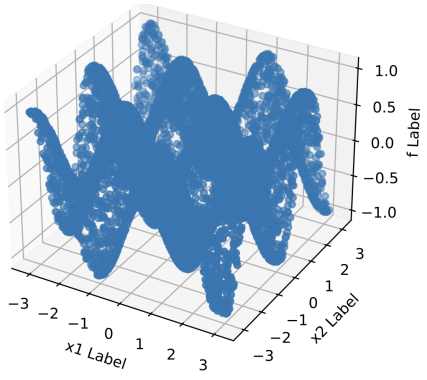
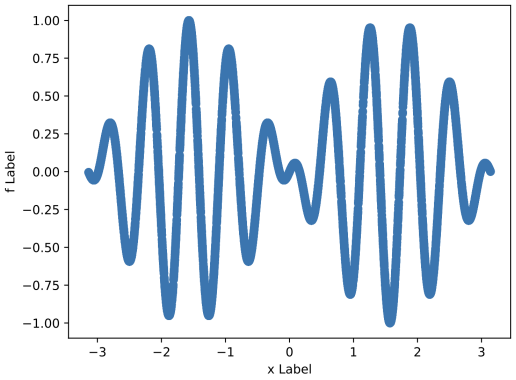
Home assignment

Description:

1. The desired functions for approximations are standing waves, either in standard 2d waves with frequencies parameterized by a1,a2, or 1d with an inner and outer envelope with a single frequency parameter a.

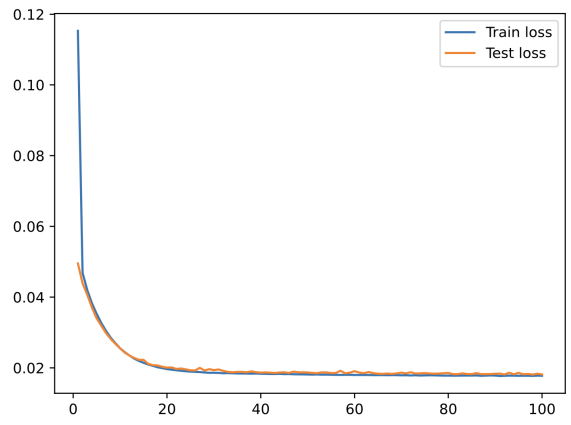
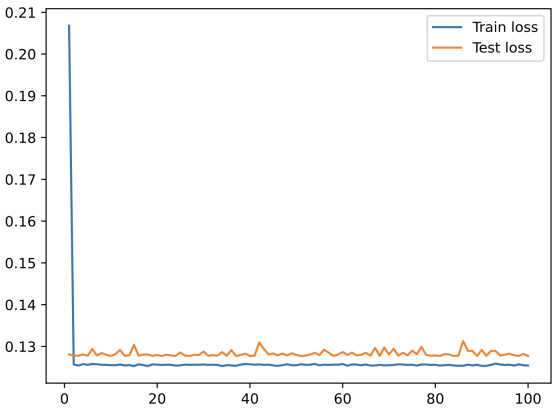




