Solving ODEs with Neural Networks

Computational Case Study report
supervised by Dr. Kathryn Gillow
Hillary term 2021-2022

Candidate Number 1064551

Contents

1 Introduction

In this report we shall look into how we can train neural networks to predict the solutions for differential equations. First, we'll build up the structure of a neural network, and how we can train it. Then we'll present the general setup for solving ordinary differential equations, including how this differs from general supervised machine learning, and give some examples. Finally we'll consider strategies for setting the hyperparameters.

2 Neural Networks

A neural network is a class of models that map from an input x to an output y using perceptron nodes. When these nodes are arranged in multiple successive layers, the model is called deep.

2.1 Perceptron

Perceptrons, schematically shown in Figure 1, also called artificial neurons (for their inspiration from biological neurons), are functions that take vector input, apply a linear mapping, and then a non-linear activation function, returning a scalar value.

$$\hat{y} = \phi\left(\mathbf{w}^T\mathbf{x} + b\right) \qquad \text{where} \quad \mathbf{x}, \mathbf{w} \in \mathbb{R}^n,\ y, b \in \mathbb{R} \tag{2.1}$$

and ϕ is a general non-linear function such as the sigmoid or tanh.

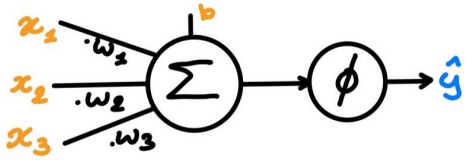

Figure 1: Visual representation of a perceptron with 3 inputs x_i, a bias value b, which are all added together, and then applied to the activation function ϕ to yield $\hat{y}$

2.2 Layer

We can arrange these perceptrons in a layer, Figure 2, such that they all take as input the same vector, and their outputs are concatenated in another vector (not necessarily of the same size). Specifically:

$$\hat{\mathbf{y}} = N(\mathbf{x}) = \phi(W\mathbf{x} + \mathbf{b}) \quad \text{where} \quad \mathbf{y}, \mathbf{b} \in \mathbb{R}^m,\ \ x \in \mathbb{R}^n,\ \ W \in \mathbb{R}^{m\times n}. \tag{2.2}$$

where thus $L : \mathbb{R}^m \to \mathbb{R}^n$, and ϕ is applied element-wise.

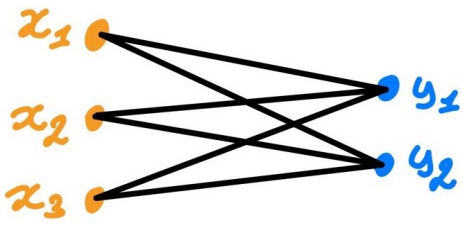

Figure 2: Layer of 2 perceptrons with three inputs. For clarity the labels of weights biases are omitted.

2.3 Loss and backpropagation

Suppose we want to train a layer (thus its parameters W and $\mathbf{b}$) to map from $x \in \mathbb{X}$ to $y \in \mathbb{Y}$. For this we choose a loss function to quantify the model performance, which we hence try to minimize. Consider the Squared Error:

$$SE = ||\mathbf{y} - \hat{\mathbf{y}}||_2^2/2 = ||\mathbf{y} - L(\mathbf{x}; W, \mathbf{b})||_2^2/2 \tag{2.3}$$

Which we try to minimize not for a single mapping $x \to y$, but for the entire dataset (with cardinality $|\mathbb{Y}|$), for all points $x^k = \mathbb{X}$ and $y^k = \mathbb{Y}$, yielding the Mean Square Error:

$$\mathcal{L} = \frac{1}{2|\mathbb{Y}|} \sum_{k=1}^{|\mathbb{Y}|} ||\mathbf{y^k} - N(\mathbf{x^k}; W, \mathbf{b})||_2^2 \tag{2.4}$$

We shall try to apply a gradient descent algorithm, hence need the gradient of the loss w.r.t. the parameters W and $\mathbf{b}$, $\nabla_W \mathcal{L}$ and $\nabla_{\mathbf{b}} \mathcal{L}$:

$$\frac{\partial \mathcal{L}}{\partial W_{ab}} = \frac{1}{2|\mathbb{Y}|} \sum_k \frac{\partial SE^i}{\partial W_{ab}} \tag{2.5}$$

$$\frac{\partial \mathcal{L}}{\partial b_a} = \frac{1}{2|\mathbb{Y}|} \sum_l \frac{\partial SE^i}{\partial b_a} \tag{2.6}$$

where the partial derivative terms are derived and given in Appendix B. Rearranging the weight matrix and bias vector into a vector θ, we can correspondingly construct $\nabla_\theta \mathcal{L}$ from the (2.5).

Lemma 1. *[2, Lecture 1]. Let $\mathcal{L}(\theta) \in C^1$, $\theta, s \in \mathbb{R}^n$ and $s \neq 0$. Then:*

$$\nabla \mathcal{L}(\theta)^T s < 0 \quad \to \quad \forall \alpha > 0 \ \textit{sufficiently small:} \quad \mathcal{L}(\theta + \alpha s) < \mathcal{L}(\theta).$$

Proof. If $\nabla\mathcal{L}(\theta)^T s < 0$, and $\nabla\mathcal{L}$ is continuous, then there exists a sufficiently small $\bar{\alpha}$ such that $\forall\alpha \in [0, \bar{\alpha}]$: $\nabla\mathcal{L}(\theta + \alpha s)^T s < 0$. Next, performing a Taylor expansions of $\mathcal{L}(\theta + \alpha s)$ around θ:

$$\mathcal{L}(\theta + \alpha s) = \mathcal{L}(\theta) + \alpha\nabla\mathcal{L}(\theta + \tilde{\alpha} s) \quad \text{for some } \tilde{\alpha} \in (0, \alpha)$$

And hence restricting $\alpha \in (0, \bar{\alpha})$ yields the desired result. □

Therefore, we conclude that for a sufficiently small α, the update step $\theta := \theta - \alpha s$ with $s = \nabla\mathcal{L}$ yields a decrease in loss, until $s = 0$ (which implies an extremum). Hence, with this technique, we can update the weight matrix and bias vector to (asymptotically) bring the loss to a minimum, though it does not provide any guarantees about whether the minimum is local or global.

