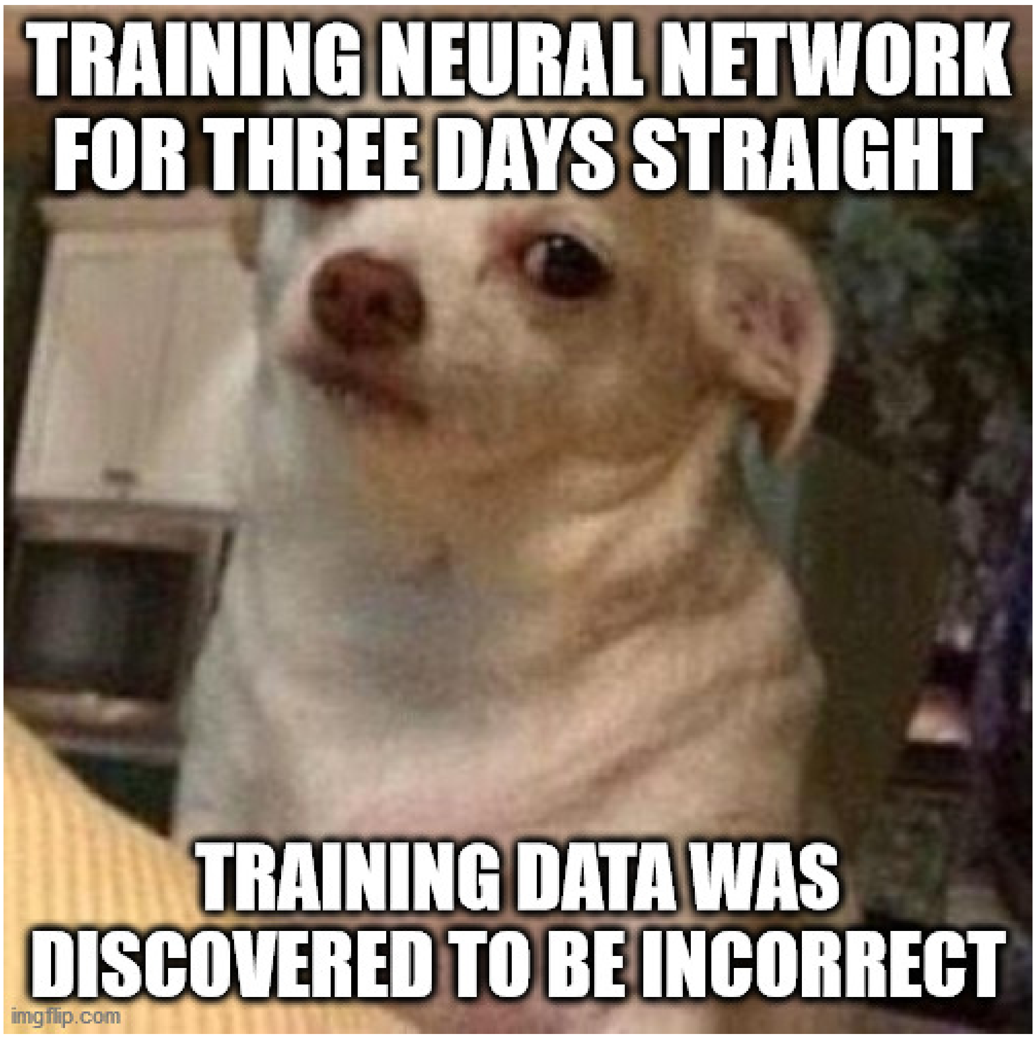
Numerical Mathematics Project 2: Report

November 2020


TRAINING NEURAL NETWORK FOR THREE DAYS STRAIGHT
TRAINING DATA WAS DISCOVERED TO BE INCORRECT
imgflip.com

1 Implementation of ANN

The artificial neural network is a function approximator $\tilde{F}(y;\theta) : \mathbb{R}^d \to \mathbb{R}$ which can be defined as a composition of maps Φ_k and G.

The maps $\Phi_k : \mathbb{R}^d \to \mathbb{R}^d, 0 \leq k \leq K-1$ and $G : \mathbb{R}^d \to \mathbb{R}$ are given as:

$$\Phi_k(y) = y + h\sigma\left(W_k y + b_k\right), \qquad G(y) = \eta\left(w^T y + \mu\right) \tag{1}$$

Φ_k is the transformation from one hidden layer to the next, while G is the transformation collapsing the dimensions on the hidden layers into the correct terminal dimension. The function approximator as a composition is therefore:

$$\tilde{F}(y;\theta) = G \circ \Phi_{K-1} \circ \Phi_{K-2} \circ \cdots \circ \Phi_1 \circ \Phi_0(y) \tag{2}$$

In addition, Φ_k is defined in terms of σ, which is the activation function, and G is defined in terms of η, which is the hypothesis function. The activation function was in this project taken to be the sigmoid function, and the hypothesis function to be a function in terms of tanh, both described below:

$$\sigma(x) = \tanh x, \qquad \eta(x) = \frac{1}{2}\left(1 + \tanh\frac{x}{2}\right) \tag{3}$$

Using the neural network consists in selecting some input data and performing a forward sweep defined as:

$$z^{(0)} = y, \qquad z^{(k+1)} = \Phi_k(z^{(k)}) \qquad \tilde{F}(y;\theta) = \Upsilon = G\left(\left(z^{(K)}\right)\right) \tag{4}$$

The implementation follows the details in the project description. It is possible to send different input data through the network simultaneously by placing them in a matrix $Y = [y_1, \ldots, y_I]$ and similar to the project, the following is defined:

Approximate values of $\tilde{F}(Y;\theta)$ and exact values of $F(Y)$:

$$\Upsilon = \left[\tilde{F}(y_1;\theta), \ldots, \tilde{F}(y_I;\theta)\right] \qquad c = [F(y_1), \ldots, F(y_I)]$$

The objective function:

$$J(\theta) = \frac{1}{2}\|\tilde{F}(y;\theta) - c\|^2 = \frac{1}{2}\|\Upsilon - c\|^2 \tag{5}$$

The back propagation vectors (collection of I vectors):

$$P^{(K)} = \frac{\partial J}{\partial Z^{(K)}} = w \cdot \left[(\Upsilon - c) \odot \eta'\left((Z^{(K)})^T w + \mu\vec{1}\right)\right] \tag{6}$$

$$P^{(k)} = \frac{\partial J}{\partial Z^{(k)}} = P^{(k+1)} + hW_k^T \cdot \left[\sigma'\left(W_k Z^{(k)} + b_k\right) \odot P^{(k+1)}\right] \tag{7}$$

And, lastly, the expressions necessary to compute $\nabla_\theta J$:

$$\frac{\partial J}{\partial \mu} = \eta'\left(\left(Z^{(K)}\right)^T w + \mu\mathbf{1}\right)^T (\Upsilon - c) \tag{8a}$$

$$\frac{\partial \mathcal{J}}{\partial w} = Z^{(K)}\left[(\Upsilon - c) \odot \eta'\left(\left(Z^{(K)}\right)^T w + \mu\right)\right] \tag{8b}$$

$$\frac{\partial J}{\partial W_k} = h\left[P^{(k+1)} \odot \sigma'\left(W_k Z^{(k)} + b_k\right)\right] \cdot \left(Z^{(k)}\right)^T \tag{8c}$$

$$\frac{\partial J}{\partial b_k} = h\left[P^{(k+1)} \odot \sigma'\left(W_k Z^{(k)} + b_k\right)\right] \cdot \mathbf{1} \tag{8d}$$

To update the neural network with the gradient found by the back propagation, a simple formula is used:

$$\theta_{new} = \theta_{old} - \tau\nabla_\theta J(\theta_{old}) \tag{9}$$

Simplifications to reduce re-computations

In order to save re-computation of common parts and keep the code simpler by adding some layers of abstraction, the following expressions were introduced:

$$Y_c = \Upsilon - c \tag{10a}$$

$$\nu = \eta'\left(\left(Z^{(K)}\right)^T w + \mu\right) \tag{10b}$$

$$H^{(k)} = h\left[P^{(k+1)} \odot \sigma'\left(W_k Z^{(k)} + b_k\right)\right] \tag{10c}$$

These abstractions result in a revision of $\nabla\theta J$ given as:

$$\frac{\partial J}{\partial \mu} = \nu^T Y_c \tag{11a}$$

$$\frac{\partial \mathcal{J}}{\partial w} = Z^{(K)}\left(Y_c \odot \nu\right) \tag{11b}$$

$$\frac{\partial J}{\partial W_k} = H^{(k)} \cdot \left(Z^{(k)}\right)^T \tag{11c}$$

$$\frac{\partial J}{\partial b_k} = H^{(k)} \cdot \mathbf{1} \tag{11d}$$

This eliminates the duplicative computation of Y_c, ν and $H^{(k)}$.

A guide on where to find the implementations of these are given below:

All implementation of these functions are given in the `ann.py` file. The table below gives an explanation to which functions correspond to which equations:

Table 1: Linking equations with desired functionality and proper function name in python.

Equation	Action	Function name
eq. (4)	Forward sweep	`getZ`
eq. (4)	Approximate values from the ANN	`getUpsilon`
eq. (6)	Start back propagation	`getPK`
eq. (7)	Rest of the back propagation	`getP`
eq. (10a)	Y_c simplification	`getYc`
eq. (10b)	ν simplification	`getNu`
eq. (10c)	$H^{(k)}$ simplification	`getHk`
eq. (11)	Computation of $\nabla_\theta J$	`getdelJ`
eq. (9)	Updating θ	`updateTheta`

1.1 Using suggested functions and systematically varying parameters

The functions that we used in the testing was $F(y) = \frac{1}{2}y^2$ and $F(y) = y^3 - y$. In order to notice a trend in the choice of parameters they were changed systematically. The approximated functions, some of the gradients of the approximated functions and the objective functions were displayed in plots. By observing an approximate function that fits the exact function and a smooth objective function that converges, the optimal parameters can be found.

1.1.1 Adjusting d and K

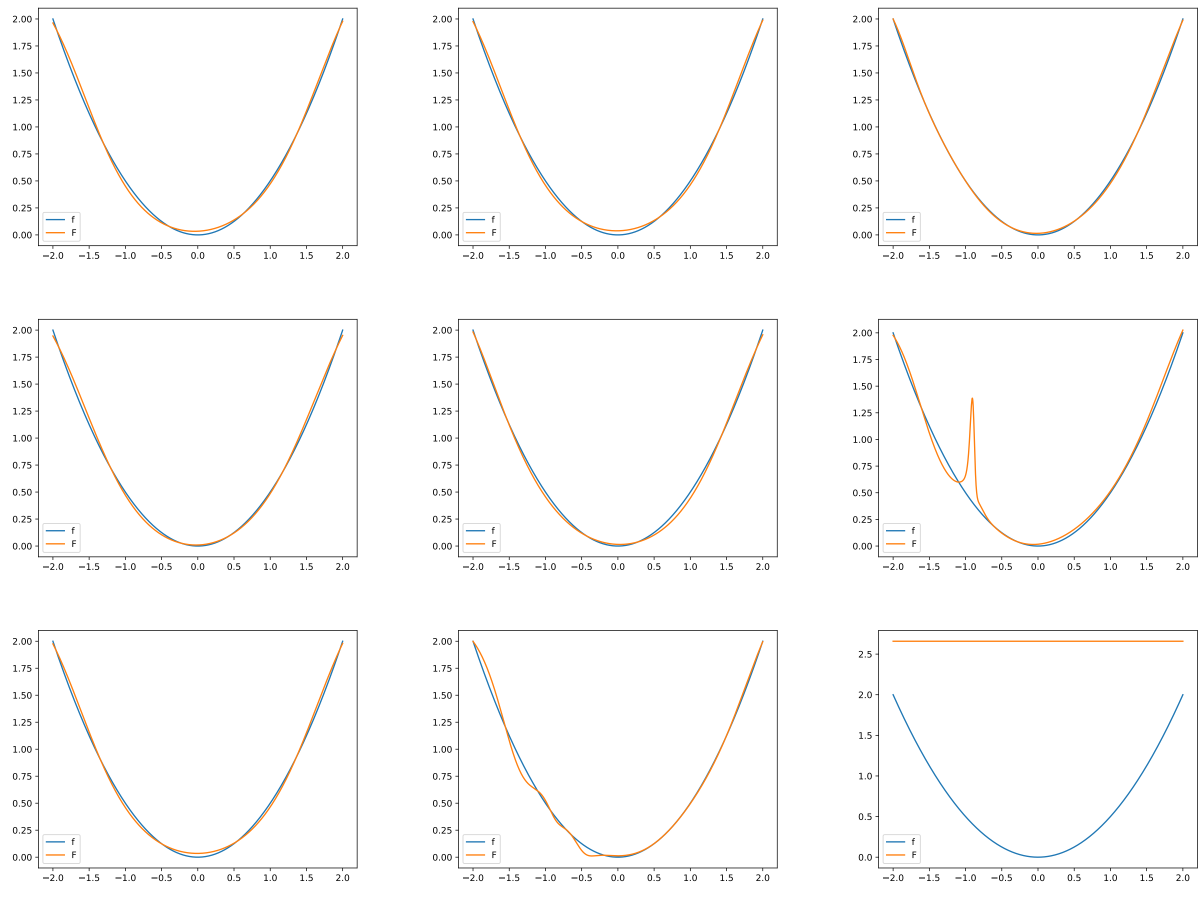


Figure 1: The plots display the approximated functions, in orange, with K = 2, 4 and 8 for every column and d = 2, 4, and 8 for every row when training the ANN on $F(y) = \frac{1}{2}y^2$.