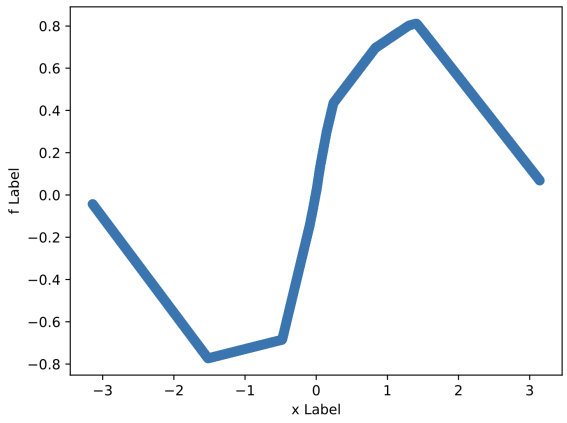
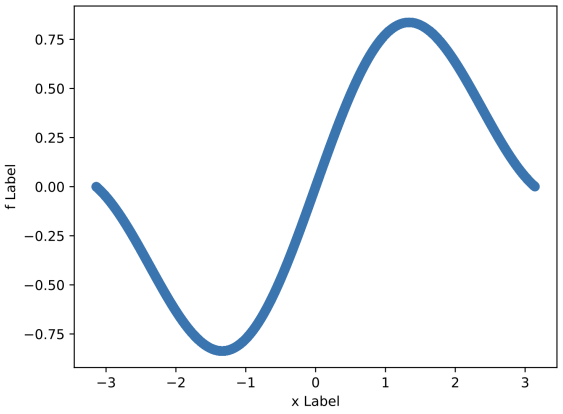
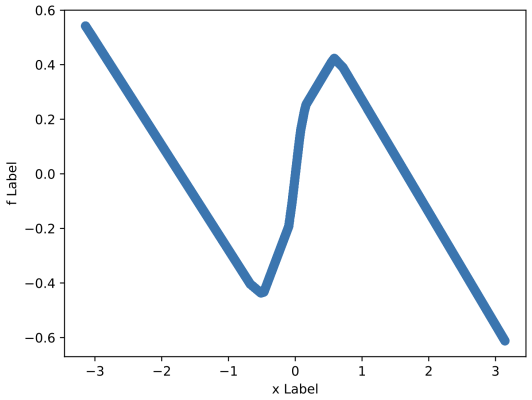
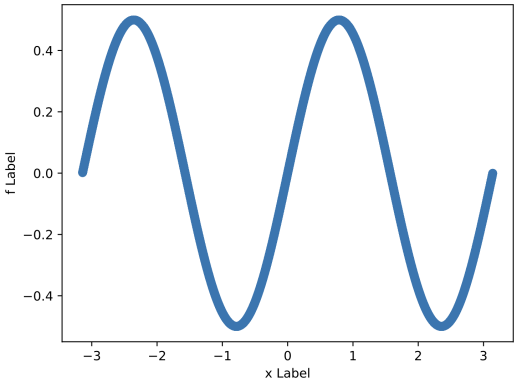
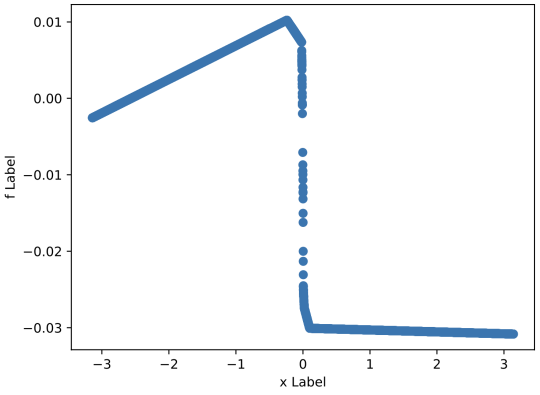
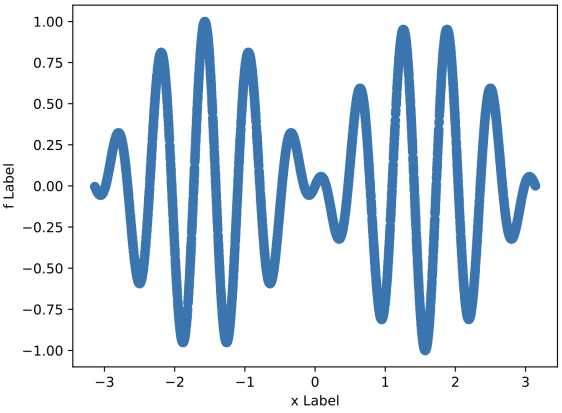
2. We used a minimal learning scheme to approximate these functions. The network is composed of 3 layers and one activation function (ReLU in our case), and the gradient descent method has a single unchanged learning rate over the L2 loss.

Experiments:

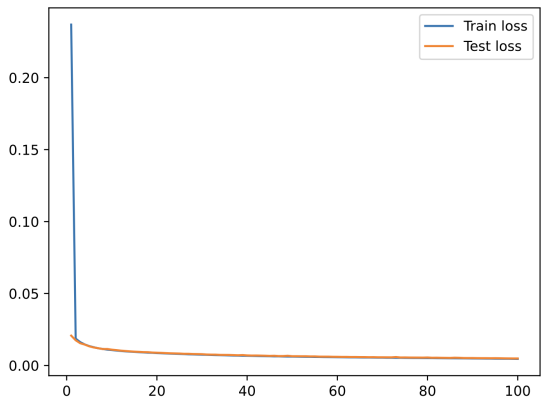
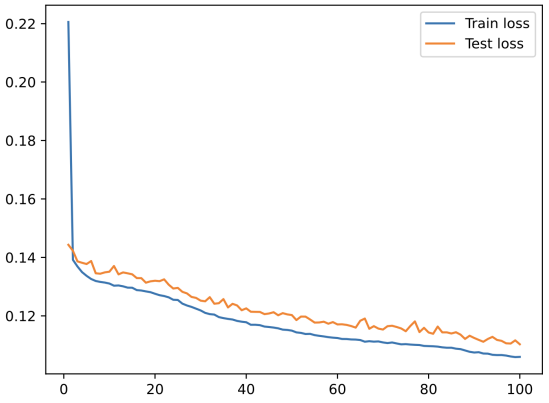
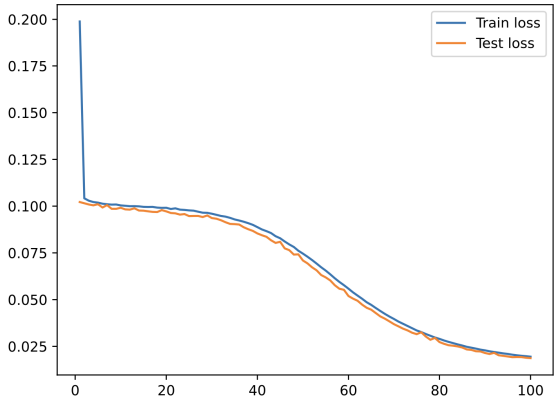
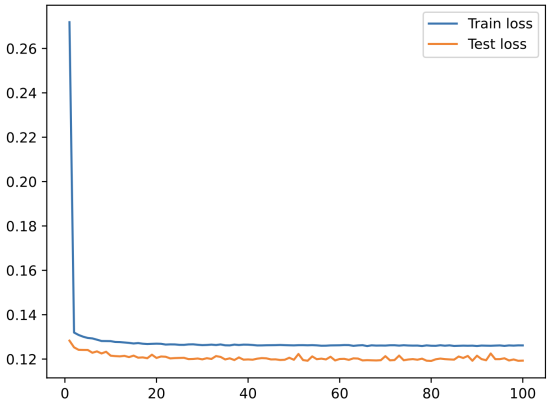
1. f(x) = sin(x)cos(ax). Here are train/test losses,