2.4 Deep Neural Network

Appending multiple layers forms a deep neural network, Figure 3. The output of one layer then feeds into the next one. The loss definition doesn't change, but the computation of the gradient becomes more involved.

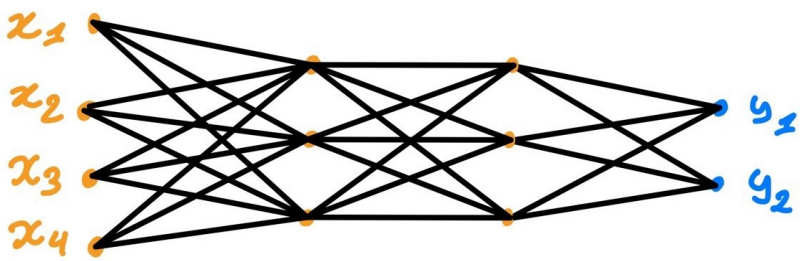

Figure 3: Deep neural network consisting of an input layer (x_i), 2 hidden layers and an output layer (y_i)

3 General problem

The aim of this case study is to create neural networks that map from coordinate vector (or scalar in 1D case) to the value of the solution of a differential equation. Let's consider a general second order ODE

$$a(x)\frac{\mathrm{d}^2\phi}{\mathrm{d}x^2} + b(x)\frac{\mathrm{d}\phi}{\mathrm{d}x} + c(x)\phi = d(x) \tag{3.7}$$

with Dirichlet boundary conditions:

$$\phi(x_a) = a \qquad \phi(x_b) = b. \tag{3.8}$$

Note that the neural network must have a single input and a single output, but we otherwise have no requirements on its architecture.

Boundary condition enforcement. Let us modify the output of the neural network such that the boundary conditions are always satisfied. To do this, we use a special activation function in the final layer of the ODE, mapping to a space of solutions where the BCs are always satisfied:

$$\phi = A(x) + B(x)N(x) = a + (b-a)\frac{x - x_a}{x_b - x_a} + (x - x_a)(x - x_b)N(x) \tag{3.9}$$

where $N(x)$ is the neural network, $A(x)$ was designed to satisfy the BCs and $B(x)$ to ensure the network contribution is zero at the BCs.

Loss function. We want the network to best approximate a solution to the ODE. Hence, we want the error between the left hand and right hand sides to be minimal. Let us again define the MSE over a set of points $\{x^i\}_{i=1}^n$:

$$\mathcal{L} = \frac{1}{2n}\sum_{i=1}^{n}\left(a(x^i)\frac{\mathrm{d}^2\phi}{\mathrm{d}x^2} + b(x^i)\frac{\mathrm{d}\phi}{\mathrm{d}x} + c(x^i)\phi - d(x^i)\right)^2. \tag{3.10}$$

One major difference with the general network case is that here the evaluation of the loss function involves the computation of the derivative(s) of the neural network. Though this is possible to do manually, it isn't regularly done because:

1. The manual computation is tedious and error prone
2. The final result isn't easily adaptable to a new differential equation or neural network architecture

To resolve these issues, we use automatic differentiation. It is a computational technique whereby you define the function (in this case the neural network) from input to output, and an algorithm analytically computes the derivative w.r.t. any desired input variable.

Algorithm The autodifferentiation library used, Jax [1], is functional, hence it dictates a certain code structure. Assume we have a function `forward_pass(w_b, x)` that takes both an input `x` and the neural network's weights and biases `w_b` (previously called θ), and that `ode_error_func` computes the error between the left and right hand sides of the ODE. A generic implementation of the gradient descent updating algorithm for the weights and biases is then:

```
def loss(w_b, x):
        phi = forward_pass(w_b, x)
        dphi_dx = jacobian(forward_pass)(w_b, x)
        ddphi_dxx = hessian(forward_pass)(w_b, x)
        return mean(ode_error_func(ddphi_dxx, dphi_dx, phi, x)**2)

def update(w_b, x):
        grad = grad(loss)(w_b, x)
        w_b = w_b - alpha * grad
        return w_b
```

We first define the computation of the loss function, which includes evaluations of the derivative(s) of the neural network (jacobian for first derivative and hessian for second). To update the loss we then sample the gradient of the loss function, and apply the update function.

The implementation in Python of the neural network has vectorization (so that the entire dataset can be evaluated in parallel), Just-In-Time (JIT) compilation and GPU aceleration to speed up computation. The result is that for (4.11), it takes just 2.6 to perform JIT compilation and 10^4 training iterations over 64 points. Note that the number of points is a multiple of 2 as this improves the performance when using GPU acceleration.

Rather than using gradient descent with a constant step size, we use Adam [5], which adapts its learning rate for each coordinate based on exponential averages of the gradient and squared gradient, as it was empirically found to perform better than gradient descent (with or without momentum).