When testing for different dimensions and layers, the parameters were set to d, K = 2, 4 and 8. The plots of the approximated function of $F(y) = \frac{1}{2}y^2$, its gradient and the objective function are displayed in Figure 1, 2 and 3. A common theme throughout this testing period is finding a balance. Not only will computational time increase significantly when increasing the different parameters, but even when doings so they do not necessarily give better results. Worst case scenario, too high values give no result at all. This is observed in all the figures for d = 8 and K = 8, such as in Figure 1. The optimal values of K and d seems to be 8 and 4. The approximated function, the gradient of the approximated function and the objective function all look ideal. Though, other combinations such as K, d = 2, 4 gives good results, but does not fit the gradient as well.

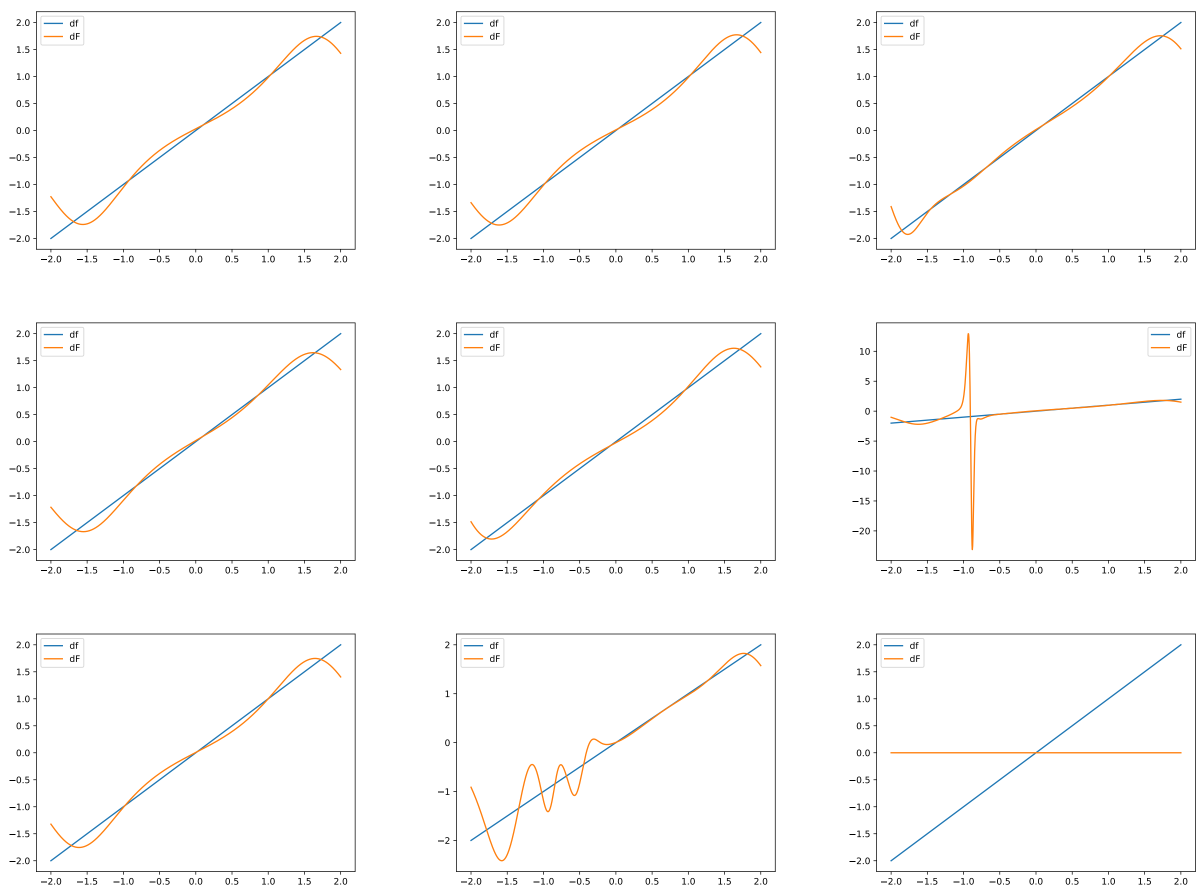


Figure 2: The plots display the gradient of the approximated functions, in orange, with K = 2, 4 and 8 for every column and d = 2, 4, and 8 for every row when training the ANN on $F(y) = \frac{1}{2}y^2$.

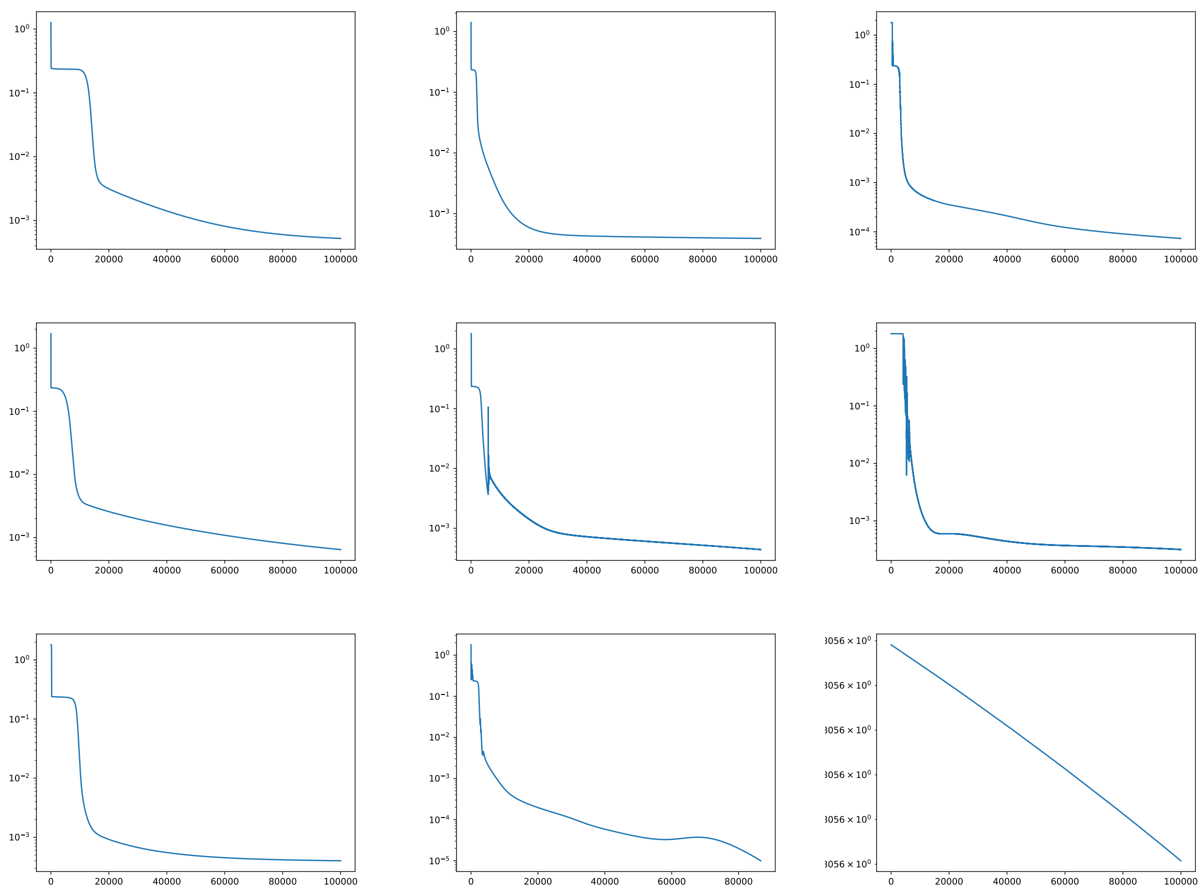


Figure 3: The plots display the objective function when training the ANN on $F(y) = \frac{1}{2}y^2$ with with K = 2, 4 and 8 for every column and d = 2, 4, and 8 for every row.

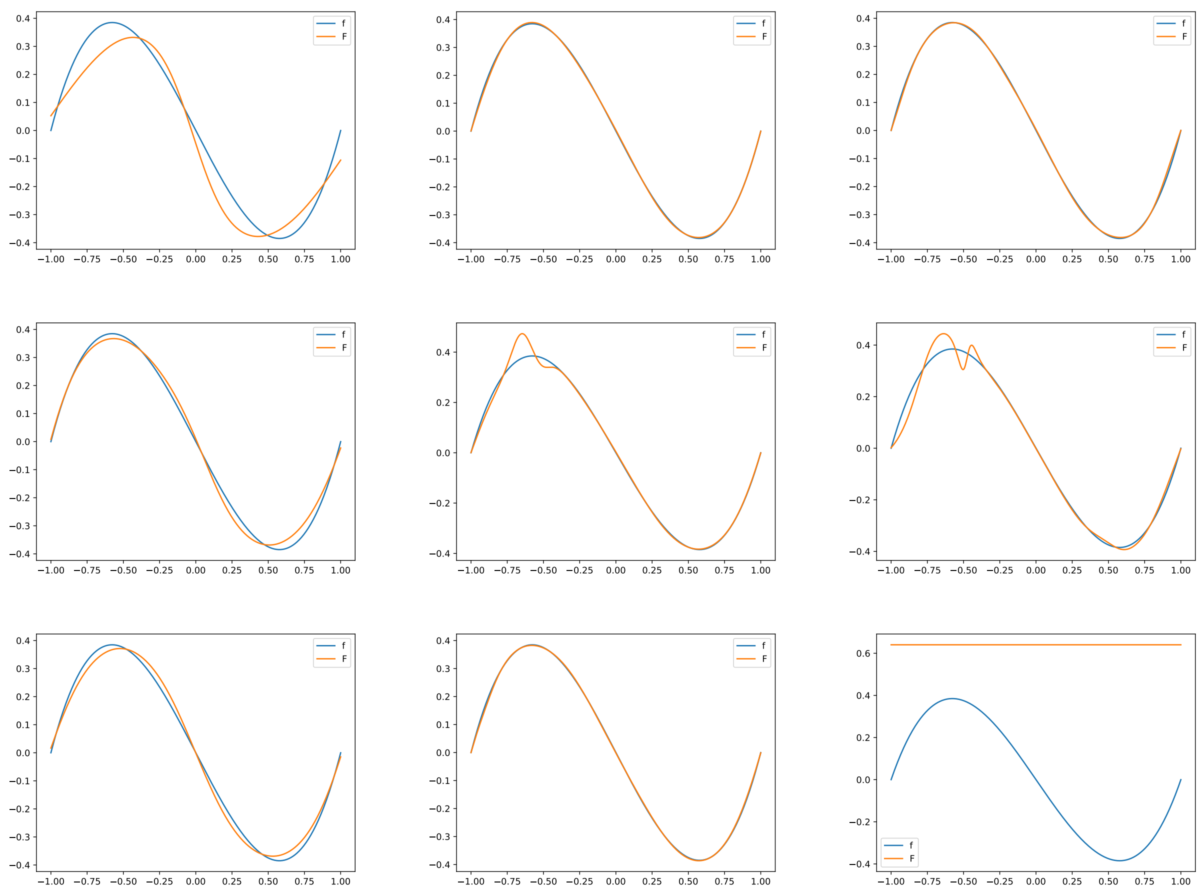


Figure 4: The plots display the approximated functions, in orange, with K = 2, 4 and 8 for every column and d = 2, 4, and 8 for every row when training the ANN on $F(y) = y^3 - y$.