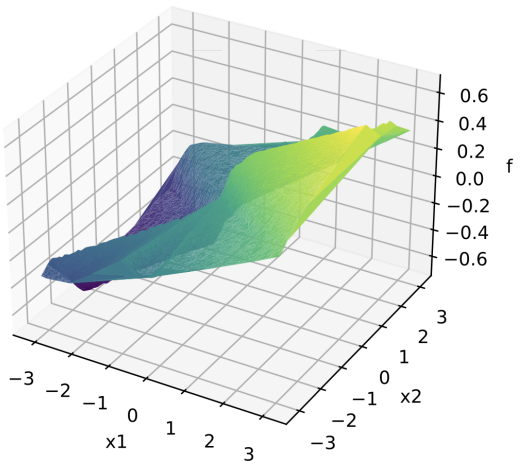
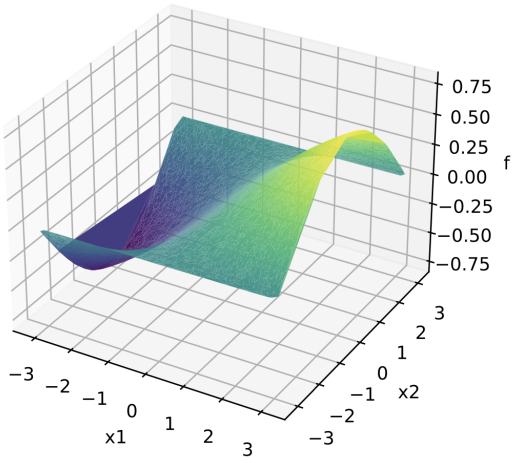
train/test losses for a=1	train/test losses for a=10
	

