Independent Study

Matthieu Meeus

March 2020

1 Introduction

This document marks the beginning of the independent research study in Spring 2020, under supervision of Dr. Sondak, Dr. Protopapas and Dr. Mattheakis.

The goal of the overall project is to experiment with different neural network architectures (with an initial focus on activation functions) to solve partial differential equations (PDEs) in an unsupervised, data-free way.

This document goes beyond the initial exploration from the last couple of weeks.

First, recall the analytical of the two-dimensional, homogeneous Laplace equation in a circular domain. The same method, being separation of variables will now be used to solve the same equation but in three dimensions while maintaining the same circular form, therefore in cylindrical coordinates. The hope is obtain Bessel functions in the solution, rather than the infinite sum of polynomials and trigonometric functions of last time. The general form of this solution inspires us to implement a similar mathematical procedure in our neural network.

Next, a simple neural network architecture is being examined mathematically. The mathematical formulation of a regular neural network with two inputs, an arbitrary number of nodes in one layer and a sinusoidal activation function is derived. This approach is then extended for two layers. Note that this has been incorporated in the document since the beginning.

Next, recall the Pytorch implementation of the neural network to solve a simple ODE. First, the theory about Chebyshev interpolation was tested and appeared to be not very useful. Second, the performance of the proposed masternode to guarantee periodicity in the solution is evaluated. This done for the classic harmonic oscillator the Van der Pol equation and the Duffing equation.

Next, we will explore two-dimensional PDE's. A polar coordinate version of Feiyu's code has been implemented and used to solve the 2D Helmholtz equation. After verifying that it works, we continue to enforce periodic boundary conditions in the variable θ and try to implement different activation function. With this we hope that we can leverage the knowledge from the physics/analytical solution behind the problem to boost the training of the neural network.

2 Problem formulation and analytical solution

2.1 Problem formulation

As an example PDE, we will solve the three-dimensional, homogeneous Laplace equation on a hollow cylinder. Note that because of the circular dimension, cylindrical coordinates will be used in the analysis. Figure 1 below illustrates the domain that will be considered.

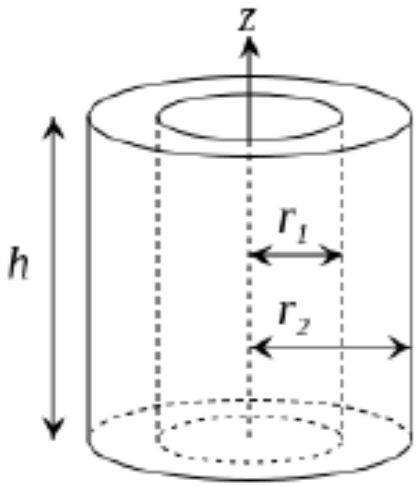

Figure 1: The considered cylinderical domain for the PDE, (ref)

The general, homogeneous Laplace equation is equal to:

$$\nabla^2 u(r, \theta, z) = 0 \tag{1}$$

For polar coordinates in particular, the Laplace operator has the following mathematical form:

$$\nabla^2 u = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} + \frac{\partial^2 u}{\partial z^2} = 0. \tag{2}$$

We now have to specify the boundary conditions applicable to the problem. For the radial dimensions, we will consider be a homogeneous Dirichlet boundary condition for $r = r_1$ and a non-homogeneous Neumann boundary condition for $r = r_2$, or:

$$u(r = r_1, \theta, z) = 0 \tag{3}$$

$$\frac{\partial u}{\partial r}\Big|_{r=r_2} = g(\theta, z) \tag{4}$$

Note that the circular geometry requires the solution to be periodic in θ, which will also play a crucial part in the analytical solution.

For the z-dimension, we will also consider a homogeneous Dirichlet boundary condition for $z = 0$ and a non-homogeneous one for $z = z_1$.

$$u(r, \theta, z = 0) = 0 \tag{5}$$

$$u(r, \theta, z = z_1) = h(r, \theta) \tag{6}$$

The section below will attempt to find the analytical solution of equation (1) with the specified definition of the Laplace operator (2) with boundary conditions (3), (4), (5) and (6).

2.2 Analytical solution through separation of variables

The general method that we will use is the separation of variables, meaning that we could write the solution of the PDE $u(r, \theta, z)$ as a product of decoupled functions in its three variables r, θ and z. Or:

$$u(r, \theta, z) = R(r)T(\theta)Z(z) \tag{7}$$

Let's plug this formulation into the cylindrical formulation of equation (1):

$$\nabla^2 u(r, \theta, z) = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} + \frac{\partial^2 u}{\partial z^2} = \frac{TZ}{r}\frac{d}{dr}\left(r\frac{dR}{dr}\right) + \frac{RZ}{r^2}\frac{d^2T}{d\theta^2} + RT\frac{d^2Z}{\partial z^2} = 0 \tag{8}$$

Which leads to:

$$\frac{TZ}{r}\frac{dR}{dr} + TZ\frac{d^2R}{dr^2} + \frac{RZ}{r^2}\frac{d^2T}{d\theta^2} + RT\frac{d^2Z}{\partial z^2} = 0 \tag{9}$$

$$\frac{1}{r}\frac{R' + rR''}{R} + \frac{1}{r^2}\frac{T''}{T} + \frac{Z''}{Z} = 0 \tag{10}$$

Note that the first two terms can only depend on r and θ, while the third can only depend on z and their sum should be equal to zero. This can only be true if -with λ being a real number:

$$\begin{aligned} -\frac{Z''}{Z} &= \lambda \\ \frac{1}{r}\frac{R' + rR''}{R} + \frac{1}{r^2}\frac{T''}{T} &= \lambda. \end{aligned} \tag{11}$$

Note that through the separation of variables, we are able to decouple the PDE into multiple equations.

Let's start solving for $Z(z)$, or:

$$\frac{Z''}{Z} = -\lambda \tag{12}$$

We now suppose that λ is a negative number, or $\lambda = -\gamma^2$. With this, we know that solution for $Z(z)$:

$$Z_0(z) = A_0 + B_0 z \tag{13}$$

$$Z_\gamma(\theta) = A_\gamma cosh(\gamma z) + B_\gamma sinh(\gamma_j z) \tag{14}$$

Considering the boundary condition (5), Z should be equal to zero when $z = 0$, which leads to $A_0 = A = 0$.

Next, we proceed with the second part of the (11) and multiply both sides by r^2:

$$r\frac{R' + rR''}{R} + \frac{T''}{T} = \lambda r^2 \tag{15}$$

Again, we know that $\frac{T''}{T}$ should be equal to a real number, so:

$$-\kappa = \frac{T''}{T} = \lambda r^2 - r\frac{R' + rR''}{R} \tag{16}$$

Let's say $\kappa = \nu^2$ and thus a positive number. For our solution to make physical sense, we need $T(\theta)$ to be periodic, or that $T(\theta) = T(\theta + 2\pi)$ and $T(\theta) = T(\theta + 2\pi)$ for all θ. From online resources (link), we know that the general solution for this is equal to the following:

$$T_0(\theta) = C_0 \tag{17}$$

$$T_\nu(\theta) = C_\nu cos(\nu\theta) + D_\nu sin(\nu\theta) \tag{18}$$

Where ν should be a strictly positive integer, so $\kappa_j = \nu^2$ for $\nu = 1, 2, ...$

We can know return to equation (16) and solve for $R(r)$, which will be more complicated.

$$\begin{aligned} -\kappa &= \lambda r^2 - r\frac{R' + rR''}{R} \\ -\kappa R &= \lambda r^2 R - rR' - r^2 R'' \\ r^2 R'' + rR' + (-\kappa - \lambda r^2)R &= 0 \\ r^2\frac{R''}{R} + r\frac{R'}{R} + (-\nu^2 + \gamma^2 r^2) &= 0 \end{aligned} \tag{19}$$

We can now substitute $x = \gamma r$ and $r = \frac{x}{\gamma}$, which leads to -after some algebra:

$$\frac{d^2R}{dx^2} + \frac{1}{x}\frac{dR}{dx} + \left(1 - \frac{\nu^2}{x^2}\right)R = 0. \tag{20}$$

The solution of this ODE is a linear combination of the Bessel function of the first and second kind, or $J_{\nu x}$ and $Y_{\nu x}$ respectively. This leads to:

$$R_\nu(r) = E_\nu J_\nu(\gamma r) + F_\nu Y_\nu(\gamma r) + G_\nu \tag{21}$$

Using the results from above and the separability of variables, we can write the overall, general solution for $u(r, \theta, z)$:

$$u(r, \theta) = R(r) * T(\theta) * Z(z) = \sum_\nu \sum_\gamma \begin{Bmatrix} sinh(\gamma z) \\ cosh(\gamma z) \end{Bmatrix} \begin{Bmatrix} \cos(\nu\theta) \\ \sin(\nu\theta) \end{Bmatrix} \begin{Bmatrix} J_\nu(\gamma r) \\ Y_\nu(\gamma r) \end{Bmatrix} \tag{22}$$

With the coefficients of $cosh(\nu z)$ being zero and the incorporating the constants, we get:

$$u(r,\theta,z) = A_0 + \sum_{\nu}\sum_{\gamma}\left\{\begin{array}{c} A_{\nu\gamma}sinh(\gamma z)\cos(\nu\theta)J_\nu(\gamma r) \\ +B_{\nu\gamma}sinh(\gamma z)\cos(\nu\theta)Y_\nu(\gamma r) \\ +C_{\nu\gamma}sinh(\gamma z)\sin(\nu\theta)J_\nu(\gamma r) \\ +D_{\nu\gamma}sinh(\gamma z)\sin(\nu\theta)Y_\nu(\gamma r)\end{array}\right\}. \tag{23}$$

The only task that remains is determining all coefficients associated with each γ and ν. This can be done using non-homogeneous boundary conditions (4) and (6).

I do have some difficulties moving forward into the details here. My question is if we could simplify this by saying that $r_1 = 0$, which leads to the exclusion of the Bessel function of the second kind $Y_\nu(\gamma r)$ as this is not finite for $r = 0$.

3 Exploring the neural network architecture

3.1 One hidden layer

This section explores the mathematical formulation of a neural network architecture with two input, one hidden layer and a continuous, single output. Note that the two inputs correspond to the two dimensions of the PDE in the section above, and the output function is a neural network prediction $\tilde{u}_{NN}$ of the exact solution u. The following figure illustrates a basic architecture with two nodes:

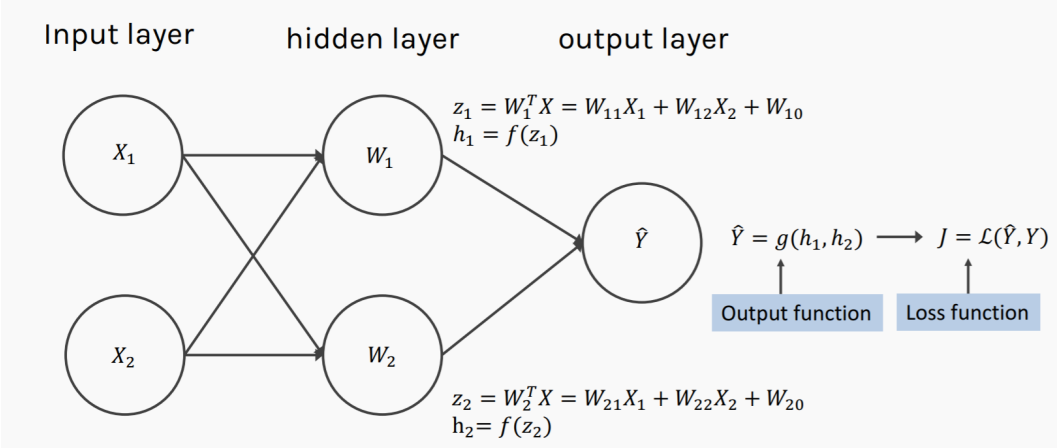

Figure 2: Illustration of a simple Artificial Neural Network (ANN), ref. to CS209a

The input layer has two continuous values, X_1 and X_2. Two nodes z_1 and z_2 result from a linear combination of these two values with an added bias. Each node therefore has two weights and a bias term, which will be tuned during training. Next, the resulting linear combinations z_1 and z_2 are 'activated' with a non-linear activation function $f(z)$. In what follows, this will be assumed to be a sinusoidal function. As the output should be a single, continuous value $\tilde{u}_{NN}$, the output function will again be a simple linear combination of the activated outputs of all nodes, with according weights and bias term.

Let's now try to find the mathematical expression of $\tilde{u}_{NN}$ in terms of the two input variables X_1 and X_2 and all the weights and biases. First, it is important to clearly define notation. W_j^i corresponds to the vector containing the weights and bias term for layer i and node j. $W^{(o)}$ contains the weights of the output layer. Capital letters correspond to the vectors, and small letters to the real numbers.

The two nodes are being computed as follows:

$$z_1 = W_1^{(1)} \cdot X = \begin{bmatrix} w_{11}^{(1)} \\ w_{12}^{(1)} \\ w_{10}^{(1)} \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_2 \\ 1 \end{bmatrix} = w_{11}^{(1)} X_1 + w_{12}^{(1)} X_2 + w_{10}^{(1)} \tag{24}$$

$$z_2 = W_2^{(1)} \cdot X = \begin{bmatrix} w_{21}^{(1)} \\ w_{22}^{(1)} \\ w_{20}^{(1)} \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_2 \\ 1 \end{bmatrix} = w_{21}^{(1)} X_1 + w_{22}^{(1)} X_2 + w_{20}^{(1)} \tag{25}$$

These nodes will now be 'activated' as follows:

$$h_1^{(1)} = sin(z_1) = sin(W_1^{(1)} \cdot X) = sin(w_{11}^{(1)} X_1 + w_{12}^{(1)} X_2 + w_{10}^{(1)}) \tag{26}$$

$$h_2^{(1)} = sin(z_2) = sin(W_2^{(1)} \cdot X) = sin(w_{21}^{(1)} X_1 + w_{22}^{(1)} X_2 + w_{20}^{(1)}) \tag{27}$$

With the output weights contained in vector $W^{(o)}$, we can now write the output:

$$\tilde{u}_{NN} = W^{(o)} \cdot H^{(1)} = \begin{bmatrix} w_1^{(o)} \\ w_2^{(o)} \\ w_0^{(o)} \end{bmatrix} \cdot \begin{bmatrix} h_1^{(1)} \\ h_2^{(1)} \\ 1 \end{bmatrix} = w_1^{(o)} h_1^{(1)} + w_2^{(o)} h_2^{(1)} + w_0^{(o)} \tag{28}$$

Or in terms of the input variables, this becomes:

$$\tilde{u}_{NN} = w_1^{(o)} sin(w_{11}^{(1)} X_1 + w_{12}^{(1)} X_2 + w_{10}^{(1)}) + w_2^{(o)} sin(w_{21}^{(1)} X_1 + w_{22}^{(1)} X_2 + w_{20}^{(1)}) + w_0^{(o)} \tag{29}$$

For an total number of N nodes in one hidden layer, the general expression becomes:

$$\tilde{u}_{NN} = \sum_{i=1}^{N} w_i^{(o)} sin(w_{i1}^{(1)} X_1 + w_{i2}^{(1)} X_2 + w_{i0}^{(1)}) + w_0^{(o)} = \sum_{i=1}^{N} w_i^{(o)} sin(W_i^{(1)} \cdot X) + w_0^{(o)} \tag{30}$$

As a sidenote, we keep in mind the classic mathematical expressions for the sine and cosine of sums:

$$\sin(\alpha + \beta) = \sin\alpha \cos\beta + \cos\alpha \sin\beta \tag{31}$$

$$\cos(\alpha + \beta) = \cos\alpha \cos\beta - \sin\alpha \sin\beta \tag{32}$$

This means that sines and cosines of sums can be written as sums of products of sines and cosines. Hence, one could possibly rewrite equation (22) such that only sines and cosines of every input value individually appear.

