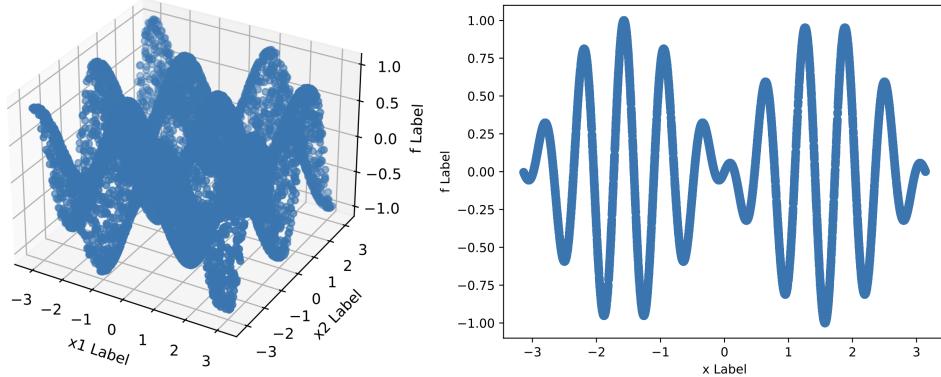


Home assignment

Description:

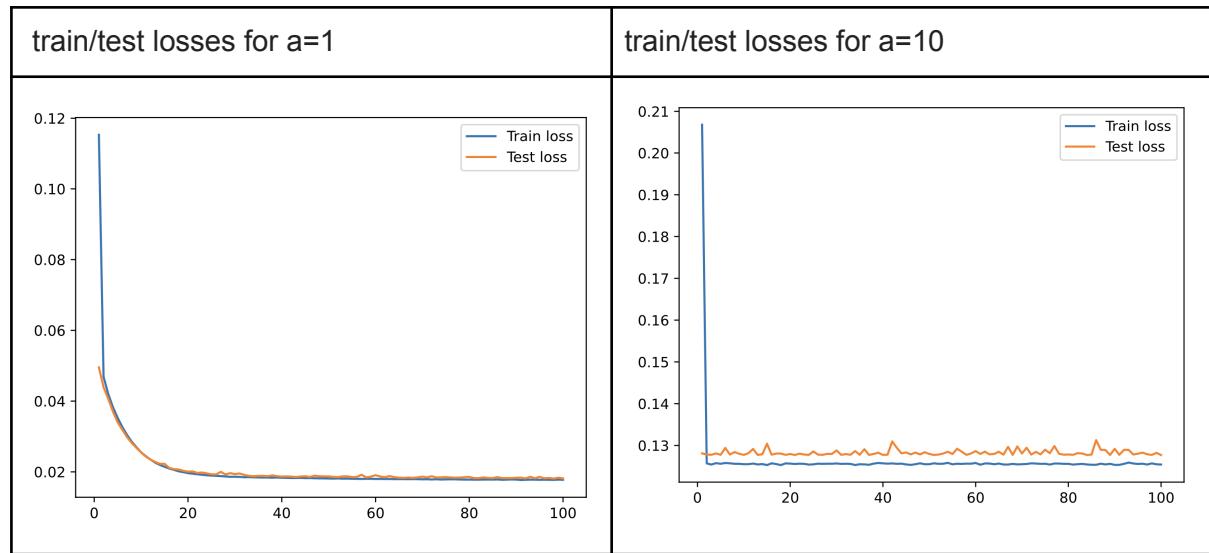
1. The desired functions for approximations are standing waves, either in standard 2d waves with frequencies parameterized by a_1, a_2 , or 1d with an inner and outer envelope with a single frequency parameter a .



2. We used a minimal learning scheme to approximate these functions. The network is composed of 3 layers and one activation function (ReLU in our case), and the gradient descent method has a single unchanged learning rate over the L2 loss.

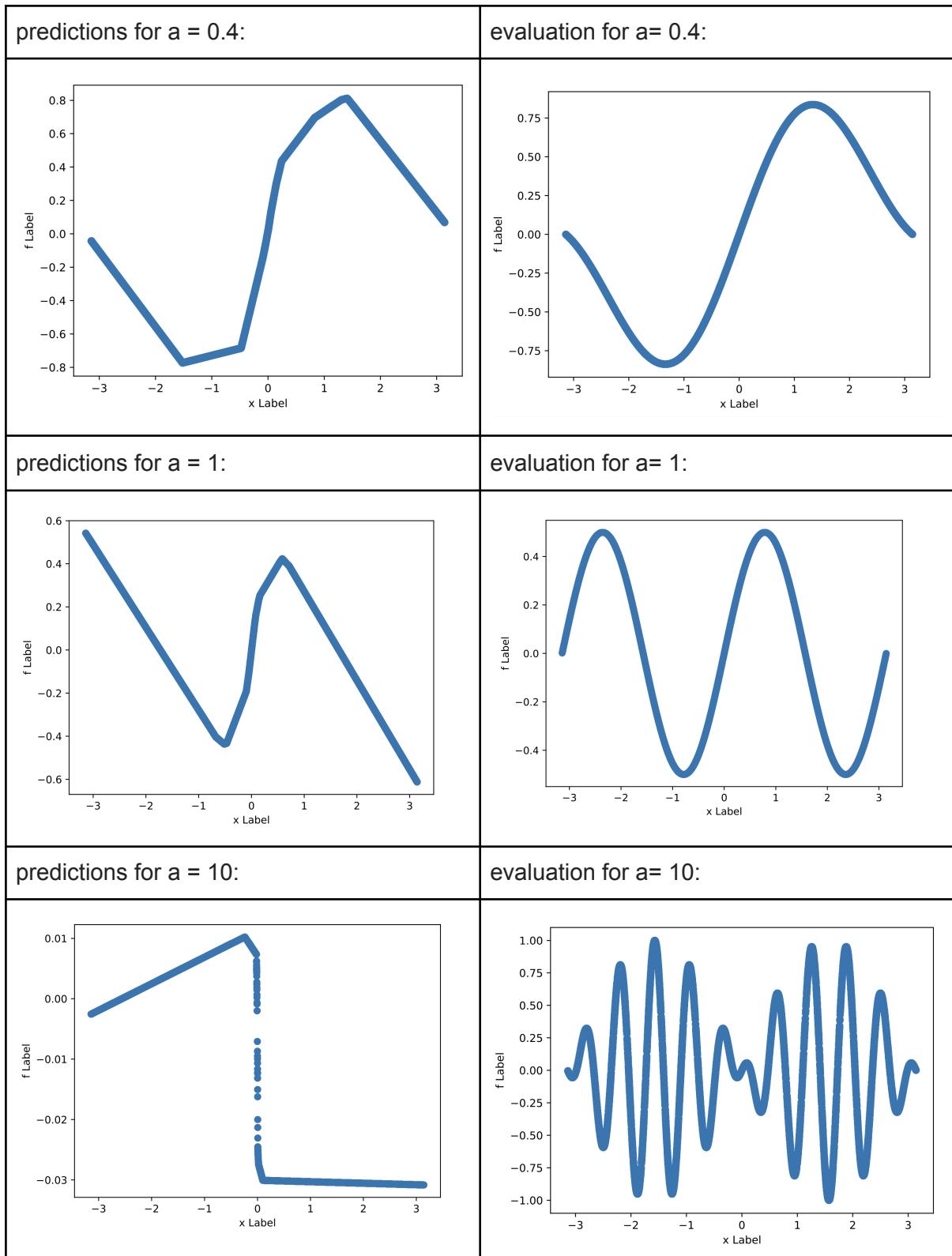
Experiments:

1. $f(x) = \sin(x)\cos(ax)$. Here are train/test losses,

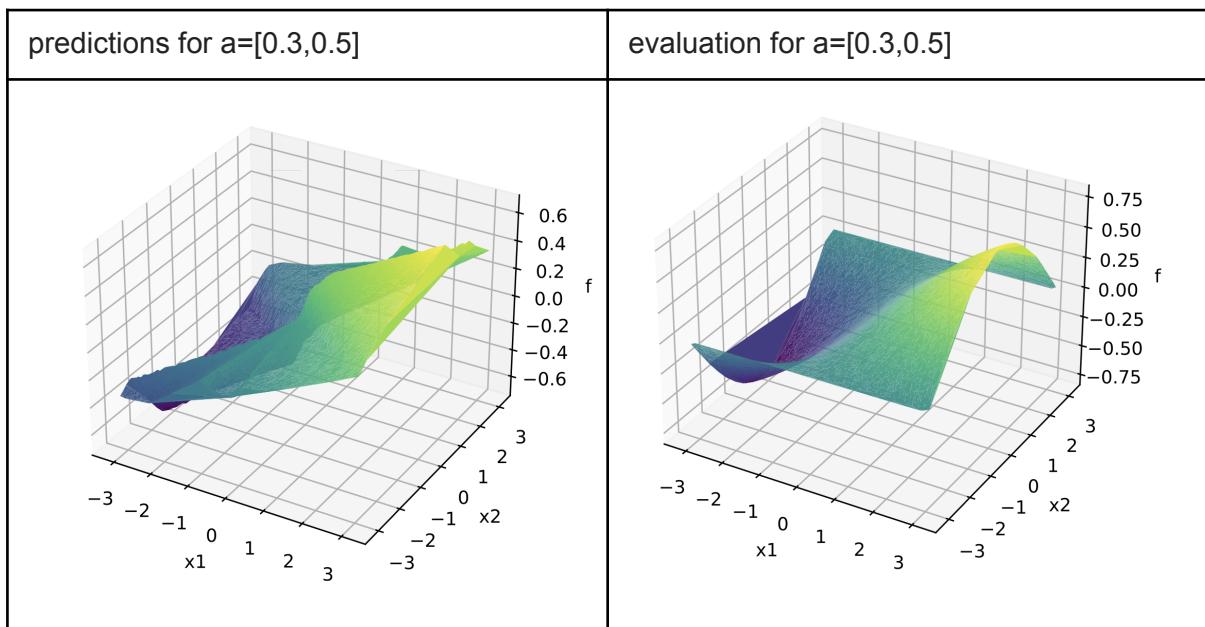
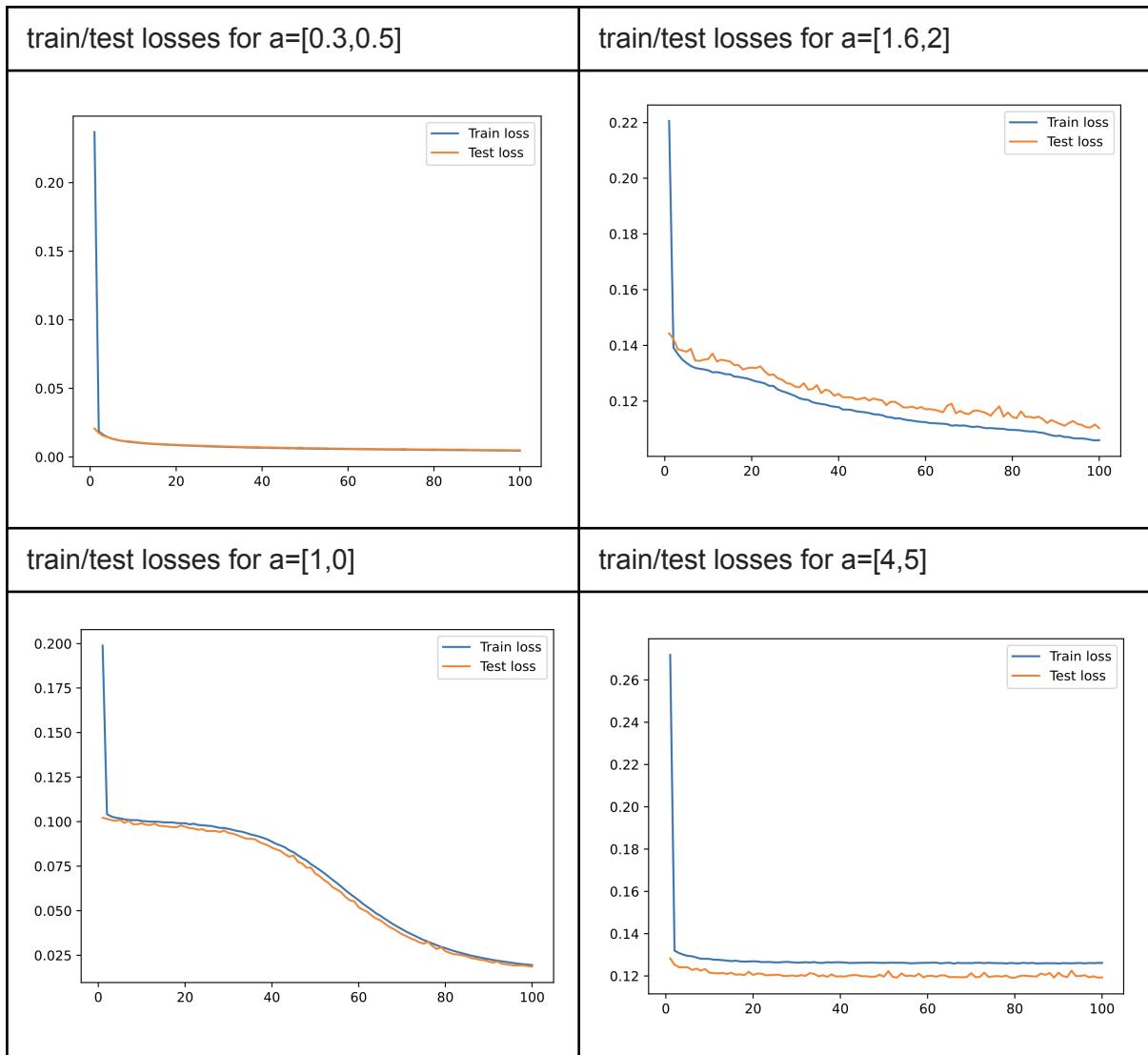


