likely because of the sheer length of the analytic solution in terms of the genes, which is 24 genes long for the diffusion equation. Randomly reproducing this sequence is far less likely than randomly generating the genetic sequence coding for e.g. $\sin(x)\cos(y)$. Note that the likelihood of randomly selecting the genes for a given analytical expression will not just be a function of the length of the expression, but also of the types of components needed from the grammar. This is because the different components are found taking the modulus with respect to different numbers.

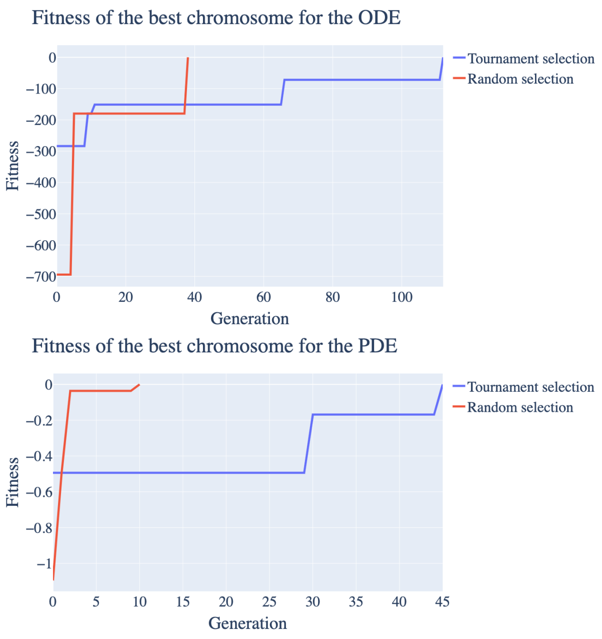

Figure 6. Plots of the fitness of the top gene as function of generations for the ODE and PDE using tournament selection and randomly changing all the genes at each generation.

Finally, it is important to stress that the results here obtained are fairly unreliable because of the sheer lack of data. As mentioned, there is a strong stochastic element to the functioning of the algorithm, so many repetitions would be needed to make reliable comparisons between the different methods. However, given the theoretical arguments above and the lack of falsifying evidence, it seems fair to assume that the genetic algorithm tested does not work through the intended mechanism, though further research would be needed to draw any definite conclusions regarding this.

One way to adress the problem of genes not having meaning independent of the loci and other genes could be to re-structure the way encoding happens. Instead of parsing the chromosome linearly, adding to the final expression, a tree could be built, where each expression form a sub-tree. It would then be possible for the crossover scheme to swap sub-trees, and not the entire remining section of the chromosme below the crossover point. This reduces the change of the expression changing completely during reproduction, rather only changing the particulars of it. This method has not been tested, and is only presented as inspiration for future work.

4.3 Finding the Eigenvectors of a Symmetric Matrix

A symmetric matrix A was generated using (3.3). The matrix used was

$$A = \begin{bmatrix} -0.0005 & 0.0130 & 0.5017 & 1.0018 & -0.8294 & 0.5650 \\ 0.0130 & 0.5403 & -0.5811 & -0.0028 & 0.3717 & -1.2035 \\ 0.5017 & -0.5811 & -0.8638 & 0.3562 & 0.2334 & -0.8832 \\ 1.0018 & -0.0028 & 0.3562 & 0.9973 & 0.5236 & -1.4400 \\ -0.8294 & 0.3717 & 0.2334 & 0.5236 & 0.3191 & 1.1358 \\ 0.5650 & -1.2035 & -0.8832 & -1.4400 & 1.1358 & -0.1203 \end{bmatrix}.$$

Using a standard linear algebra library, the eigenvalues were obtained as

$$\begin{aligned} \lambda_1 &= -3.1388, \\ \lambda_2 &= 2.5308, \\ \lambda_3 &= -0.8786, \\ \lambda_4 &= 0.0559, \\ \lambda_5 &= 1.2885, \\ \lambda_6 &= 1.0142. \end{aligned}$$

The start-vector generated was

$$x(0) = \begin{pmatrix} 0.7598 \\ 1.8274 \\ -0.6607 \\ -0.8078 \\ 0.8878 \\ -0.2174 \end{pmatrix}.$$

A neural network with 2 hidden layers, each with 30 nodes, a constant learning rate of $\eta_0 = 0.0003$, and no momentum or L_2-regularisation parameter was initialised with the activation function *tanh*. Using the network to solve (2.7) with this A and $x(0)$, after 50 epochs, the resulting vector at $t = 100$ was

$$x(t_{max}) = \begin{pmatrix} 0.6751 \\ 0.5289 \\ -0.3373 \\ -0.0054 \\ 0.0156 \\ 0.3878 \end{pmatrix}.$$

Using (2.8), the corresponding eigenvalue was

$$\lambda = 0.05587217366620945.$$

This is similar to the smallest (in absolute terms) eigenvector, λ_4. Using the non-rounded value, the log of the relative error was -4.97. The log of the largest relative error of each element in the computed eigenvector was -1.33.