3.1 Empirical vs true risk minimization

While we minimize the loss over a finite subset of $[x_a, x_b]$, we actually wish to minimize over the entire space, [4, Section 5.2]. However, computers can only deal with finite

quantities, and with increasing number of points comes increasing computational cost. Hence we must measure and optimize the loss on a finite set of points (empirical risk) while we want to minimize the error over all points (true risk).

In effect, during the training process, we not only want to minimize the empirical error, but also the gap between the empirical and true error. As we can't sample the true error, we must approximate it. For this, we divide the dataset into two: training data and test data. The training data is used in the optimization algorithm, while the test error is simply used to approximate the true error.

In this specific situation where the data (x points) can be generated as needed, we simply define the test dataset to be another linear spacing between x_a and x_b, with more points than the training dataset (here chosen to be 1024).

3.2 Underfitting vs overfitting

When training a machine learning problem, we need to balance underfitting and overfitting. Underfitting occurs when the model is not able to obtain a sufficiently low error on the training set (for example fitting a linear model to a underlying quadratic distribution). Overfitting occurs when the gap between training and test error is too large. The relationship between the two is mediated by the model capacity, [4].

Figure 4 compares the test error of neural networks trained with differing numbers of training points. It might be surprising to see that a neural network trained on 2 points (thus $x = 0$ and $x = 1$) can learn. This is because the loss function does not only evaluate the value of the neural network, but also its derivatives, which aren't fixed by the boundary condition enforcing activation function.

We can draw a few conclusions from Figure 4. Firstly, we don't seem to be experiencing overfitting, even when the number of points is just 2. One hypothesis for this is that, placing Dirichlet boundary conditions, and using the derivative in the loss function, there isn't much freedom left for the network to overfit. Secondly, increasing the number of training points does increase the model accuracy, which will not be visible in the loss function, but will greatly affect model performance. We must still be careful to avoid overfitting, hence, we never use less than 10 points for training.

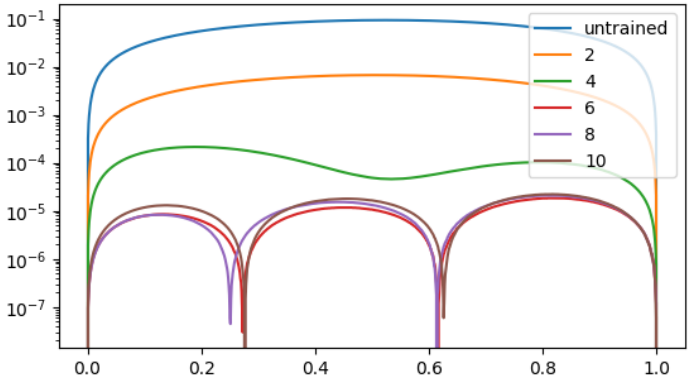

Figure 4: Absolute error (w.r.t. the exact solution) of a neural network trained on $n_{points} \in \{2, 4, 6, 8, 10\}$ points, evaluated on a mesh of 1024, for the second order ODE of (4.13).

4 Problem Solving

We can now use the neural network to approximate solutions to differential equations, and have recreated the examples presented by Lagaris et al. [6].

4.1 First order ODE

Consider Example 2 from Lagaris et al. [6, p. 8]:

$$\frac{\mathrm{d}}{\mathrm{d}x}\phi + \frac{1}{5}\phi = e^{-\frac{x}{5}}\cos(x) \quad \text{with} \quad \phi(0) = 0. \tag{4.11}$$

It has exact solution $\phi = e^{-x/5}\sin(x)$. To enforce the BC, the final layer activation function must be $\hat{y} = 0 + x \cdot N(x;\theta)$. Figure 5 shows the resulting neural network prediction, error, and loss progression. We note that at $x = 0$ (and only there) the error is zero, which is due to the final activation layer that enforces the left boundary condition. Note that the troughs to zero are sign changes of the error (whose absolute value is plotted).

4.2 Sinusoidal first order ODE

One might say that approximating the solution to (4.11) isn't very complicated, as the solution doesn't very much from the line between the two imposed Dirichlet boundary conditions. To demonstrate that this technique is valid for more curvy

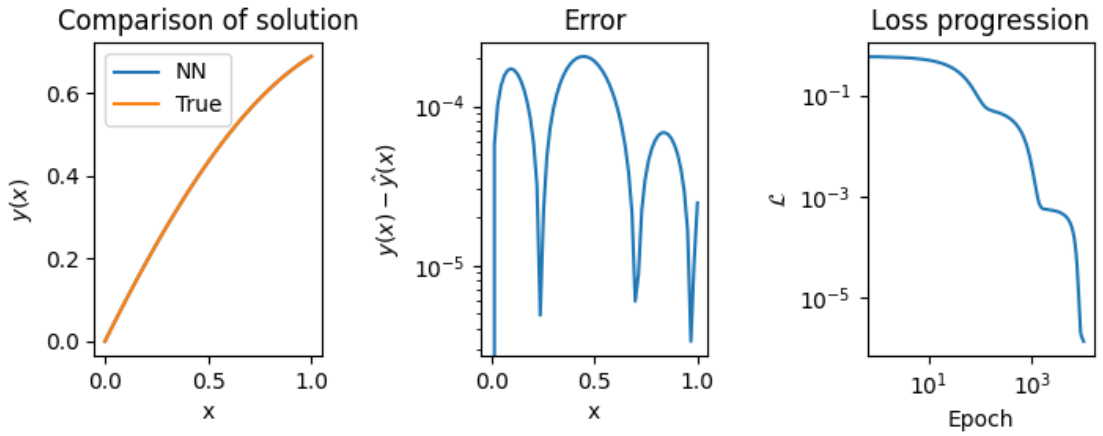

Figure 5: Comparison of the neural network output and the true value, for (4.11), for a single training run of 10^4 epochs, using the Adam [5] optimizer with a learning rate $\alpha = 0.001$ and 64 points, linearly spaced between 0 and 1.