3.2 Two hidden layers

It is now interesting to check what happens to this mathematical expression if there are two hidden layers. The following figure illustrates what such an architecture would look like

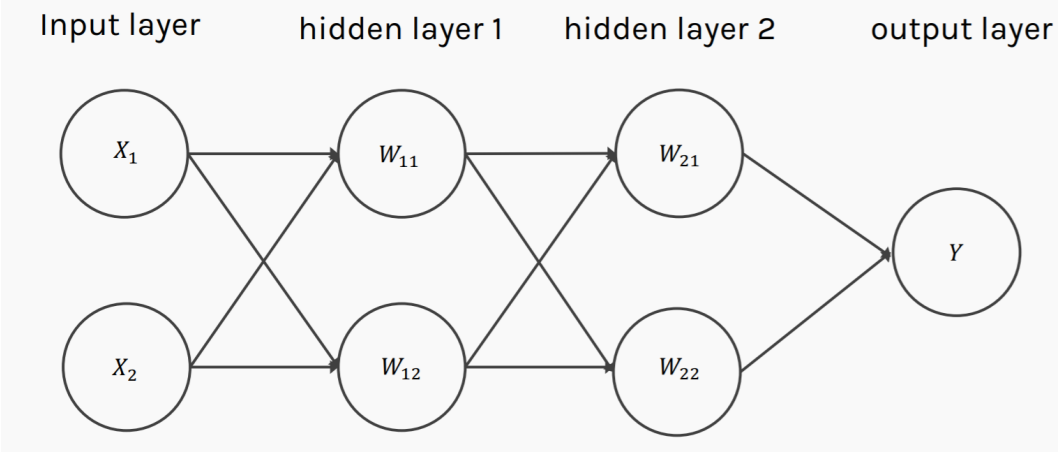

Figure 3: Illustration of a simple Artificial Neural Network (ANN) with 2 layers, ref. to CS209a

For this, let's define vector $H^{(1)}$ containing all activated nodes from the first layer and a 1 for the bias term. Using derivations from before we get:

$$H^{(1)} = \begin{bmatrix} h_1^{(1)} \\ h_2^{(1)} \\ 1 \end{bmatrix} = \begin{bmatrix} sin(W_1^{(1)} \cdot X) \\ sin(W_2^{(1)} \cdot X) \\ 1 \end{bmatrix} = \begin{bmatrix} sin(w_{11}^{(1)} X_1 + w_{12}^{(1)} X_2 + w_{10}^{(1)}) \\ sin(w_{21}^{(1)} X_1 + w_{22}^{(1)} X_2 + w_{20}^{(1)}) \\ 1 \end{bmatrix} \tag{33}$$

For the second layer, this now becomes:

$$H^{(2)} = \begin{bmatrix} h_1^{(2)} \\ h_2^{(2)} \\ 1 \end{bmatrix} = \begin{bmatrix} sin(W_1^{(2)} \cdot H^{(1)}) \\ sin(W_2^{(2)} \cdot H^{(1)}) \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} sin(w_{11}^{(2)} sin(w_{11}^{(1)} X_1 + w_{12}^{(1)} X_2 + w_{10}^{(1)}) + w_{12}^{(2)} sin(w_{21}^{(1)} X_1 + w_{22}^{(1)} X_2 + w_{20}^{(1)}) + w_{10}^{(2)}) \\ sin(w_{21}^{(2)} sin(w_{11}^{(1)} X_1 + w_{12}^{(1)} X_2 + w_{10}^{(1)}) + w_{22}^{(2)} sin(w_{21}^{(1)} X_1 + w_{22}^{(1)} X_2 + w_{20}^{(1)}) + w_{20}^{(2)}) \\ 1 \end{bmatrix} \tag{34}$$

The output can then be written as:

$$\begin{aligned}\tilde{u}_{NN} &= W^{(o)} \cdot H^{(2)} \\ &= w_1^{(o)} sin(w_{11}^{(2)} sin(w_{11}^{(1)} X_1 + w_{12}^{(1)} X_2 + w_{10}^{(1)}) \\ &+ w_{12}^{(2)} sin(w_{21}^{(1)} X_1 + w_{22}^{(1)} X_2 + w_{20}^{(1)}) + w_{10}^{(2)}) \\ &+ w_2^{(o)} sin(w_{21}^{(2)} sin(w_{11}^{(1)} X_1 + w_{12}^{(1)} X_2 + w_{10}^{(1)}) \\ &+ w_{22}^{(2)} sin(w_{21}^{(1)} X_1 + w_{22}^{(1)} X_2 + w_{20}^{(1)}) + w_{20}^{(2)}) \\ &+ w_0^{(o)}\end{aligned} \tag{35}$$

Interestingly, implementing more layers leads to a serial application of the activation function (so a sine of a sum of sines), while more nodes increases the length of the linear combinations within one sine. Both options lead to an equal increase in weights.

4 Solving an ODE with a Neural Network

In this section, we will attempt to solve a simple ordinary differential equation (ODE) with a neural network (NN). We first discuss the equation to be solved and its exact solution and subsequently expand on the neural network architecture needed to solve this.

4.1 Problem statement and exact solution

The ODE we wish to solve is the following:

$$\frac{d^2x}{dt^2} + \omega^2 x = 0 \tag{36}$$

With initial conditions $x(0) = x_0$ and $\frac{dx}{dt}(0) = v_0$. The analytical solution is equal to:

$$x(t) = \frac{v_0}{\omega} \sin(\omega t) + x_0 \tag{37}$$

For $x_0 = 0$, $v_0 = 1$ and $\omega = 2$, this leads to the following graphical result:

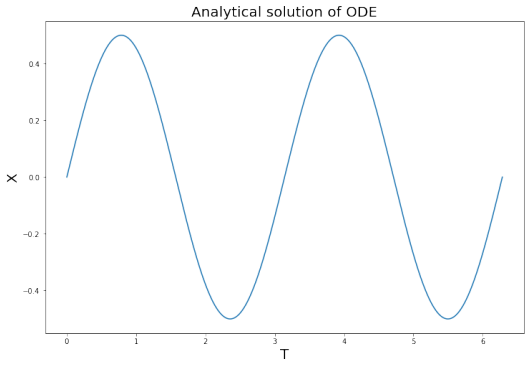

Figure 4: Solution of ODE

4.2 Neural Network solution

We now wish to solve equation (42) with a NN. The following figure illustrates how such a network can be designed:

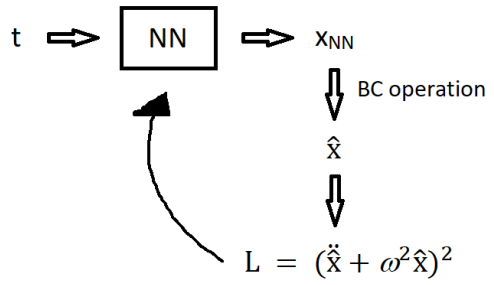

Figure 5: NN representation

The input variable is one scalar t, which leads through a series of nodes and layers of linear combinations and activation functions to a value x_{NN}. In order to force the solution to satisfy the specified boundary conditions, the following operation is computed:

$$\hat{x} = x_0 + (1 - e^{-t})v_0 + (1 - e^{-t})^2 x_{NN} \tag{38}$$

The loss function L is then computed based on the structure of the ODE. For a correct solution, the loss must be equal to zero.

$$L = (\ddot{\hat{x}} + \omega^2 \hat{x})^2 \tag{39}$$

Note that the loss function is a complication function of t and the weights of the neural network. The derivatives that explicitly appear in the loss function are with respect to the input variable t. For minimization, the algorithm will have to compute the gradient of the loss function with respect to the weights.

This has been implemented in Pytorch, with a tanh as activation function and two layers with each 32 nodes. The following illustrates the results.

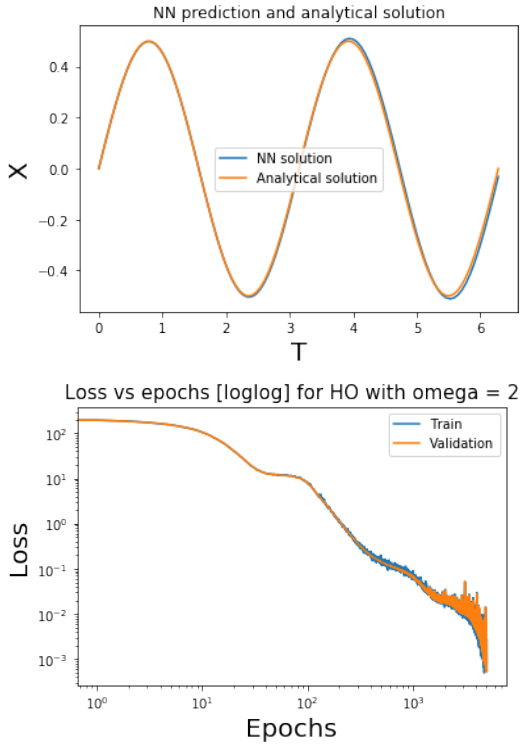

Figure 6: NN prediction of simple ODE

It is clear that the neural network does a pretty good job in predicting the solution to the ODE. Two things made me wonder about potentially improving the current design.

First, the interpolation method to generate the training data that is currently used is either uniformly distributed (default), equally spaced, or a noisy version of either one. This raises the question is there might be a modified sampling approach that leads to better results. In the section below, the Chebyshev interpolation is tested.

Secondly, it is interesting to analyse the performance of the neural network outside the training domain. The figure below shows what happens when the same NN as trained above is used to predict the solution of the ODE.

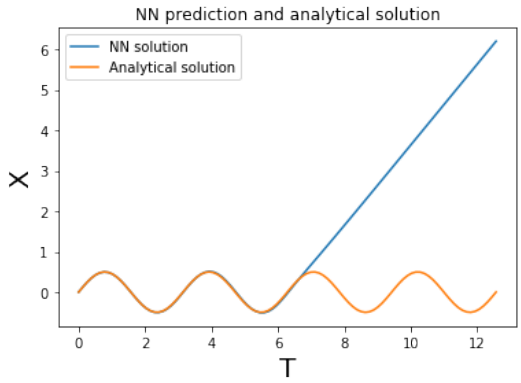

Figure 7: NN prediction outside training domain

We can also plot the difference between the predicted solution and the analytical solution over the t-domain. Here the t-domain is chosen to be slightly smaller than the training domain (up until 2.2π instead of 2π:

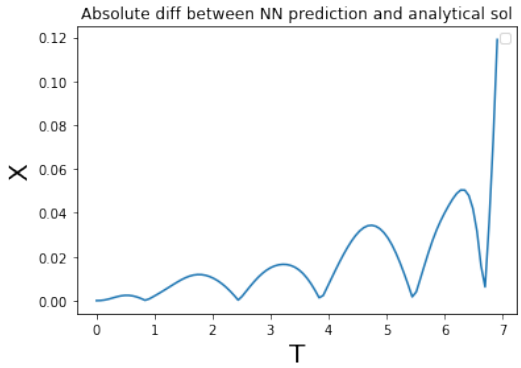

Figure 8: Difference between analytical solution and prediction

From this, we conclude that the network is not able to predict outside the training domain and predicts increasingly worse further away from the initial condition. Next, we will dive deeper into potentially incorporating periodicity into the network, in the hope to expand our performance outside the training domain.

4.3 Chebyshev Interpolation

I wondered whether it matters how you are interpolating the points in the t-domain. In the analysis above, a uniformly distrubuted sampling was used. Given that Chebyshev interpolation has the characteristic of minimizing the interpolation error when approximating functions by polynomials, this might be result in a better performance of the network. Let's find out.

Recall the Chebyshev interpolation on an interval $[a, b]$:

$$x_k = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2k-1}{2n}\pi\right), \quad k = 1, \ldots, n. \tag{40}$$

After training the network, the NN predictions with the Chebyshev interpolation points are illustrated in the figure below. The points tend to be more concentrated towards the ends of the interval.

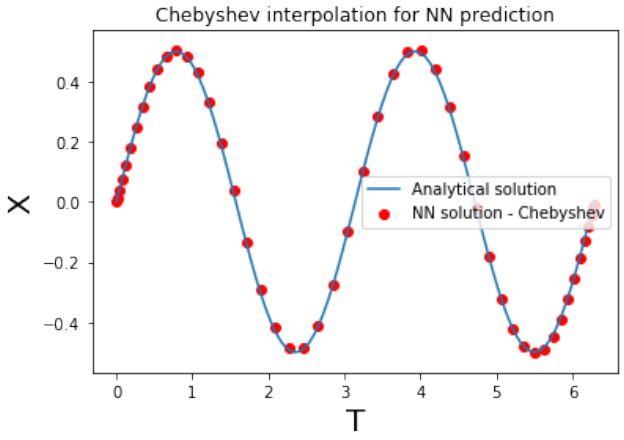

Figure 9: NN prediction with Chebyshev points

Different interpolation methods can only be compared when looking at the losses over amount of epochs. In the following three graphs, I plotted this for three different ODE's: one harmonic oscillator with $\omega = 2$, one with $\omega = 4$ and one exponential decaying function. From all three cases, the Chebyshev interpolation does not seem to have any particular advantage, which is unfortunate. Looking back at Figure 9, this might make sense, as we are actually sampling more close to the initial condition, which is already fit closely through the initial condition operator. In this regard, it might be smart to sample more the further away from the initial condition, but this should be tested (if relevant). In any case, the Chebyshev remains a nice interpolation option to include for the user, as it still might have better performance in particular application.

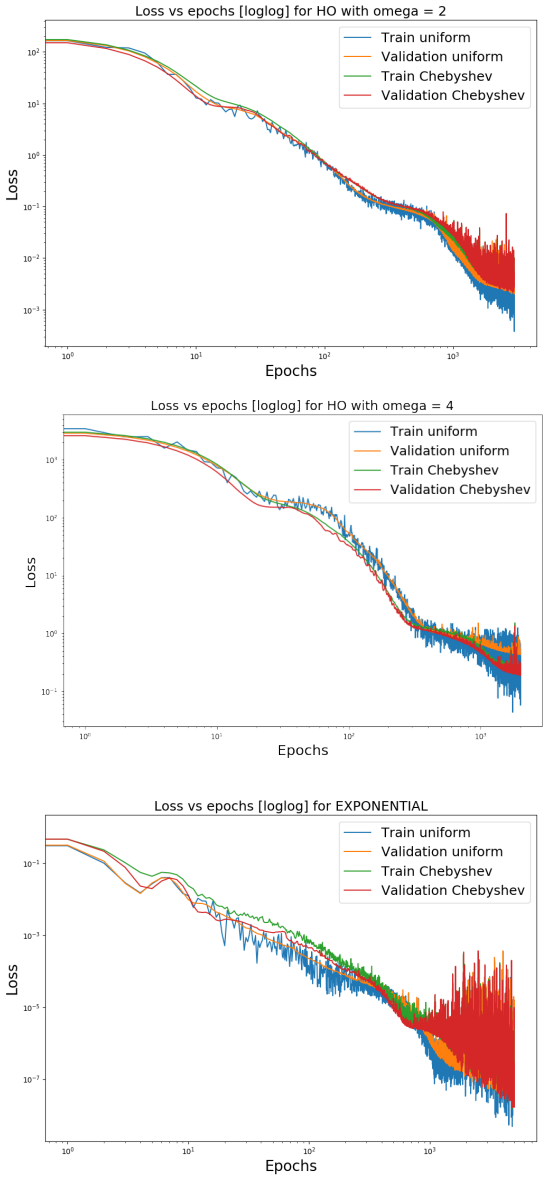

Figure 10: Loss vs epochs for different ODE's, Chebyshev vs uniform

5 Experimenting with Pavlos' Masternode

Last week, Pavlos proposed to implement one masternode in front of the neural network in order to incorporate periodicity of a solution.

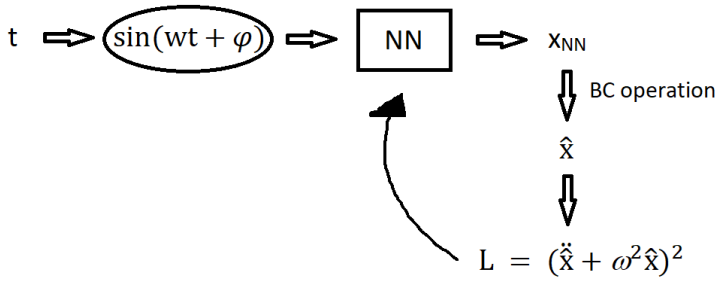

Figure 11: NN Architecture with masternode

From the figure above, it is clear that the input parameter is first 'activated' by a sine function with a certain frequency and phase that are incorporated as weights to be trained. As such, the hope is to have periodic output functions in t. This would make sense, as:

$$sin(wt + \phi) = sin(wt + \phi + 2 * \pi) \tag{41}$$

As such, every $\frac{2\pi}{w}$, the output of the masternode and thus the input of the remaining neural network will be same, resulting in a periodic output.