as we use a single activation function with hidden layer of size n the output of the network can't be too expressive so the network can't capture the rapid changes in f , if a is too large compared to n .

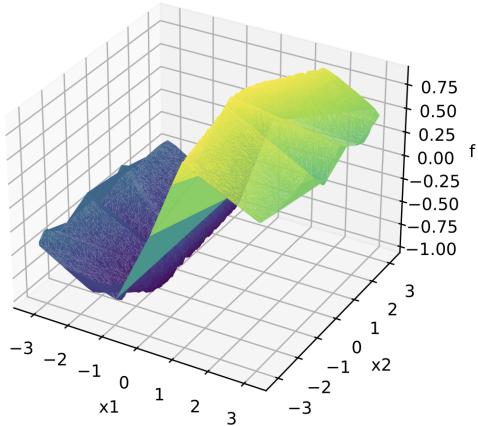
Here we show the actual evaluation of f and the network's prediction for $n=15$,



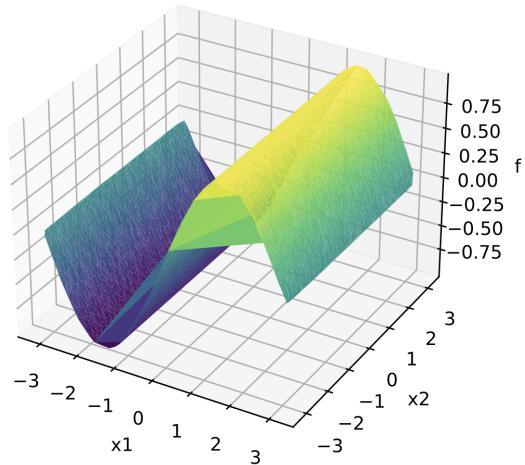
2. $f(x_1, x_2) = \sin(a_1 x_1) \cos(a_2 x_2)$ we can expect a similar trend as the function is separable. We have used a single hidden layer with 20 neurons.