ODEs, consider:

$$\frac{\mathrm{d}}{\mathrm{d}x}\phi - \phi\frac{1}{4} = e^{x/4}\pi\cos(\pi x) \quad \text{with} \quad \phi(0) = 0 \tag{4.12}$$

for which the exact solution is $\phi(x) = e^{x/4}\sin(\pi x)$. Imposing a different boundary condition doest not increase the complexity of the problem as all it changes is the final activation function, which now is $\hat{y} = xN(x;\theta)$.

From Figure 6 we can see that the neural network is able to effectively approximate the solution to (4.12). The distinct steps in the training loss seem to coincide with sudden changes in the form of the solution. During the training process, the solution approximation seems to always attach to a peak or valley, as can be seen in Figure 11, and then jump to a better approximation.

4.3 Second order ODE

Consider Example 3 from Lagaris et al. [6, p. 9]

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\phi + \frac{1}{5}\frac{\mathrm{d}}{\mathrm{d}x}\phi + \phi = -\frac{1}{5}e^{-x} \quad \text{with} \quad \phi(0) = 0,\ \phi(1) = e^{-1/5}\sin(1). \tag{4.13}$$

The exact solution is again $\phi(x) = e^{-x/5}\sin(x)$. The final layer activation function is $\hat{y} = x\sin(1)e^{-1/5} + x(1-x)N(x;\theta)$. Figure 7 shows the comparison between the neural network prediction and truth, as well as the loss progression. The main change with Figure 5 is that here the error goes to zero both at $x = 1$ and $x = 2$.

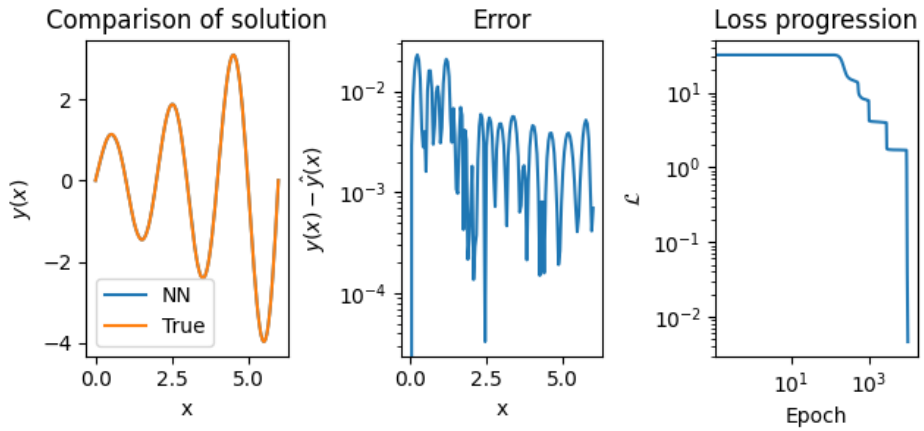

Figure 6: Comparison of solution of (4.12). Neural network has 2 hidden layers of 10 nodes each, trained using Adam with a learning rate $\alpha = 10^{-3}$, 128 equidistant points between $x = 0$ and $x = 6$ in $5 \cdot 10^5$ epochs

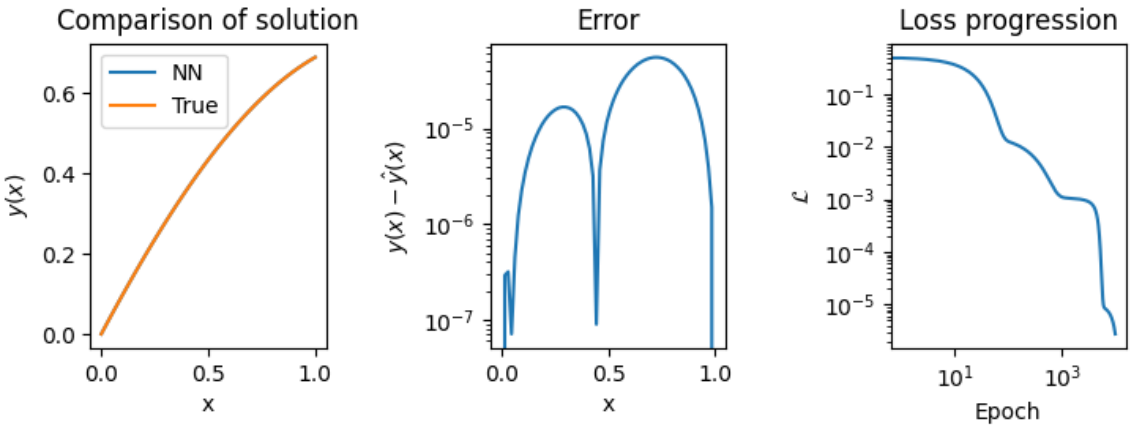

Figure 7: Comparison of the neural network output and the true value, for (4.13), for a single training run of 10^4 epochs, using the Adam [5] optimizer with a learning rate $\alpha = 0.001$ and 64 points, linearly spaced between 0 and 1.