However, recall the initial value operator:

$$\hat{x} = x_0 + (1 - e^{-t})v_0 + (1 - e^{-t})^2 x_{NN} \tag{42}$$

Despite the intrinsic periodicity introduced by the masternode, the initial value operator has non-periodic terms in t, therefore making it impossible to recover a periodic prediction from the NN. This could be solved by looking into an alternative initial value operator, that is periodic in t, and also enforces the initial values of the ODE. The general form is:

$$\hat{x} = x_0 + f_1(t)v_0 + f_2(t)x_{NN} \tag{43}$$

Where f_1 and f_2 should both be periodic in t, and $f_1(0) = 0$, $f_2(0) = 0$, $f_1'(0) = 1$ and $f_2'(0) = 0$. Before going any further, it is important to note that the periodicity in t only makes sense when it's consistent in the masternode and the initial value operator. If we call the first weight used as frequency in the masternode w_0, we should have the same periodicity in t for both functions f_1 and f_2. Keeping this in mind and the specified conditions, the following operator is proposed:

$$\hat{x} = x_0 + \frac{1}{w_0}\sin(w_0 t)v_0 + \sin^2(w_0 t)x_{NN} \tag{44}$$

It is important to notice that the weight w_0 from the masternode now returns in the initial condition operator, and thus in the loss function.

It took a while to incorporate this into Feiyu's code but I believe it worked. The following graphs illustrates the loss in the training domain over the epochs for three methods: the original code with no masternode and the non-periodic initial value operator, the code with the masternode and the non-perodic operator and the code with the masternode and the periodic operator.

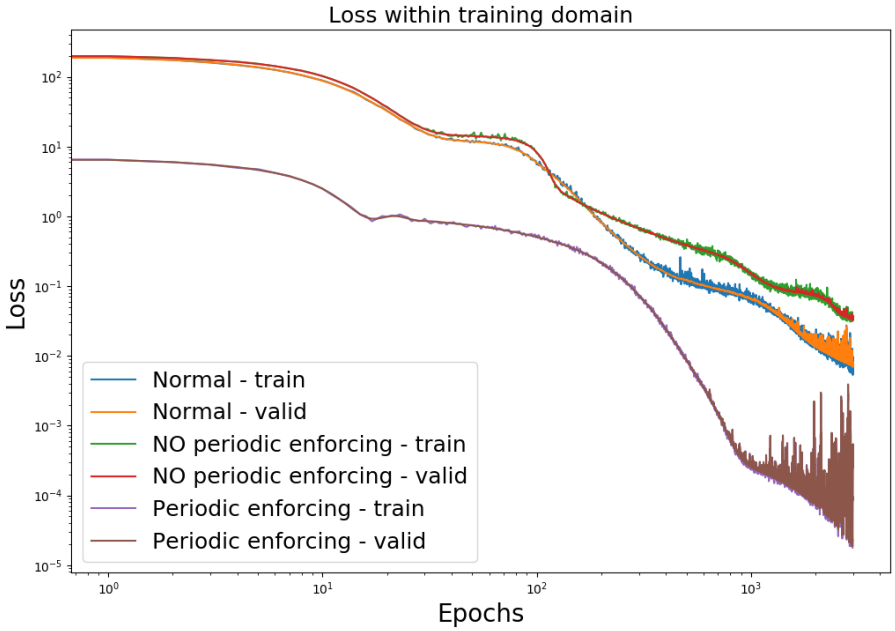

Figure 12: Losses for masternode design

This graph is very interesting! First, it is clear that including the masternode, without adapting the initial value operator, leads to worse performance. This makes sense in my opinion, as you're only partially forcing periodicity. Interestingly, the loss of the network with the masternode and the periodic initial value operator starts with a lower value and decreases faster than the other networks. The first makes sense, as you're using a different initial value operator and thus a different loss function - which appears to be lower. For more iterations, the network seems to benefit from the enforced periodicity, leading to significantly lower values.

Note that the loss computed above is the loss in the training domain. It is now interesting to verify the performance outside the training interval, which is illustrated on the graph below. As expected, the network with masternode and periodic initial value operator is entirely periodic in t, therefore predicting equally well outside the training domain, while the other networks do not have this periodicity and perform very poorly as discussed previously.

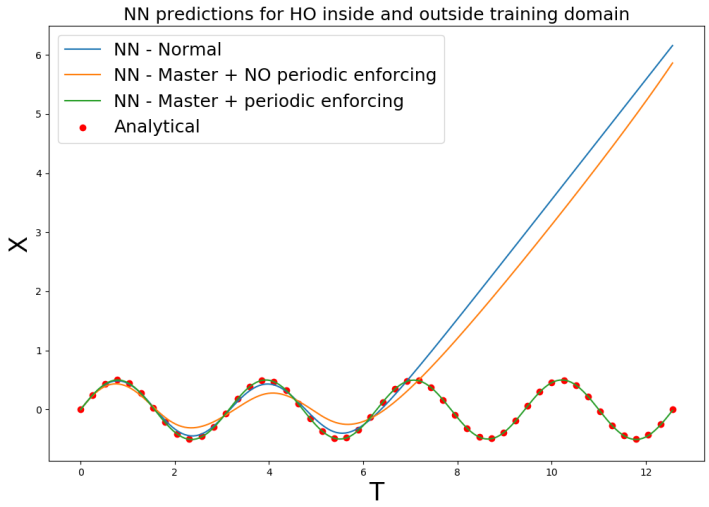

Figure 13: Outside training domain performance for masternode design

We can also plot the error between the predicted values and the corresponding analytical solution. This leads to the following graph:

Figure 14: Errors inside and outside training domain

Knowing that the training domain for the network above was between 0 and 2π, the errors are relatively small even far from the initial condition and outside the training domain. This accuracy was not possible to achieve with other non-periodic implementations.

It is clear that the masternode potentially has some positive impact on performance. Before making any statements, we need to dive deeper, which will be done in the following paragraphs.

First, it is crucial to know whether the enforced periodicity that is used as a trainable parameters is predicted correctly by the neural network. Therefore, the analysis is repeated for the same ODE as before but with different values of ω. The following table lists both the trained value of the masternode weight w_0 as the loss in the training domain associated with the network prediction.

Omega	Masternode weight	Loss after 7k epochs
0.75	-0.25000116	1.8429933e-08
1	-1.0000070333	2.69562495e-07
1.25	-0.416494817	4.952709787e-05
1.5	0.49993616	1.56429246e-06
1.75	-0.350078344	0.0073964195
2	0.40003186464	6.240822131e-06
2.25	-0.322204768	0.01389456
2.5	0.833506167	0.00035869

Table 1: Caption

In none of the cases above, the model is able to exactly predict the value of omega. Interestingly, the model is still able to achieve very low losses with these 'wrong' frequencies. Let's have a look at the evolution of the loss vs epochs for the model with ω equal to 0.75 and 2.5.

It is great to see that the loss function decreases faster for the network with the masternode, and that the latter's prediction shows some periodicity which leads to good predictions also outside the training domain. Note that this periodicity arises despite the frequencies being entirely wrong. My thoughts are that this might be because of the goniometric relationships between sines and cosines and multiples of their angles. For instance, the predicted frequencies for $\omega = 0.75$ and $\omega = 1.5$ correspond to approximately $\frac{1}{3}$ of the real value, while this seems to be a fraction of $\frac{1}{5}$ for $\omega = 1.75$ and $\omega = 2$. This would mean that the masternode could predict a frequency or a fraction of a frequency for a periodic function.

Next, it is necessary to verify whether the solutions show true periodicity. The definition of a periodic function $x(t)$ is that there exists some non-zero constant P for which:

$$x(t + P) = x(t) \tag{45}$$

which should count for all t. If we look at the figures above, this might be the case. However, during the iteration over values for ω, the following plots arose as well:

Looking at the predictions of the masternode outside the training domain, it seems that the solution is not automatically periodic in time. In order to be entirely sure, we should maybe look at the period associated with the predicted frequency. We know that the following relationship holds:

$$T = \frac{2\pi}{\omega} \tag{46}$$

In both cases above, the values of the predicted frequencies (the masternode weights) lead to $T \approx 15$ and $T \approx 17.5$. That means that the periodicity cannot be judged based on

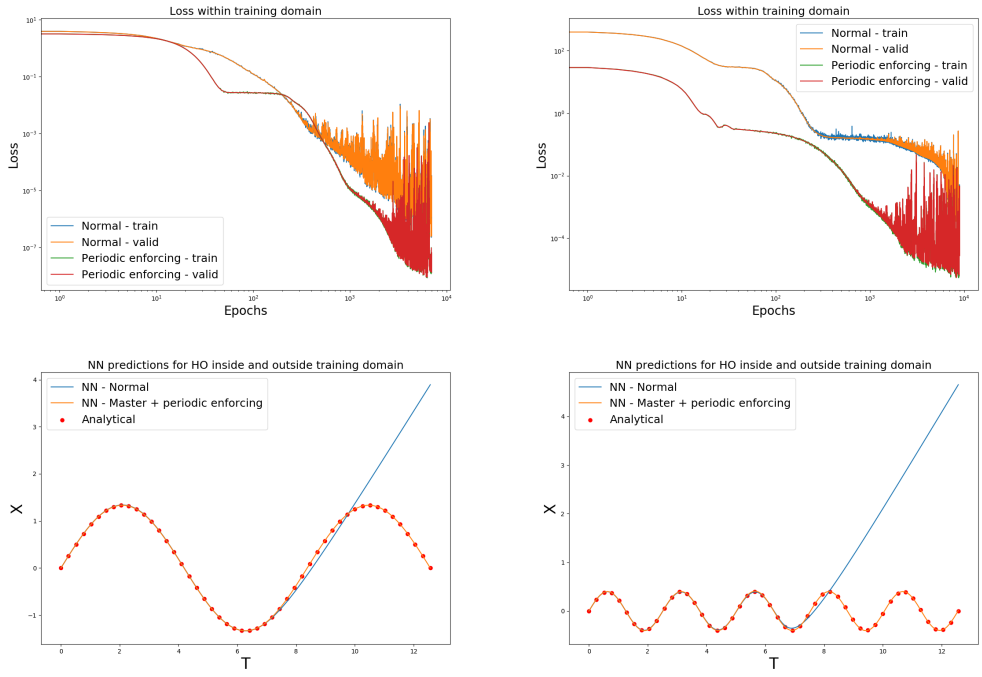

Figure 15: Loss (above) and solution (below) for omega equal to 0.75 and 2.5

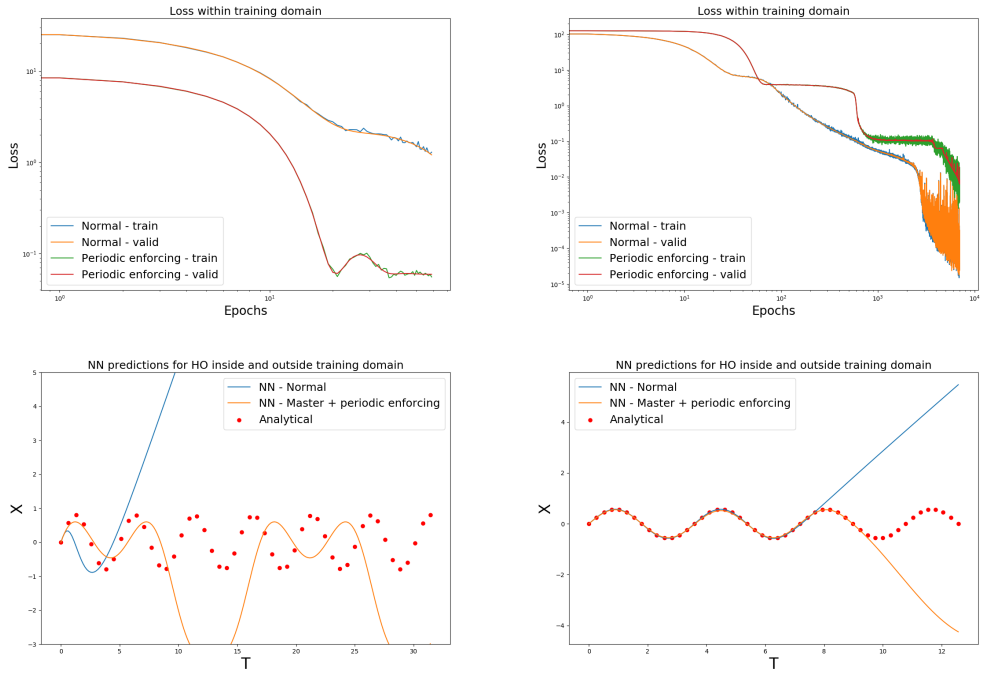

Figure 16: Loss (above) and solution (below) for omega equal to 1.25 and 1.75

the graphs above and that the domain in t should be expanded. Let's have a look now:

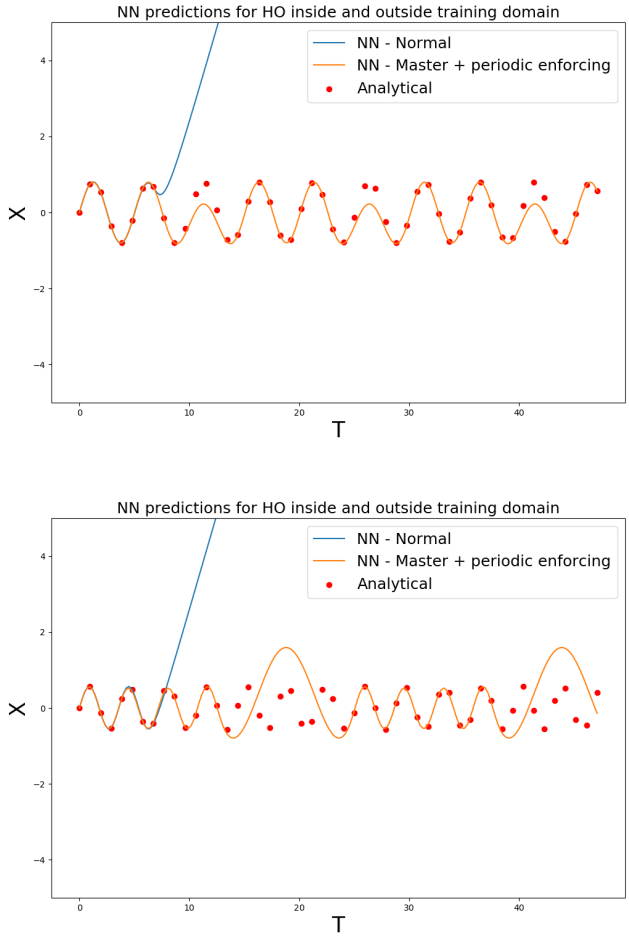

Figure 17: NN solutions far outside the training domain for omega equal to 1.25 and 1.75

These graphs show that, how bad the prediction of the frequency may be, there will always be some periodicity in time for the masternode network solution.

From this analysis, we could make some conclusions:

- The mathematical formulation of the network is intrinsically periodic with a frequency as trainable parameter.
- In the cases tested thus far, this seems to have a positive impact on the loss vs epochs evolution.
- Despite the two points above, the masternode network becomes incredibly sensitive with respect to the value of the frequency. This raises the question whether there can be put more 'weight' on the training on this masternode weight/frequency. Should

we/I think about a clever way to implement this in the loss function? Or is this not useful direction to follow?

6 Non-linear ODE's

It is now interesting to assess the performance of the model for non-linear Ordinary Differential Equations. With the purpose of still evaluating the masternode, two non-linear, but periodic ODEs have been chosen.

6.1 Van der Pol Equation

Firstly, the Van der Pol equation has been chosen:

$$\frac{d^2x}{dt^2} - \mu(1-x^2)\frac{dx}{dt} + x = 0, \tag{47}$$

In dynamics, the Van der Pol oscillator is a non-conservative oscillator with non-linear damping. The scalar parameter μ determines the strength of the non-linear damping. For μ non-zero, the equation does not have an analytical solution. The figure below illustrates the numerically computed solution for $\mu = 5$ (source).

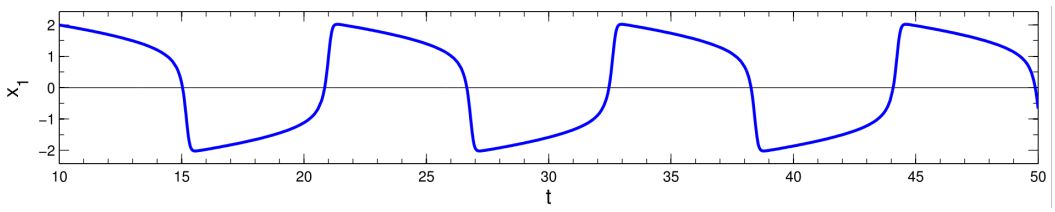

Figure 18: Solution to the Van der Pol equation for $\mu = 5$

Clearly, the solution shows periodicity in the time domain. Hence, together with the fact that the solution does not contain any goniometric functions, this is the perfect function to test the functionality of the neural network with the masternode and the periodic initial value enforcing.