By observing the plots for $F(y) = y^3 - y$, Figure 4, 5 and 6, one can conclude with K, d = 2, 2 being the better option. This highlights the need to vary the parameters with different functions and some values for K and d that works well for one function does not necessarily work with a different function.

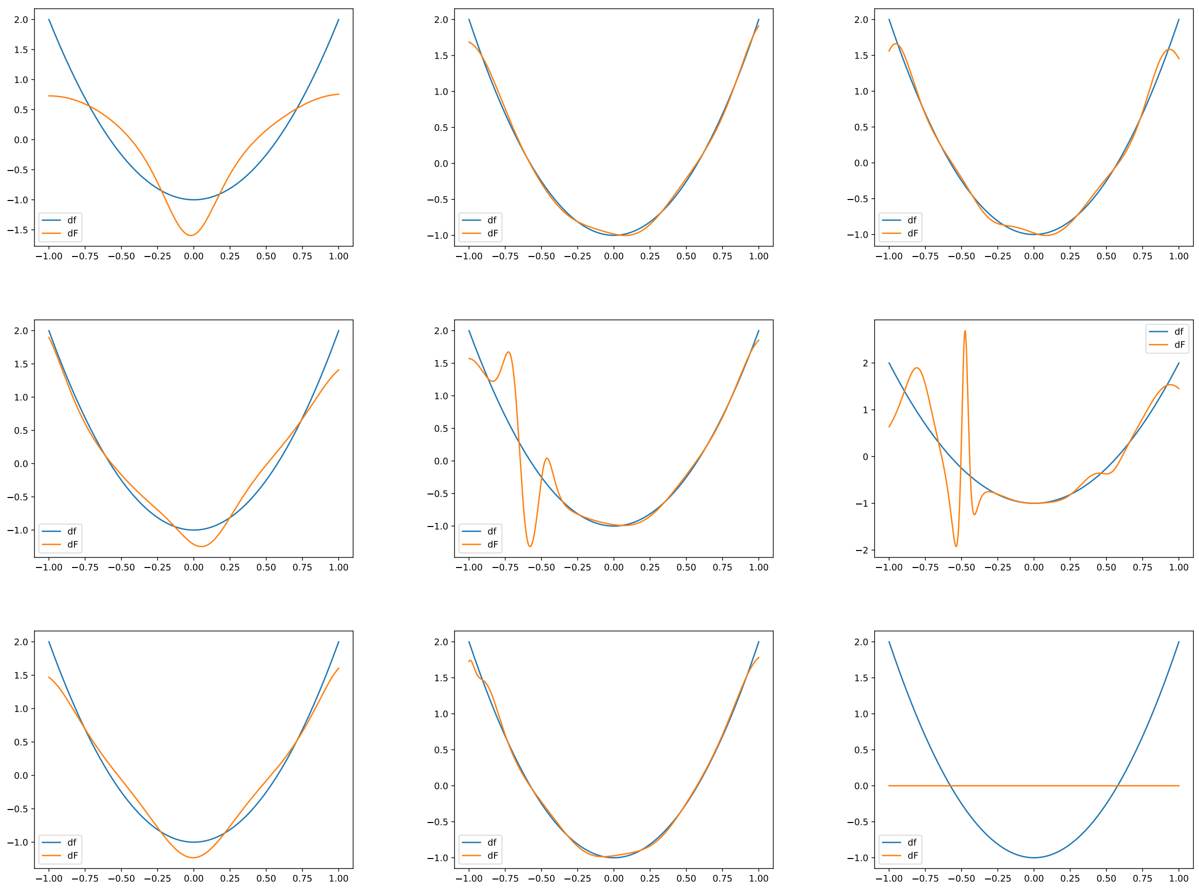


Figure 5: The plots display the gradient of the approximated functions, in orange, with K = 2, 4 and 8 for every column and d = 2, 4, and 8 for every row when training the ANN on $F(y) = y^3 - y$.

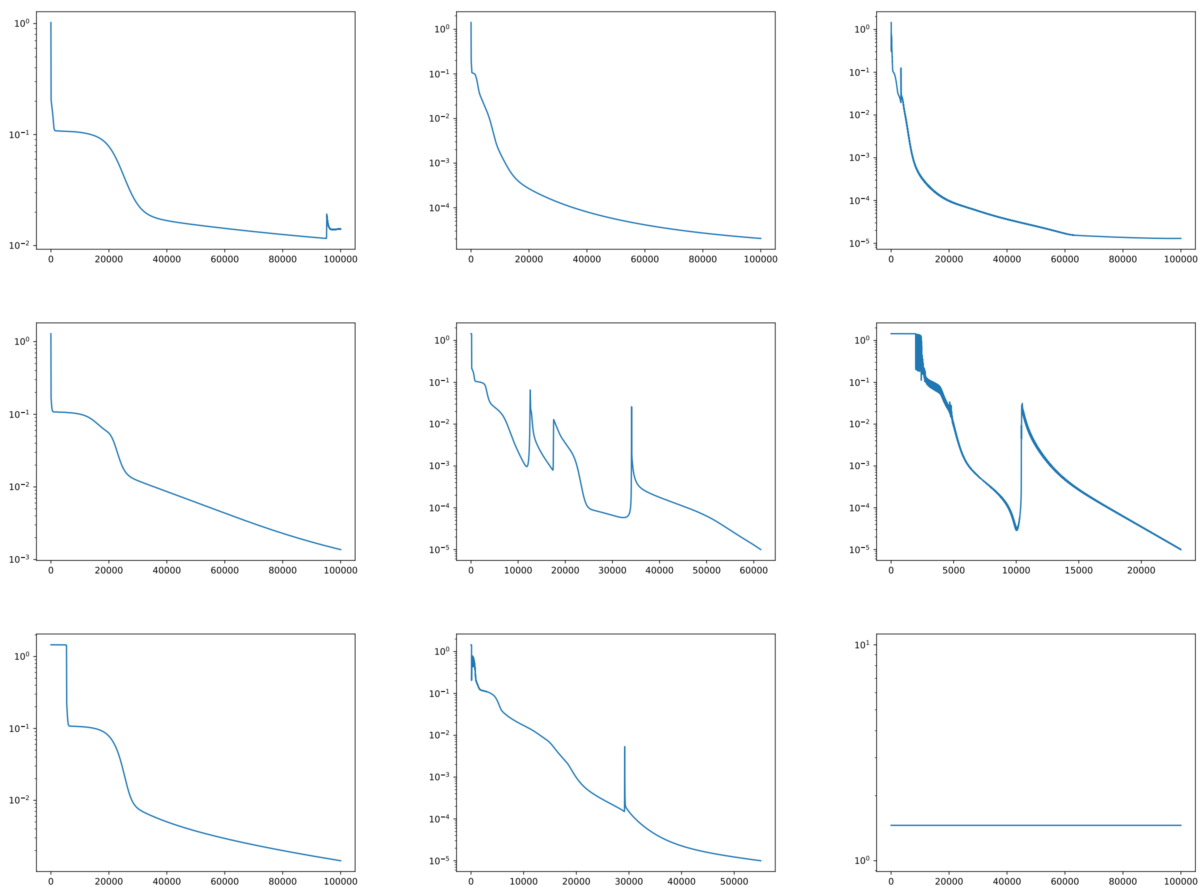


Figure 6: The plots display the objective function when training the ANN on $F(y) = y^3 - y$ with with K = 2, 4 and 8 for every column and d = 2, 4, and 8 for every row.

1.1.2 Adjusting I

The amount of data points, I, was set to 5, 10, 15, 20, 50 and 100 in order to test its effect. To approximate $F(y) = y^3 - y$ it seems like 10 data points is enough as the approximated function and its gradient fits the actual functions well, seen in Figure 7 and Figure 8. It is worth noticing that there is an overfitting occurring when using only 5 data points which is cured by increasing I to 10. For high values of I, the objective function, Figure 9, is jagged which is believed to be due to a too high τ.

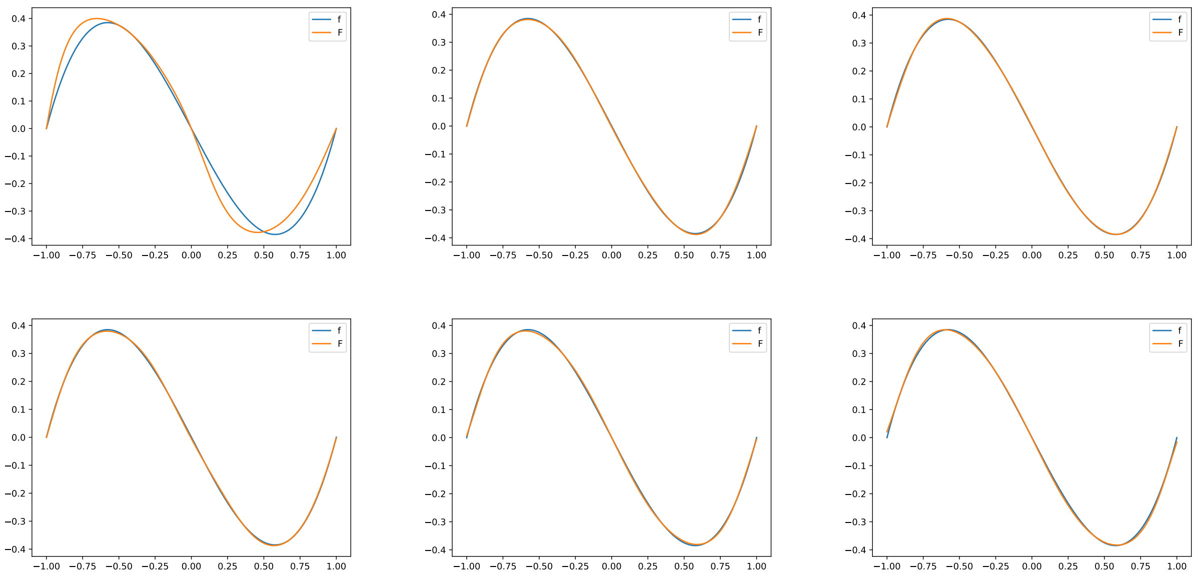


Figure 7: The plots display the approximated functions, in orange, from left to right with I = 5, 10, 15, 20, 50 and 100 when training the ANN on $F(y) = y^3 - y$.

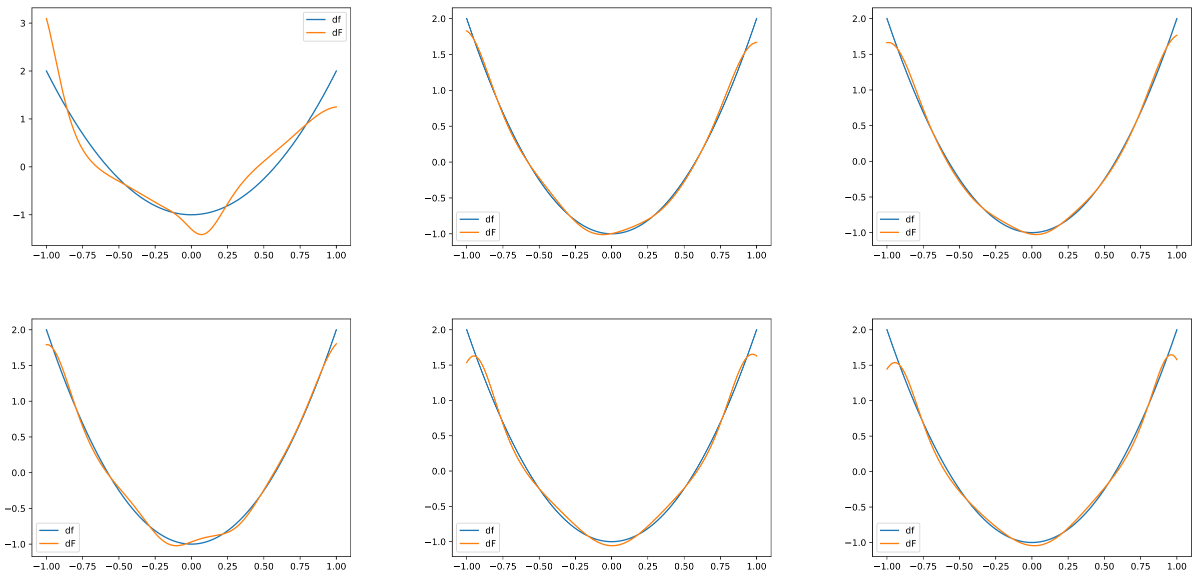


Figure 8: The plots display the gradient of the approximated functions, in orange, from left to right with I = 5, 10, 15, 20, 50 and 100 when training the ANN on $F(y) = y^3 - y$.