4.4 Solving systems of ODEs

We can extend the framework presented in Section 3 to also solve systems of ODEs, by increasing the number of outputs of the neural network. Consider Example 4 presented by Lagaris et al. [6, p. 9]

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}x}\phi_1 &= \cos(x) + \phi_1^2 + \phi_2 - (1 + x^2 + \sin(x)^2) \qquad \text{with} \quad \phi_1(0) = 0 \\ \frac{\mathrm{d}}{\mathrm{d}x}\phi_2 &= 2x - (1 + x^2)\sin(x) + \phi_1\phi_2 \qquad \text{with} \quad \phi_2(0) = 1 \end{aligned} \tag{4.14}$$

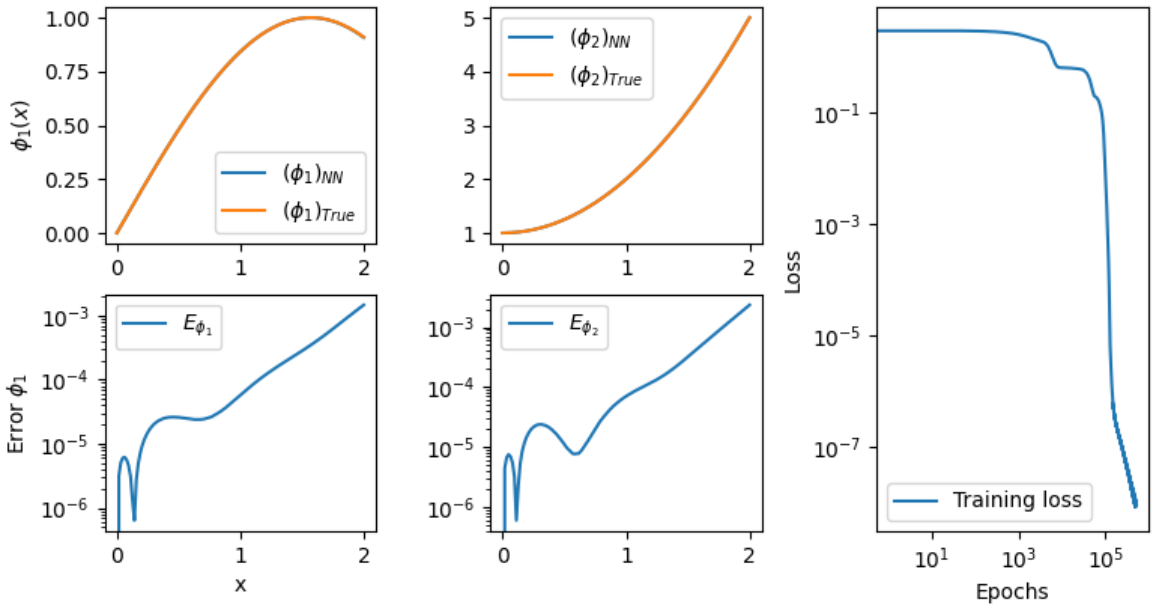

Figure 8: Comparison of the neural network prediction and the true solution of (4.14). The neural network has 2 hidden layers of 20 nodes each, was trained using Adam with $\alpha = 10^{-5}$, trained using 128 equidistant points between 0 and 2 for $5 \cdot 10^5$ epochs.

on $x \in [0, 3]$. The analytic solutions are $\phi_1(x) = \sin(x)$ and $\phi_2(x) = 1 + x^2$. The final layer activation function is then $\hat{y}_1 = xN_1(x;\theta)$ and $\hat{y}_2 = 1 + xN_2[x;\theta]$, where N_1 and N_2 are the first and second entries of the neural network output vector. The two outputs imply that the MSE is defined as $||\mathbf{e}||_2^2$, where e_i is the error (left hand side minus right hand side) of the i-th differential equation.

Figure 8 shows the neural network results. As this is a system of first order ODEs, with a boundary condition applied on the left, we see the error goes to zero towards $x = 0$, and increases with x. We note that, even though the final loss is lower than the previous examples, the accuracy (absolute error) is worse. Two hidden layers of 20 nodes each, and a learning rate of $\alpha = 10^{-5}$ was needed to achieve this performance.

5 Hyperparameter tuning

When building a neural network and training it, many parameters must be chosen, initially arbitrarily or using intuition. These hyperparameters greatly affect the performance of the final model and must hence be optimized for. The problem we shall consider in this optimization process is the sinusoidal ODE, presented in Subsection 4.2, as its learning process was empirically found to be more sensitive than the others

to the hyperparameters.

Hyperparameters are parameters that aren't set during the main training procedure. This include the shape of the network (number of layers and node in each layer), the optimization algorithm (which itself has hyperparameters such as the learning rate), the initialization strategy, the activation functions, and many more.

General strategies. There are many strategies for hyperparameter tuning, which one is optimal will depend on the specific problem. Hand tuning is a good for initially getting the optimization working. Grid search and random search are two techniques that aim to sample most of the parameter space. Other techniques take an initial guess and sample around it to make new guesses, taking into account the previous samples.

5.1 Hand tuning

Though it takes experience to acquire, intuition for choosing hyperparameters, and understanding their effect, is of great importance when developing a neural network model. Initially guesses may also be obtained from literature. For example, Diederik and Kingma, when presenting Adam [5] give reasonable initial values for α, β_1 and β_2. Sometimes, looking at previous work can be useful to gauge an appropriate number and size of layers. By manually tweaking these initial guesses (e.g informed by looking at the loss progression or test accuracy) we can find reasonable parameter values.

Using hand tuning, we found the following hyperparameters: a network structure $[1, 10, 10, 1]$ (2 hidden layers with 10 nodes each) with sigmoid and BC enforcing activation functions; a learning rate $\alpha = 0.001$ and 128 equidistant points. To enable us to vary number of points without explicitly favouring more points, we limit the number of epochs at $10^7/n_{points}$.