The loss function now becomes:

$$L = (\ddot{\hat{x}} - \mu(1-\hat{x}^2)\dot{\hat{x}} + \hat{x}) \tag{48}$$

Where $\hat{x}$ still comes from the same operation executed on the output of the neural network x_{NN}:

$$\hat{x} = x_0 + \frac{1}{w_0}\sin(w_0 t)v_0 + \sin^2(w_0 t)x_{NN} \tag{49}$$

Recall the unsatisfying results from last week, illustrated in the figure below. Note that this was due to the memory issue with the ODE code, which made it impossible to go beyond 500 epochs.

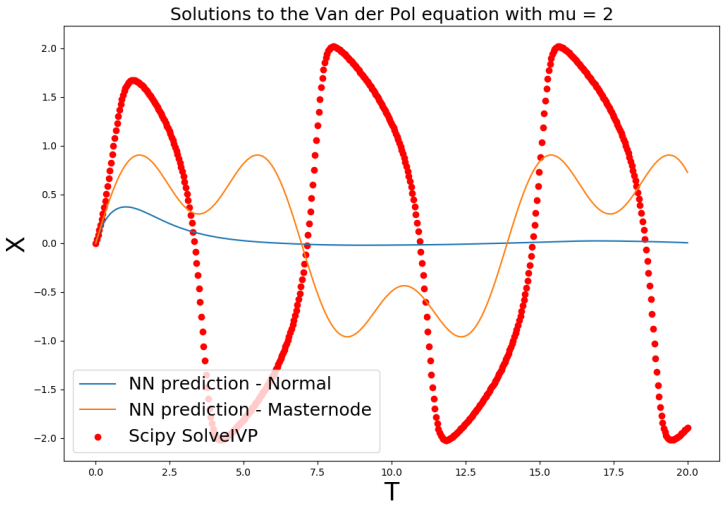

Figure 19: Prediction of the Van der Pol equation for $\mu = 2$

Two things are now tried to increase the performance of the model. First the memory problem was easily fixed with the help of Feiyu. This made it possible to increase the number of epochs to 10,000 and above. Also, the complexity of the network can be upgraded. With 10 layers, and 10,000 epochs, we still get an unfortunate result, given in the figures below:

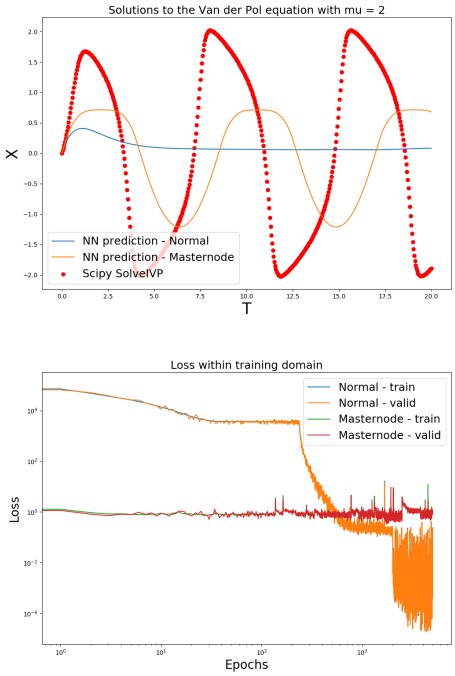

Figure 20: Prediction of the Van der Pol equation for $\mu = 2$

6.2 Duffing Equation

A somehow less strongly non-linear equation would be the Duffing equation. The general formulation is given by:

$$\ddot{x} + \delta\dot{x} + \alpha x + \beta x^3 = \gamma\cos(\omega t) \tag{50}$$

From https://en.wikipedia.org/wiki/Duffing$_e quation$, we get the following physical meaning behind the parameters in the non-linear oscillation $x(t)$:

- δ controls the amount of damping,
- α controls the linear stiffness,
- β controls the amount of non-linearity in the restoring force; if $\beta = 0$, the Duffing equation describes a damped and driven simple harmonic oscillator,
- γ is the amplitude of the periodic driving force; if $\gamma = 0\gamma = 0$ the system is without a driving force,
- ω is the angular frequency of the periodic driving force.

For $\alpha = 3$, $\beta = 0.2$, $\delta = 0$, $\gamma = 2$ and $\omega = 7$, we get a moderately non-linear, periodic solution. The graph below illustrates the solution (through Scipy IVP), the normal neural network prediction and the prediction with the masternode. The second plot illustrates the corresponding losses.

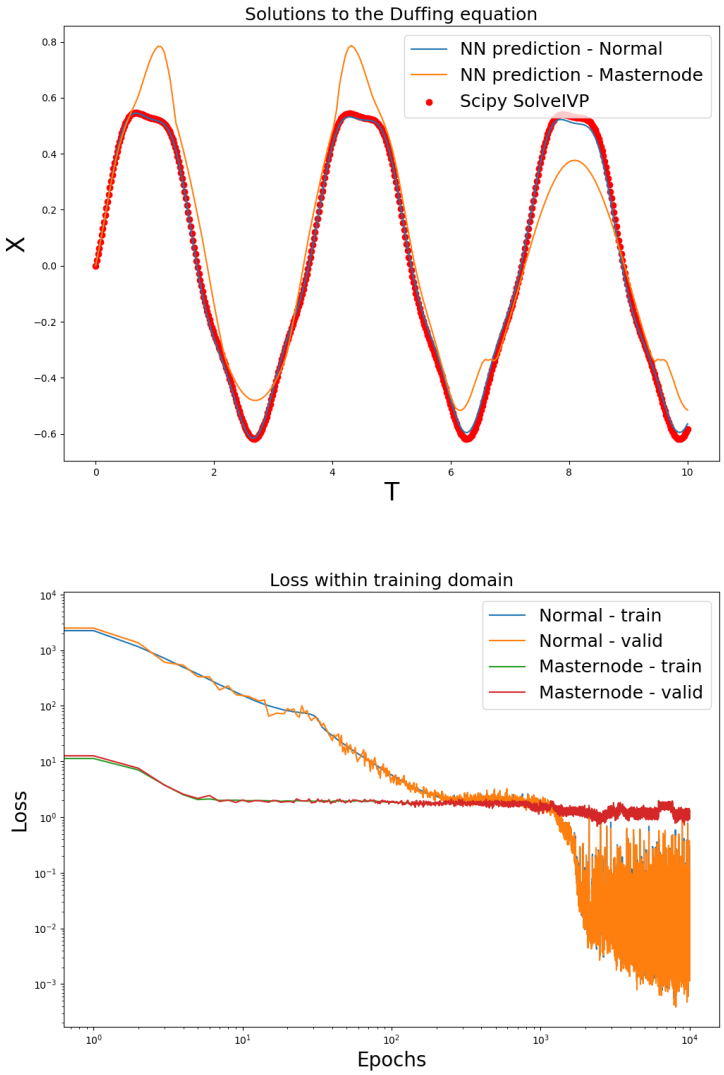

Figure 21: Prediction of the Duffing equation

The network trained in the figure above had 3 hidden layers with each 32 nodes, being trained for 10,000 epochs. Clearly, the normal neural network implementation is able to reach very low losses, while this is not the case for the masternode version.

7 2D Boundary Value Problems

7.1 Easy Laplace to start with

Last week, I started to use Feiyu's code to solve two-dimensional boundary value problems. First, let's just solve the two dimensional Laplace equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{51}$$

The boundary conditions are the following:

$$\begin{aligned} u(x,y)\Big|_{x=0} &= \sin \pi y \\ u(x,y)\Big|_{x=1} &= 0 \\ u(x,y)\Big|_{y=0} &= 0 \\ u(x,y)\Big|_{y=1} &= 0 \end{aligned} \tag{52}$$

The analytical solution is the following:

$$u(x,y) = \frac{\sin(\pi y)\sinh(\pi(1-x))}{\sinh \pi} \tag{53}$$

Implementing this in Feiyu's code results in the following graphs:

Both the low loss for a relatively low amount of epochs and the smooth solution, with the boundary conditions seemingly met, this looks very good. This initial (brief) exploration will enable us to go further into the PDE's.

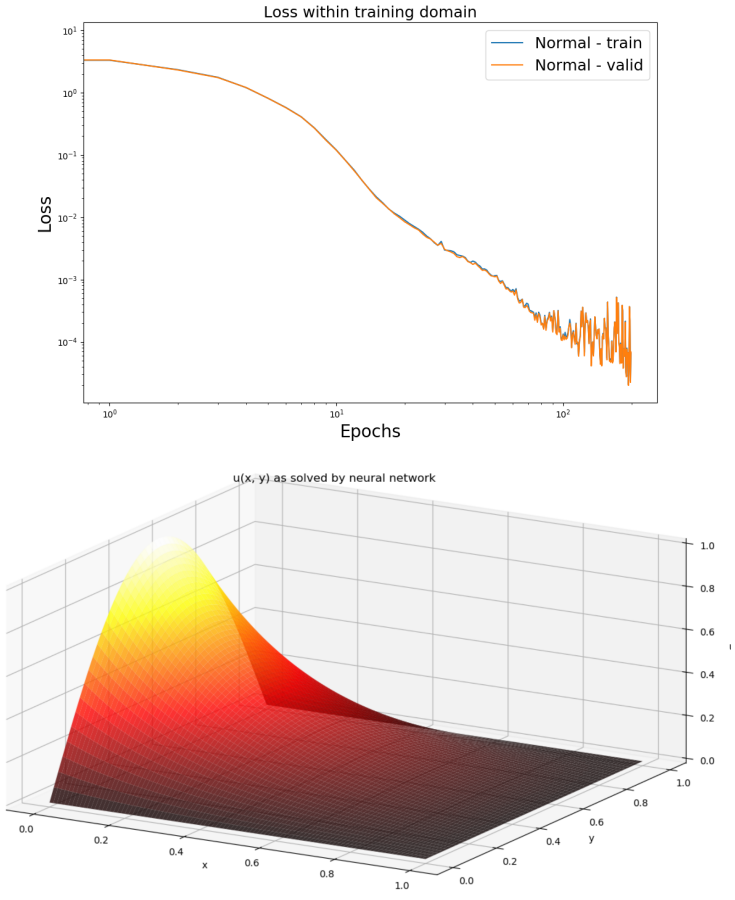

Figure 22: NN solutions of the two dimensional Laplace equation

8 The 2D Helmholtz equation

Now the initial exploration for two dimensional boundary values has been done, we can shift to a more complicated problem: the Helmholtz equation. This equation is a time-independent form of the wave equation and is given by:

$$\nabla^2 u = -k^2 u \tag{54}$$

Let's consider this equation in a circular, 2D space and formulate the problem in polar coordinates r, θ:

$$\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} + k^2 u = 0 \tag{55}$$

8.1 Analytical Solution

Using the classic separation of variables as we are now familiar with, we get an idea of the analytical solution for $u(r, \theta)$. As the problem shows similarity with the three-dimensional one solved previously, we can go through the math quite fast. Starting with $u(r, \theta) = R(r)T(\theta)$, we get the two following ODE's:

$$T'' + n^2 T = 0 \tag{56}$$

$$r^2 R'' + rR' + r^2 k^2 R - n^2 R = 0 \tag{57}$$

With the solution being periodic in θ, we get the following general solution for $T(\theta)$, with n being an integer:

$$T(\theta) = A_n \cos(n\theta) + B_n \sin(n\theta) \tag{58}$$

For the ODE $R(r)$ we will get a linear combination of Bessel functions of the first and second kind and order n. For $k = 1$, we get as general solution:

$$R_n(r) = C_n J_n(nr) + D_n Y_n(nr) \tag{59}$$

Putting it all together, we know that the general solution should look like the equation below. Note that the exact values for the constants depend on the specified boundary conditions.

$$u(r, \theta) = A_0 + \sum_n \Big(A_{1n} J_n(nr) + A_{2n} Y_n(nr)\Big) \cos(n\theta) + \sum_n \Big(B_{1n} J_n(nr) + B_{2n} Y_n(nr)\Big) \sin(n\theta) \tag{60}$$

8.2 Neural Network solution

8.2.1 Basic implementation

We can now proceed solving the equation starting from the NN PDE package developed by Feiyu. Based on the sphericalPDE module, I started a new module 'polarPDE', which should be able to solve two-dimensional PDE's in polar coordinates, so in r and θ. The

changes needed to shift from spherical to polar were not too complicated. Importantly, I noted that the spherical module does not enforce periodic boundary conditions in θ or ϕ and I will first use the same boundary condition enforcer but translated to 2D polar coordinates. When the following boundary conditions hold:

$$\begin{aligned} u(r,\theta)\Big|_{r=r_0} &= f(\theta) \\ u(r,\theta)\Big|_{r=r_1} &= g(\theta) \end{aligned} \tag{61}$$

The following enforcing transformation is executed to go from the NN prediction u_{NN} to the $\hat{u}$:

$$\hat{u} = f(\theta)(1-\tilde{r}) + g(\theta)\tilde{r} + (1 - e^{(1-\tilde{r})\tilde{r}})u_{NN} \tag{62}$$

Where:

$$\tilde{r} = \frac{r - r_0}{r_1 - r_0} \tag{63}$$

Note that this successfully enforces the solution to meet the boundary conditions as specified above. However, this does not force the output to be periodic in θ, which might lead to problems in the solution. Let's first implement it as specified above and find out if we run into any problems. The following specific boundary conditions are considered for the Helmholtz equation:

$$\begin{aligned} u(r,\theta)\Big|_{r=r_0} &= f(\theta) = 1 \\ u(r,\theta)\Big|_{r=r_1} &= g(\theta) = sin(3\theta) \end{aligned} \tag{64}$$

The solutions and the according loss vs epoch are illustrated in the figures below:

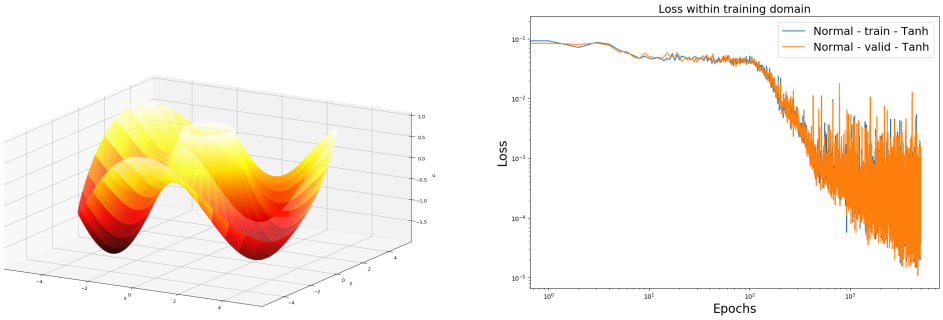

Figure 23: NN solutions of the two dimensional Helmholtz equation

8.2.2 Periodic boundary enforcing

The network above was trained for 5000 epochs and it looks like it is fairly well able to figure out periodicity in θ. Let's take a look at the results of the neural network when it

is only trained for 200 epochs. The two figures below illustrate the results in 3D and the difference between the solution for $\theta = 0$ and $\theta = 2\pi$ after 200 epochs:

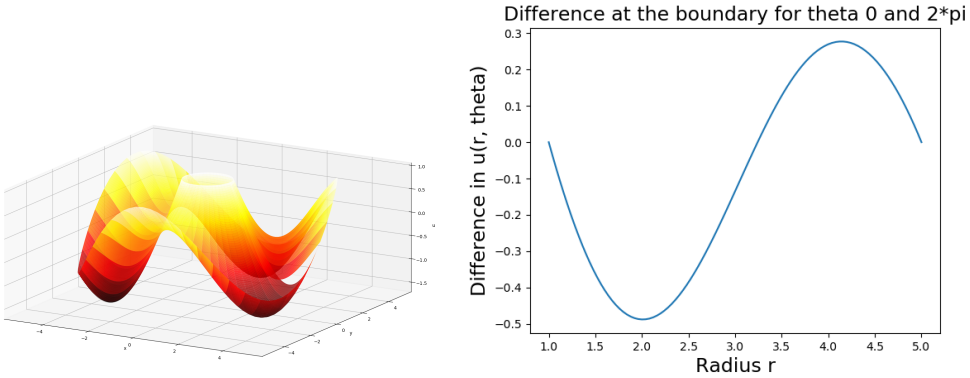

Figure 24: NN solutions of the two dimensional Helmholtz equation for small amount of epochs (200)

Here, we see that the periodicity in θ is not automatically satisfied. Assuming this would be beneficial for the training, we might think on potential ways to enforce this. After giving it some thought, I came up with the following three ideas:

- Implement the masternode, as tried in the ODE initial value problem in previous paragraphs, but only for the θ input variable.
- Embedding the periodicity in the loss function.
- Increase the θ domain from 2π to 4π and see whether the network can figure out periodicity like that.