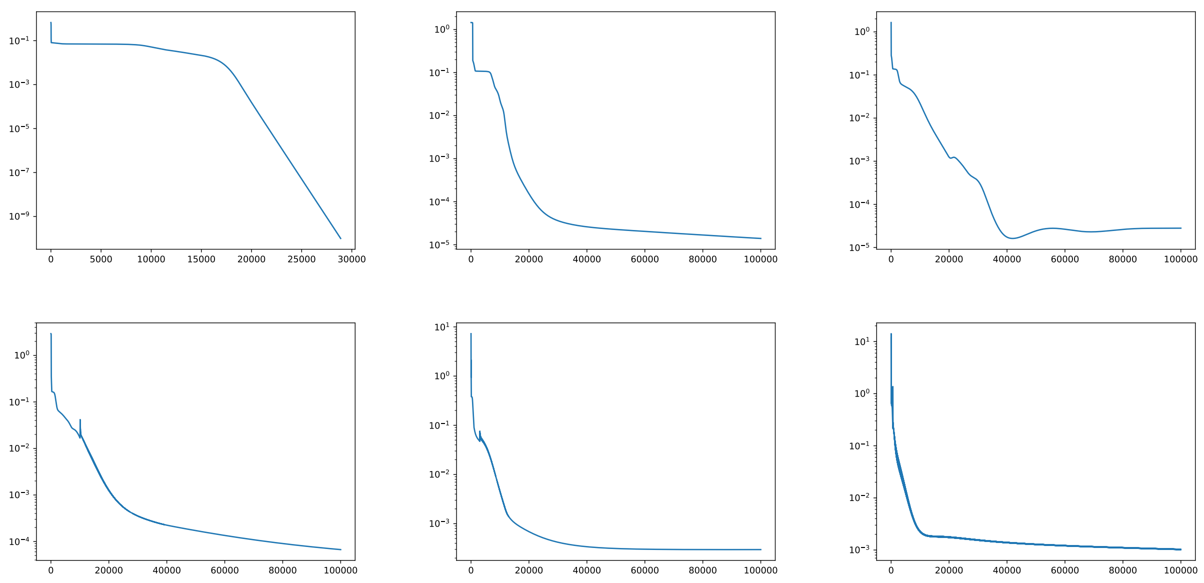


Figure 9: The plots display the objective function when training the ANN on $F(y) = y^3 - y$ with, from left to right, I = 5, 10, 15, 20, 50 and 100.

1.1.3 Adjusting tau

When testing different values for the gradient descent stepsize, $\tau = 0.001$, 0.005, 0.01, 0.05 and 0.1 was used. To approximate $F(y) = y^3 - y$, displayed in Figure 10 it seems like $\tau = 0.01$ is appropriate. However, none of the objective functions, Figure 11, in this case gave a properly smooth objective function, and a too high τ gives an extremely jagged objective function. This is due to the method being too coarse. Too low τ takes too long and the amount of iterations were not high enough to give a proper result.

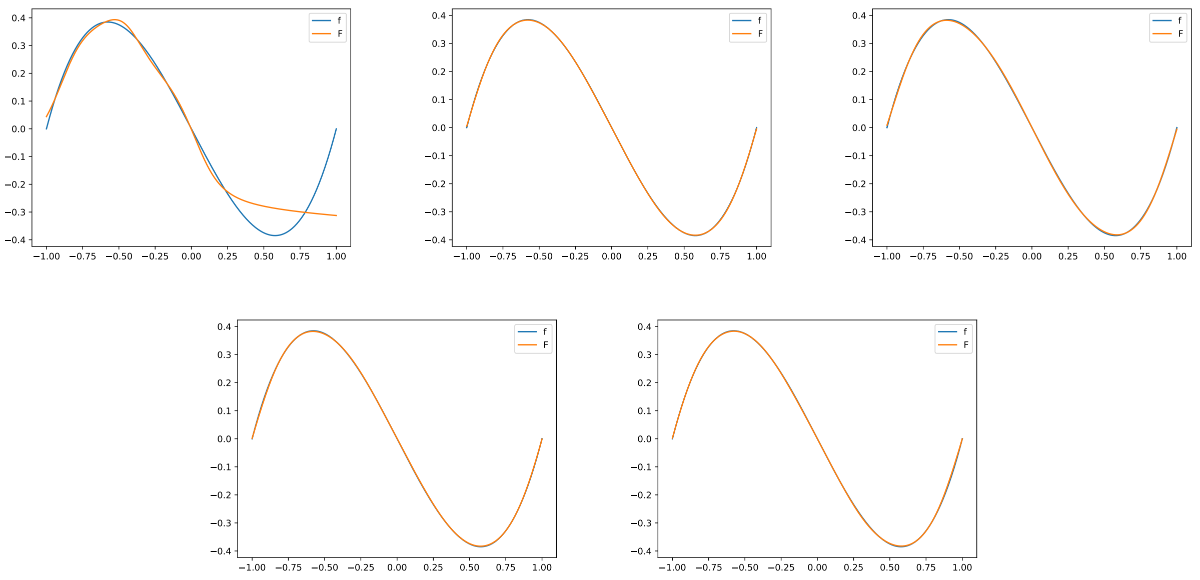


Figure 10: The plots display the approximated functions, in orange, from left to right with $\tau = 0.001$, 0.005, 0.01, 0.05 and 0.1 when training the ANN on $F(y) = y^3 - y$.

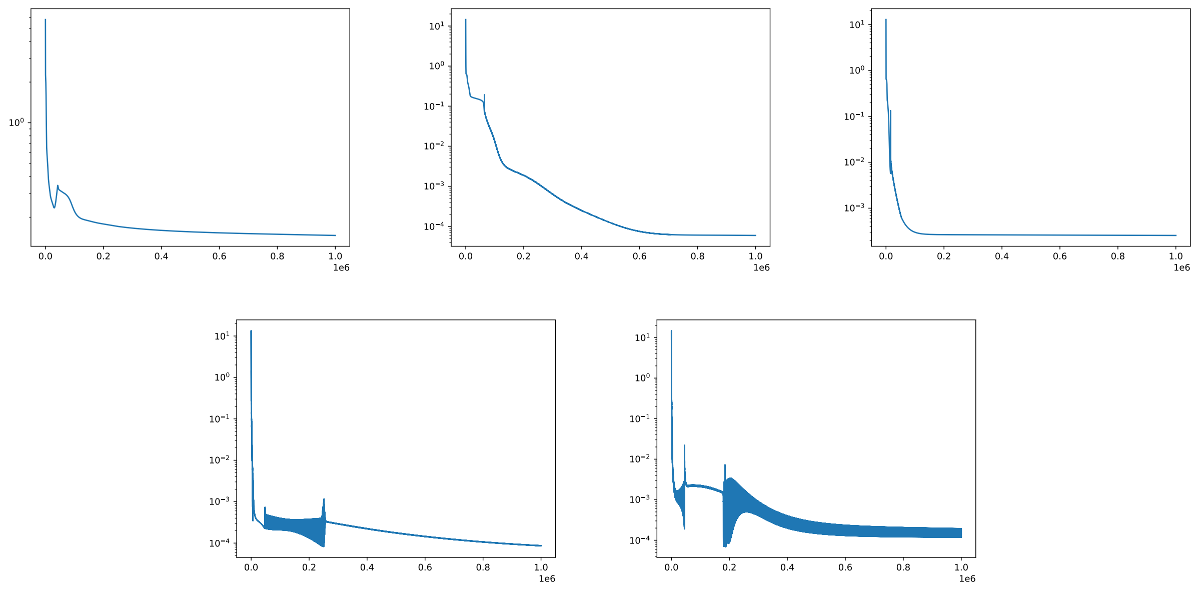


Figure 11: The plots display the objective function when training the ANN on $F(y) = y^3 - y$ with, from left to right, $\tau = 0.001$, 0.005, 0.01, 0.05 and 0.1.

1.1.4 Adjusting h

In order to test the effect of varying the impact of a layer, h = 0.001, 0.1, 0.5, 1 and 10 was used. As seen in Figure 12, h = 0.5 was sufficient to approximate the function. However, using h = 1, the objective function, Figure 13, converges faster while also giving a good function approximation.

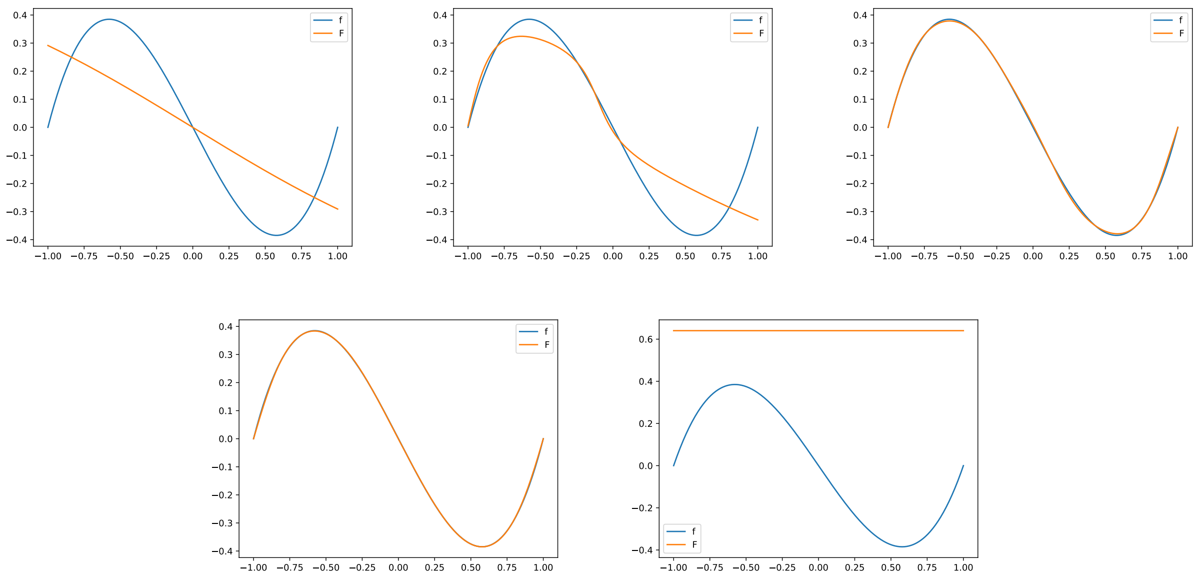


Figure 12: The plots display the approximated functions, in orange, from left to right with h = 0.01, 0.1, 0.5, 1 and 10 when training the ANN on $F(y) = y^3 - y$.