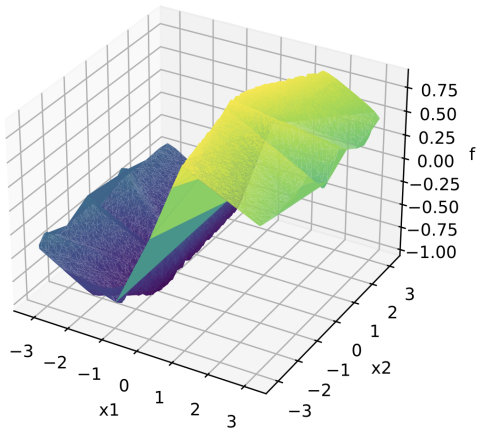
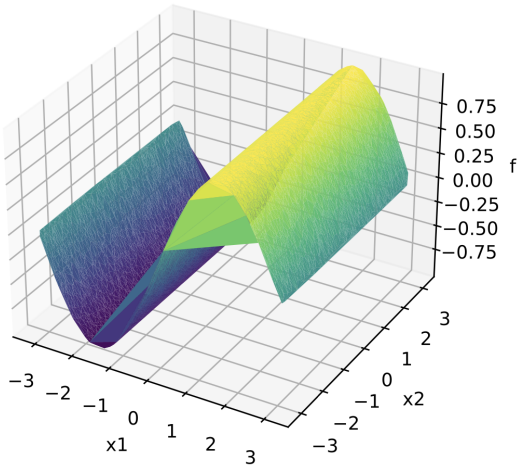
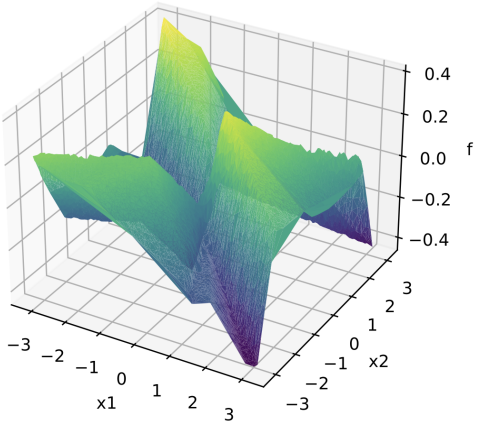
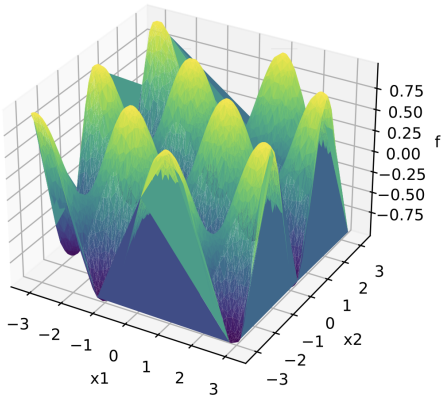
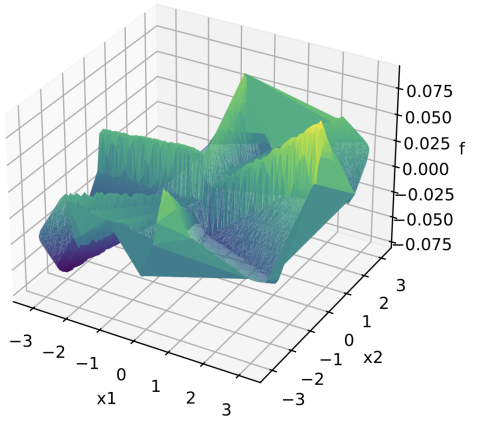
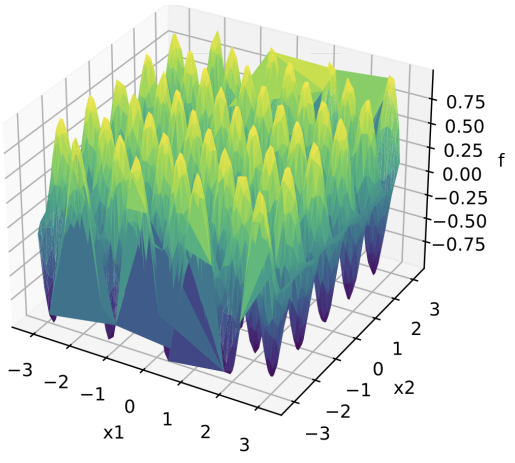
as we use a single activation function with hidden layer of size n the output of the network can't be too expressive so the network can't capture the rapid changes in f, if a is too large compared to n.
Here we show the actual evaluation of f and the network's prediction for n=15,

predictions for a = 0.4:	evaluation for a= 0.4:
	