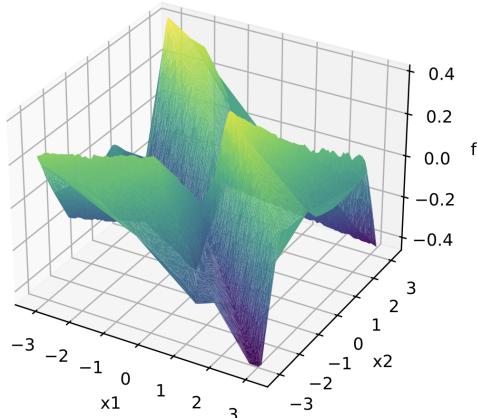
predictions for $a=[1,0]$



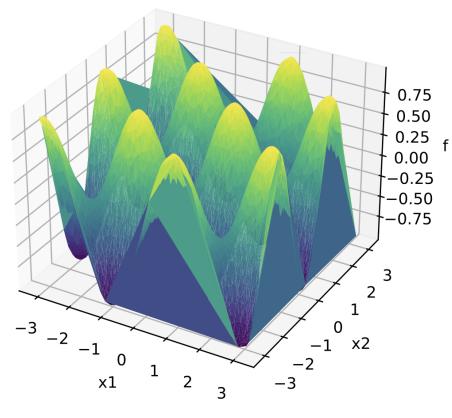
evaluation for $a=[1,0]$



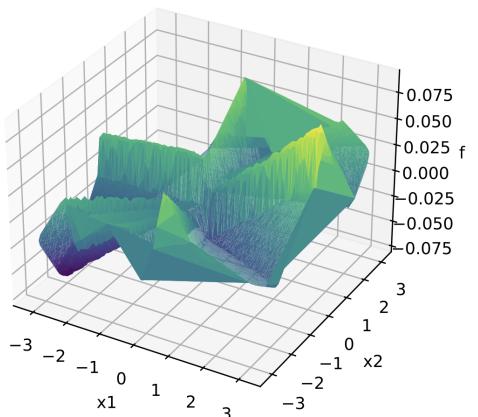
predictions for $a=[1.6,2]$



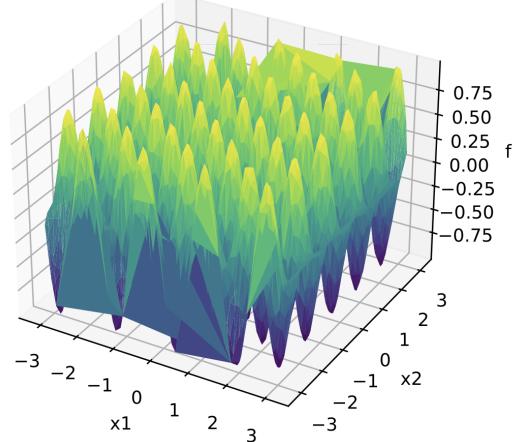
evaluation for $a=[1.6,2]$



predictions for $a=[4,5]$

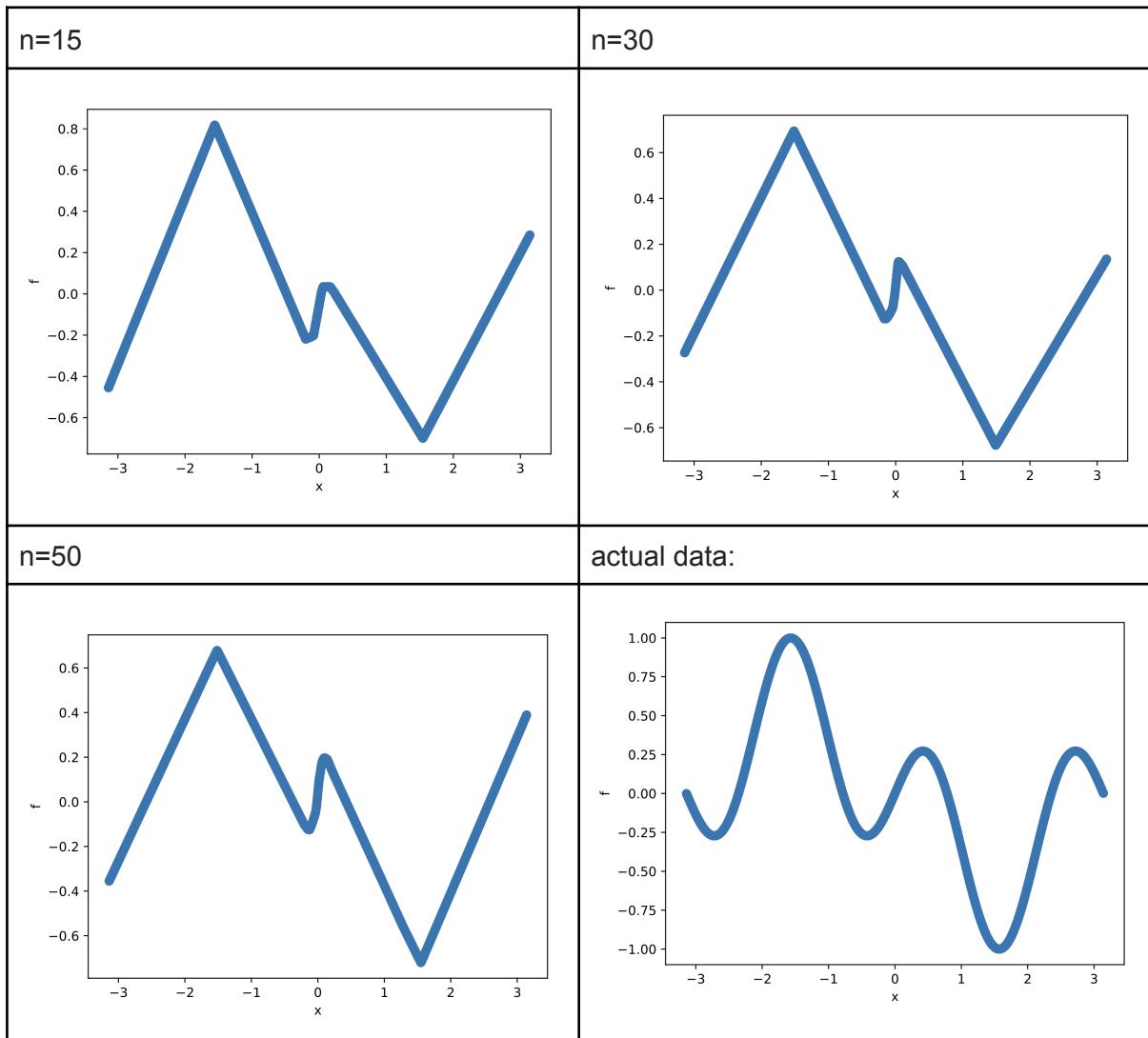


evaluation for $a=[4,5]$



3. We now investigate the relation between the hidden layer dimension and the parameter a . As we've discussed, if a is large (meaning that frequency of f is large) then we cannot expect to mimic f using a small finite sum of relu functions as each component makes a single sharp change of direction over the reals.

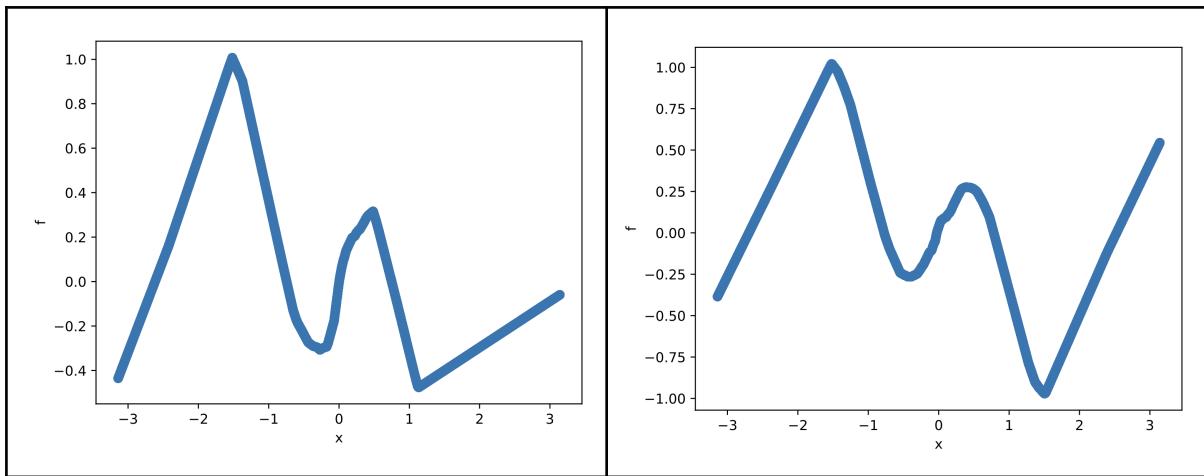
Therefore we fix a and increase n and plot the results,



we can see that around $x=0$ the prediction gets smoother as n increases but besides that the prediction is not quite accurate. Perhaps there is some bias towards small x values or just the fact that we can approximate sine and cosine as polynomials near zero.

