TMA4215 - Numerical Mathematics

Project 2

1 Introduction

This project aims to investigate the use of a neural network to train a model for finding the Hamiltonian for a given dataset. The code developed is structured into several files, and can be summarized in the following:

- `main.py` - the main entry point for the program, which gives an example of ...
- `ResNet.py` - the neural network, in addition to functions for computing gradients
- `integration.py` - the numerical integration schemes
- `examples.py` - some examples of how the neural network is used on synthetic data
- `hamiltonians.py` - example of nonlinear pendulum hamiltonian
- `project_2_data_acquisition.py` - supplied file for acquiring Hamiltonian data

2 Neural network

The neural network is implemented in `ResNet.py`, with some examples of its use provided in `examples.py`. The `ResNet` class provides a general framework for the ResNet architecture, where the user can set activation and hypothesis function, optimization method and optimization parameters, and architecture parameters such as number of hidden layers and dimension of hidden layers.

Synthetic data

Synthetic data was first generated to test the network and validate its performance. This was done by uniformly sampling input data for a variety of functions, training the network on the input and output data, and comparing the output of the network to the actual function output for test data. As seen from the results in Figure 1, the network adequately estimates the functions $F_1(y) = \frac{1}{2}y^2$ and $F_2(y_1, y_2) = \frac{1}{2}(y_1^2 + y_2^2)$.

Validation of performance

The performance of the network is a combination of accuracy and computational time. The accuracy can be measured on the test set by the mean squared error (MSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(c_i - \tilde{F}(y_i;\theta))^2, \tag{1}$$

where n is the number of data points, c_i is the output data for the test set and $\tilde{F}(y_i;\theta)$ is the output of the network with test set input data y_i. The training set is not used for measuring accuracy, as it will often lead to overfitting, unless the training data captures the estimated function well. Computational time is simply measured while training, where we desire a low training time, but enough to give accurate predictions. This is coupled with convergence, where we desire a fast convergence, meaning low computational time. The convergence time is dependent on some criteria for when the optimization has converged.

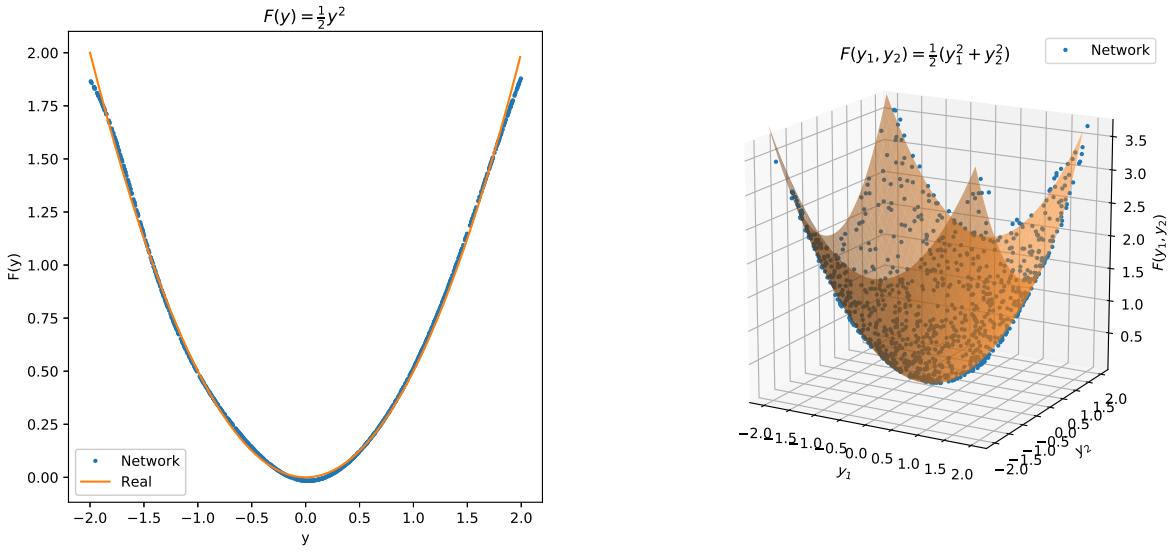
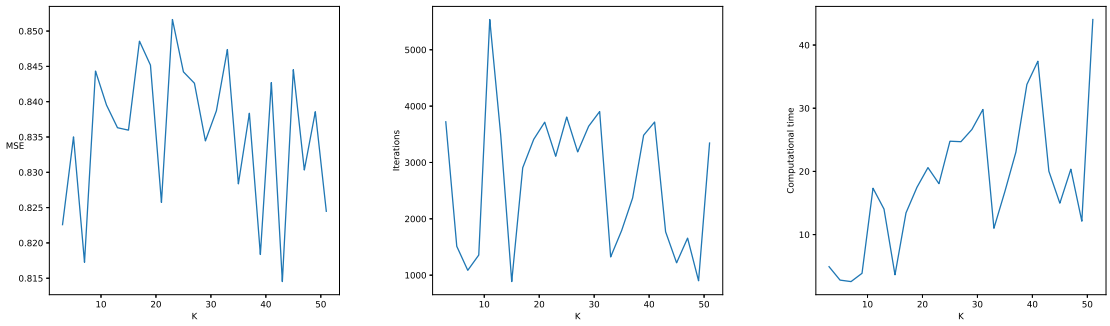