predictions for a = 1:	**evaluation for a= 1:**
	
predictions for a = 10:	**evaluation for a= 10:**
	

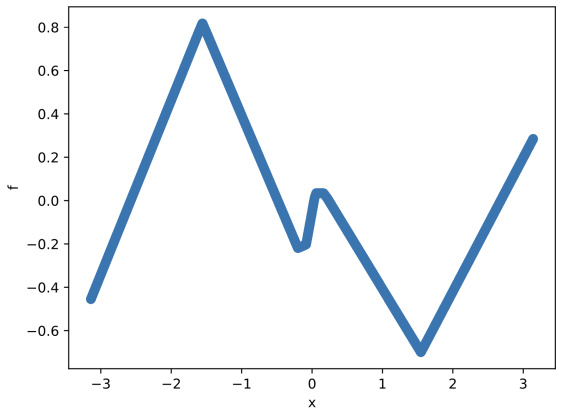
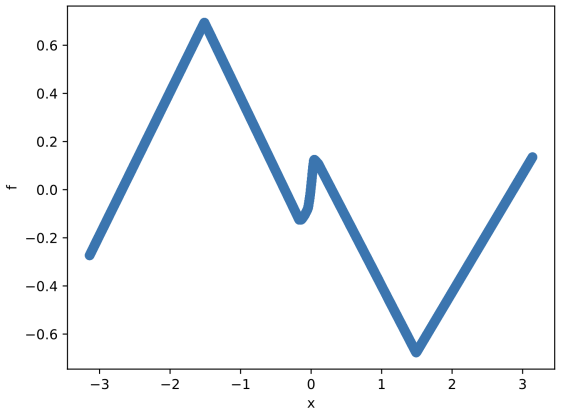
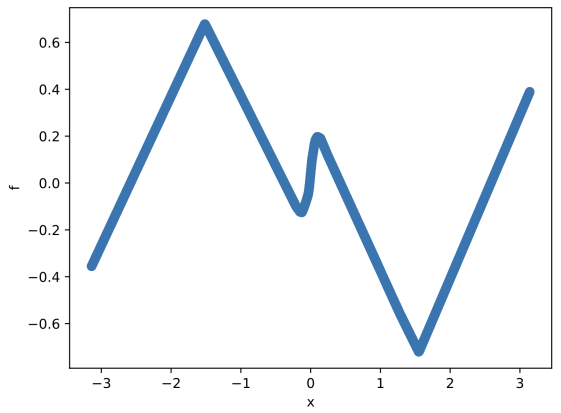
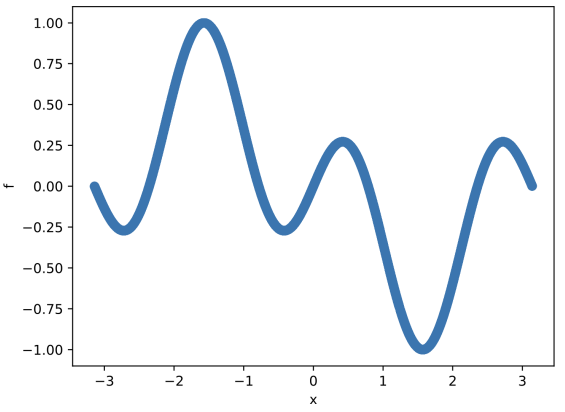
2. f(x1,x2) = sin(a1x1)cos(a2x2) we can expect a similar trend as the function is separable. We have used a single hidden layer with 20 neurons.

train/test losses for a=[0.3,0.5]	train/test losses for a=[1.6,2]
	
train/test losses for a=[1,0]	train/test losses for a=[4,5]
	

predictions for a=[0.3,0.5]	evaluation for a=[0.3,0.5]
	

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]
	

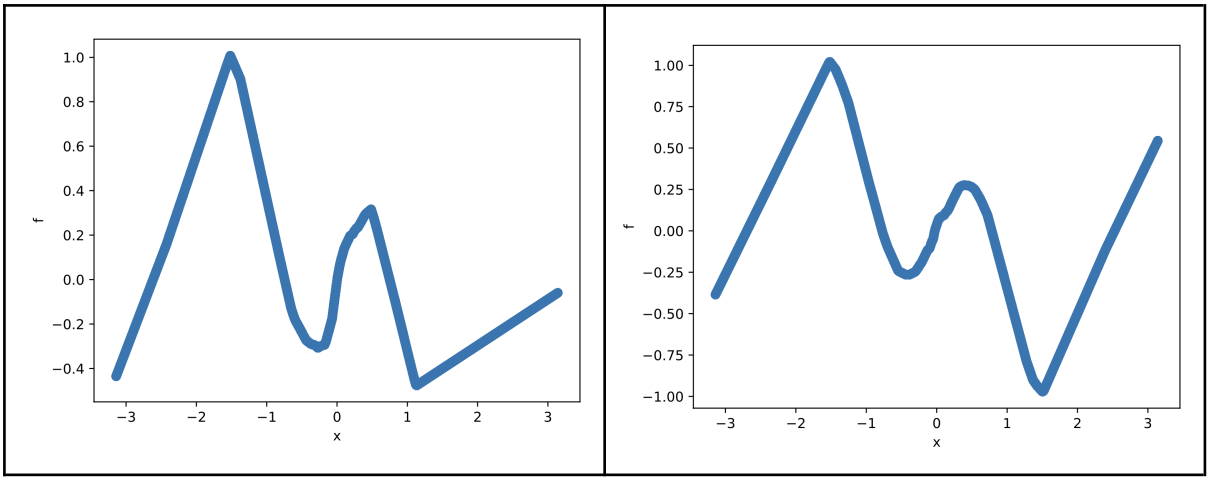
3. We now investigate the relation between the hidden layer dimension and the parameter a. As we've discussed, if a is large (meaning that frequency of f is large) then we cannot expect to mimic f using a small finite sum of relu functions as each component makes a single sharp change of direction over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

n=15	n=30
	
n=50	actual data:
	

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.