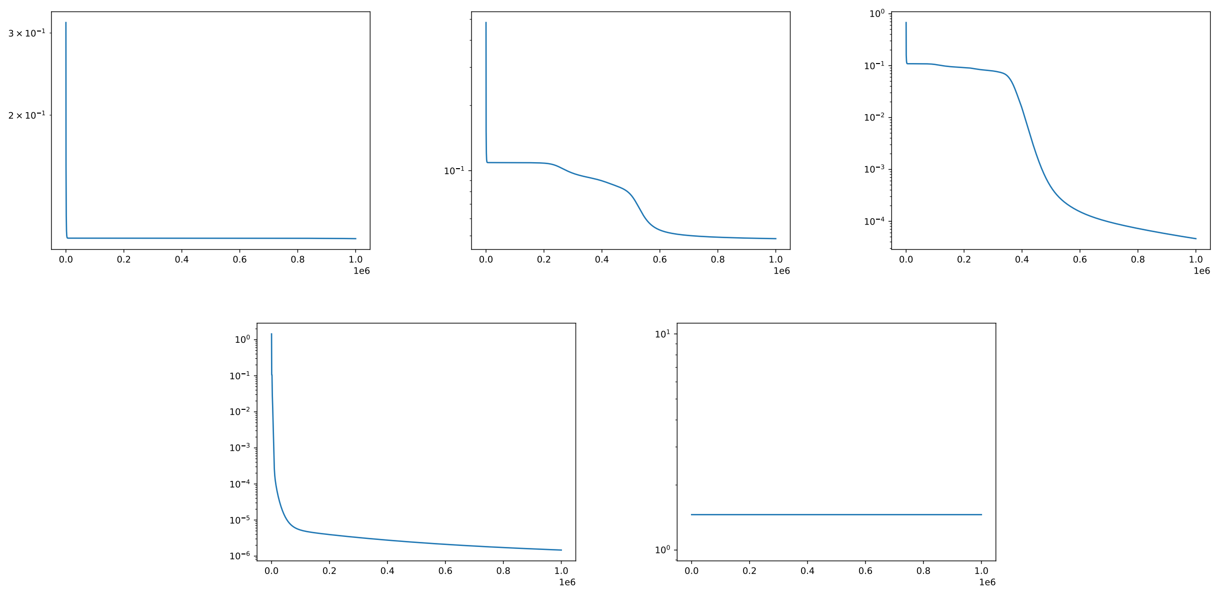


Figure 13: The plots display the objective function when training the ANN on $F(y) = y^3 - y$ with, from left to right, h = 0.01, 0.1, 0.5, 1 and 10.

1.1.5 Adjusting d and K for 2d functions

After testing out different parameters on 1d functions, we continued by testing on the 2d function $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$. Adjusting the number of dimensions and layers of the ANN gives similar results as for the 1d functions. These results are displayed in Figure 14. For K, d = 2, 2 the objective functions does not converge fast enough which would most likely result in a subpar approximated function. However, by increasing d, the objective function converges faster and one could assume by results from testing on 1d functions that the approximated function fits the exact function well. By increasing K, the objective function is not smooth and one cannot know for sure whether or not the result of the function approximation is good enough.

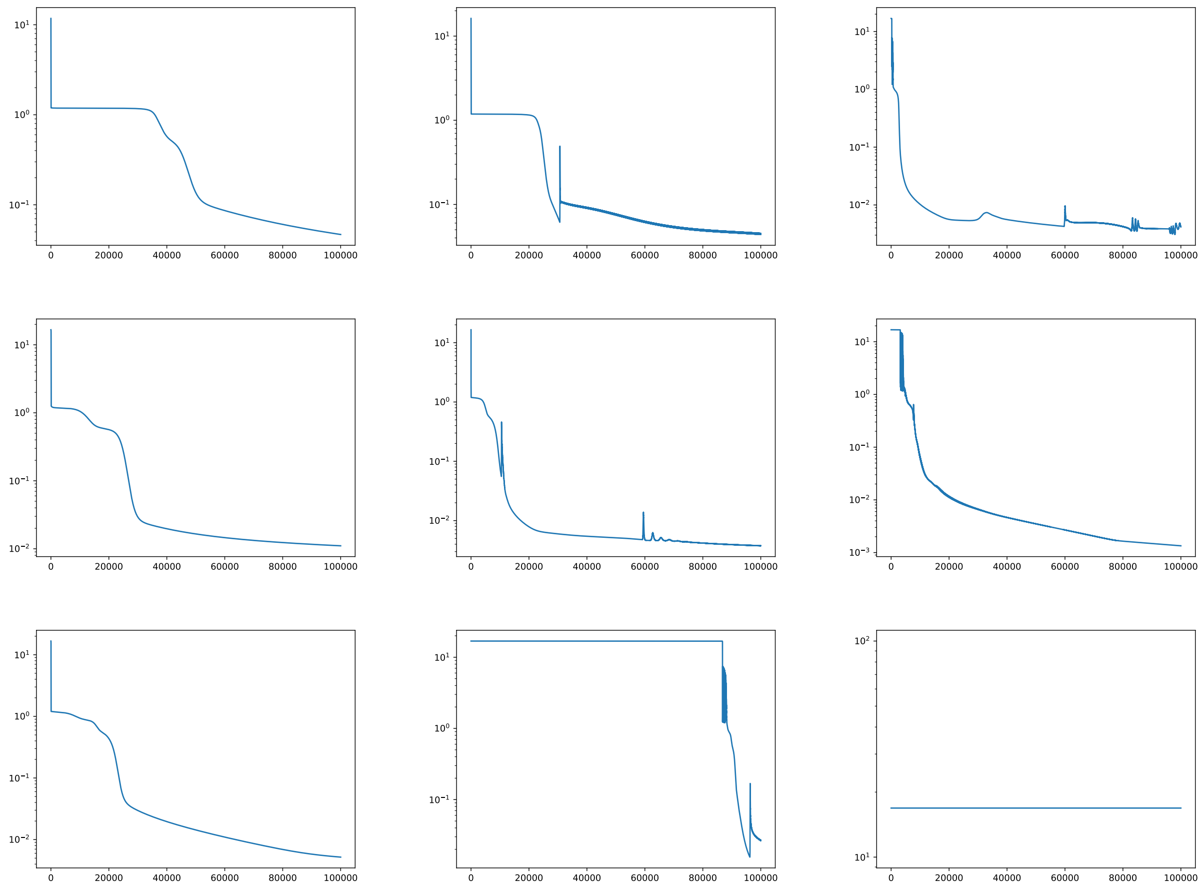


Figure 14: The plots display the objective function when training the ANN on $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$ with with K = 2, 4 and 8 for every column and d = 2, 4, and 8 for every row.

1.1.6 Adjusting tau for 2d functions

The results of adjusting τ when approximating $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$ is displayed in Figure 15. Again, when τ is too large, the objective function becomes jagged. By decreasing τ to 0.01, the resulting objective function is smooth and converges relatively fast. Decreasing it further gives slow convergence and without necessarily giving better results.

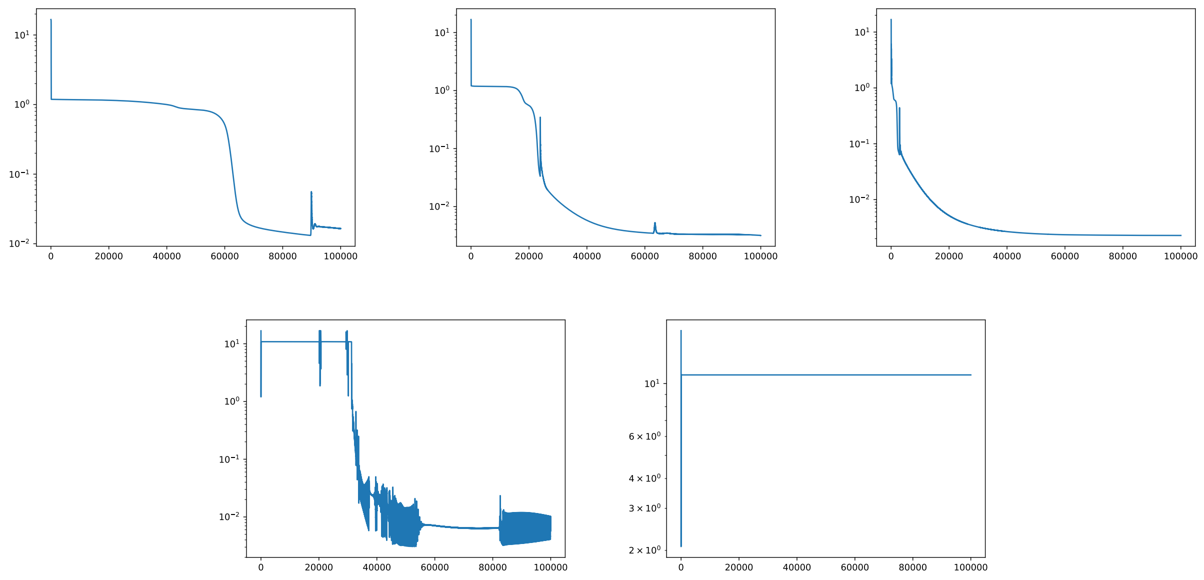


Figure 15: The plots display the objective function when training the ANN on $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$ with, from left to right, $\tau =$ 0.001, 0.005, 0.01, 0.05 and 0.1.

1.1.7 Adjusting I for 2d functions

The results of adjusting I when approximating $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$ is displayed in Figure 16. The testing was performed with I $= 10$, 20 and 50. The objective function for I $= 50$ is for the most part smooth and converges the fastest. It is therefore important to train on a large enough data set, but it is essential to keep in mind that when increasing I, one often has to compensate by decreasing τ.

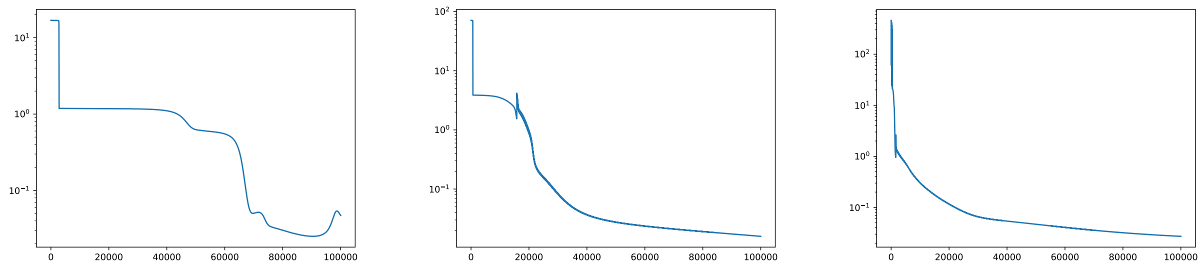


Figure 16: The plots display the objective function when training the ANN on $F(y) = \frac{1}{2}(y_1^2 + y_2^2)$ with, from left to right, I $=$ 10, 20 and 50.

1.2 Training of Hamiltonian function data

The training process on the given Hamiltonian datasets ended up being a lot of manual labour. First of all, we used a network of 4 dimensions with 5 layers for both the potential and kinetic energy approximators. This gives θ a total dimension of $4^2 \cdot 5 + 4 \cdot 5 + 4 + 1 = 105$. Getting larger networks to converge was challenging as they ended up never getting anywhere, similarly to what we saw for 1d functions in Figure 1 and 4, where the highest dimension neural networks ended up doing nothing. Still, even for the 4x5 network, to get it started we had to run a few hundred iterations on a very small data set (100 - 1000 values) with quite a large gradient step size, τ, to get fast but chaotic descent that can work as a starting point for training with larger datasets.

Afterwards, we started training on two data batches at a time (using the concatenate function that was given). Two batches was chosen at a time as this let us first train on batch 0 and 1, then 1 and 2, etc. such that we never surprised the network with completely unfamiliar data. This was done hoping that not completely changing the training data each epoch would reduce the amount the network "unlearns" from the previous data when trying to approximate the new data. Since 50 batches were given, and we wanted some data the network had never seen during the training process, we decides that we would only train on the first 40 batches and leave the last 10 for testing. As such, the procedure of training on batch 0 and 1, and then 1 and 2, continued up to batch 38 and 39, doing 10000 iterations on each set. After this, we trained on larger datasets of 20 batches each, still making sure they overlap sufficiently across epochs so an additional 10000 iterations was done on batch 0-19, then 10-29, and finally 20-39.