Figure 1: *Network output vs. real output for synthetic test data*

Parameter identification

The parameters to tune in the network are K, d, h and optimization parameters as τ in gradient descent and α in ADAM. The optimal parameters are those that give optimal performance, where optimal performance is some middle-ground between accuracy and computational time. With the criteria of performance defined, we can systematically search for the best combination of parameters. An example is given in Figure 2, where the optimal number of hidden layers, K, for estimating the function $F_1(y)$ is searched for. We see that the MSE is not heavily affected by the number of layers, and that number of iterations before convergence seemingly random. The computation time, however, is seen to increase with the number of hidden layers. For this use case, we then conclude that fewer hidden layers give better performance.



(a) *MSE* ***(b)*** *Iterations* ***(c)*** *Computational time*

Figure 2: *Tests for finding the optimal number of hidden layers, K.*

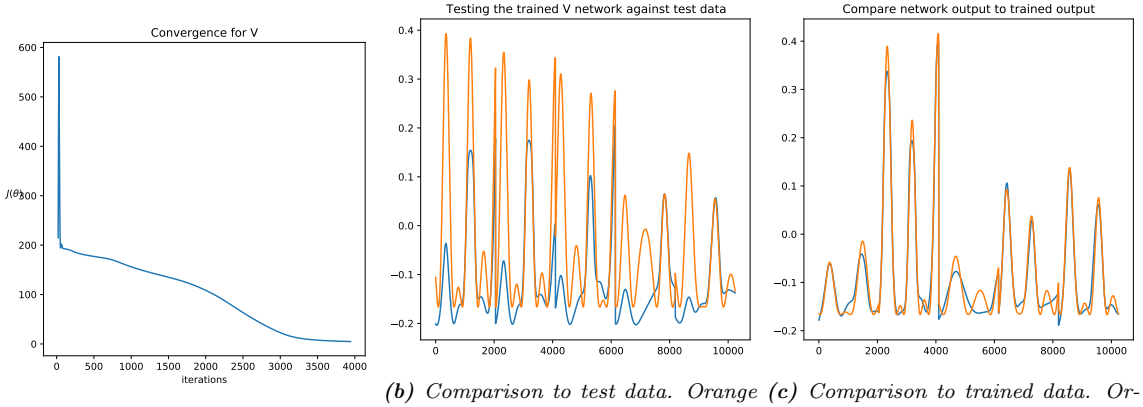
A grid-search over all parameters can be conducted, but this takes a very long time, to the extent that it is not feasible. This was especially the case for the Hamiltonian data, which was much more demanding to train than the synthetic data. An approach by trial-and-error is therefore more convenient, at the cost of not necessarily finding the optimal parameters.

3 Performance on provided Hamiltonian data

The network was trained on the provided data set for approximating the unknown Hamiltonian $H(p,q) = T(p) + V(q)$, where two independent networks were trained for $T(p)$ and $V(q)$, respectively. Training the network was done with $K = 5$, $d_0 = 2 \cdot d = 6$ and $h = 0.1$, as these parameters proved a middle way between training time and accuracy on test data. Training with both gradient descent and ADAM was tested, and showed that gradient descent was actually unable to converge to anything, and so ADAM was exclusively used for its faster and robust convergence.

