Project 2

Thomas Torkildsen

October 2020

1 Introduction

In this project Hamiltonian systems shall be considered and a model shall be trained to find a Hamiltonian based on only the momentum and the position. The data consists of different values of the Hamiltonian at points in the solution space. The Hamiltonian will be estimated by a neural network. The neural network will be of ResNet architecture.

In this project the Hamiltonian framework is a energy function and the parameters are generalised as positions and momenta, $y = (q, p)$. The Hamiltonian (H) is a separable function, $H(p, q) = T(p) + V(q)$. $T(p)$ can be considered the kinetic and $V(q)$ the potential energy of the system.

The goal in this project is to train two neural network to approximate $T(p)$ and $V(q)$. After they are approximated we shall calculate the neural networks gradient and check if the properties $\dot{p} = -\frac{\partial H}{\partial q}$ and $\dot{q} = \frac{\partial H}{\partial p}$ hold. This will be done using symplectic Euler and Størmer-Verlet.

2 The neural network

In the jupyter file, section 1, the implementation of the training algorithm alongside explaination of parameters can be seen. With the training function for the neural network implemented it is to decide which parameters and integrated functions to use. We want to find the case that gives the highest precision as well as the lowest training time for the neural network.

All plots in the file has also been plotted in the Jupyter file.

2.1 Testing the model

When testing the model for $\sigma_1(x) = \max\{0, x\}$, $\eta_1(x) = x$ and $\sigma_2(x) = \tanh(x)$, $\eta_2(x) = \frac{1}{2}(1 + \tanh\frac{x}{2})$ the results vary a lot. From figure 1 it is clear that $J(\theta)$ diverges after a few iterations for some of the multidimensional training functions. Even tho it seems as if $J(\theta)$ moves quicker towards zero for the other cases, σ_1 and η_1 are not suited in the neural network. Therefore σ_2 and η_2 will be used further from this point on.

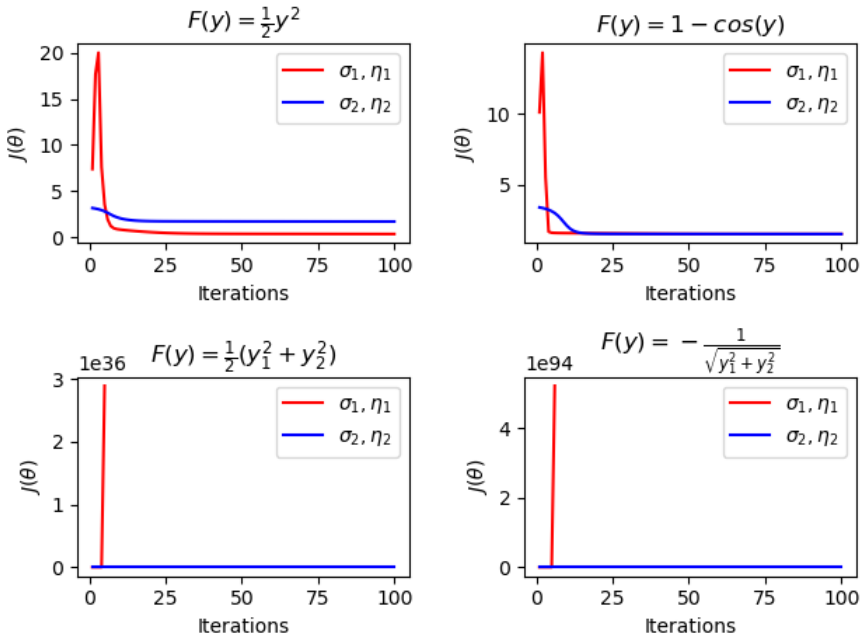

Figure 1: convergence plots with the different test functions for $\sigma_1(x) = \max\{0, x\}$, $\eta_1(x) = x$, $\sigma_2(x) = \tanh(x)$ and $\eta_2(x) = \frac{1}{2}(1 + \tanh\frac{x}{2})$ ($K = 5, I = 106, \tau = 0.03, h = 0.2$)

2.2 Optimizing variable choices

When optimizing the variable choices one variable were looked at while the other remained constant. Still it is important to make sure that there always are less or equal unknown parameters than equations i.e. $K(d^2 + d) + d + 1$ has to be smaller then or equal to I. If this is not the case the result will not be reliable. h represents the weight of each layer in the neural network. Since we always want the possibility to at least take one full step throughout the layers and $\sigma_2, \eta_2 \in [0, 1]$, we need $k \cdot h \geq 1$.