We utilised every little bit of time after the new and correct training data was released and let the network train on the entire data set of the first 40 batches. The training process from this point on was checked on periodically to make sure that the objective function was decreasing smoothly. The gradient stepsize, τ, was fine-tuned to accomplish this. Since the new training data was released on Friday, there was about 70-72 available hours of training time. This ended up being a bit less than 10^6 iterations on the entire data set for both T and V. Both ended up having a slow and controlled convergence over this last training phase. However, V seemed to converge much slower than T. After the training process, the network approximating V had an objective function value of J = 13.5 for the entire 40 batches, while the approximation for T had J = 2.75. Since the objective function depends heavily on the amount of simultaneous data that is used, we can divide by the amount of data points. Each batch is 2048 points, which becomes $40 \cdot 2048 = 81920$ data points. This gives an average variance of a single point $\frac{13.5}{81920} = 1.6 \cdot 10^{-4}$ for V and $\frac{2.5}{81920} = 3.6 \cdot 10^{-5}$ for T. The square root of this is the magnitude of the average error for a single point which is approximately 1.3% for V and 0.6% for T.

1.3 Evaluation on test data not used in training phase

Figure 17 shows both true values for the potential V and kinetic term T given from the first ten datasets as well as the approximate values for these calculated from the positions and momenta from the same datasets. The datasets 0 to 9 are sets which the ANN has been trained on, and the degree of fitness for these sets can be attributed to the actual training process.

Figure 18, however, shows true and approximate values for the *last* ten datasets, 40 to 49, which are sets the ANN has never seen. The degree fitness of these datasets cannot be attributed to the sets being familiar to the neural network due to exposure in training, and the fitness must therefore be due to the ANN's actual approximation of the Hamiltonian. The ANN does capture the behaviour of the system to a certain extent.

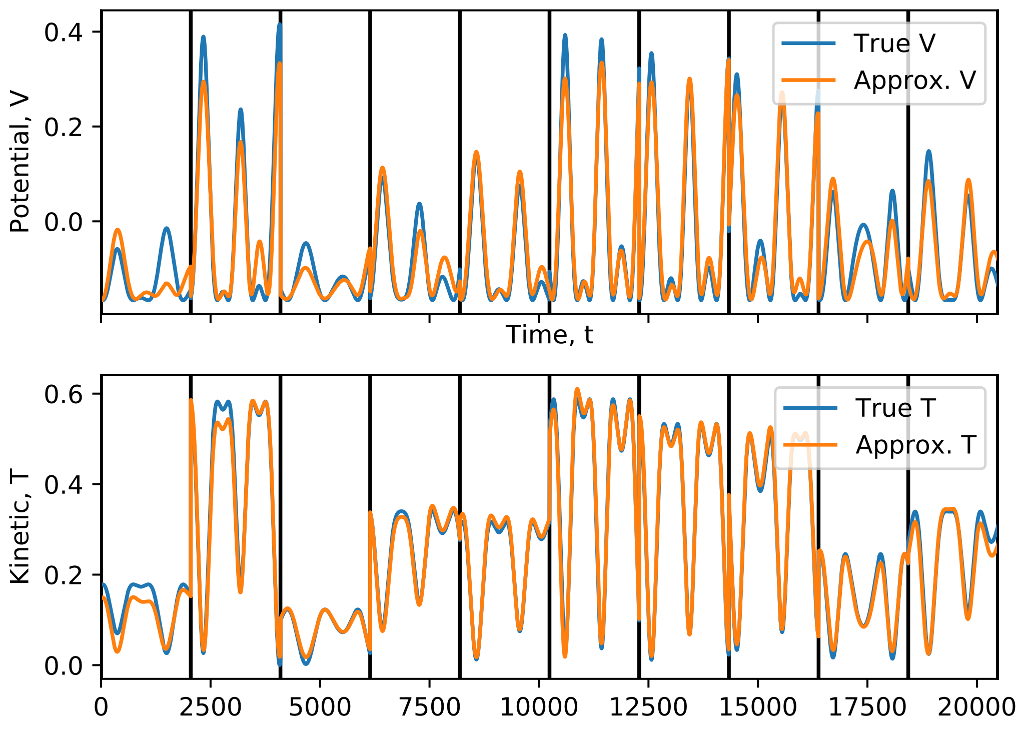


Figure 17: True and approximate values for potential term V and kinetic term T for the first 10 datasets.

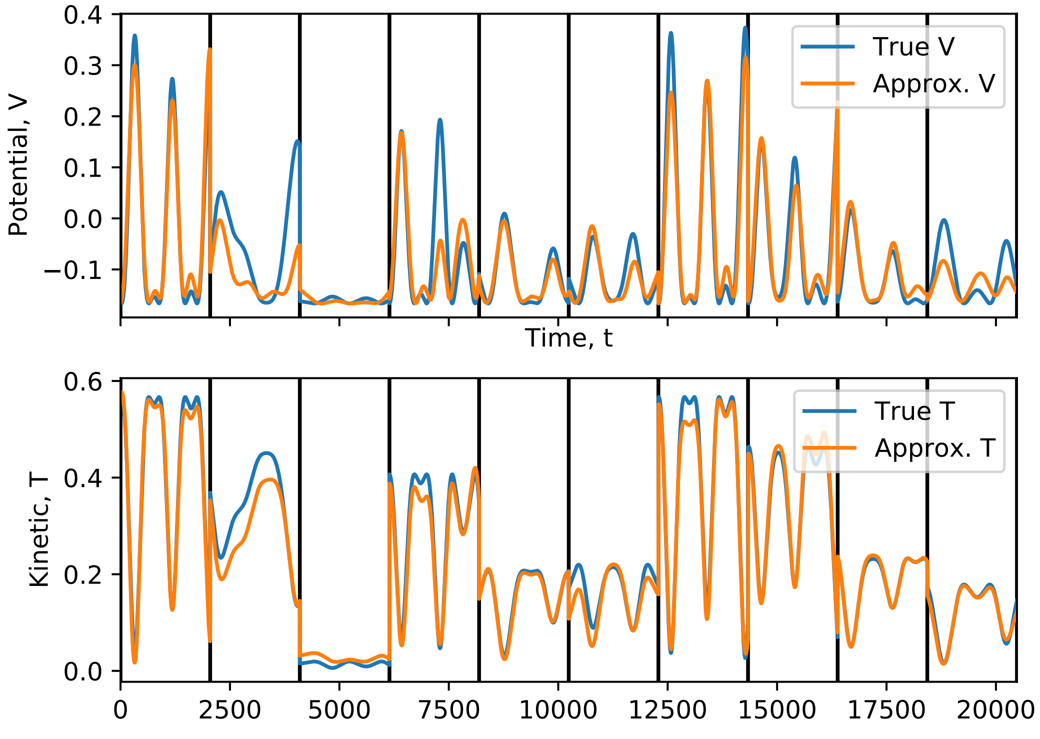


Figure 18: True and approximate values for potential term V and kinetic term T for the last 10 datasets.

2 Gradient of the Neural Network

To compute the gradient with respect to the input of the neural network, we must find the gradient of the composition given in eq. (2). This will result in the chain rule being applied to all $Z^{(k)}$, since they depend on the input. From the forward sweep of the ANN, we get a set of $Z^{(k)}$ which are the result of the successive transformation of the input Y to the k-th hidden layer. Alternatively, the forward sweep can be written as a composition of maps:

$$Z^{(k)}(Y) = \Phi_{k-1} \circ \ldots \Phi_0(Y) \tag{12}$$

From eq. (4) we get that $\tilde{F}(Y) = G \circ Z^{(K)}$ and $Z^{(K)} = \Phi_{K-1} \circ Z^{(K-1)}(Y)$. The gradient of G is therefore:

$$\nabla_y \tilde{F}(Y) = \left(DZ^{(K)}\right)^T \nabla G\left(Z^{(K)}\right) \tag{13}$$

Due to the stepping-procedure in eq. (4), the Jacobian of the k-th intermediate value is:

$$DZ^{(k)} = D\Phi_k\left(Z^{(k)}\right) DZ^{(k-1)} \Rightarrow \left(DZ^{(k)}\right)^T = \left(DZ^{(k-1)}\right)^T \left(D\Phi_k\left(Z^{(k)}\right)\right)^T \tag{14}$$

Note that the input Y is just a series of completely independent vectors y_i, and the gradient is thus computed "per column". The gradient of the ANN with respect to the input y is given in eq. (13), and together with eq. (14), this can be summarized into a matrix multiplication of Jacobian matrices.

$$\nabla_y \tilde{F}(Y;\theta) = \left(D\Phi_0\left(Z^{(0)}\right)\right)^T \ldots \left(D\Phi_{K-1}\left(Z^{(K-1)}\right)\right)^T \nabla G\left(Z^{(K)}\right) \tag{15}$$

It is now necessary to compute $\nabla G(Z^{(K)})$ and $(D\Phi_k)^T$ using the expressions from eq. (1). First, the gradient of G:

$$\begin{aligned} G(y) &= \eta\left(\sum_{i=1}^{d} w_i y_i + \mu\right) \\ \Rightarrow \frac{\partial G}{\partial y_j} &= \eta'\left(\sum_{i=1}^{d} w_i y_i + \mu\right) w_j \\ \therefore \nabla G(y) &= \eta\left(w^T y + \mu\right) w \end{aligned} \tag{16}$$

Then, the more involved process of working out $(D\Phi_k)^T$:

$$\begin{aligned}
\Phi_k(y) &= y + h\sigma\left(W_k y + b_k\right) \\
\left[\Phi_k(y)\right]_i &= y_i + h\sigma\left(\sum_{j=1}^{d}(W_k)_{ij}y_j + b_i\right) \\
\Rightarrow \frac{\partial(\Phi_k)_i}{\partial y_r} = \left[D\Phi_k\right]_{ir} &= \delta_{ir} + h\sigma'\left(\sum_{j=1}^{d}(W_k)_{ij}y_j + b_i\right)(W_k)_{ir} \\
\therefore (D\Phi_k)^T &= \delta_{ir} + h\sigma'\left(\sum_{j=1}^{d}(W_k)_{ij}y_j + b_i\right)(W_k)_{ir} \qquad (17)
\end{aligned}$$

It should be noted that $(D\Phi_k) = \left[D\Phi_k\right]_{ri}$ by the definition given in the Project 2 Supplement. This results in $(D\Phi_k)^T = \left[D\Phi_k\right]_{ir}$. For sake of simplicity, let $\sigma'_{k,i} = \sigma'\left(\sum_{j=1}^{d}(W_k)_{ij}y_j + b_i\right)$ Conveniently, multiplication of $(D\Phi_k)^T$ and a vector v gives an r-th component of:

$$\begin{aligned}
\left[(D\Phi_k)^T v\right]_r &= \sum_{i=1}^{d}\left[D\Phi_k\right]_{ir} v_i = v_i\delta_{ir} + h\sum_{i=1}^{d}\sigma'_{k,i}[W_k]_{ir}v_i \\
\sigma'_{k,i}\cdot v_i \Rightarrow \sigma'_k \odot v, &\qquad (W_k)_{ir} = W_k^T, \qquad v_i\delta_{ir} = v_r \\
\left[(D\Phi_k)^T v\right]_r &= v_r + h\sum_{i=1}^{d}(W_k)_{ir}(\sigma'_{k,i}\odot v_i) \\
\therefore (D\Phi_k)^T v &= v + W_k^T\left(h\sigma'(W_k y + b_k)\odot v\right) \qquad (18)
\end{aligned}$$

Since $Z^{(k)}$ are already known, it is possible to perform the computation of the gradient of the ANN if we also have the derivatives of the activation- and hypothesis functions. These are:

$$\sigma'(x) = 1 - \tanh^2 x \qquad \eta'(x) = \frac{1}{4}\left(1 - \tanh^2\frac{x}{2}\right)$$

With these expressions it is possible to implement this method. The pseudocode for this implementation is given in the following subsection.

Implementation of the derived gradient

The implementation of the derived gradient can be written as:

```
1: compute $Z^{(k)}$ in forward sweep
2: acc = $\nabla G(Z^{(K)}) = \eta'(w^T Z^{(K)} + \mu)w$        ▷ acc is accumulator
3: for $k = K-1,\dots,0$ do
4:     acc = acc + $W_k^T\left(h\sigma'(W_k Z^{(k)} + b_k)\odot \text{acc}\right)$        ▷ Expansion of $(D\Phi_k)^T$
5: end for
6: return acc
```

The python-implementation is found as the `getGradANN` function in the `ann.py` file.

2.1 Numerical gradient of the ANN

When running test it became apparent that the implementation of $\nabla_y \tilde{F}$ did not function properly. It is not known whether this is an issue with the implementation in code, faulty logic in the pseudocode or an error in the derivation. Since the pseudocode is faithful to the project supplement, this is least likely to be the case. Code implementation and derivation error are still likely sources of error.