5.2 Grid search

One strategy we might use is to search all combinations of hyperparameters, where each hyperparameter is one of a finite set. In our case we let these be:

1. Network structure: $N \in \{[1, 20, 1], [1, 10, 10, 1], [1, 7, 7, 6, 1], [1, 5, 5, 5, 5, 1]\}$
2. Learning rate $\alpha \in \{0.01, 0.001, 0.001\}$

3. Number of points $n_{points} \in \{10, 100, 1000\}$

4. Scaling of the random initial weights matrices and bias vectors $\epsilon \in \{0.1, 0.01, 0.001\}$

This yields 108 different options. However, to be able to make any statistically significant conclusions, we must run each option multiple times, we chose 10 multiples. This is an optimization over more than 10 Billion points.

The results, shown in Figure 9, enable us to draw some conclusion. Firstly, the training dataset should compromise only of 10 or 100 points, more points leads to strictly worse performance. Next, we can see that a small learning rate ($\alpha = 0.001$) and many layers, leads to poor performance, probably because the small step size does not let the algorithm make much progress in this epoch constrained environment. Somewhat surprisingly, both 3 and 4 layers are have worse performance than 1 and 2.

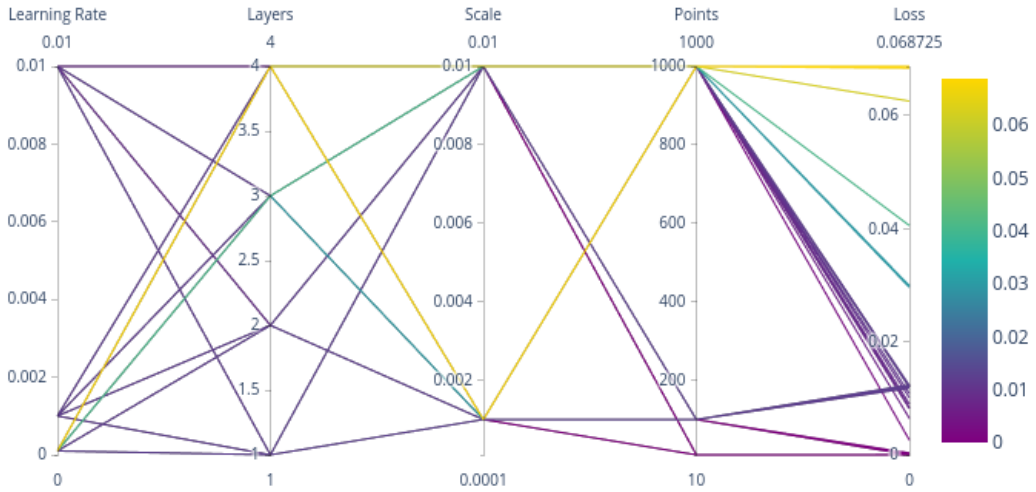

Figure 9: Parallel coordinates plot illustrating the results of the grid search. The colour of each line is related to the final loss.

The best performance, with an (estimated) true loss of $\hat{\mathcal{L}} = 1.6 * 10^{-6}$ is obtained by setting $\alpha = 10^{-4}$, $\epsilon = 10^{-3}$, $n_{points} = 10$ and having one hidden layer.

5.3 Random Search

As grid search is quite computationally expensive, we might wish to consider randomly subsampling. While a uniform distribution is possible, it might lead to oversampling of certain areas and undersampling of others. What we want is an even spread of

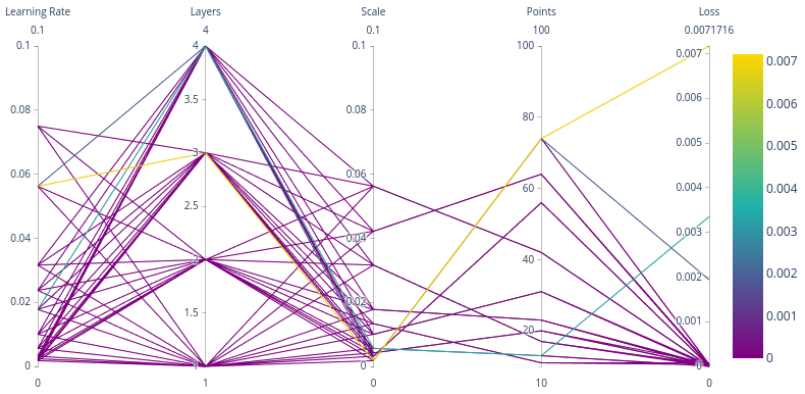

Figure 10: Parallel plot of the hyperparameters evaluated and the accompanying loss.

all points over the parameter space. One technique is to use a quasi-random Sobol sequence [7].

Given that we don't know the order of magnitude of many of these parameters, we must choose the range of the random values carefully. Let

1. Learning rate $\alpha = 10^{c_\alpha}$ where $c_\alpha \in [-3, -1]$
2. Number of points $n_{points} = 10^{c_{np}}$ where $c_{np} \in [1, 2]$
3. Scaling $\epsilon = 10^{c_\epsilon}$ where $c_\epsilon \in [-3, -1]$

Note that we can't randomly sample the network structure, so we'll have to perform a grid search over, over all 4 options. We choose to try $4 \cdot 10$ samples, each repeated 10 times.

The results are shown in Figure 10. The best performance, with an (estimated) true loss of $\hat{\mathcal{L}} = 3.0 * 10^{-7}$ is obtained by setting $\alpha = 5.6 \cdot 10^{-3}$, $\epsilon = 1.7 \cdot 10^{-2}$, $n_{points} = 23$ and having two hidden layers.