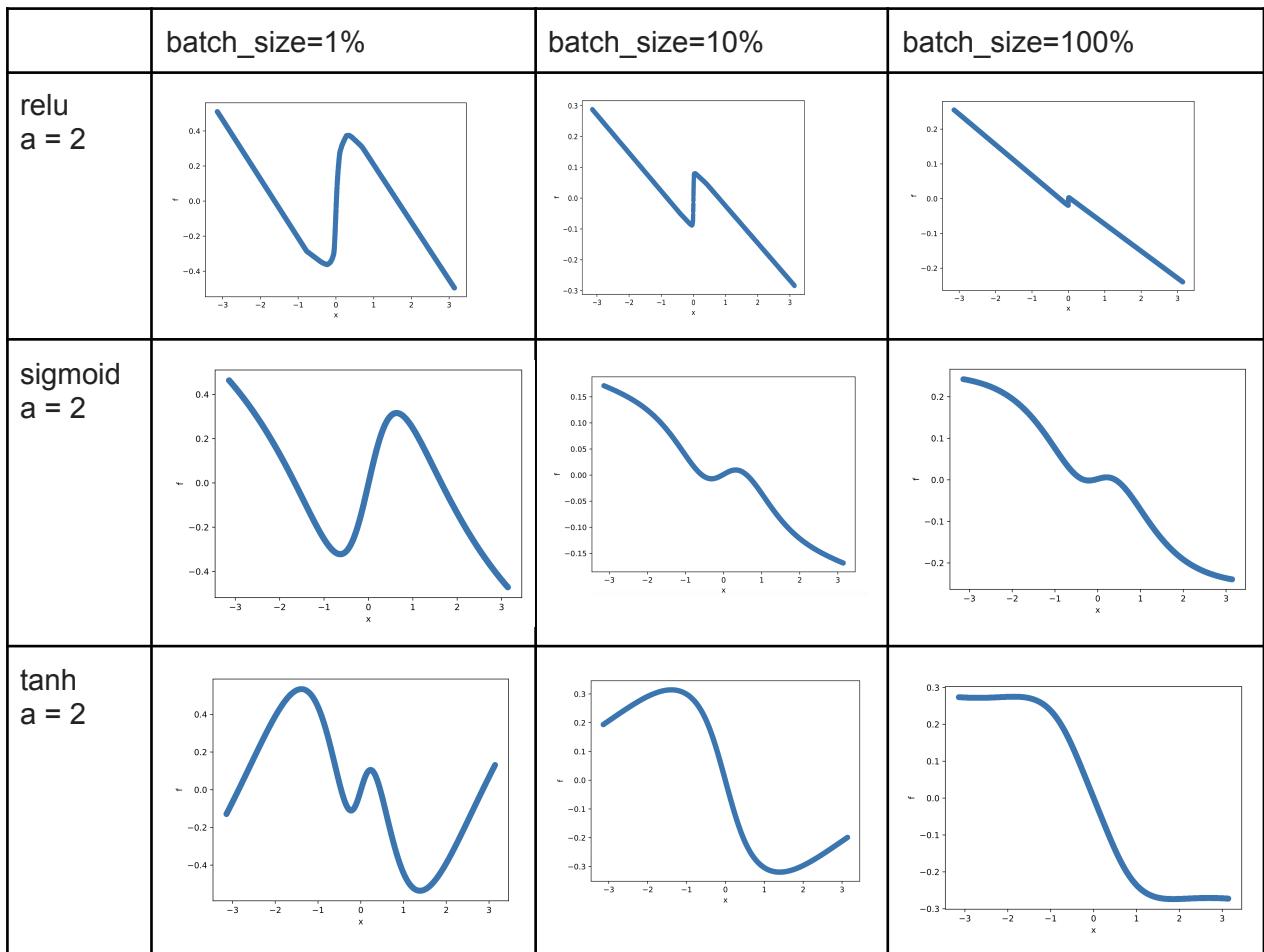
4. Lastly, we can also look at larger networks with more layers, here are two examples for the same function as above, $f(x)=\sin(x)\cos(2x)$

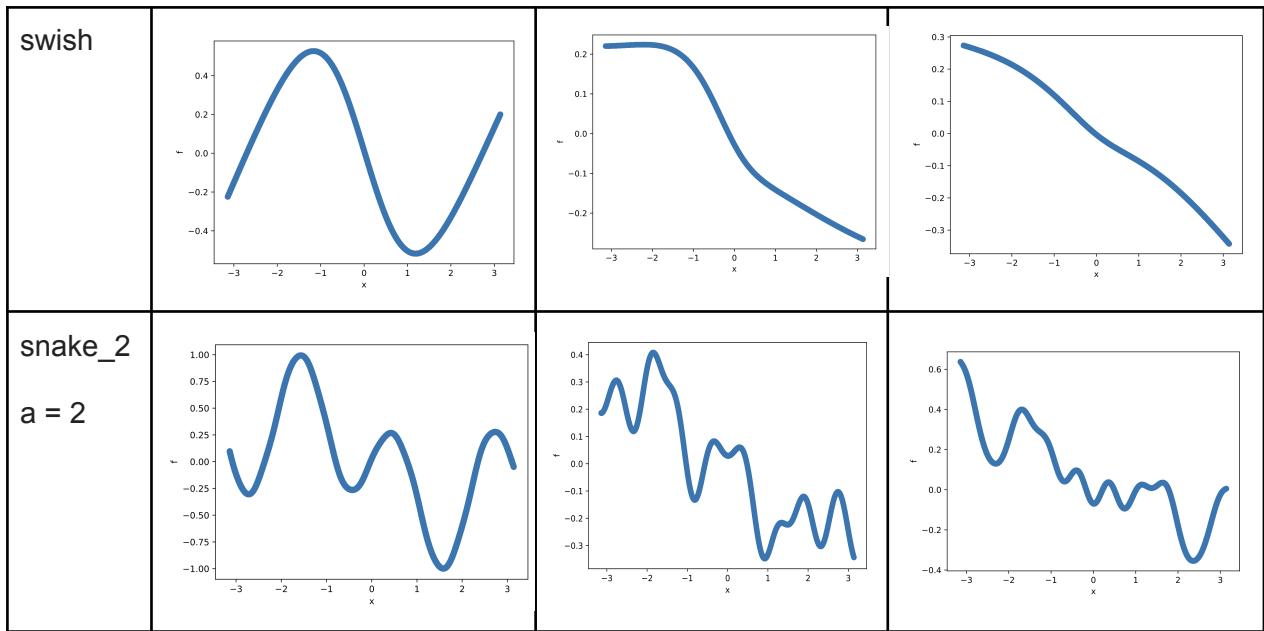
layers=[1,20,40,15,1]	layers=[1,20,50,40,15,1]
-----------------------	--------------------------