Nevertheless, it was decided to construct a numerical computation of the gradient by computation of a small perturbation around the point of interest. The following approximation to the gradient was used:

$$\left[\nabla_y \tilde{F}(y;\theta)\right]_i \approx \frac{\tilde{F}(y + \frac{\Delta y}{2} e_i) - \tilde{F}(y - \frac{\Delta y}{2} e_i)}{\Delta y} \tag{19}$$

In eq. (19) the factor e_i is the i-th canonical basis vector of $\mathbb{R}^d$. Suppose $E = [e_1, \dots, e_{d_0}]$ is the collection of the first d_0 canonical basis vectors and the padding scheme to be to expand a vector by appending zeros. Since the only elements of interest in the gradient are the first d_0 elements, the matrix E can be sent through the ANN simultaneously to compute all required components of the gradient with respect to the input.

Therefore, the following holds:

$$\nabla_y \tilde{F}(y;\theta) \approx \frac{\tilde{F}(y + \frac{\Delta y}{2} E) - \tilde{F}(y - \frac{\Delta y}{2} E)}{\Delta y} \tag{20}$$

The code for implementing this is more straightforward than the analytical gradient of the ANN, but yielded great results. The analytical method computes the gradient with respect to all I input vectors in one forward sweep and one sweep resembling a back propagation, while the numerical method requires a forward sweep of dimension d_0 to be computed for each input vector, and is therefore slower. For the given problem size, this is not an issue, as the training time is short. For larger scale problems this numerical method of computing the gradient is likely to become unwieldy.

3 Integral methods for the Hamiltonian

In order to find trajectories of the Hamiltonian function with any starting values, integral methods such as the symplectic Euler method and the Størmer-Verlet method can be applied to the system.

3.1 Testing on the given Hamiltonians

In order to make sure the code is correct and the methods are sufficiently accurate, testing was performed on two different separable Hamiltonian problems.

The nonlinear pendulum

For the nonlinear pendulum problem, p and q are scalars.

$$H(p,q) = T(p) + V(q) = \frac{1}{2}p^2 + mgl(1 - \cos q) \tag{21}$$

The code is constructed in such a way that $\frac{\partial T}{\partial p}$ and $\frac{\partial V}{\partial q}$ needs to be found.

$$\frac{\partial T}{\partial p} = p, \qquad \frac{\partial V}{\partial q} = mgl \sin q \tag{22}$$

Both the symplectic Euler method and the Størmer-Verlet method, fig. 19, preserves the Hamiltonian along trajectories as the plot shows a closed loop. The values used for the testing was $m = l = 1$, $g = 9.81$, $p_0 = 1.3$ and $p_1 = 0$.

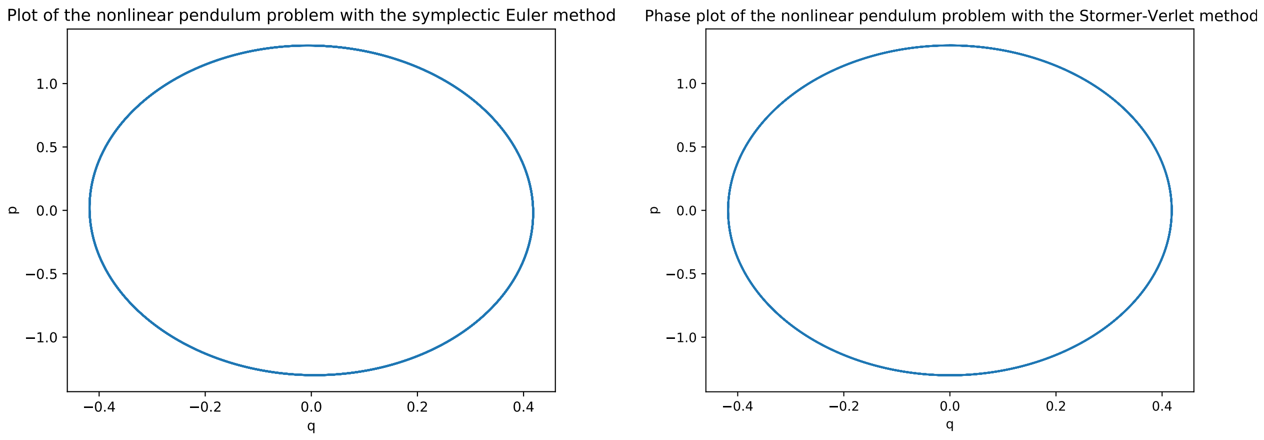


Figure 19: The phase plots display the relationship between the position, q, and momentum, p in the nonlinear pendulum problem with the symplectic Euler method and the Størmer-Verlet method

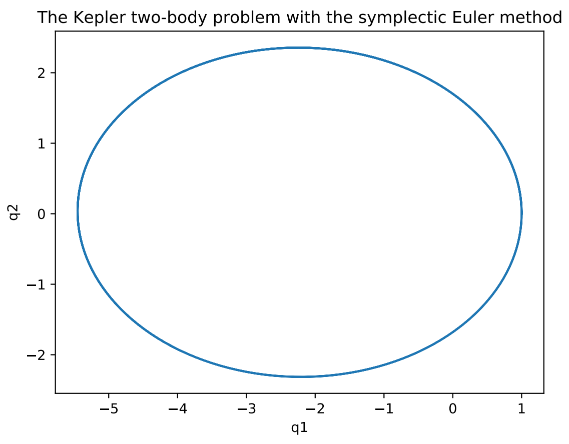
Kepler two-body problem

The same procedure as for solving the nonlinear pendulum problem is used. However, for the Kepler two-body problem there are vectors of the momentum and positions and thus the gradient.

$$\frac{\partial T}{\partial p_1} = p_1, \qquad \frac{\partial T}{\partial p_2} = p_2$$
$$\Rightarrow \frac{\partial T}{\partial p} = p$$

$$\frac{\partial V}{\partial q_1} = \frac{q_1}{(q_1^2 + q_2^2)^{\frac{3}{2}}}, \qquad \frac{\partial V}{\partial q_2} = \frac{q_2}{(q_1^2 + q_2^2)^{\frac{3}{2}}}$$
$$\Rightarrow \frac{\partial V}{\partial q} = \begin{bmatrix} \frac{q_1}{(q_1^2+q_2^2)^{\frac{3}{2}}} \\ \frac{q_2}{(q_1^2+q_2^2)^{\frac{3}{2}}} \end{bmatrix}$$

Both the symplectic Euler method and the Størmer-Verlet method, fig. 20, preserve the Hamiltonian along trajectories as the plot shows a closed loop. The starting values used was $p_0 = [0\ 1.3]^T$ and $q_0 = [1\ 0]^T$.



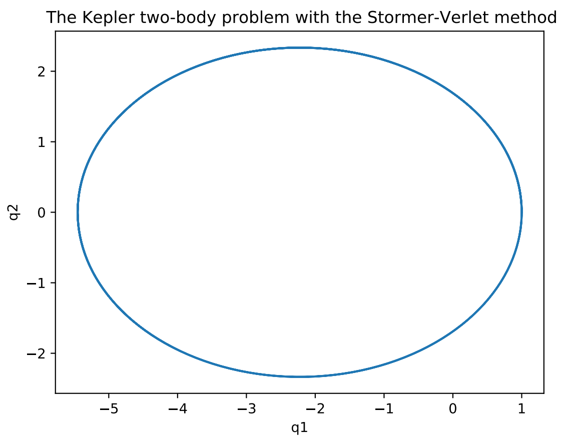


Figure 20: The plots display the relationship between the positions q_1 and q_2 in the Kepler two-body problem with the symplectic Euler method and the Størmer-Verlet method

3.2 Testing on the unknown Hamiltonian

Figure 21 shows the evolution of the total energy of the system with respect to time in batch number 2, one of the trained sets. As expected, the true values for T and V are combined into H so that the energy is constant. The ANN disagrees with this energy as it is not a perfect approximation of the true Hamiltonian, for the given positions and momenta, the energy is all but constant. Since the symplectic Euler and Størmer-Verlet methods are concerned with a particle navigating the space where the ANN approximation of the Hamiltonian applies, the positions and momenta from these methods represent an actually valid path given the ANN as Hamiltonian. From the figure, it is clear that both methods succeed in preserving the energy as the variation is negligible. The methods will obviously result in a different path than that in the true Hamiltonian, so it remains to be seen whether the actual paths are close.

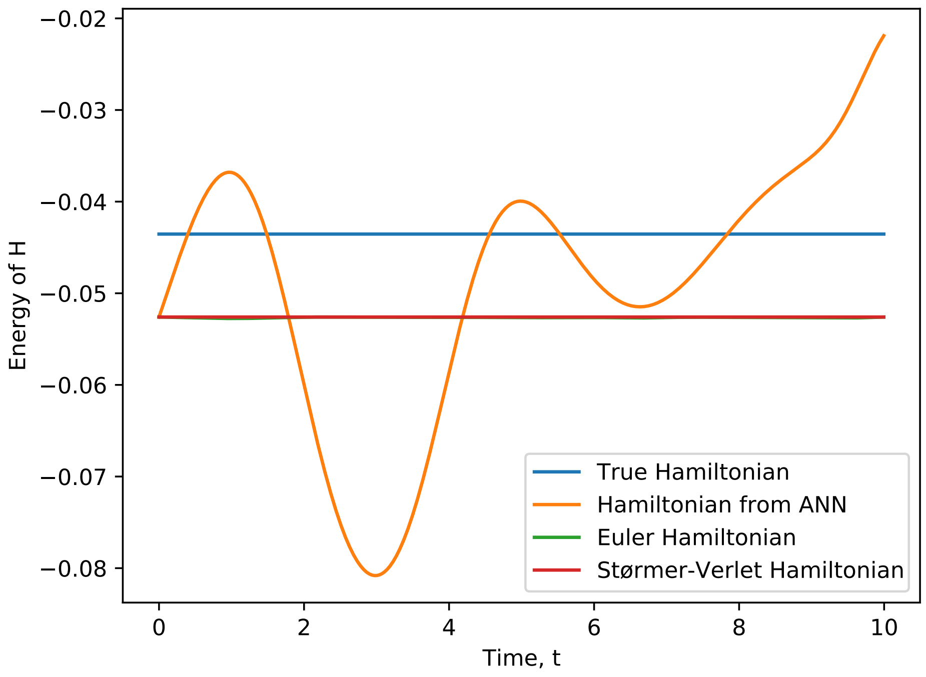


Figure 21: The plot displays the total energy, H, over time for the true Hamiltonian, the Hamiltonian from the ANN, the symplectic Euler method for the approximated Hamiltonian and the Størmer-Verlet method for the approximated Hamiltonian for batch 2.

Figure 22 shows the total energy for batch 47, one of the foreign datasets. The discrepancies between symplectic Euler and Størmer-Verlet are more visible in this dataset, and it is clear that Størmer-Verlet is better at preserving the total energy.

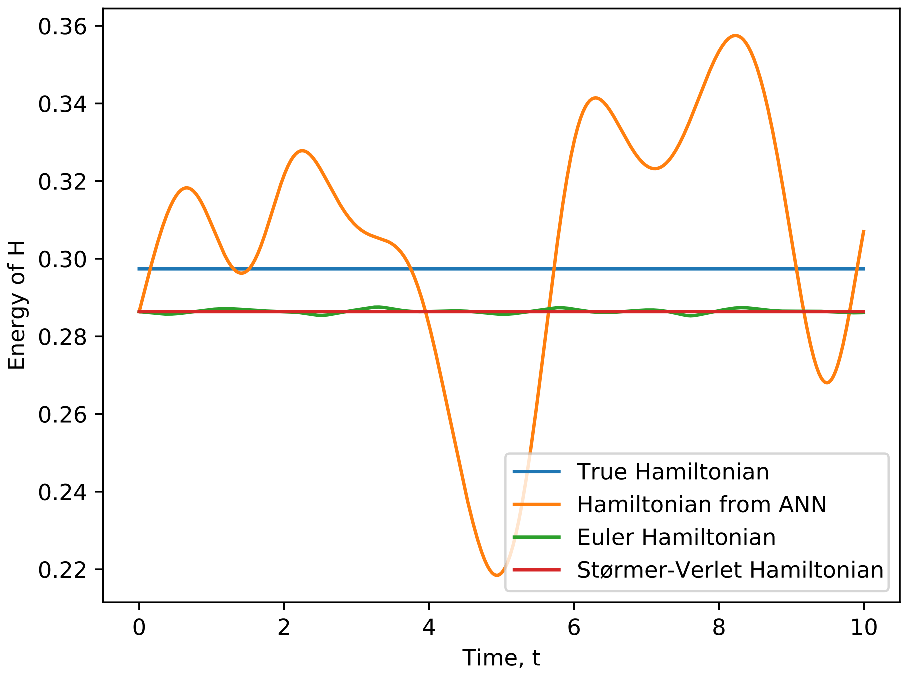


Figure 22: The plot displays the total energy, H, over time for the true Hamiltonian, the Hamiltonian from the ANN, the symplectic Euler method for the approximated Hamiltonian and the Størmer-Verlet method for the approximated Hamiltonian for batch 47.