Let's try the first idea, being the masternode for only the θ input variable. This means that, instead of the normal input θ, we use $sin(\theta)$ as input. As such, periodicity is ensured. This leads to the following results, where the first graph corresponds to the solution after only 10 epochs, and clearly illustrates that the periodicity is ensured from very early on. However, the loss function appears to decrease more slowly than before.

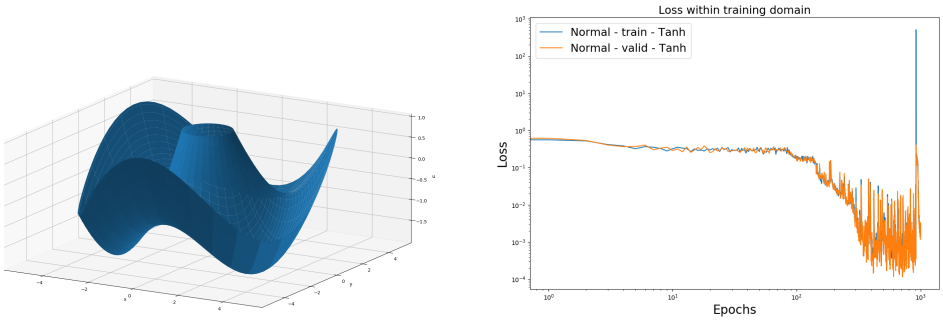

Figure 25: NN solutions of the two dimensional Helmholtz equation with the masternode

The next thing I'd like to try is to embed the periodicity in the loss function. Note that the loss function so far was equal to the mean squared error (MSE) of the solution $\hat{u}$ plugged into the PDE and an array of zeros. We can now try to embed the periodicity requirement inside the loss function. In the following trial, I added an MSE of the NN predictions for a range of radii and $\theta = 0$ and $\theta = 2\pi$. This should drive the the training towards periodicity, as this will minimize the loss function. Note that a hyperparameter can be tuned to increase the effect of this periodic boundary condition operator. Let's have a look at what this does to the periodic boundary condition after only 20 epochs (left) and to the loss function during training:

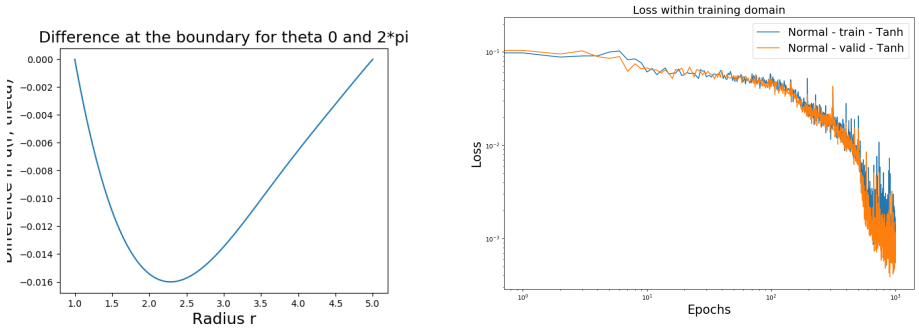

Figure 26: Solutions for the penalization of the boundary conditions in the loss function

Note that these errros are two order of magnitudes smaller than the ones in Figure 24, which were the results of the network with the regular loss function just after 200 epochs. This means that, although periodicity is not strictly ensured, it is possible to force the model to enforce it more and more during training. The following figure illustrates the effect on the training error, which looks actually more stable and faster decreasing than the one in Figure 23.

Lastly, I tried to expand the training domain in θ from the usual $[0, 2\pi]$ to the increased $[0, 4\pi]$. With this, I hope that the network is able to find the periodicity faster, as this should be inherently present in the PDE. For an equal amount of training points, but now

in the larger interval, and equal amount of epochs as in Figure 24, we get:

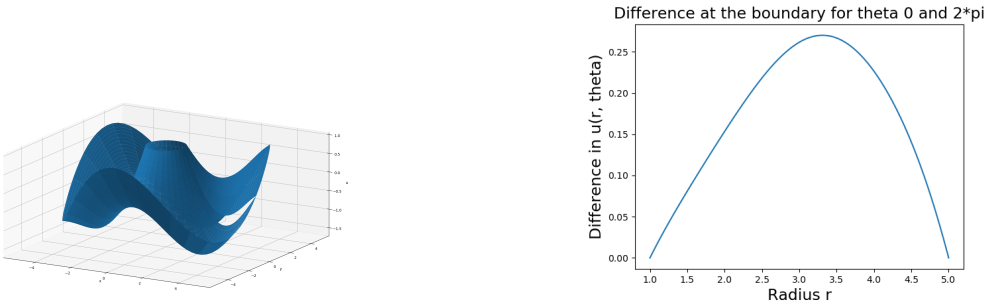

Figure 27: NN solutions of the two dimensional Helmholtz equation for small amount of epochs and (200) $[0, 4\pi]$

Apparently, the periodic boundary conditions are not more satisfied than previously.

8.2.3 Activation functions

First, let's explore the results of the 2D Helmholtz equation in a 2D countourplot plot the boundary conditions at $r = 5$. This is done in the figure below for the basic network with Tanh and 5000 epochs.

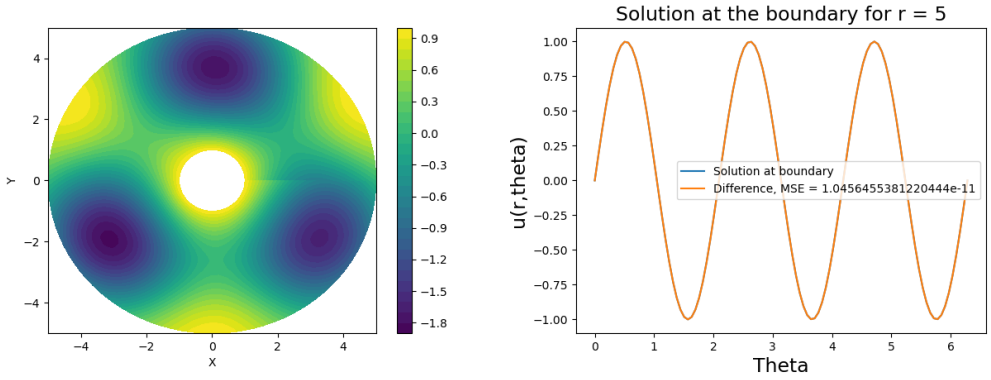

Figure 28: Contourplot and the boundary conditions at r=5 of the two dimensional Helmholtz equation

We conclude that the contourplot is indeed an interesting way to visualize the 2D cylindrical solutions. The solutions seem to be smooth, with a slight imperfection at the periodic boundary condition for $\theta = 0$ and $\theta = 2\pi$. From the second figure above, it is clear that the boundary condition is perfectly met thanks to the enforcing, with an inaccuracy MSE of magnitude 10^{-11}.

We will now move forward using the basic code, so with no enforcing for the periodic boundary conditions.

Looking at the analytical solution in equation (60), we might think of potential benefits of changing the activation functions inside the neural network. In all the analyses above, Tanh has been used to introduce non-linearity in the network, and it will be interesting to compare its performance to the one of network using different activation.

First, we will use Sine as activation function throughout the entire network. We will repeat this for using zero, one, two and three hidden layers and compare the loss function vs epochs of the network with Sine activation and Tanh activation. The following four figures summarize all results:

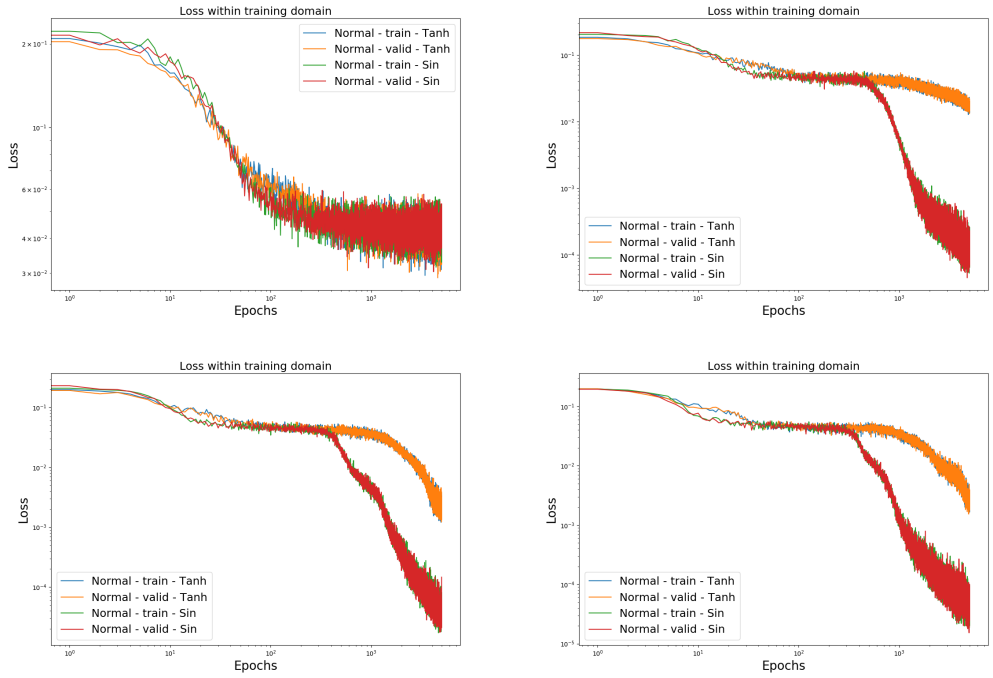

Figure 29: Comparison of Tanh vs Sine activation for 0,1,2 and 3 hidden layers top left, top right, bottom left and bottom right respectively

Clearly, the performance of the network using Sine as activation function is significantly better! For 0 hidden layers, nothing particular changes but for an increasing amount of layers, the loss function decreases more steeply and for a smaller amount of epochs. This is a very cool result. Let's take the network with three layers and train it for 50,000 epochs instead to 5,000:

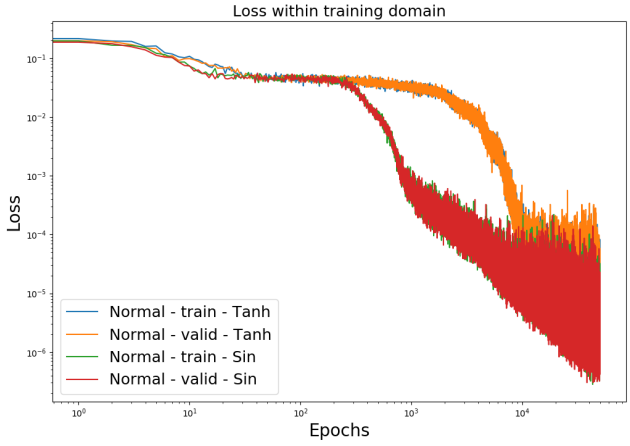

Figure 30: Network performance for sine vs tanh

From this extended training time, it is still clear that the network with Sine as activation leads to a loss function that decreases more rapidly and is able to reach lower minima $(10^{-7} - 10^{-6})$.

Next, we can wonder whether such a good performance would also arise when we use Bessel functions as activation. Thanks to [source], we can have a look at how the Bessel and modified Bessel functions of order 1,2 and 3 look like:

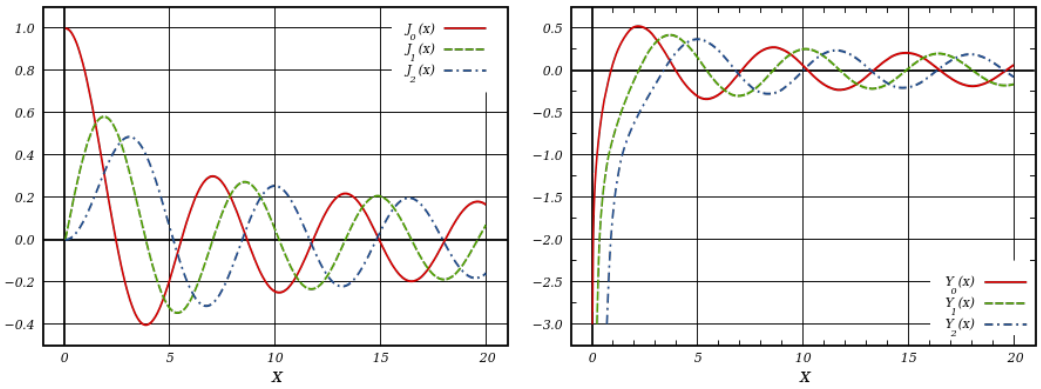

Figure 31: Bessel functions and modified Bessel functions of the 1st, 2nd and 3rd order

It is clear that the modified Bessel functions reach negative infinity for x approaching zero. This makes it hard to use it as activation function. We will therefore try to use the normal Bessel functions $J_n(x)$ as activation function for $n = 1, 2$ and for one and two hidden layers. The following figure illustrates how the validation loss decreases vs epochs in all cases:

Given that we trained the network for 5000 epochs, the resulting loss is quite high in all

Figure 32: Validation loss for the 2D Helmholtz equation using Bessel functions as activation

four cases. Therefore, it seems that using Bessel functions as activation function does not lead to a similar performance boost as using Sine.

Lastly, we will examine what happens if we combine the Sine and the Bessel function as activation. We will train two networks, both existing of two hidden layers with each a different activation. One network will first be activated with the Bessel function of the first kind (which seemed to perform slightly better on the last figure) and then with Sine. The other network will do the opposite. The results are illustrated below:

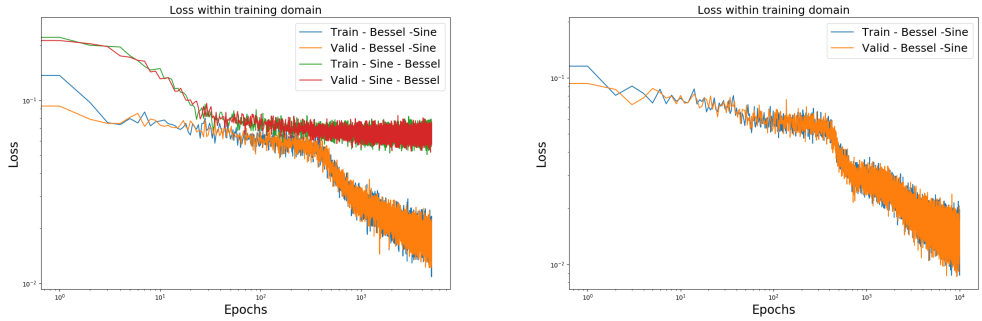

Figure 33: Validation loss for the 2D Helmholtz equation using Bessel and Sine as activation (5k and 10k epochs)

Clearly, the network with first Bessel and then Sine as activation performs better and is not done training. Let's do another 5000 epochs:

Unfortunately, the performance of the combination of Bessel and Sine is still very poor. It is definitely better to just use the Sine as activation function.

8.3 Road to heterogeneous layers

8.3.1 Analytical solution

With an eye on incorporating Bessel functions as activation functions, it is important to know which ones in the analytical solution are important. In this section we will solve the 2D Helmholtz equation analytically and check how the solution changes for an increasing amount of terms. We will use the following Dirichlet boundary conditions on a finite disk:

$$\begin{aligned} u(r,\theta)\Big|_{r=r_0} &= f(\theta) = 0 \\ u(r,\theta)\Big|_{r=r_1} &= g(\theta) \end{aligned} \tag{65}$$

The first, homogeneous condition leads to:

$$\frac{D_n}{C_n} = -\frac{J_n(nr_0)}{Y_n(nr_0)} = \kappa_n \tag{66}$$

The general solution can now be formulated as follows:

$$u(r,\theta) = A_0 + \sum_n A_n\Big(J_n(nr) + \kappa_n Y_n(nr)\Big)\cos(n\theta) + \sum_n B_n\Big(J_n(nr) + \kappa_n Y_n(nr)\Big)\sin(n\theta) \tag{67}$$

IF we wish to satisfy the second boundary condition at $r = r_1$, we get an easy Fourier series that approximates $g(\theta)$ for n approaching ∞. The coefficients A_0, A_n and B_n are now determined as follows:

$$\begin{aligned} A_0' = A_0 &= \frac{1}{\pi}\int_0^{2\pi} g(\theta)d\theta \\ A_n' = A_n\Big(J_n(r_1) + \kappa_n Y_n(r_1)\Big) &= \frac{1}{\pi}\int_0^{2\pi} g(\theta)cos(n\theta)d\theta \\ B_n' = B_n\Big(J_n(r_1) + \kappa_n Y_n(r_1)\Big) &= \frac{1}{\pi}\int_0^{2\pi} g(\theta)sin(n\theta)d\theta \end{aligned} \tag{68}$$

As such, each coefficient can be determined for each value of n.