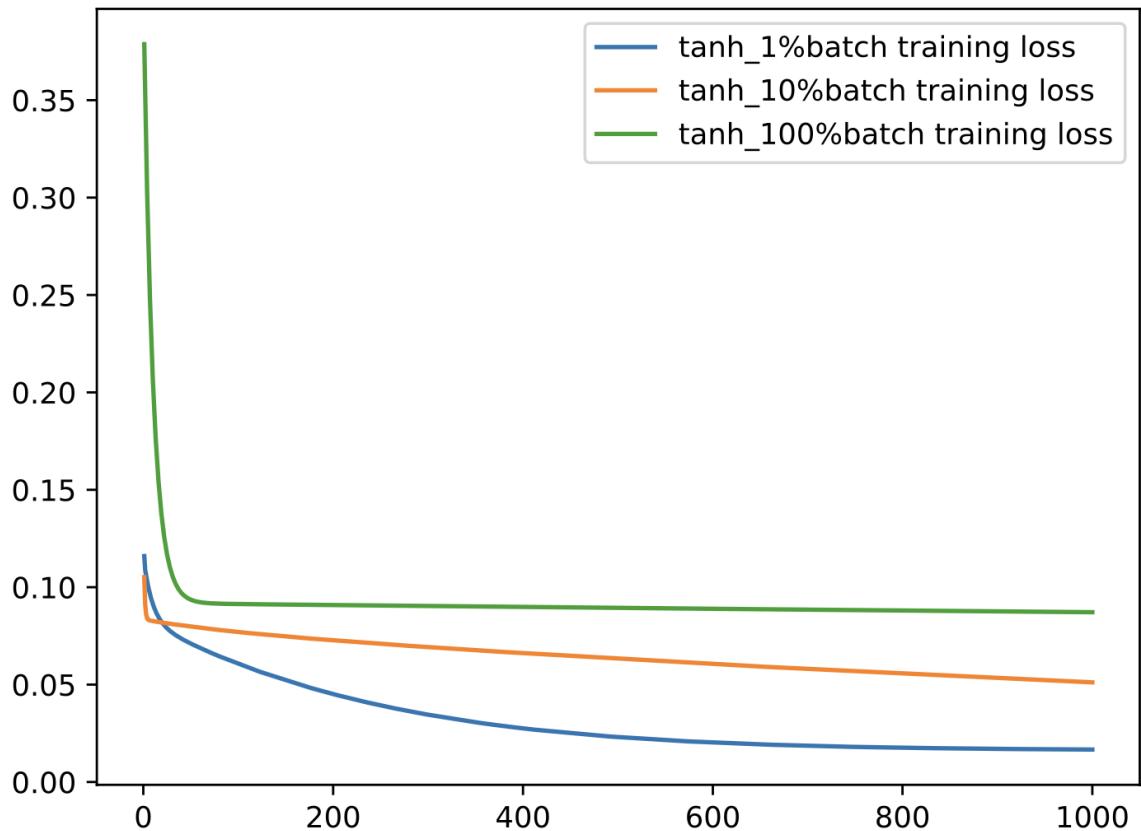
This improves the approximation around zero, but remains inaccurate further away.