As the training process was much slower on the provided data set, the number of data points that were used for training was restricted to five batches, meaning 10240 data points. Additionally, different termination criteria were used, with $J(\theta) = 0.5$ and $J(\theta) = 0.05$ before termination being tested and compared.

The first batch was tested with $J(\theta) = 0.5$. This seemed to perform well, and was compared to the performance of the network with only one batch of data points, or 2048 data points. Both networks were initially tested with the trained function output as a sanity check that the network was trained properly. It can be seen that the V network does properly predict the output. The T network is able to properly predict the shape of the function, but is severely off by a scale. See Figure 6c for results.

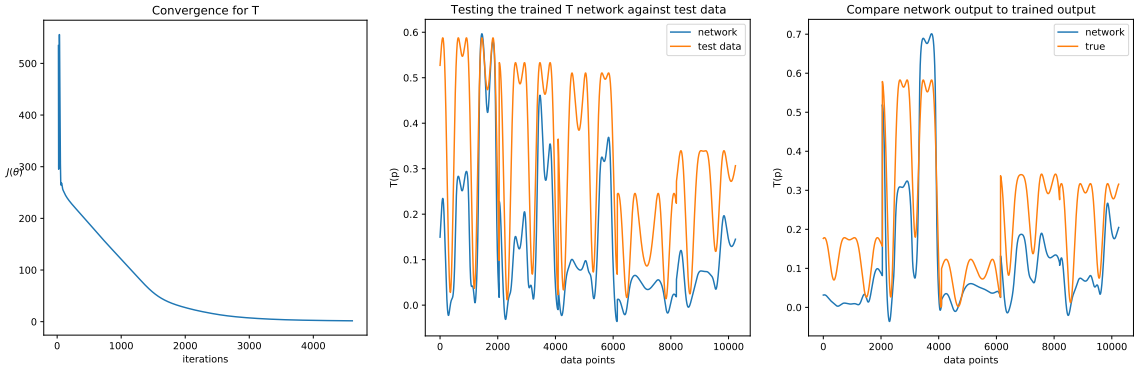


(a) *Convergence of V network*

(b) *Comparison to test data. Orange is test data, and blue is network prediction.*

(c) *Comparison to trained data. Orange is test data, and blue is network prediction.*

Figure 3: *Results of training V network with 5 batches of data, or 10240 datapoints.*

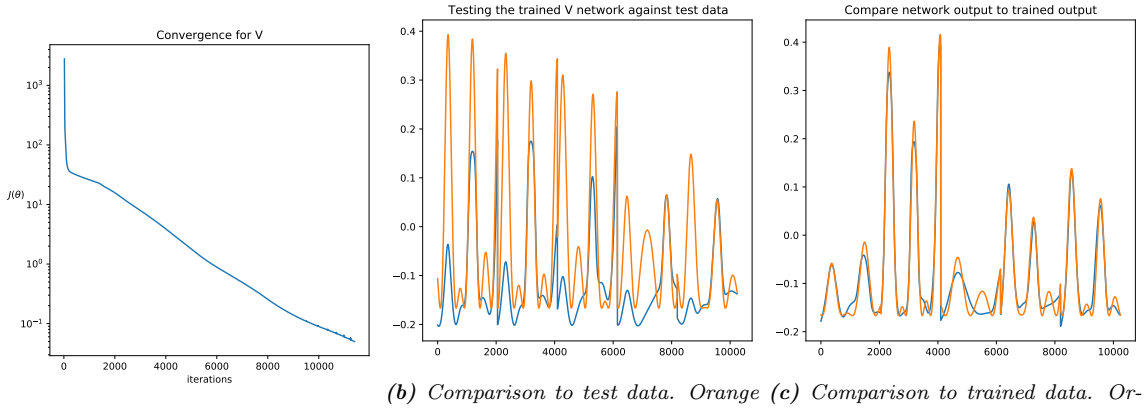


(a) *Convergence of T network*

(b) *Comparison to test data*

(c) *Comparison to trained data*

Figure 4: *Results of training T network with 5 batches of data, or 10240 datapoints with $J(\theta) = 0.5$ as termination criterion used.*