After playing around with multiple expressions for $g(\theta)$, I decided to go for:

$$g(\theta) = sin(3\theta) * cos(\theta) \tag{69}$$

This function is periodic and is not very straight forward to approach by Fourier series. I implemented the code to compute this solution, for a different value of n. Note that for instance for $N = 2$, this means that all Bessel functions order less than or equal to 2 are incorporated, with each a cosine and sine term. Let's first take a look at the values for A_n and B_n.

Order of Bessel n	κ_n	A_n (cosine)	B_n (sine)
0	-8.6701	-1.0859e-16	0.0
1	0.5633	-1.8843e-17	-4.2392e-17
2	0.0696	3.8022e-16	6.9292
3	3.3605e-3	-1.2074e-16	1.5309e-16
4	7.4422e-05	7.6768e-16	1.2781
5	9.5911e-07	-8.0464e-16	-2.2079e-16

Table 2: Coefficients of analytical solution for increasing n

The following figure shows how the solution changes for increasing amount of terms, so for $N = 1, 2, 3, 5$. On the left side, the 3D solution is plotted, in the middle the corresponding 2D contourplot and on the right side a comparison of the actual boundary condition $g(\theta)$ and the Fourier series approximation.

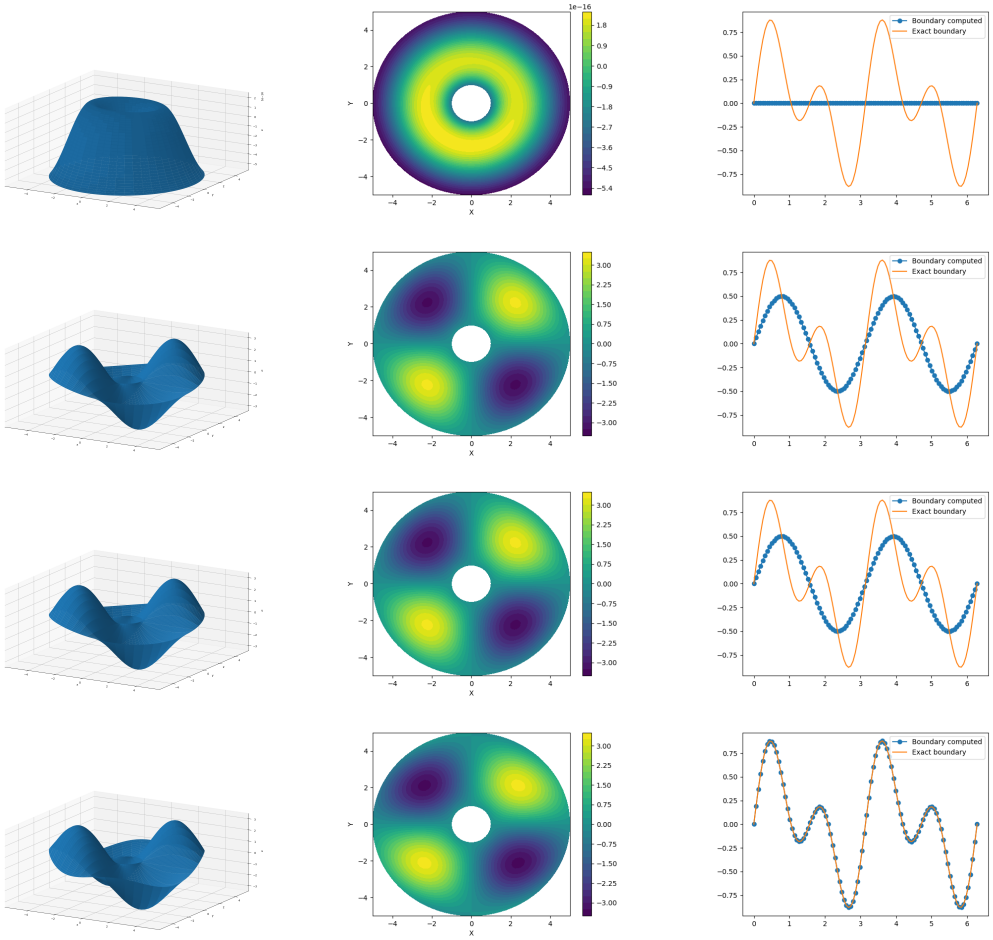

Figure 34: Analytical solution with terms up until Bessel functions of order N = 1, 2, 3 and 5. Left is the 3D solution, mid is the contourplot and right is the boundary condition at $r = r_1$

From both the table and the figures below, we can detect the most important modes/terms in the analytical solution. With all values of A_n close to machine precision, the solution does not contain any relevant cosine waves. Only two values for B_n are of significant magnitude, being B_2 and B_4. These correspond to the sines with frequencies 2 and 4. Let's take a look at the trigonometric identity that transforms a sum of sines to a product and vice versa:

$$sin(\alpha) + sin(\beta) = 2sin(\frac{\alpha + \beta}{2})cos(\frac{\alpha - \beta}{2}) \tag{70}$$

Looking more closely, it is clear that we can reduce the product $g(\theta) = sin(3\theta) * cos(\theta)$ into a sum of two sines with frequencies 2 and 4. It's cool to see that this is verified by the analytical solution.

Also, it is interesting to have a look at the values for κ. For increasing n, this decreases. This means that for increasing value of n, the Bessel function of the first kind is increasingly more important than the one of the second kind.

8.3.2 Fourier series

Next, we wish to play around with heterogeneous layers in the network. This means that we incorporate multiple activation function inside the same layer, for distinct nodes. The first, intuitive thing to do is to mimic the Fourier series approximation. From mathematical theory, we know that each function can be approximated by an infinite sum of sines and cosines with increasing frequencies. This inspires us to build a neural network with one heterogeneous layer, consisting of an equal amount of nodes activated by a sine and cosine. With the weights and biases corresponding to a frequency and phase shift, it is possible that the network could approach the solution in a similar way as the Fourier series. This is done for an increasing amount of nodes in 1 layer for the 2D polar Helmholtz equation with the boundary condition $g(\theta)$ as specified in equation (69). The following figure illustrates the results in terms of training loss:

Figure 35: Fourier network performance for different number of nodes

This was the graph of last week. It was then recommended to check what happens if we would take 16 and 32 sines and cosines in a layer. This is illustrated below:

My take-away with these would be that for a good amount of nodes (64+), we can already reach low losses 10^{-3} for a fairly limited amount of epochs ($10^2 - 10^3$) and only one layer.

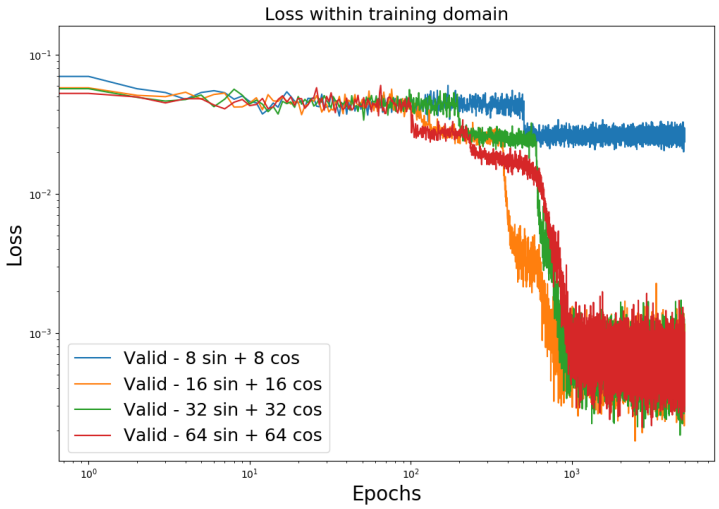

Figure 36: Fourier network performance for different number of nodes

Next, It is interesting to check out the actual trained weights of the network. Note that all Fourier networks have two inputs and $2N$ nodes with $N = 8, 16, 32, 64, 128$ in the training above. For each input and each node there will be a weight and a bias, corresponding to a trained 'frequency' and 'phase shift' respectively. In the figure below, it is illustrated how the frequencies are spread over all the nodes for the two inputs θ (left) and the radius r (right) for both the sine and cosine nodes separately, and for an increasing amount of $N = 16, 32, 64$. The network has been trained long enough to reach losses of around 10^{-4}.

For θ, the frequencies are spread of a range $[-3, 3]$. This makes sense as the analytical solution contains modes of sines and cosines of frequency equal to 3. Note that frequency 2 is also very common, the remaining weights are located close to zero. When it comes to the radius r, we see a range of only $[-1, 1]$ and most frequencies located around 0.

I would conclude the following about these results:

- The network is clearly able to figure out the most relevant frequencies (for θ in any case).
- It seems not very beneficial to increase the amount of layers, as the loss does not decrease significantly when doing so.
- The trained frequencies seem to stay within a certain, rather small region. This is entirely different than the concept of Fourier series, where the frequencies are increasing integers. Would it be possible to force the network in this direction?

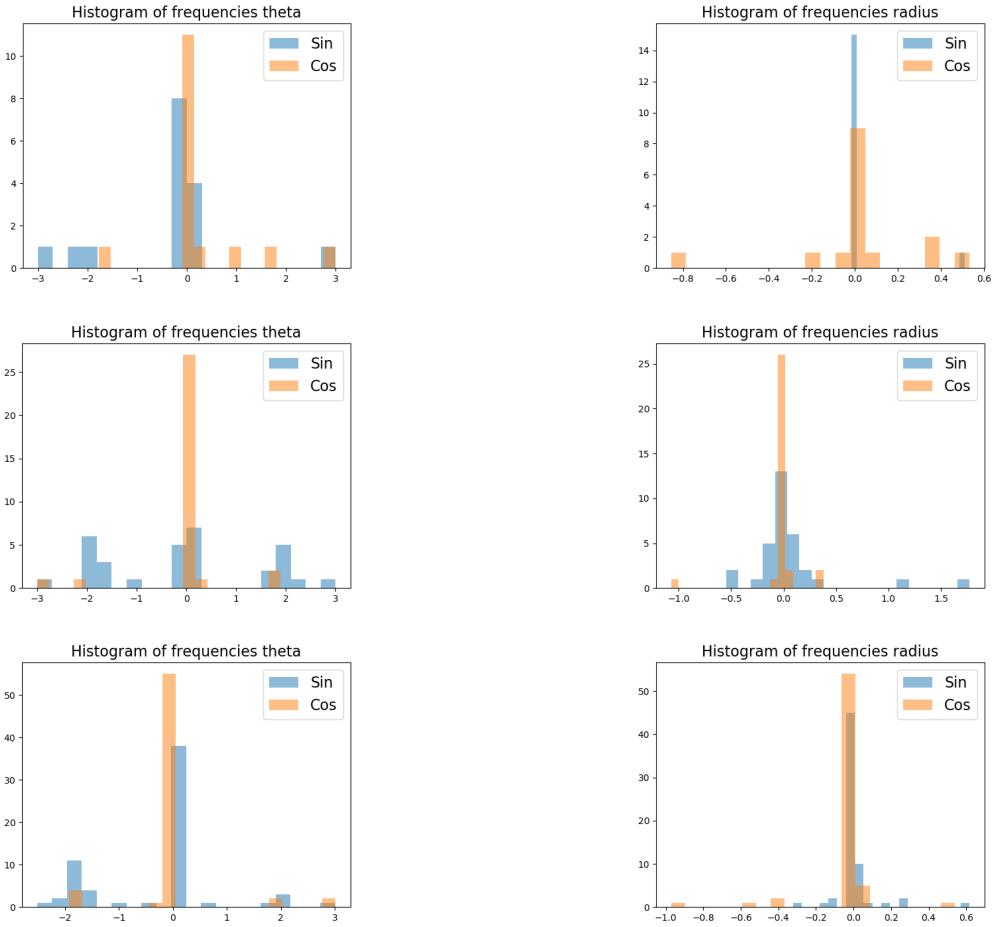

Figure 37: Histograms of the 'frequency' weights of the Fourier network for the θ (left) and r (right) input for increasing amount of layers (top 16, mid 32, bottom 64)

Before going any further with this, a short literature review is needed. The idea of building neural networks with sines and cosines to mimic Fourier series approximation sounds so straight-forward that it needs to be tried before.

A first and very relevant publication to look at is Zhumekenov et al. (link). It's a 2019 paper discussing the performance of so-called 'Fourier Neural Networks (FNNs)', laying out an overview of the most significant work that has been done regarding Fourier series resembling neural networks. The first mentioning of FNN was in 1988, by Gallant and White, who used the following 'cosine squasher' as activation function:

$$\sigma_{GW}(x) = \begin{cases} 0, & x \in (-\infty, -\frac{\pi}{2}) \\ \frac{1}{2}(cos(x + \frac{3\pi}{2}) + 1), & x \in [-\frac{\pi}{2}, \frac{\pi}{2}] \\ 1, & x \in (\frac{\pi}{2}, \infty). \end{cases} \tag{71}$$

The paper claims that of all FNN's, the one proposed by Gallant and White returns the

best results, but that none of the FNNs can outperform the standard feedforward neural network with sigmoidal activation, except for synthetic data. To me it's not entirely clear what is meant by 'synthetic' (they mention 'from a known function'). They also tried the Fourier activation function for image recognition on MNIST and language modeling, which resulted in a similar and worse performance respectively.

The specific cosine squasher as activation made me curious, so I implemented this and compared its performance to using a regular sine function. This leads to the following loss curve:

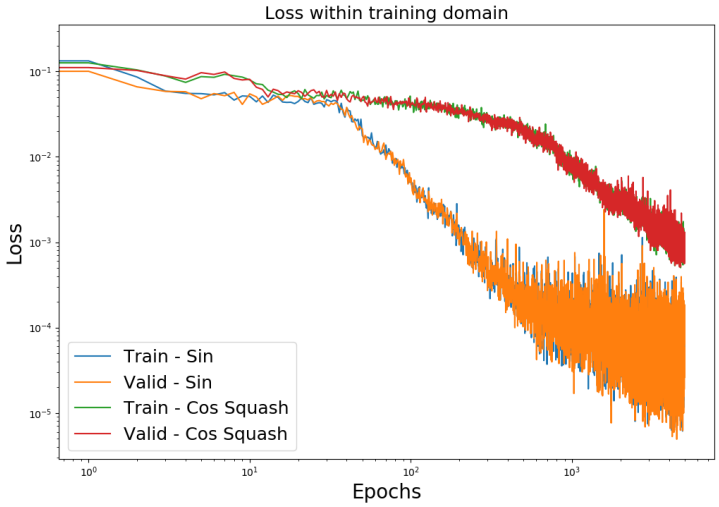

Figure 38: Loss curve for sine vs cosine squash

Clearly, the sine function still performs better.

I also found this paper (link) about using a Fourier Series Neural Network for system identification. It mentions how its network is able to approximate the Fourier coefficients, which enables it to identify the most relevant frequencies. I could however not read beyond the abstract.

After some search on Google Scholar, I found no more specific papers mentioning a neural network that mimics Fourier Series approximation with heterogeneous layers. Let alone for the specific application of solving PDE's in a unsupervised, data-free way. To me, using Fourier series knowledge inside the network makes more sense than in supervised applications.

Next, it would be useful to fit Bessel functions with our Fourier network in a supervised fashion. As such, we can make sure it is actually feasible to approach this closely by a sum of sines and cosines and what the frequencies are. The following figure illustrates the results for the Bessel functions of the first and second kind of order 0,1 and 2. The ground truth and the predictions are plotted on the left, the histogram of the trained frequencies in the middle and the loss vs epochs on the right.