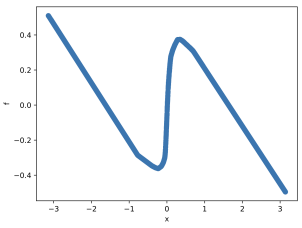
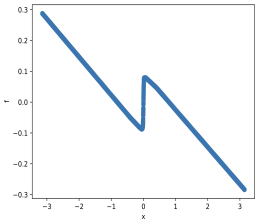
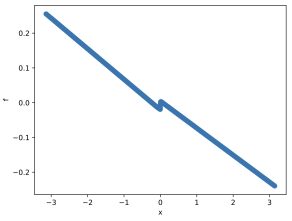
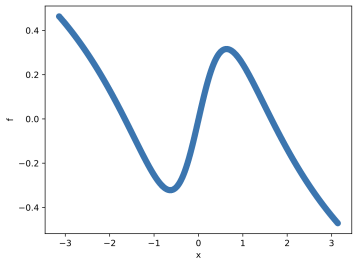
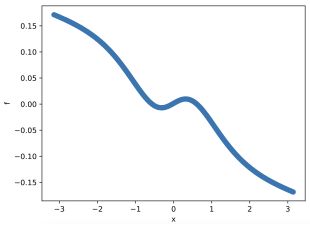
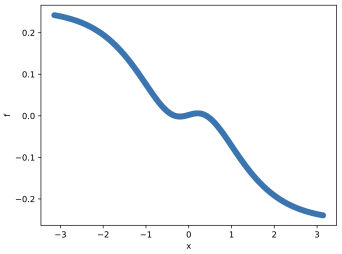
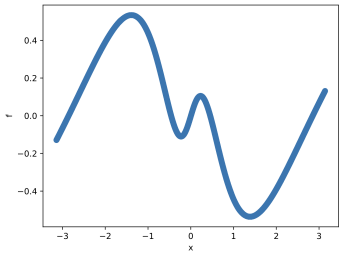
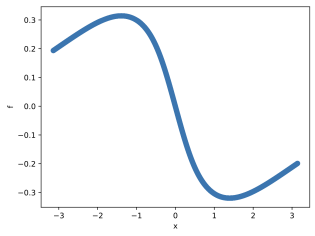
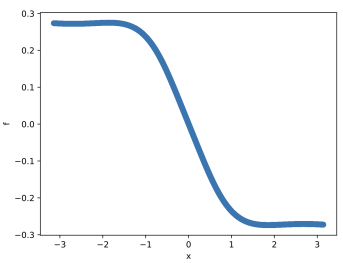
4. Lastly, we can also look at larger networks with more layers, here are two examples for the same function as above, f(x)=sin(x)cos(2x)

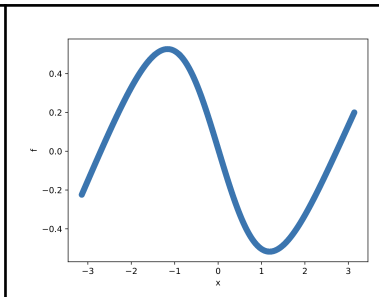
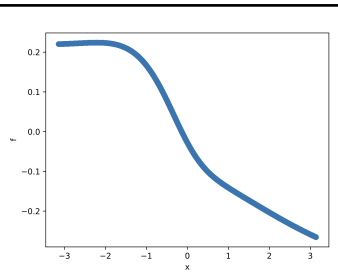
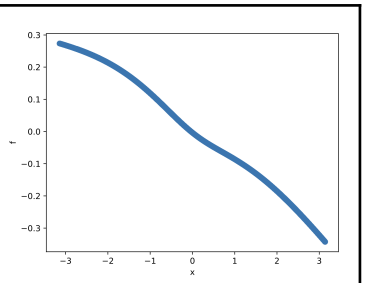
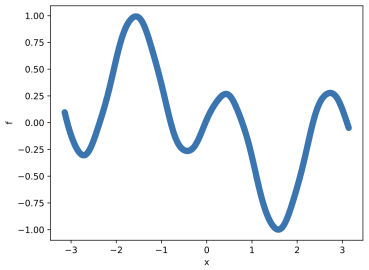
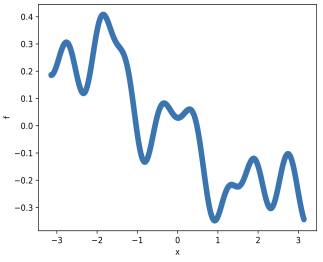
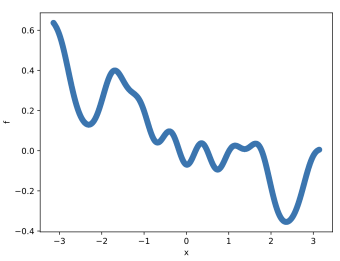
layers=[1,20,40,15,1]	layers=[1,20,50,40,15,1]