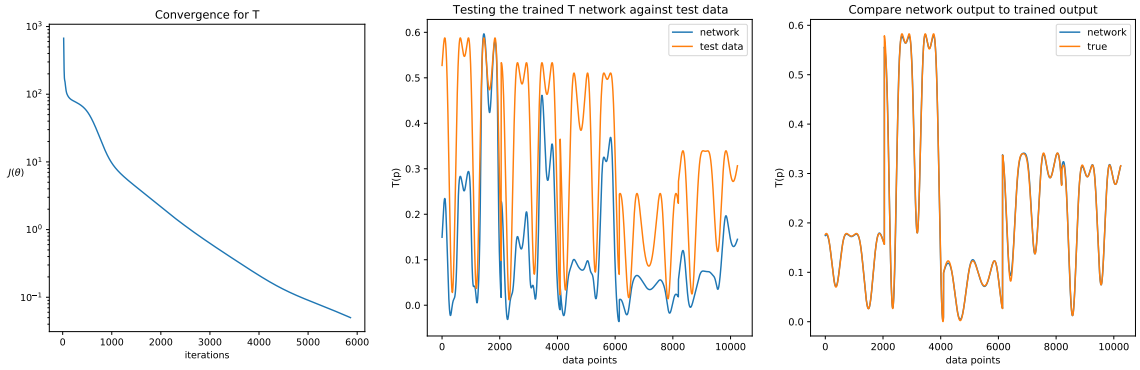


(a) Convergence of V network

(b) Comparison to test data. Orange is test data, and blue is network prediction.

(c) Comparison to trained data. Orange is test data, and blue is network prediction.

Figure 5: *Results of training V network with 5 batches of data, or 10240 data points. Termination criterion is $J(\theta) = 0.05$*



(a) Convergence of T network

(b) Comparison to test data

(c) Comparison to trained data

Figure 6: *Results of training T network with 5 batches of data, or 10240 datapoints with $J(\theta) = 0.05$ as termination criterion used.*

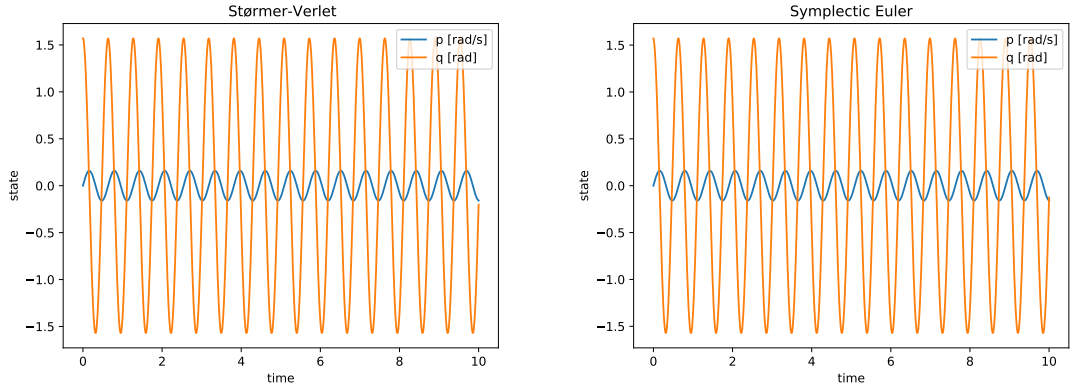
4 Integration methods

An implementation of both symplectic Euler and Størmer-Verlet were made for integrating the state space p, q with the gradient of the learnt $V(q)$ and $T(p)$, and can be found in `integrate.py`. Both implementations were tested on the nonlinear pendulum system as test system, with the trajectories given in Figure 7 and the phase plots in Figure 8.