Each component of the computed eigenvector was plotted as a function of epoch, to see how fast it converges. This is shown in Figure 7. Most of the elements seem to have converged already after about 30 epochs, though then the relative error of the eigenvalue was bigger, with the log of it at -3.25. After 100 epochs, it was even better, at -5.94, though this seems to be close to a limit,

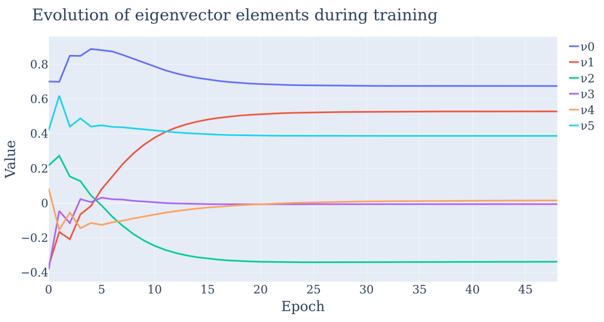

Figure 7.

as after 200 epochs, it was only marginally better, at -5.95.

While for some purposes the value obtained here might be accurate enough, this is in general not a great method for finding an eigenvalue, as standard libraries give much more accurate results, and optimal parameters for the network need to be determined. Further, this method is only good for finding a *single* eigenvalue and its corresponding vector at a time. If the symmetric matrix represents an ODE, and only a single solution is needed, solving it with a neural network can be viable. It would, however, be just as easy to use the neural network to solve the original ODE directly. Further, it would be beneficial if the neural network was generalisable to find an eigenvalue of a symmetric matrix without needing to be retrained, though whether this is the case was not tested. Such an analysis is outside the scope of this paper, and left for future work.

Theorem 4 in [7] states that the solution should converge to the eigenvector corresponding to the largest eigenvalue of A. We got the opposite, having it converge to the smallest. To determine if this was always the case, 1000 different symmetric matrices and start vectors $x(0)$ was generated. Table 2 shows what percentage each rank of eigenvalue was observed, with 1 corresponding to the smallest, 6 to the largest, and None meaning the training resulted in overflow and thus no result. The middle column presents the count for each rank when the matrix A was changed every time, while the right column shows the result keeping the matrix A constant. In both cases, the results are similar, so the choice of A affects the rank less than the choice of $x(0)$. This was to be expected. In both cases, about half the eigenvalues was the smallest, with the frequency decreasing for higher ranks. The properties of the vectors giving the different ranks were not studied.

Due to the problems with the genetic algorithm discussed above, it was not attempted to be used for find the eigenvalue of a symmetric matrix.

5 Conclusion

A neural network was used to numerically solve a partial differential equation. While the results obtained were

Table 2. Number of times the resulting eigenvalue had the given rank in the sorted ordering of all eigenvalues of A, with 1 being the smallest, and 6 largest.

Rank	Random A	Constant A
1	506	544
2	184	244
3	84	69
4	57	65
5	27	15
6	29	12
None	113	51

close to the analytic solution, explicit solvers performed much better, and was much more efficient. For more complex PDEs, for instant non-linear ones, the explicit schemes can be difficult to set up, so a neural network might be the best tool.

The genetic algorithm used was unable to find the analytical expression desired, and major flaws were uncovered in the method used. There is strong reason to believe that the algorithm does not work through cumulative adaptations of the chromosomes, as intended, but rather through randomly creating a new set of chromosomes at each generation and picking out the fittest. To change this, a different encoding scheme would be needed for the genetic code, one which gives each locus in the genome an unambiguous interpretation, independent of the other genes. This would preserve the structure of the expressions between mutations, and would allow for small, step-wise adaptations. The development of such an encoding scheme, however, is beyond the scope of this paper, but is certainly a worthy topic for further research, given the importance and wide applicability in many fields of being able to solve complicated differential equations.

The neural network-differential equation solver was applied to find the eigenvalues of a symmetric matrix. While being a quite quick and accurate method, it only gives one eigenvalue for a choice of initial vector. Thus, many vectors are needed to find them all. This does not take very much time, and so can be used to solve differential equations which can be reformulated as eigenvalue problems.

References

[1] John H Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. Complex adaptive systems. MIT Press, Cambridge, Mass, 1992.

[2] Sara Pernille Jensen and Daniel Johan Aarstein. Regression and classification, using stochastic gradient descent, feed forward neural networks and logistic regression. *University of Oslo*, October 2021.

[3] Håkon Olav Torvik, Vetle Vikenes, and Sigurd Sørlie Rustad. Machine learning: Using regression and neural networks to fit continuous functions and classify data. *University of Oslo*, October 2021.

[4] I.G. Tsoulos and I.E. Lagaris. Solving differential equations with genetic programming. *Genetic Programming and Evolvable Machines*, 7:33–54, 2006.

[5] Eyal Wirsansky. Hands-on genetic algorithms with python : applying genetic algorithms to solve real-world deep learning and artificial intelligence problems, 2020.

[6] S. Wright. The roles of mutation, inbreeding, crossbreeding and selection in evolution. *Proceedings of the XI International Congress of Genetics*, 8:209–222, 1932.

[7] Zhang Yi, Yan Fu, and Hua Jin Tang. Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix. *Computers & Mathematics with Applications*, 47(8):1155–1164, 2004.