It is reasonable to say that more input data (I) will result in more iterations until convergence. $j(\theta)$ will move slower towards zero for a large number of input values (I) since $\Upsilon - C$ has I elements and with more elements each element has to be smaller in order for the norm to be the same as a case with fewer elements.

When looking at figure (2) it is clear that the test function is approximated worse for a high number of layers (K) for almost every test function. The cost is significantly higher for a high amount of layers as well. With more layers it is also reasonable to expect more noise when converging. A slight change a high number of parameters will naturally make a bigger change in the result than a slight change in a small number of parameters.

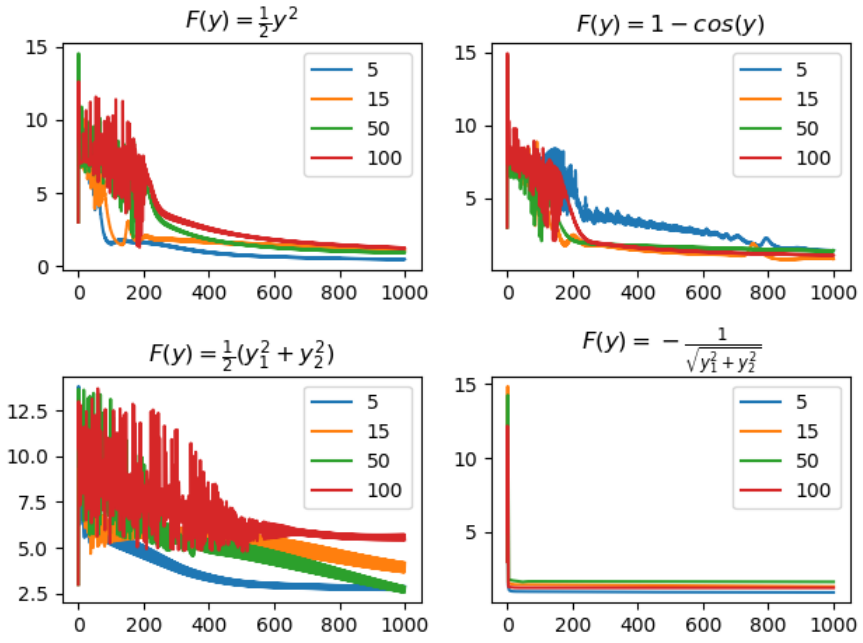

Figure 2: convergence plots of the test functions with different amount of layers. Costs where: $K = 5 \rightarrow 18$ seconds, $K = 15 \rightarrow 62$ seconds, $K = 50 \rightarrow 203$ seconds and $K = 100 \rightarrow 383$ seconds ($I = 2010, \tau = 0.01, h = \frac{1}{K}$)

When I is constant and $d \geq d_0$ is changed like in figure (3). With a high d we get more parameters θ so that more parameters are changed each iteration and fluctuation occurs. With more parameters it should also be easier to approximate Hamiltonian, but the cost increases quite a bit as well. Therefor the ideal choice of d seems to be $d \approx 2 \cdot d_0$

The optimal choice of τ is not easy to decide. From figure 4 it seems like the training function behaves similarly for all choices between 0.01 and 0.09, but with more oscillation for higher choices of τ. In the next section we will see that choosing a optimal τ is not important.

2.3 The Adam method vs gradient descent

Looking at section 3 in the jupiter file it is clear that Adams descent takes a lot fewer steps to train network then standard gradient descent does. When running the training algorithm with Adams descent and standard gradient descent for 10 000 iterations, 10 layers, $h = 0.1$ and $\tau = 0.01$ the results where:

	time	J(θ)
Adam:	29.0 sec.	0.055
std. grad.:	22.4	0.23

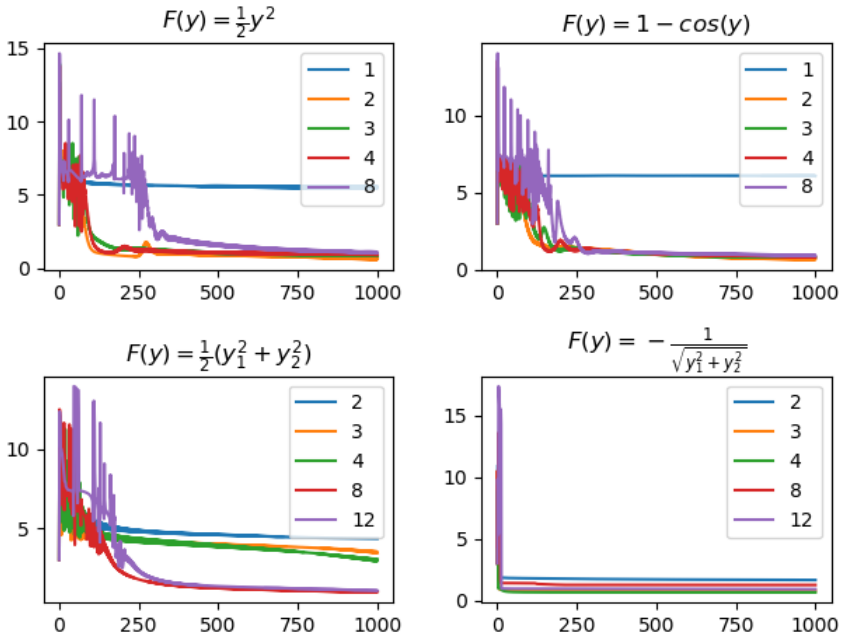

Figure 3: convergence plots of the test functions for different dimensions in the neural network. costs: $d = 1v2 \rightarrow 16$ sec, $d = 2v3 \rightarrow 25$ sec, $d = 3v4 \rightarrow 34$ sec, $d = 4v8 \rightarrow 37$ sec, $d = 8v12 \rightarrow 40$ sec ($K = 10, I = 1575, \tau = 0.01, h = 0.1$)

A 29% increase in cost reduced the error ($J(\theta)$) by 76%. The plot in jupyter make it clear that Adams give a better approximation to the exact result for a given number of iterations. Therefore, from this point on Adams descent will always be used in the training algorithm.

2.4 Training with unknown Hamiltonian function

When training with an unknown Hamiltonian we used a lot more data to train the neural network. After testing the training algorithm a few times it was clear that the using all the parameters at made the training algorithm to costly. Another problem where that the training algorithm seemed to get trapped in local minimums, which resulted in a constant $J(\theta) = c \gg 0$. So a new training function where implemented. Instead of running the program for all input parameters at the same time it is now run with a small, random section of the parameters. For each iteration a new, random section is picked until all parameters are used. Now every time the $J(\theta)$ is near a local minimum a new sett of parameters are used and helps escape. The drawback of using this method is that the convergence criteria is no longer depending on the entire batch of training data, but on the small portion of it. This might cause convergence

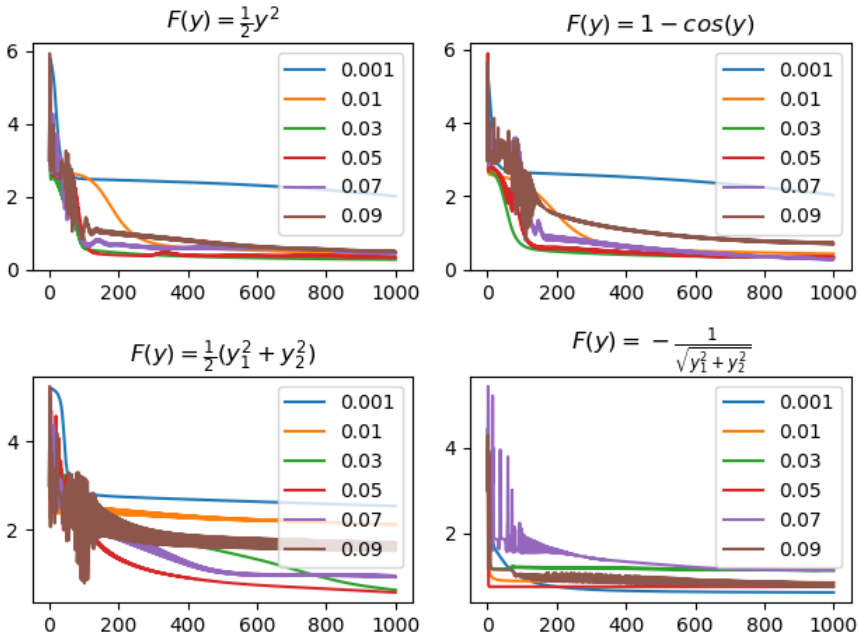

Figure 4: convergence plots of the test functions for different τ-values. ($K = 10, I = 300, h = 0.1$)

even tho only a little part of the training data corresponds well to the neural network.