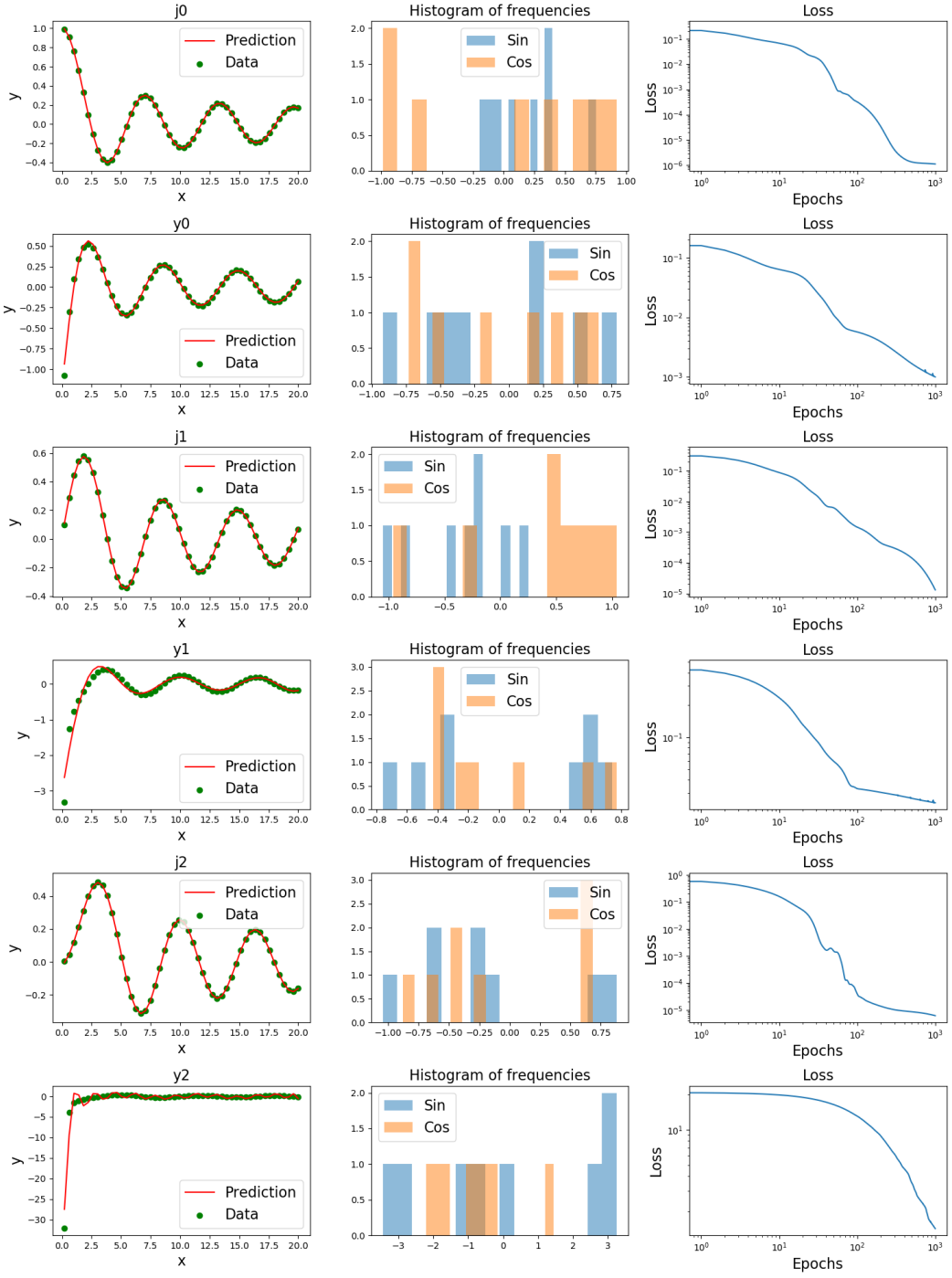

Figure 39: Fourier network for Bessel functions

Note that only one layer of 16 nodes, 8 times sines and 8 times cosine activations, has been trained for 1000 epochs. The Bessel functions of the first kind are very well approximated, with losses going towards 10^{-6}. For Bessel functions of the second kind, the value of y for x close to 0 goes to negative infinity, which makes it harder to fit. This is results in higher losses, but still a fairly good performance. The frequencies still seem to stay within a relatively small range around 0.

8.3.3 Last week's analysis

First, it is important to note that in all the graphs above, we have used one layer of sines and cosines using bias in the first layer. It is understood that this actually does not make sense, as including bias comes down to including a phase shift in the sines and cosines, which means that there is simply no distinction between both activation functions. Therefore, we will now examine the difference between one layer of sines that includes bias and one layer of a mixture of sines and cosines without bias.

We first test it on fitting the Bessel functions in a supervised way.The results are illustrated in the figure below. Note that we also use barplots instead of histograms to plot the weights from now on. The x-axis corresponds to the 'trained frequency weight', while the y-axis is equal to the magnitude of the output weight. As such, we have one figure that includes both the mode and the contribution of the mode to the output. As such, it is possible to distinguish the relative importance of different modes. The figure compares the performance of the two Fourier networks, where one only contains sines and includes biases, while the other includes an equal amount of sines and cosines but without bias.

I would say that the loss curves are similar, with a slight advantage for the sine with bias network. This makes sense, as the bias introduces another set of trainable parameters and a higher degree of freedom. When 16 sines and 16 cosines are used in one network without bias, there are 32 trainable weights, while 32 sines with bias have 64 weights, or double the amount. I do realize this is not an entirely fair comparison and am wondering what would be better?

It is also interesting to look at the trained weights. For both networks, the frequencies seem to be grouped somehow together, but still different enough. From this, there are no clear 'main frequencies' to capture.

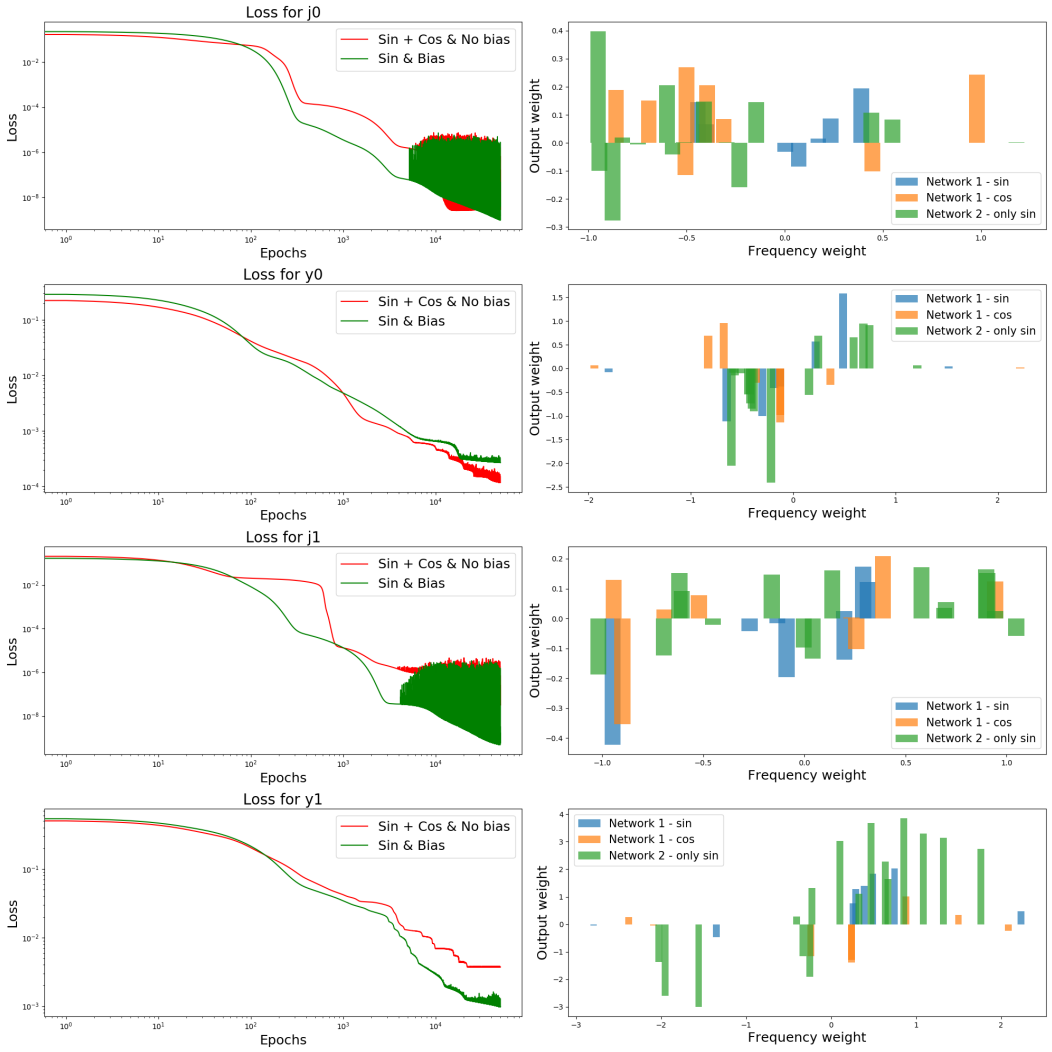

Figure 40: Fourier network for Bessel functions - comparison between Sine only with bias and Sine and Cosine without bias

Next, we have done the same comparison of both Fourier networks for the Helmholtz equation. Below you can find the loss function and the trained weights for r and θ for both networks. Again, the network for sine only performs better, as it has double the amount of parameters. For the variable θ, both networks are able to capture the main frequencies, while for r they seem to do so in a different way.

Given that the sine network has more parameters, the comparison might not be entirely fair. Therefore, we now try two networks with one Fourier layer, one with 32 sines and 32 cosines without bias and one with 32 sines with bias. This comes down to an equal amount

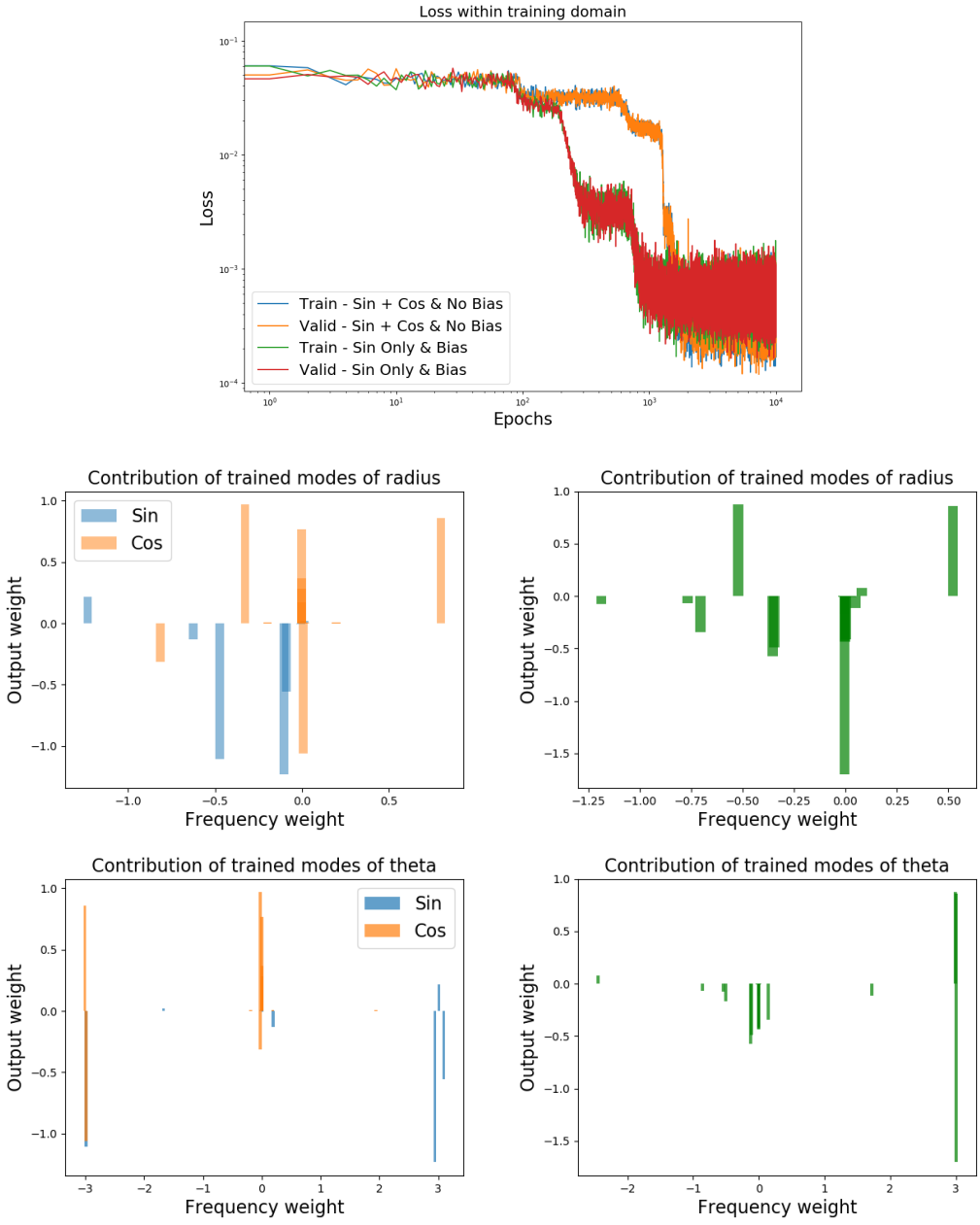

Figure 41: Loss of Fourier network for Helmholtz - comparison between Sine and Cosine without bias (left) and Sine only with bias

of parameters. The following curve illustrates the loss curve in that scenario:

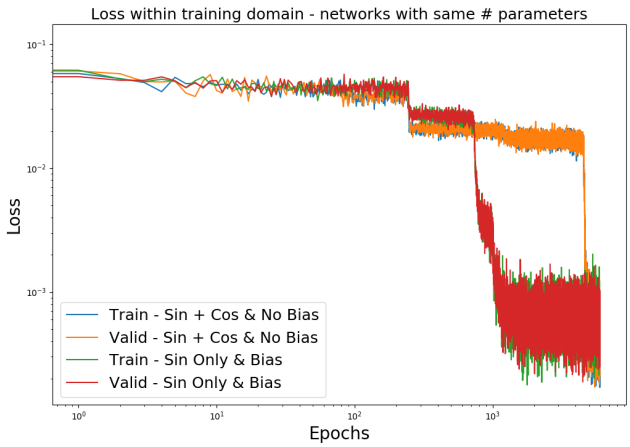

Figure 42: Loss Fourier network for Helmholtz - comparison between Sine only with bias and Sine and Cosine without bias

Somehow unexpectedly to me, it turns out that the loss function for Sine only still decreases faster. Having weights and biases for 32 sine nodes turns out the be better than having only weights for 32 sine and 32 cosine nodes.

Next, I wondered whether we could initialize the weights of the Fourier layer in a way similar to the actual Fourier series. Recall that the mathematical formulation of the N-order Fourier series approximation of function $f(x)$ is the following:

$$S_N = \frac{a_0}{2} + \sum_{n=1}^{N} \left[a_n cos\Big(\frac{2\pi n}{P} x\Big) + b_n sin\Big(\frac{2\pi n}{P} x\Big) \right] \tag{72}$$

I tried to initialize the 'frequency' weights exactly equal to $\frac{2\pi n}{P}$ in the hope that the corresponding local minimum of the network will be lower than with the default local minimum. For the supervised approximation of the Bessel functions, this leads to:

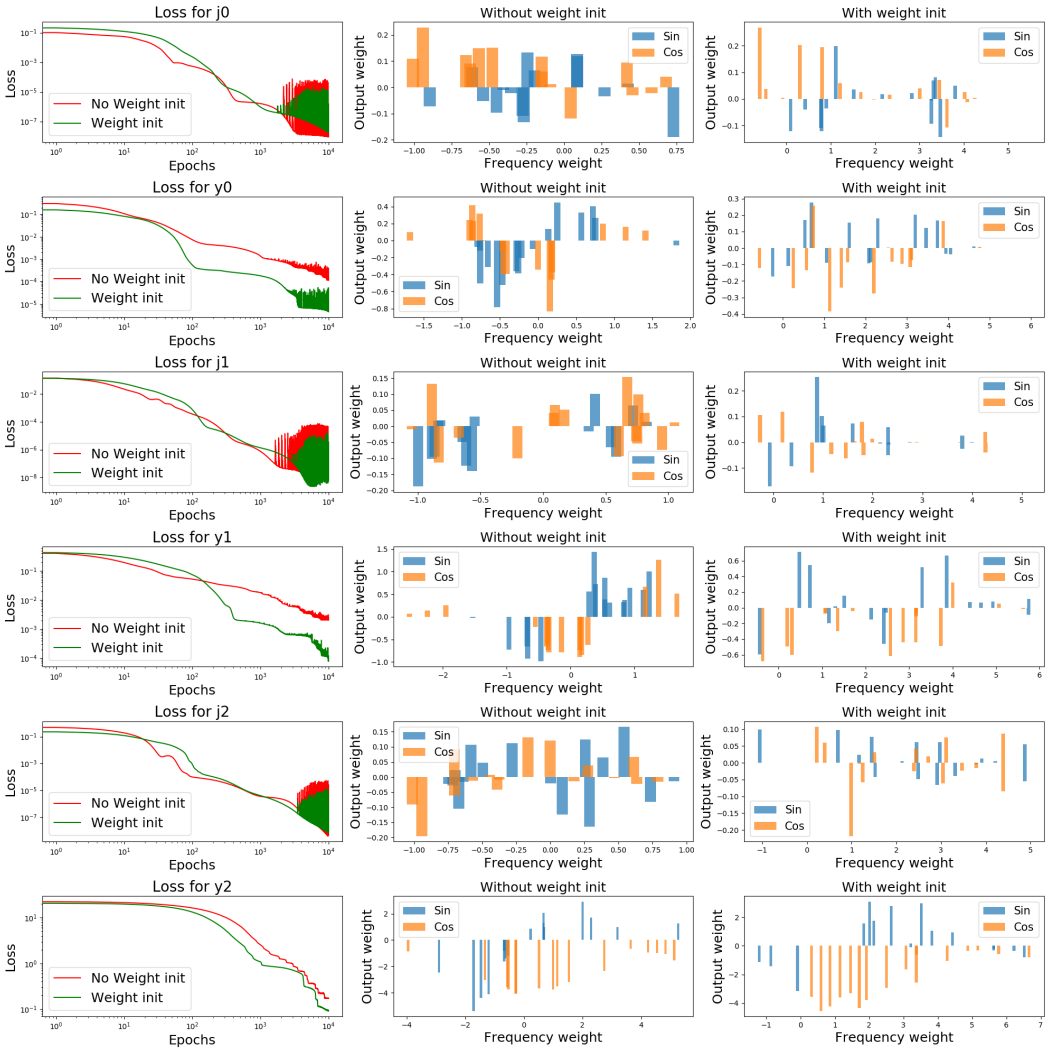

Figure 43: Fourier network for Bessel functions - evaluation of weight initialization

I was trying to do the same for Helmholtz, but could not find a learning rate for which the loss did not blow up/the loss was actually decreasing.