5 Computing the gradient of a neural network

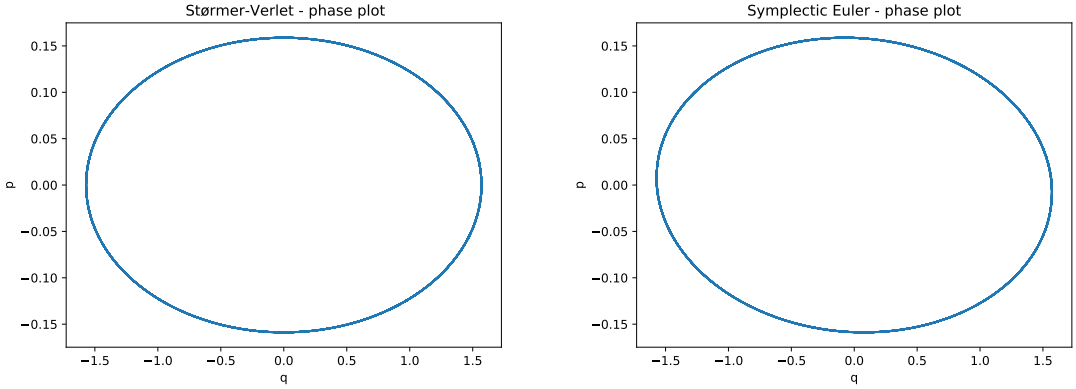
Derivation

The main goal for the network implementation was to be able to train the network on data of two unknown functions $V(q)$ and $T(p)$ that satsfy the Hamiltonian relation



(a) Størmer-Verlet. *(b) Symplectic Euler.*

Figure 7: *Integrated state space with the two integration methods symplectic Euler and Størmer-Verlet. The initial conditions are $p = 0$ and $q = \pi/2$*



(a) Størmer-Verlet. *(b) Symplectic Euler.*

Figure 8: *Phase plots with the two integration methods symplectic Euler and Størmer-Verlet. The initial conditions are $p = 0$ and $q = \pi/2$*

$$H(p,q) = V(q) + T(p), \tag{2}$$

$$\dot{q} = \frac{\partial H}{\partial p}(p,q) = \frac{\partial T}{\partial p}(p), \tag{3}$$

$$\dot{p} = -\frac{\partial H}{\partial q}(p,q) = -\frac{\partial V}{\partial q}(q). \tag{4}$$

In order to do this, the gradients, now called $\nabla T = \partial T/\partial p$ and $\nabla V = \partial V/\partial q$, are needed. The mathematics to obtain these are based on the mathematical framework of the ResNet architecture, and is based on the hint note posted on Blackboard. The formula for the gradient function is computed from the form of the learnt function $F(y)$, which is given by

$$F(y) = G(\Psi_K(y)) \tag{5}$$

where $G(y)$ is the transfer function of the final layer and $\Psi_K(y)$ is the convolution of the transfer functions of the K hidden layers. Computing the gradient gives, by the chain rule,

$$\nabla F = \left(\frac{\partial \Psi}{\partial y}\right)^{\mathsf{T}} \cdot \nabla G. \tag{6}$$

We find ∇G as

$$\begin{aligned}\nabla G &= \nabla_y(\eta(w^\mathsf{T} y + \mu)) \\ &= \eta'(w^\mathsf{T} y + \mu) w\end{aligned}$$

by the chain rule. Note that this in practice is just w, as $\eta(x) = x$ is used.

The Jacobian $\dfrac{\partial \Psi_K}{\partial y}$ is computed recursively as

$$\begin{aligned}\frac{\partial \Psi_K(y)}{\partial y} &= \frac{\partial \Phi_K(\Psi_{K-1}(y))}{\partial y} \\ &= \frac{\partial \Phi_K}{\partial \Psi_{K-1}} \cdot \frac{\partial \Psi_{K-1}}{\partial y}\end{aligned}$$

where $\Phi_K(y)$ is the transfer function of hidden layer K. Inserting into (6) gives

$$\nabla F = \left(\frac{\partial \Psi_{K-1}}{\partial y}\right)^\mathsf{T} \cdot \left(\frac{\partial \Phi_K}{\partial \Psi_{K-1}}\right)^\mathsf{T} \cdot \nabla G \tag{7}$$

which holds recursively. In practice, we only need to evaluate the vector that is retrieved from $\left(\dfrac{\partial \Phi_k}{\partial \Psi_{k-1}}\right)^\mathsf{T} A$, which is, given that $\Phi_k(\Psi_{k-1})$ is given by

$$\Phi_k(\Psi_{k-1}) = \Psi_{k-1} + h\sigma\left(W\Psi_{k-1} + b\right) \tag{8}$$

is found to be

$$\left(\frac{\partial \Phi_k}{\partial \Psi_{k-1}}\right)^\mathsf{T} A = A + W^\mathsf{T}\left(h\sigma'\left(W\Psi_{k-1} + b\right) \odot A\right) \tag{9}$$

where $\odot$ denotes elementwise multiplication, or the Hadamard product.