The components for the momentum vector for the particle in batch 2 are shown in Figure 23 and the components for the positional vector are shown in Figure 24. The stepsize, dt, used for the integrator methods is the same as the stepsize in the dataset. For the momentum, the evolution is rather faithful to the true values until around the 300th step, where the curves diverge. The position is a good fit until around the 600th step, but after that some of the components fail to mimic the shape of the true values.

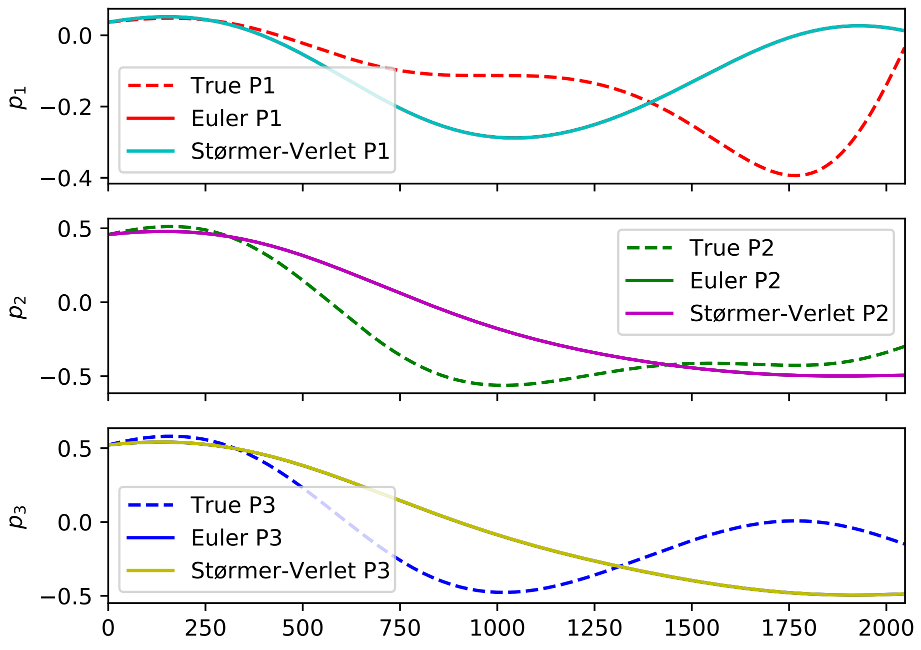


Figure 23: The plots display componentwise evolution of momentum of the exact Hamiltonian, the symplectic Euler method for the approximated Hamiltonian and Størmer-Verlet method for the approximated Hamiltonian for batch 2.

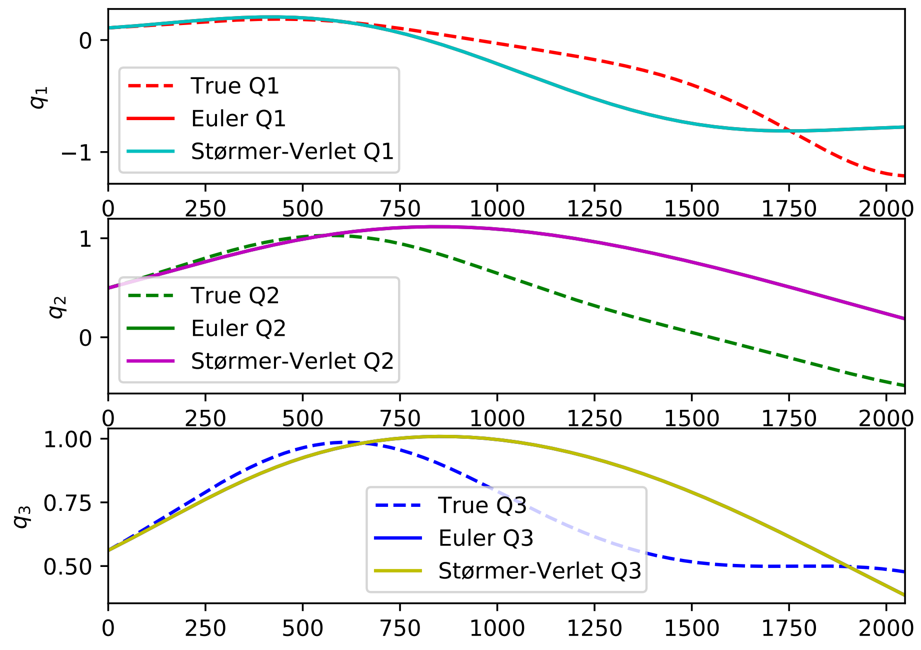


Figure 24: The plots display componentwise evolution of position of the exact Hamiltonian, the symplectic Euler method for the approximated Hamiltonian and Størmer-Verlet method for the approximated Hamiltonian for batch 2.

Figure 25 and Figure 26 show the components of the momentum- and position vectors respectively for batch 47. The behaviour of the true path is mimicked by all components except component two for both momentum and position. The approximation is good until the 700th step for momentum and the 1000th step for position.

It is worth noting that the second component of the momentum vector diverges after about 250 steps, and the second component becomes rather constant after 1000 steps. The fitness of the third component of position is not especially good, even before 250 steps, but the shape of the path is not far off.

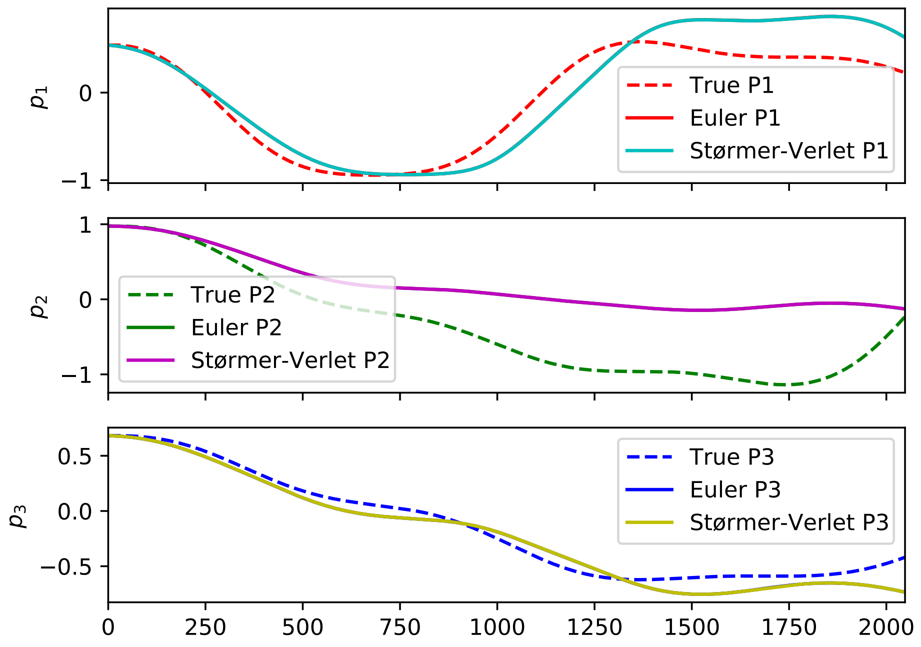


Figure 25: The plots display componentwise evolution of momentum of the exact Hamiltonian, the symplectic Euler method for the approximated Hamiltonian and Størmer-Verlet method for the approximated Hamiltonian for batch 47.

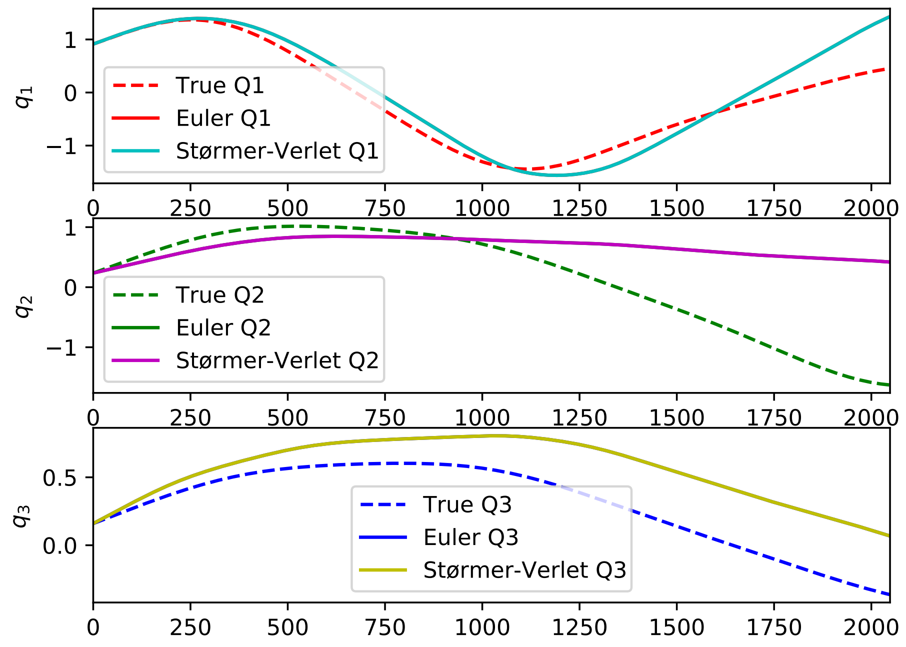


Figure 26: The plots display componentwise evolution of position of the exact Hamiltonian, the symplectic Euler method for the approximated Hamiltonian and Størmer-Verlet method for the approximated Hamiltonian for batch 47.

4 Conclusion

In this project we implemented a ResNet model artificial neural network and trained it to approximate an unknown Hamiltonian of a system. The trained ANN approximated the Hamiltonian to a certain extent, and numerical integration methods to propagate a particle in the given system showed promising results. The propagation had a good degree of fitness for at least 200 steps in time. The stepsize in these methods was similar to the time-step in the given data and 200 steps is equivalent to one second.

We discovered that it was infeasible to increase the number of hidden layers and the dimension of the hidden layers above a certain threshold for our implementation. Further work could be to attempt to understand and resolve this issue. It is reasonable to assume that an increase in the number of layers or dimension of layers would give the ANN a greater ability to approximate the Hamiltonian.

A Explanation of symbols in the code

Table 2 is a dictionary of symbols used in the code, and what they mean in relation to the project:

`Y`	Uppercase Y	(d0xI) matrix of separate input data
`c`	Lowercase c	I-vector of exact values
`d`	Lowercase d	Dimension of hidden layers vector spaces
`K`	Uppercase K	Number of hidden layers
`h`	Lowercase h	Stepsize between layers
`tau`	Greek letter τ	Learning factor with which to update θ
`it_max`		Number of iterations before forced stopping
`tol`		Target degree of fitness
`W`	Uppercase W	Weights between hidden layers
`b`	Lowercase b	Offsets between hidden layers
`w`	Lowercase w	Weights for collapsing the terminal layer
`mu`	Greek letter μ	Offset for collapsing the terminal layer
`(W, b, w, mu)`	$=\theta$	The θ parameter for the ANN, complete description of the ANN
`dJ`	$\nabla_\theta J$	Gradient of objective function wrt. `W, b, w, mu`
`J`	Uppercase J	Objective function measuring error

Table 2: Explanation to some symbols used in the code and their mathematical counterparts.