5.4 Evolutionary algorithms

The main idea to improve upon the performance of the grid and random search is to use the information from previously performed function evaluations to guide new hyperparameters to explore. One way to do this is using an evolutionary algorithm.

This class of algorithms is loosely modelled upon the evolution of species in nature. It starts, as explained by Verlag and Smith in [3], by generating an initial population, where each invidual is defined by a set of parameters (in our case these are the hyperparameters). Then, their fitness is evaluated (here quantified by the final test

loss). The best performing 5%, called the elites, are kept for the next generation. Then, from the remaining population, 25% is randomly selected to be a parent, to create the next generation. Note that the percentages mentioned are not fixed, simply reasonable values. The parents are then randomly combined, and offspring are generated by apply genetic mutation operators to their parameters (we considered crossover and mutation). This new generation then has its fitness evaluated and the loop restarts. As termination condition, we went for 10 generations, but one can also choose to stop after a certain number of iterations of no new offspring outperforming the best elite.

The best performance, with an (estimated) true loss of $\hat{\mathcal{L}} = 8.4 * 10^{-7}$ is obtained by setting $\alpha = 8.8 \cdot 10^{-3}$, $\epsilon = 6.3 \cdot 10^{-2}$, $n_{points} = 32$ and having three hidden layers. Comparing with the result of grid and random search, we see that the learning rate, scaling, and number of points are comparable, but the number of layers is different for all three.

5.5 Gradient free non-linear optimization algorithms

To minimize complicated non-linear functions, there exist many algorithms, such as Nelder Mead, BFGS, Newton and many more. However, the main problem with using these is the stochastic nature of the optimization problem. We want to reduce $\hat{\mathcal{L}} = \mathbb{E}[\mathcal{L}]$, but can only perform noisy sampling. This is an issue because the search direction of these algorithms is based on the function evaluations, whose differences the algorithm doesn't know are statistically significant or not. Repeatedly sampling the function does not solve this problem.

5.6 Bayesian optimization with acquisition function

Bayesian optimization is a constrained optimization technique for noisy functions. The main idea is that it repeatedly samples the function at parameters which have the best potential loss, according to a Bayesian model.

It starts by randomly sampling the function at multiple parameter values. Then builds a Bayesian model around those samples, to build a 95% confidence interval for the value of $\mathbb{E}[\mathcal{L}]$. Then, using an acquisition function, it samples the function at the potentially most promising value, defined by having the highest value for the sum of the mean and two standard deviations. After updating the model, the loop continues. Termination can occur either when the confidence interval of the best option is small, or after a fixed number of iterations.

We ran this optimization for $\alpha = 10^{c_\alpha}$ and $\epsilon = 10^{c_\epsilon}$ where $c_\alpha \in [-4, -2]$ and $c_\epsilon \in [-3, -1]$, keeping the number of points at 32 and 2 layers of 10 nodes each. After just 50 total evaluations (no repeats), it find the optimal parameters to be $\alpha = 8.8 \cdot 10^{-3}$ and $\epsilon = 1.7 \cdot 10^{-3}$, with an approximated true loss $\hat{\mathcal{L}} = 4.7 \cdot 10^{-8}$.

6 Conclusion

The first conclusion we can draw from this case study, is that indeed, neural networks can be used to solve differential equations, also systems, even those whose solution oscillates strongly. The difference between testing and train loss is very small, because we train for the derivatives of the function, such that (with the limited degrees of freedom of a small network) it can't vary much between training points.

The two main complications with hyperparameter tuning is that the loss function is very noisy, and costly to evaluate. Therefore hand tuning, is a suitable technique to guess initial parameters. Once the network runs, further tuning could be done using grid search or random search, but this is very costly, especially because every evaluation needs to be repeated to obtain an estimate of the standard deviation, and hence statistical significance of the result. Bayesian optimization on the other hand is very well suited to optimizing expensive and noisy functions, yielding a statistically significant optimal result. Another conclusion to draw regards the value of intuition. It is useful not only for finding initial hyperparameters, but also to appropriately set the optimization bounds, greatly diminishing the parameter space.

The code used in this special topic is available at `https://gitlab.com/th3et3rnalz/optimization_special_topic`.

References

[1] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: `http://github.com/google/jax`.

[2] Coralia Cartis. *Continuous Optimisation, Lecture slides*. 2022. URL: `https://courses.maths.ox.ac.uk/course/view.php?id=164`.

[3] A. E. Eiben and James E. Smith. *Introduction to evolutionary computing*. Springer Berlin, 2016.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[5] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: `http://arxiv.org/abs/1412.6980`.

[6] I.E. Lagaris, A. Likas, and D.I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 987–1000. DOI: `10.1109/72.712178`. URL: `https://doi.org/10.1109%2F72.712178`.

[7] I.M Sobol'. "On the distribution of points in a cube and the approximate evaluation of Integrals". In: *USSR Computational Mathematics and Mathematical Physics* 7.4 (1967), pp. 86–112. DOI: `10.1016/0041-5553(67)90144-9`.