Results

The gradient implementation was tested on the test function $1 - \cos(y)$ with derivative $\sin(y)$. The results are found in Figure 9.

As the computation of the gradient was seemingly working, it was now tested on the trained p and q data to inspect whether it was possible to integrate a solution of p and q close to the solution. The results of this attempt can be found in Figure 10. The results are, unfortunate, not accurate enough to be practically usable. The results are from networks trained on 10240 data points that terminates at $J(\theta) = 0.05$, but this may not be enough training to properly capture the gradient of the Hamiltonian. Another explanation is that the gradient implementation is not numerically stable, and so small offsets causes the integration to diverge. Unfortunately, the results were not further inspected.

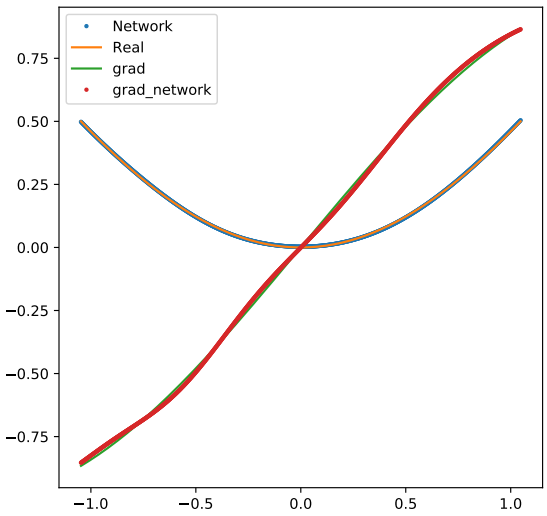
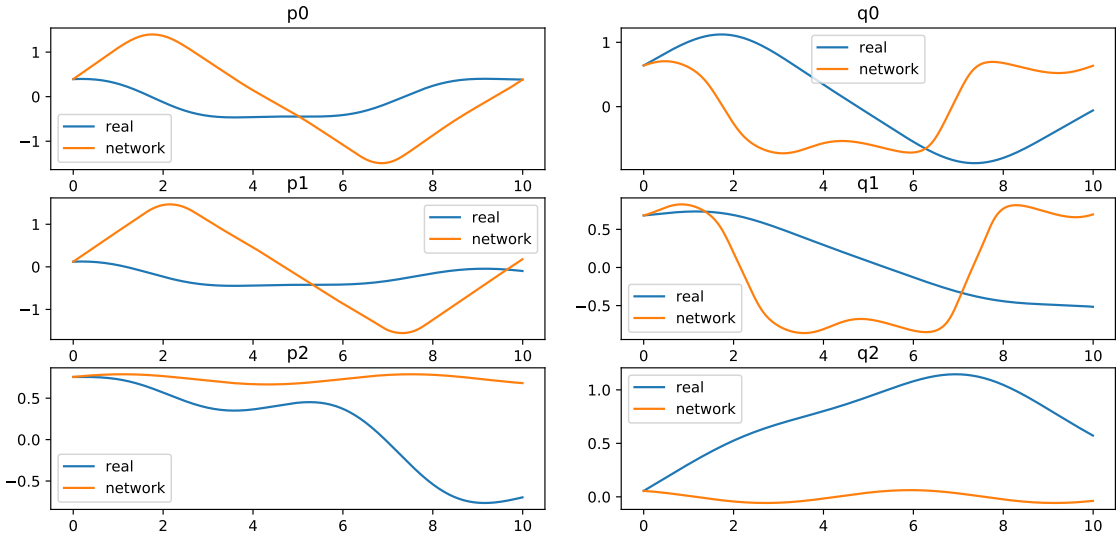


Figure 9: *Gradient of test function* $1 - \cos(y)$. $K = 5$, $h = 0.1$.



(a) *Integrated* p *compared to real* p.

(b) *Integrated* q *compared to real* q.

Figure 10: *Results of attempt to integrate* p *and* q.