Due to a high number of parameters we now look at $\frac{J(\theta)}{\text{num. elements}}$ as the error. In section 4.8 in jupyter a approximation of the Hamiltonian with a tolerance of 10^{-5} can be seen. It is quite clear that the neural network give a good approximation of the Hamiltonian, but however it does not come anywhere close to being constant. Therefore a tolerance of $0.5 \cdot 10^{-5}$ is used further.

2.5 evaluation

The neural network are running well once it has been trained, but is very costly to train. The script i wrote would have been a lot faster if θ had been written as several Numpy arrays with a preallocated space in the memory. This was not realised until the very end, so it was no time to implement that in the script. Although the cost would have been lower, it does not affect the result for a given number of iterations.

The Hamiltonian calculated from our neural network was not constant like the Hamiltonian from the test data. The plot of the Hamiltonian in section 4.7 in jupyter shows that H is varying a lot, this effect might not be as big as it looks and hopefully we still can get the properties $\dot{p} = -\frac{\partial H}{\partial q}$ and $\dot{q} = \frac{\partial H}{\partial p}$ still

hold.

3 Derive formulas for computation of $\Delta_y F(y)$

$\tilde{F}(y)$ is our trained function and can be written on the form

$$\tilde{F}(y) = \tilde{T}(p) + \tilde{V}(q) = G_p \circ \Phi_{K-1,p} \circ \Phi_{K-2,p} \circ ... \circ \Phi_{0,p}(p) + G_q \circ \Phi_{K-1,q} \circ \Phi_{K-2,q} \circ ... \circ \Phi_{0,q}(q)$$

Where

$$G_{y_i}(y_i) = \eta(\omega_{y_i}^T y_i + \mu_{y_i}),\ \Phi_{k,y_i}(y_i) = y_i + h\sigma(W_{k,y_i} y_i + b_{y_i}) \\ ,\ 0 \leq k \leq K-1,\ y_i \in y = \{p, q\} \tag{1}$$

$\omega_{y_i} \in \mathbb{R}^d$, $\mu_{y_i} \in \mathbb{R}$, $W_{k,y_i} \in \mathbb{R}^{d \times d}$, $b_{y_i} \in \mathbb{R}^d$ and $h \in \mathbb{R}$ are parameters calculated in the training function. We define the functions

$$\Psi_{0,y_i} = \Phi_{0,y_i},\ \Psi_{k,y_i} = \Phi_{k,y_i} \circ \Psi_{k-1,y_i} = \Phi_k \circ ... \circ \Phi_0,\ k = 1, ..., K-1,\ y_i \in \{p, q\}$$

Where $Z_{y_i}^{(k)} = \Psi_{k-1,y_i}(y_i),\ k = 1, ..., K$ are known from the training function. We can now write $\tilde{F}(y) = G_p \circ Z_p^K + G_q \circ Z_q^K$. The gradient of $\tilde{F}(y)$ is then

$$\Delta_y \tilde{F}(y) = \{\Delta_p \tilde{F}(y), \Delta_q \tilde{F}(y)\} = \{\Delta_p \tilde{T}(p), \Delta_q \tilde{V}(q)\}$$

Since $\tilde{T}(p)$ and $\tilde{V}(q)$ both behave similarly it is sufficient to look at $\tilde{T}(p)$:

$$\Delta \tilde{T}(p) = (D\Psi_{K-1}(y))^T \Delta G(Z^{(K)})$$

Since $Z^{(K)} = \Psi_{K-1}(y) = \Phi_{K-1}(Z^{(K-1)}) = \Phi_{K-1} \circ \Psi_{K-2}(y)$ we get

$$D\Psi_{K-1}(y) = D\Phi_{K-1}(Z^{(K-1)}) \cdot D\Psi_{K-2}(y)$$

$$\Leftrightarrow (D\Psi_{K-1}(y))^T = (D\Psi_{K-2}(y))^T \cdot (D\Phi_{K-1}(Z^{(K-1)}))^T$$

This can be repeated until we get

$$(D\Psi_{K-1}(y))^T = (D\Phi_0(Z^{(0)}))^T \cdot ... \cdot (D\Phi_{K-1}(Z^{(K-1)}))^T$$

Now we rewrite $\Delta \tilde{T}(p)$

$$\Delta \tilde{T}(p) = (D\Phi_0(Z^{(0)}))^T \cdot ... \cdot (D\Phi_{K-1}(Z^{(K-1)}))^T \Delta G(Z^{(K)})$$

$$= (D\Phi_0(Z^{(0)}))^T \cdot ... \cdot (D\Phi_k(Z^{(k)}))^T P_{k+1}$$

$$P_k = (D\Phi_{k+1}(Z^{(k)}))^T \cdot ... \cdot (D\Phi_{K-1}(Z^{(K-1)}))^T \Delta G(Z^{(K)}),\ k = 0, ..., K-1$$