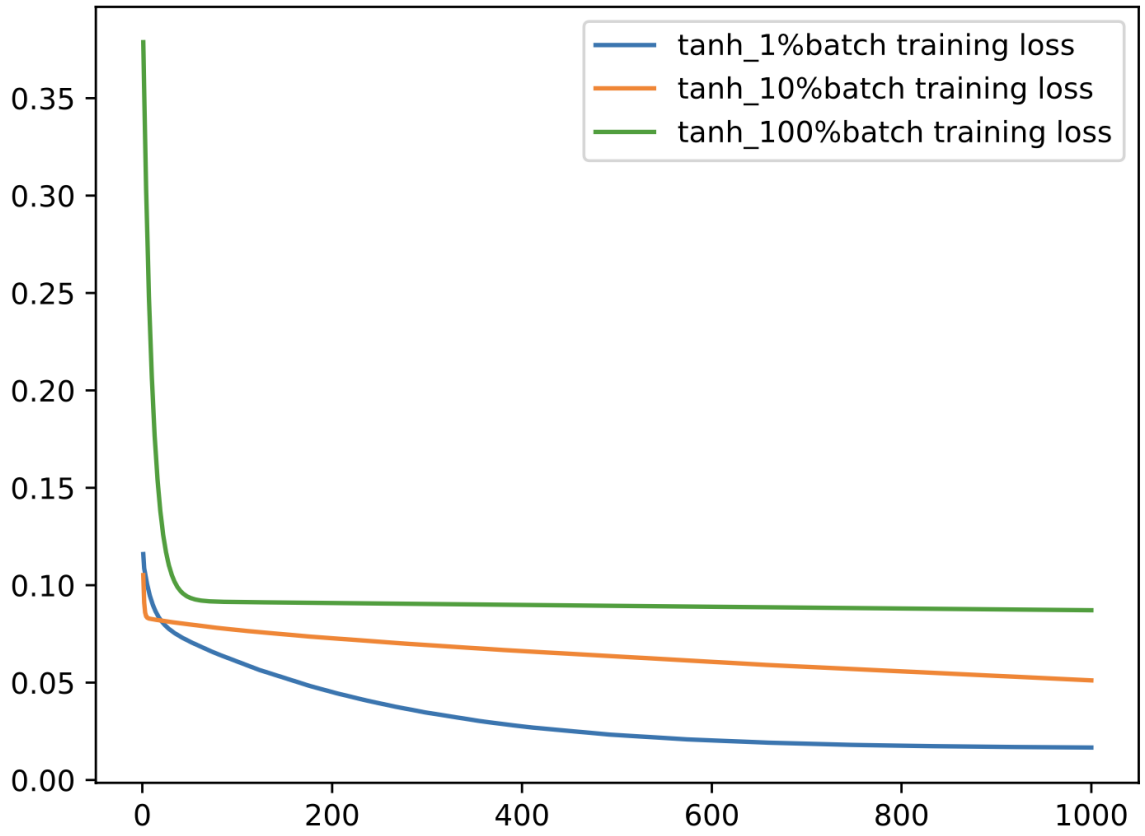
This improves the approximation around zero, but remains inaccurate further away.