A Images

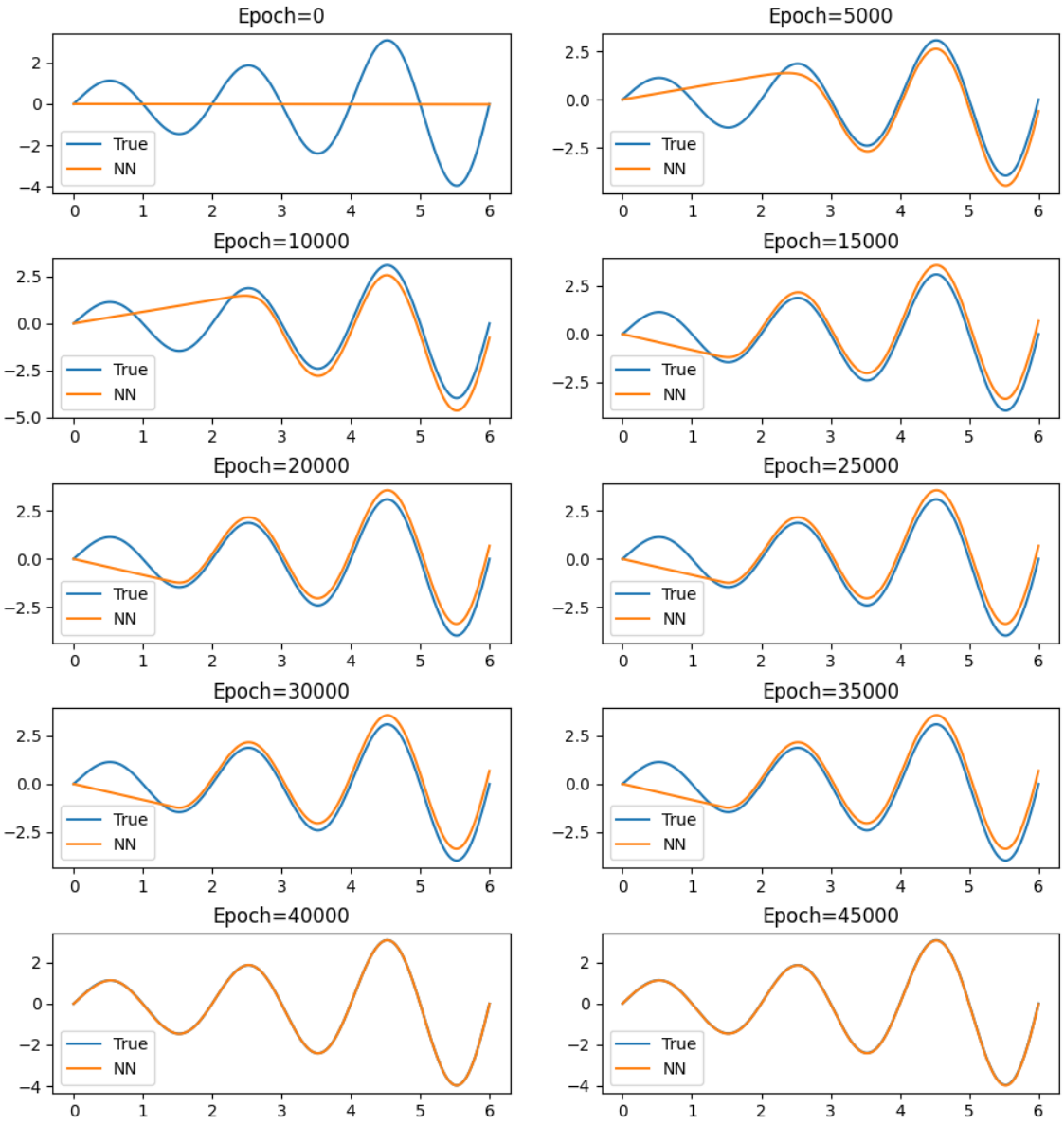

Figure 11: Neural network solution prediction of the solution of (4.12) during the training process, at various epochs.

B Backpropagation Derivation

$$\begin{aligned}
\frac{\partial MSE}{\partial W_{ij}} &= \frac{\partial}{\partial W_{ij}} \left((\mathbf{y} - \hat{\mathbf{y}})^T (\mathbf{y} - \hat{\mathbf{y}})/2 \right) \\
&= \frac{\partial}{\partial W_{ij}} \left(\mathbf{y}^T \mathbf{y} - 2\mathbf{y}^T \hat{\mathbf{y}} + \hat{\mathbf{y}}^T \hat{\mathbf{y}} \right)/2 \\
&= \frac{\partial}{\partial W_{ij}} \left(-\sum_{l=1}^{n} y_l \hat{y}_l + \frac{1}{2} \sum_{l=1}^{n} \hat{y}_l^2 \right) \\
&= -y_i \frac{\partial \hat{y}_i}{\partial W_{ij}} + \hat{y}_i \frac{\partial \hat{y}_i}{\partial W_{ij}} \\
\frac{\partial MSE}{\partial W_{ij}} &= (\hat{y}_i - y_i) \frac{\partial \hat{y}_i}{\partial W_{ij}}
\end{aligned}$$

where we used (2.1) to note that only y_i depends on W_{ij}. Then

$$\frac{\partial \hat{y}_i}{\partial W_{ij}} = \phi'(\hat{y}_i) \frac{\partial}{\partial W_{ij}} \left(W_{i:}\mathbf{x} + b_i \right) = \phi'(\hat{y}_i) x_j.$$

Similarly

$$\frac{\partial \hat{y}_i}{\partial b_i} = \phi'(\hat{y}_i) \cdot 1$$

Hence, for points x^k and y^k we find the MSE partial derivatives to be:

$$\frac{\partial MSE^k}{\partial W_{ij}} = (\hat{y}_i{}^k - y_i^k) \phi'(y_i^k) x_j^k \tag{B.15}$$

$$\frac{\partial MSE^k}{\partial b_i} = (\hat{y}_i{}^k - y_i^k) \phi'(y_i^k) \tag{B.16}$$