Note that

$$P_{k-1} = (D\Phi_{k-1}(Z^{(k-1)}))^T P_k \tag{2}$$

P_k is the same parameter as in the training function, except for $P_K = \Delta G(Z^{(K)})$. Now we compute $\Delta G(Z^{(K)})$ and $(D\Phi_{k-1})^T$ using (1). Using components we get

$$G(p) = \eta(\sum_{i=1}^{d} \omega_i p_i + \mu) \;\Rightarrow\; \frac{\partial G}{\partial p_j}(p) = \eta'(\sum_{i=1}^{d} \omega_i p_i + \mu)\omega_j$$

This gives us the gradient

$$\Delta G(p) = \eta'(\omega^T p + \mu)\omega \tag{3}$$

To simplify for an arbitrary k we write $\Phi_k(p) = \Phi(p) = p + h\sigma(Wp + b)$ and $P_{k+1} = P$. By using components we get

$$[\Phi(p)]_i = p_i + h\sigma(\sum_{j=1}^{d} W_{ij}p_j + b_i) \;\Rightarrow\; \frac{\partial \Phi_i}{\partial p_r}(p) = \delta_{ir} + h\sigma'(\sum_{j=1}^{d} W_{ij}p_j + b_i)W_{ir}$$

Note that from (2) that we all ways just need to compute for the vector P, the expression for $(D\Phi(p))^T A$'s r'th component is

$$\sum_{i=1}^{d} [D\Phi(p)]_{ir} P_i = P_r + h\sum_{i=1}^{d} \sigma'(\sum_{j=1}^{d} W_{ij}p_j + b_i)W_{ir}P_i$$

This can be rewritten as

$$D\Phi(p)^T P = P + W^T(h\sigma'(Wp + b) \odot P)$$

From (2) we get

$$P_{k-1} = D\Phi_{k-1}(p)^T P_k = P_k + W_{k-1}^T(h\sigma'(W_{k-1}p + b_{k-1}) \odot P_k)$$

It is clear that $P_0 = \Delta\tilde{T}(p)$ and so the gradient of the trained function is:

$$\Delta_y \tilde{F}(y) = \{\Delta_p \tilde{T}(p), \Delta_q \tilde{V}(q)\} = \{P_{0,p}, P_{0,q}\}$$

4 Computing the gradient

Computation of the gradient of functon example $H(q,p) = \frac{1}{2}p^T p - \frac{1}{\sqrt{q_1^2+q_2^2}}$ in Juyter is very similar to the gradient from the trained function $\tilde{H}(p,q)$. We found that scaling the gradient with a factor $\alpha = \frac{b-a}{b_y - a_y}$ gave the best results. a, b are the maximum and minimum values from the output training data and b_y, a_y from the input training data. This scaling makes snese, since we now are looking at derivatives.

5 Symplectic Euler and the Størmer-Verlet method for the Hamiltonian function

Here is when $\tilde{F}$ do not look to good anymore. When using the either symplectic Euler or Størmer-Verlet on the test function $H(q,p) = \frac{1}{2}p^T p - \frac{1}{\sqrt{q_1^2+q_2^2}}$ we get a totally different behavior then when it is used on the approximated Hamiltonian. This might occur for several reasons. One reason is that the scaling we have used is not sufficient for all q and p values. Even though this will make could be a reason why p and q values are different for the calculations it does not explain the inconsistent cycle. We can also see that we are inside the scope where the neural network was trained, so that cant be a reason either.

From the mumentums and positions gotten from Symplectic Euler and Størmer-Verlet we do get a constant Hamiltonian, which means that even thought p and q does not move in a perfect cycle the Hamiltonian is preserved.

In sectian 5.7 in jupyter it can be seen that p and q acts similarly for our network and the data, but they are not equivalent, But when we look at the Hamiltonian for these data points it preserved, with an error of $+ - 0.5 \cdot 10^{-5}$.

6 Conclusion

Our neural network gives a good approximation of a separable Hamiltonian and Network for the unknown Hamiltonian approximately preserves the Hamiltonian.

References

- Project 2 supplement from the course page.
- "Project2.ipynb" - Jupyter document.