4. We now use different activation functions as a parameter for comparison (for $a = 2$),

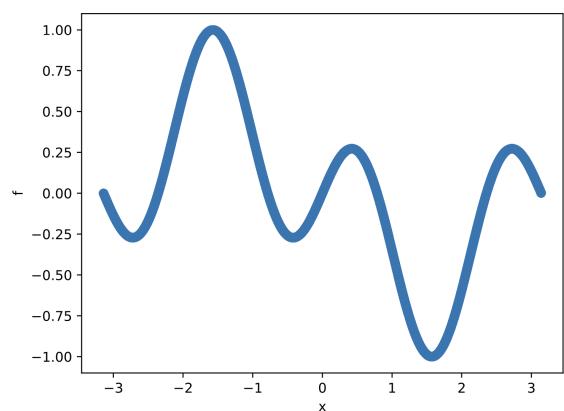
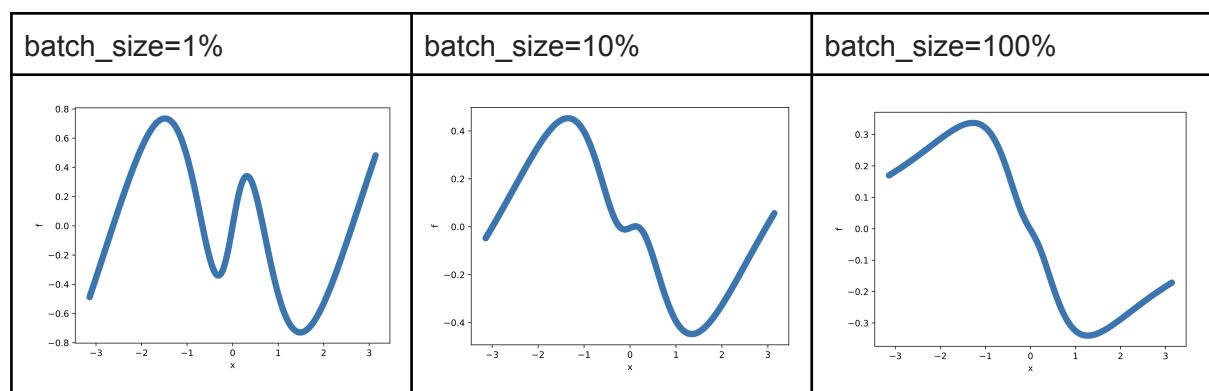
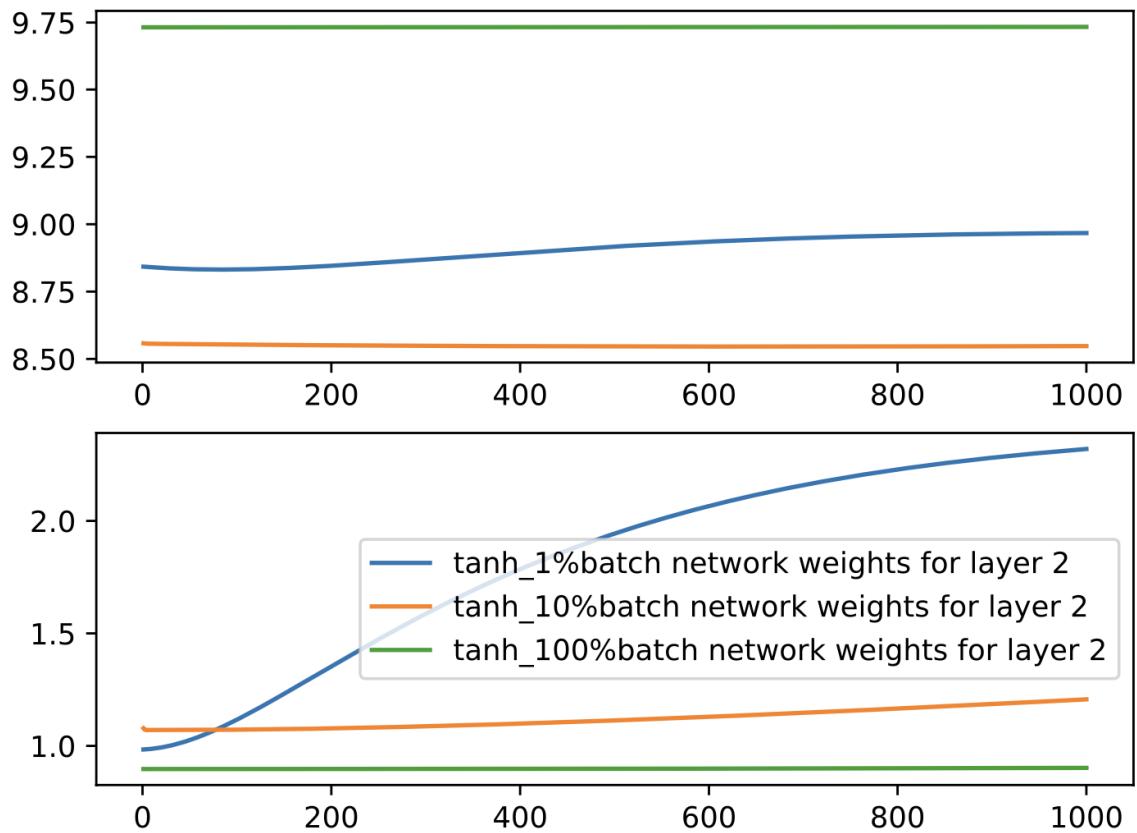




Peculiarly, it seems as if considering more data points for evaluating the empirical risk decreases the ability to learn. We may attribute this phenomena to a non robust GD method, as it seems that the larger the batch, the smaller the gradient step, and our method does not have an adaptive learning rate



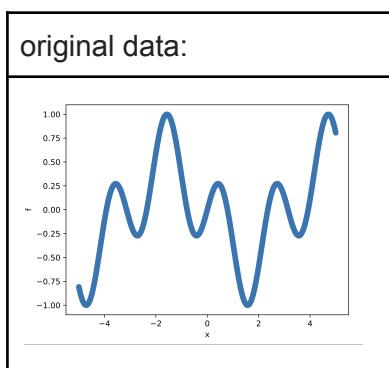
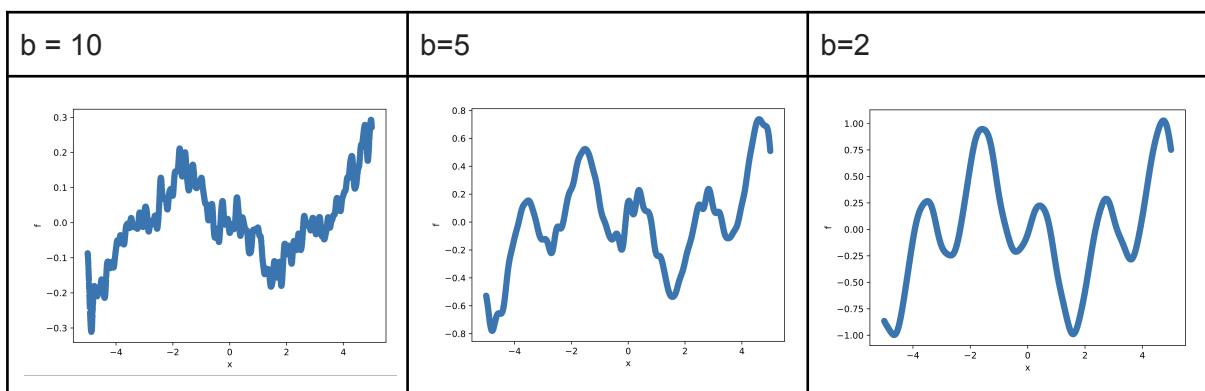
in the above x% batch means that we train the model with batches of size $|\text{train}| * x/100$.



Although not great, it does look better than before.