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

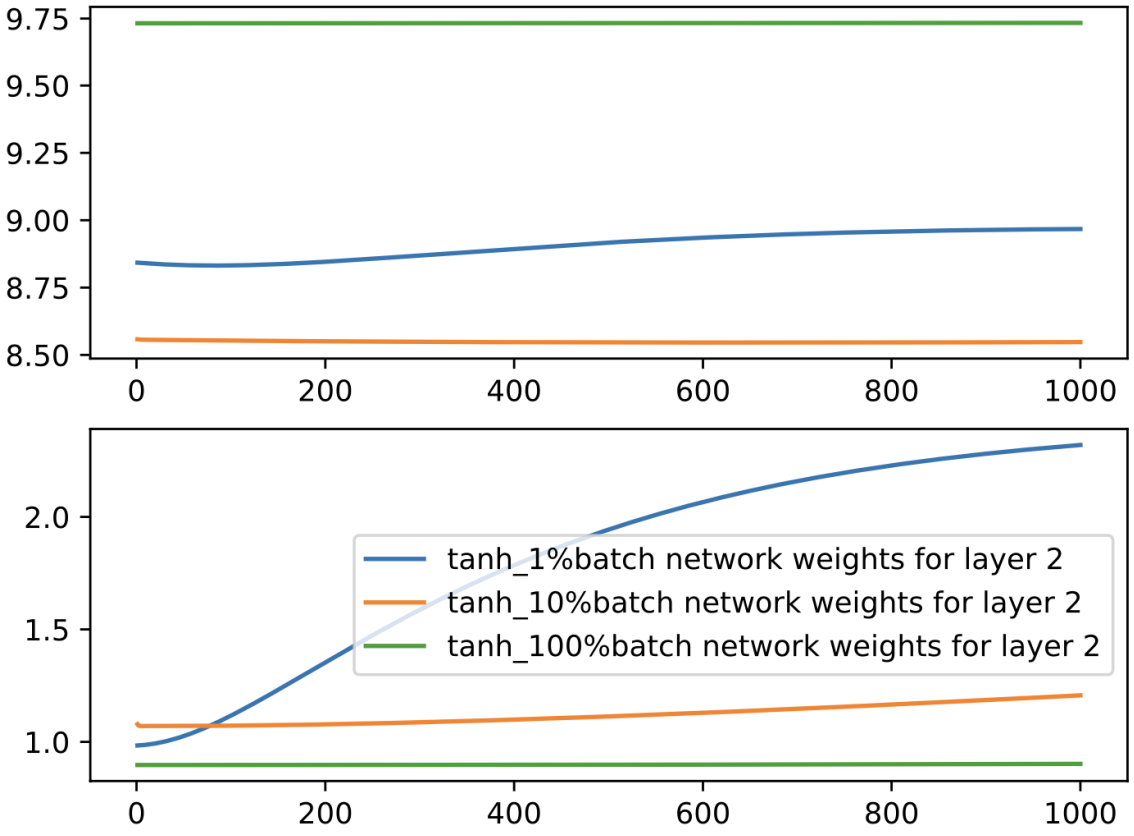
	batch_size=1%	batch_size=10%	batch_size=100%
relu a = 2			
sigmoid a = 2			
tanh a = 2			

swish			
snake_2 a = 2			

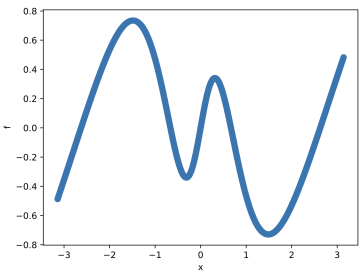
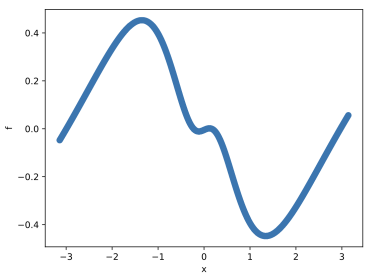
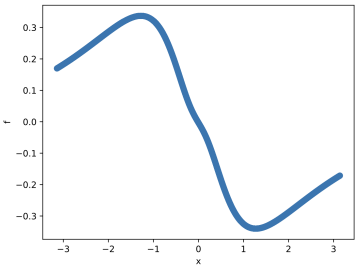
Peculiarly, it seems as if considering more data points for evaluating the empirical risk decreases the ability to learn. We may attribute this phenomena to a non robust GD method, as it seems that the larger the batch, the smaller the gradient step, and our method does not have an adaptive learning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



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



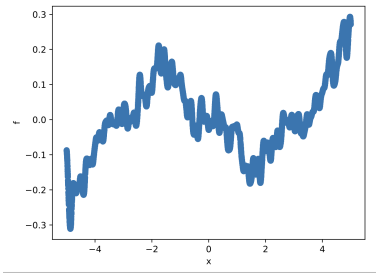
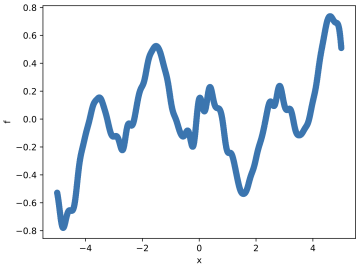
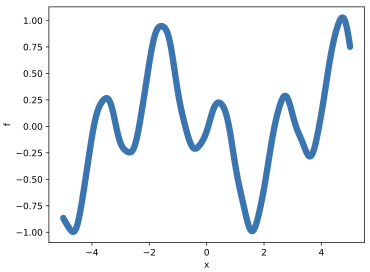
we can see how little the single batch network change over the iterations
The problem with increasing the learning rate is that in the beginning of the GD the single batch network does learn and so, a big learning rate can cause overshooting. Fine tuning this parameter is elusive and better techniques are in order.
Letting all networks train till convergence yield (on the tanh activation),

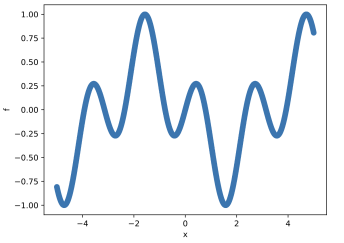
batch_size=1%	batch_size=10%	batch_size=100%
		

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 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

b = 10	b=5	b=2
		

original data:


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.