Next, I also went through the paper Marios provided, "Artificial neural networks with an infinite number of nodes" by Blekas et al, and picked up the following:

- They introduce a new type of neural network the FWNN, with weights that depend on a continuous variable, instead of the traditional discrete index. FWNN stands for 'Functionally Weighted Neural Network'. This can be interpreted as a network with an infinite number of nodes and significantly restricted number of model parameters.
- They use continuous weight functions, more specifically: Radial basis functions (RBF)

are known to be suitable for function approximation and multivariate interpolation. These functions themselves have network parameters that need be trained.

- By introducing a node density function with diverging integral over a finite domain, the network actually considers an infinite number of nodes.
- Given the limited number of trainable parameters, the network is less prone to overfitting, so much more generalizable.

This definitely looks like an interesting idea, but I'm not entirely sure what to do with it.

Also, last week it was proposed to search for Fourier Transform networks instead of Fourier Series.

It turns out that using Fourier Transform in CNN's has led to very efficient training as mentioned (here). Also, the use of Spherical Fourier Transform (SFT) has been used in CNN's as well, which led to great performance in equivariance in image classification (here). This is interesting, but not very relevant to our work here.

However, I did not find any paper that builds a network to predict the Fourier transform.

8.4 This week's analysis

This week, I have further explored the Fourier neural network, as well as a network that can perform separation of variables.

8.4.1 Comparison Sine vs Sine/Cosine

First, recall the discussion of last week about the activation and inclusion of biases in the Fourier network. We concluded that two formats with an equal amount of trainable parameters make sense:

- A network with one layer consisting of N nodes with sine activation and bias included. Further referenced as 'Sine Only'.
- A network with one layer consisting of 2N nodes, with an equal amount of sine/cosine activation and no biases.

A comparison is illustrated in the figure below (results from last week). This is a loss vs epochs plot for the 2D Helmholtz equation:

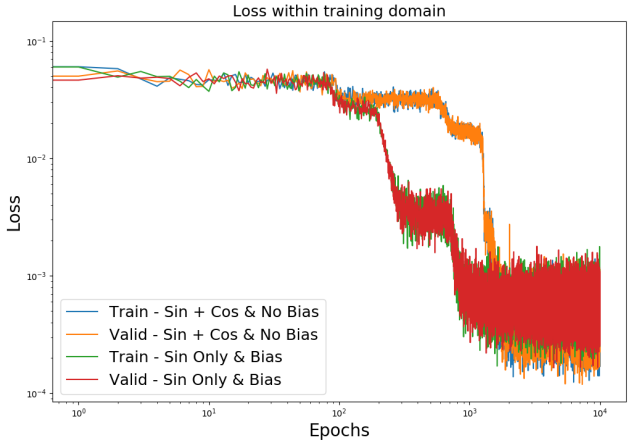

Figure 44: Loss of Fourier network for Helmholtz - comparison between Sine and Cosine without bias and Sine only with bias

From this, it seems that the sine activation leads to a faster decrease. However, we want to know which method leads to the lowest loss. In order to make a reproducible and sensible conclusion on this, we have set-up a small experiment. We want to test how one layer of only sine with bias compares to the use of a combination of sine and cosine activation without bias. Both networks have been trained for 5,000 epochs and with 32 and 64 nodes respectively. Note that because of the inclusion of bias in the latter network makes sure both networks have the same number of parameters. This has been done for 40 times and for each training, the losses corresponding to the last 500 epochs have been recorded.

	Sine	SinCos
How many wins	42.5%	57.5%
Mean improvement when win	65.60%	27.32%
Mean over all experiments	1.26e-3	3.29e-3
Min over all experiments	1.39e-3	9.42e-5
Max over all experiments	3.31e-2	2.41e-2

Table 3: Summary of experimental comparison of Sine vs SinCos Fourier layer

To me this looks promising but inconclusive. First, the SinCos approach seems to 'win' more often, which means it led to a lower loss in 57.5% of the experiments. However, when the sine network performs better, its losses are significantly lower than when the SinCos networks performs better. I believe that this is because in these cases, the SinCos network is not converged, so it's impossible to draw any conclusions. My suggestion for next step would be to repeat this experiment for 10,000 epochs (although this will take very long time).

8.4.2 Comparing Fourier network results to real Fourier Transform

Next, it is interesting to compare the results from the Fourier network to the actual Fourier transform. In order to do this we will consider the Bessel functions of the first and second kind of order $n = 0, 1, 2$. First, the Fourier transforms will be computed using the built-in Fast Fourier Transform algorithm in Numpy. The frequencies for which the absolute value of complex transformation reaches the highest value are considered to be the most relevant 'modes' of the function. These frequencies will be saved for all considered functions.

Next, we'll use the one layer Fourier network with sine and cosine activation without including bias to predict the same functions in the same domain. After training has been completed, the weights will correspond to the 'trained frequencies' and it will be interesting to compare these values to the ones achieved through FFT.

After some experimentation and thinking, I decided to train two networks. The first will have a fairly limited amount of nodes (4 sines and 4 cosines) and no regularization, while the second will use a larger amount of nodes (32 sines and 32 cosines) and a relatively large l_1-regularization. As such, it is hoped for that both networks search for the most important frequencies.

The table below summarizes the 4 main frequencies that result from the FFT, the first Fourier network with a small amount of nodes and no regularization and the second network with a large amount of nodes and regularization for 4 Bessel functions.

Importance	FFT	Small nodes - No reg	Large nodes - With reg
Frequency 1	0.1500	0.1469	-0.9395
Frequency 2	0.1000	-0.4470	-0.6974
Frequency 3	0.0500	0.5606	0.4234
Frequency 4	0.2000	-0.9250	-0.1416

Table 4: Bessel function of the first kind and order 0, or $J_0(x)$

Importance	FFT	Small nodes - No reg	Large nodes - With reg
Frequency 1	0.1500	0.2843	-0.7203
Frequency 2	0.2000	-0.4236	-0.5817
Frequency 3	0.2500	0.7115	-0.9569
Frequency 4	0.3000	0.8663	0.2931

Table 5: Bessel function of the second kind and order 0, or $Y_0(x)$

Importance	FFT	Small nodes - No reg	Large nodes - With reg
Frequency 1	0.1500	-0.8367	-0.9358
Frequency 2	0.1000	-0.9765	-0.6953
Frequency 3	0.0500	0.7939	-0.2740
Frequency 4	0.2000	-0.7386	-1.2e-4

Table 6: Bessel function of the first kind and order 1, or $J_1(x)$

Importance	FFT	Small nodes - No reg	Large nodes - With reg
Frequency 1	0.1500	-0.1696	0.2717
Frequency 2	0.1000	-0.1223	0.3911
Frequency 3	0.0500	0.2363	0.6229
Frequency 4	0.2000	0.7623	-0.2193

Table 7: Bessel function of the second kind and order 1, or $Y_1(x)$

It appears that the frequencies are significantly different in most cases. Only sometimes, the two networks are able to predict one of the main frequencies with reasonable accuracy. There seems to be a slight advantage for the network with small amount of nodes and no regularization.

8.4.3 Heterogeneous layer with Sine and Bessel

From the analytical solution of the specific problem definition as discussed in section 8.3.2, we know that the most dominating terms are products of sines and Bessel functions of frequency and order equal to 2 and 4.

Therefore, we will first try to implement a network with one heterogeneous layer, with as activation functions $sin(x)$, $J_2(x)$, $Y_2(x)$, $J_4(x)$ and $Y_4(x)$. First, I tried to apply the sine activation to the θ variable and the Bessel functions to the radius r in one layer and compute the output as a linear combination of all these nodes. This led to the following bad training performance:

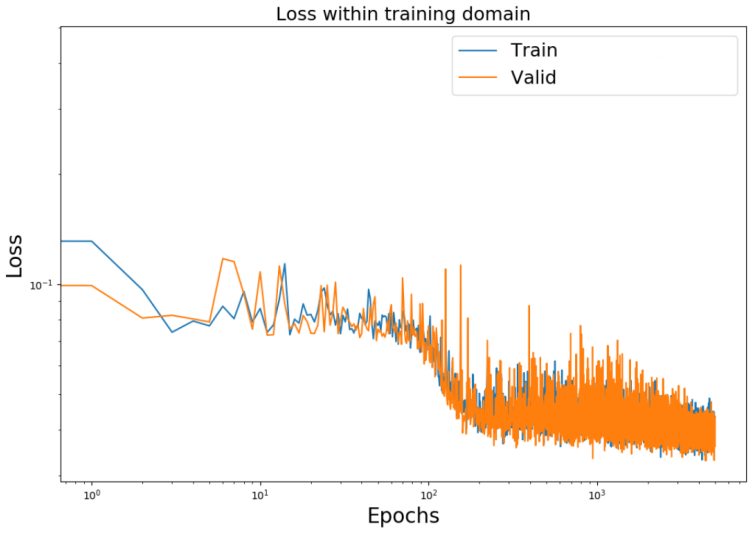

Figure 45: Solution for one layer with $sin(\theta)$, $J_2(r)$, $Y_2(r)$, $J_4(r)$ and $Y_4(r)$

Next, I want to embed the true separation of variables and their product combination into the solution. I came up with the following structure:

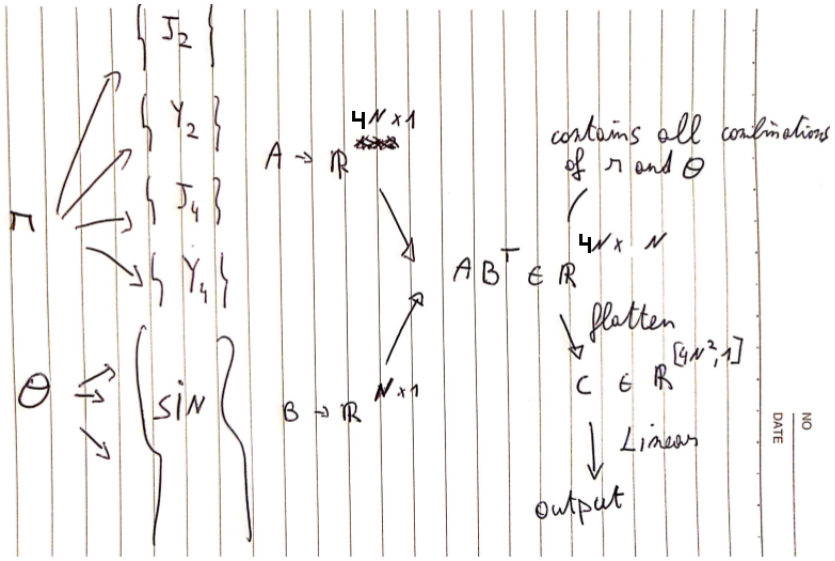

Figure 46: Proposed network architecture

I believe to have implemented this correctly (note that the Bessel functions of the second type work as well now). Although it seems to make sense, it does not result in immediate low losses, as we would expect, since the true analytical solution is embedded.

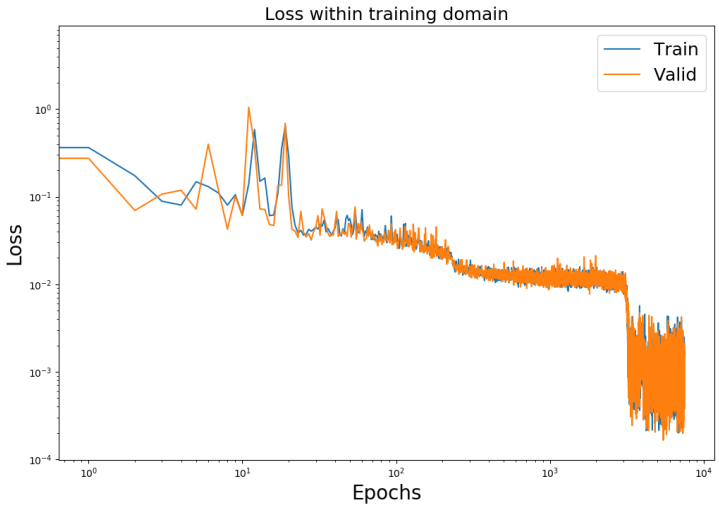

Figure 47: Loss in training domain of proposed Separation of Variable network

The losses go down until $10e-4$ for 7500 epochs, which seems ok but clearly not equal to the exact solution. I was wondering the following:

- Is there anything wrong in my reasoning?
- Do I include too many options (as $2N^2$ contains each possible combination of r and θ). I currently do not use regularization, but maybe I should?
- Maybe it is hard to train the weights for the Besselfunctions?

Looking foward to discussing this.

9 Conclusion

This documents has touched upon multiple topics.

First, it discusses the analytical solution of the Laplace equation in cylindrical coordinates with particular boundary conditions. The final solution consists of an infinite sum of multiplications of hyperbolics in z, sines and cosines of θ and Bessel functions of the first and second order in r.

Second, the mathematics behind a simple neural network with sine as activation function has been developed, both for one layer with multiple nodes as for two layers. It appears that the first leads to a linear combination of sines of the input variables, while the latter leads to weighted sums of sines of sines.

Third, a simple ODE has been solved with a neural network through a Pytorch implementation of the cleverly designed, unsupervised NN architecture. It was interesting to see how the error grows the further away from the initial condition and that the network

performs very poorly outside the training domain. As deeper exploration, I evaluated the convergence of the NN with Chebyshev interpolation rather than uniform sampling. After thorough evaluation, it seems that there is no reason to believe that Chebyshev performs better, which does not mean that it is a nice additional feature to be incorporated in the code.

Next, the masternode of Pavlos has been implemented in the same NN in the hope periodic solutions would arise. With a modified initial condition operator, it turns out that periodic solutions can be derived. In some cases, when a reasonable value for the frequency is computed, this method can lead to very good performance. However, this changes drastically for slightly different values of the frequency, which was seen in the prediction of a simple linear oscillator but more strongly in the predictions of the solutions to Van der Pol and Duffing equation. For the non-linear equations, it is however important to note that the non-linearity makes it hard to solve it without the masternode as well.

Next, a very simple PDE's, the 2D Laplace equation, has been solved using a the NN implementation of Feiyu. This basis understanding is then used to solve the 2D Helmholtz equation in polar coordinates. First, the code is developed and tested for a specific example. Afterwards, we wonder how we can enforce periodic boundary conditions in θ. It appears that using the masternode for θ leads to strict enforcing of the periodicity, but performs poorly during training. The most effective way was to add a penalization term into the loss function that incorporates 'how much the periodicity' is met. This led to good performance.

Next, we have played around with different activation functions in the network. First, using Sine instead of Tanh led to a significant improvement! The loss function decreases at a higher rate and for less amount of epochs. This could have been expected from the analytical solution. From this, we took a side-road towards experimenting with Fourier inspired neural networks. This leads to interesting results, definitely given the lack of literature around this particular topic. Similarly, we tried to embed Bessel functions as activation function. Based on the exact analytical solution, we were able to identify the relevant Bessel functions in the solution and tried to play around with these as activation function. This does not work for the moment, but a tensor product approach seems to be the way to go.