We included the “snake_b” function in the regular arsenal of activation functions. This function was introduced in the paper “Neural Networks Fail to Learn Periodic Functions and How to Fix It” <https://arxiv.org/pdf/2006.08195.pdf>, where they show the any type of ReLU/tanh network fails to learn a function which has a finite period over the reals, as these networks are asymptotically affine. In contrast, they showed that the function $\text{snake}_b(x) = x + \sin^2(bx)/b$ (usually picking $5 < b < 50$ for periodic tasks) can approximate any periodic function using a single hidden layer (a universal approximation type theorem).

Here is the prediction for $a=2$ using the snake activation function



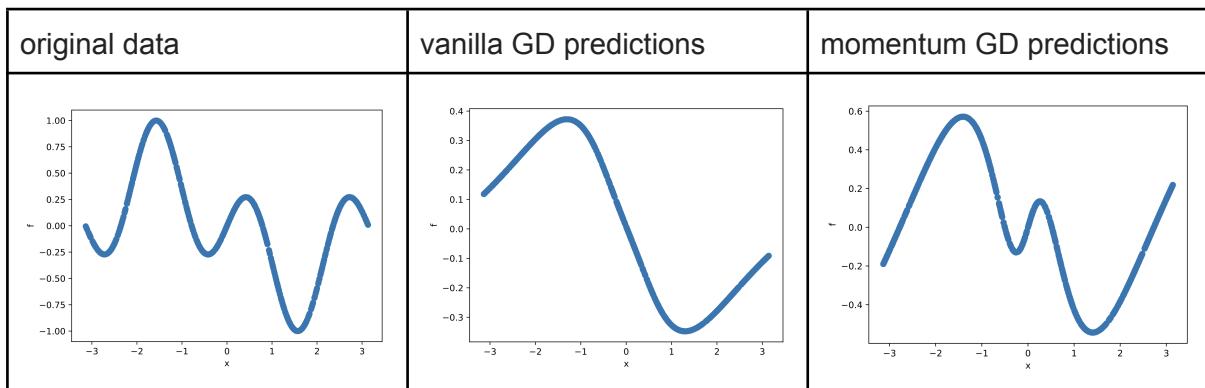
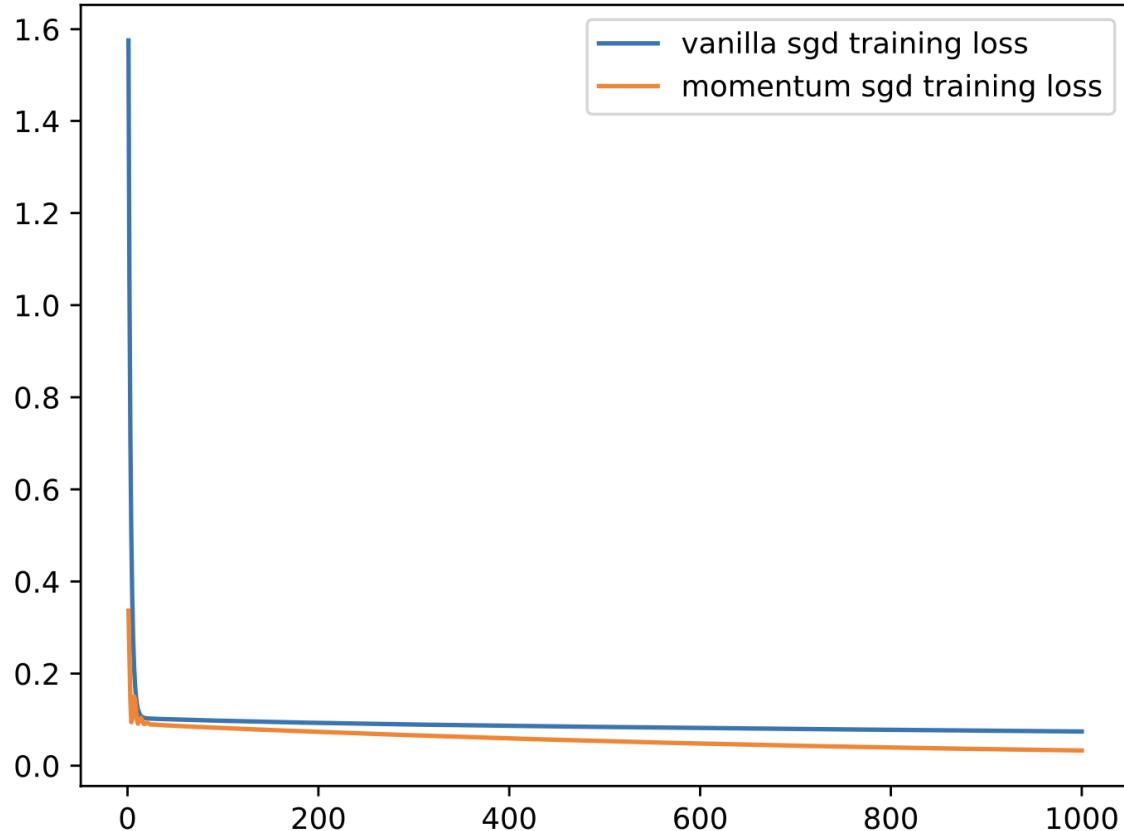
The results are not at all surprising as we pretty much used an *ax machina* solution as the snake function and the original objective are very similar.

However, these plots give a hint as to how the snake function approximates periodic functions, it spans them on the fourier basis.

5. Vanilla GD vs Momentum based GD. The momentum variation is an extension of the plain gradient descent method done by adding extra terms to the update rule that take into account the previous steps. In this way, the algorithm “remembers” the previous decisions and so is influenced by them.

Explicitly, it has the following form, $w_{t+1} = w_t - \Delta w_t$, where $\Delta w_t = \eta \nabla L(w_t) + \alpha \Delta w_{t-1}$.

Expanding the recursive formula gives, $w_{t+1} = w_t - \eta \sum_{k \leq t} \alpha^{t-k} \nabla L(w_k)$, so if $\alpha \in [0, 1]$ then the contribution of past updates decay exponentially.



We can indeed see that the momentum version works much better!

Looking at the changes in the network parameters shows that the basic GD gets stuck at some plateau whereas the momentum method variables keep changing.

