

# **MTH 602 Scientific Machine Learning**

Homework 6

11/17/2025

S. M. Mahfuzul Hasan

02181922



## I. PAPER & PENCIL WORK

1. Eq. 6.11 in the book

$$y_k(\mathbf{x}, \mathbf{w}) = f \left( \sum_{j=0}^M w_{kj}^{(2)} h \left( \sum_{i=0}^D w_{ji}^{(1)} x_i \right) \right) \quad (1)$$

- In the first layer, there are  $M$  weights ranging from 1 to  $M$ , and the number of features are  $(D + 1)$ . It is assumed based on the figure 6.9 of the book that, the bias of the input features is not connected to  $z_0$ . As a result, the number of neurons becomes  $M$  for first layer. So, the total number of learnable parameters in first layer is

$$M(D + 1)$$

In the second layer,  $j$  ranges from 0 to  $M$ . So, there are a total of  $(M + 1)$  weights. Considering only one output, and setting  $k = 1$ , the total number of learnable parameters in second layer is

$$(M + 1)$$

So, the total number of learnable parameters in eq. (1) is

$$M(D + 1) + (M + 1) = MD + 2M + 1$$

- For  $D = 1$  and  $M = 2$ , eq. (1) can be written as,

$$y_k(\mathbf{x}, \mathbf{w}) = f \left( \sum_{j=0}^2 w_{kj}^{(2)} h \left( \sum_{i=0}^1 w_{ji}^{(1)} x_i \right) \right) \quad (2)$$

where,

$$\begin{aligned} a_j^{(1)} &= w_{j1}^{(1)} x_1 + w_{j0}^{(1)} \\ z_j &= h(a_j^{(1)}) = \max(0, a_j^{(1)}) \end{aligned} \quad (3)$$

and,

$$\begin{aligned} a_k^{(2)} &= w_{k2}^{(2)} z_2 + w_{k1}^{(2)} z_1 + w_{k0}^{(2)} \\ y_k &= f(a_k^{(2)}) = \max(0, a_k^{(2)}) \end{aligned} \quad (4)$$

If we set  $\{w_{10}^{(1)}, w_{11}^{(1)}, w_{20}^{(1)}, w_{21}^{(1)}\} = \{0, 1, 0, -1\}$  and  $\{w_{10}^{(2)}, w_{11}^{(2)}, w_{12}^{(2)}\} = \{0, 2, -2\}$  in eq. (3) and (4) respectively, we get

$$\begin{aligned} a_1^{(1)} &= x_1 + 0 = x_1 \\ a_2^{(1)} &= -x_1 + 0 = -x_1 \\ z_1 &= h(a_1^{(1)}) = \max(0, x_1) = x_1 \quad [\text{just showing } x_1 > 0 \text{ case}] \\ z_2 &= h(a_2^{(1)}) = \max(0, -x_1) = 0 \\ a_1^{(2)} &= 0 + 2x_1 + 0 = 2x_1 \\ y_1 &= f(a_1^{(2)}) = 2x_1 \end{aligned} \quad (5)$$

And, If we set  $\{w_{10}^{(1)}, w_{11}^{(1)}, w_{20}^{(1)}, w_{21}^{(1)}\} = \{0, -2, 0, 2\}$  and  $\{w_{10}^{(2)}, w_{11}^{(2)}, w_{12}^{(2)}\} = \{0, -1, 1\}$  in eq. (3) and (4) respectively, we get

$$\begin{aligned}
 a_1^{(1)} &= -2x_1 + 0 = -2x_1 \\
 a_2^{(1)} &= 2x_1 + 0 = 2x_1 \\
 z_1 &= h(a_1^{(1)}) = \max(0, -2x_1) = 0 \quad [\text{just showing } x_1 > 0 \text{ case}] \\
 z_2 &= h(a_2^{(1)}) = \max(0, 2x_1) = 2x_1 \\
 a_1^{(2)} &= 2x_1 + 0 + 0 = 2x_1 \\
 y_1 &= f(a_1^{(2)}) = 2x_1
 \end{aligned} \tag{6}$$

So, it can be concluded from eq. (5) and (6) that, with the usage of different weights for the same architecture, even a simple multiplication can result in non-trivial networks, and the solutions are not unique.

- Book problem 6.4:

Given,

$$\begin{aligned}
 \sigma(a) &= \frac{1}{1 + e^{-a}} \\
 y_k(\mathbf{x}, \mathbf{w}) &= f \left( \sum_{j=0}^M w_{kj}^{(2)} \sigma \left( \sum_{i=0}^D w_{ji}^{(1)} x_i \right) \right)
 \end{aligned} \tag{7}$$

Eq. (6.14) from the book

$$\begin{aligned}
 \tanh(a) &= \frac{e^a - e^{-a}}{e^a + e^{-a}} = \frac{1 - e^{-2a}}{1 + e^{-2a}} = \frac{2}{1 + e^{-2a}} - \frac{1 + e^{-2a}}{1 + e^{-2a}} \\
 &= \frac{2}{1 + e^{-2a}} - 1 = 2\sigma(2a) - 1 \\
 \Rightarrow \sigma(2a) &= \frac{1}{2} (\tanh(a) + 1) \\
 \Rightarrow \sigma(a) &= \frac{1}{2} \left( \tanh \left( \frac{a}{2} \right) + 1 \right) \\
 \Rightarrow \sigma \left( \sum_{i=0}^D w_{ji}^{(1)} x_i \right) &= \frac{1}{2} \left( \tanh \left( \frac{1}{2} \sum_{i=0}^D w_{ji}^{(1)} x_i \right) + 1 \right) \quad [\text{from eq. (1)}]
 \end{aligned} \tag{8}$$

Now using eq.(8) in eq. (7),

$$\begin{aligned}
y_k(\mathbf{x}, \mathbf{w}) &= f \left( \sum_{j=0}^M w_{kj}^{(2)} \frac{1}{2} \left( \tanh \left( \frac{1}{2} \sum_{i=0}^D w_{ji}^{(1)} x_i \right) + 1 \right) \right) \\
&= f \left( \sum_{j=0}^M \hat{w}_{kj}^{(2)} \tanh \left( \sum_{i=0}^D \hat{w}_{ji}^{(1)} x_i \right) \right), \\
&\quad \hat{w}_{kj}^{(2)} = \frac{1}{2} w_{kj}^{(2)}; \quad j \in [1, M] \\
&\quad \hat{w}_{k0}^{(2)} = \frac{1}{2} (w_{k0}^{(2)} + 1) \\
&\quad \hat{w}_{ji}^{(1)} = \frac{1}{2} w_{ji}^{(1)}
\end{aligned} \tag{9}$$

which is clearly a network that differs by the linear transformation of the parameters compared to eq. (7). (Showed)

## 2. • Book problem 7.3

Given,

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2} (\mathbf{w} - \mathbf{w}^*)^T \mathbf{H} (\mathbf{w} - \mathbf{w}^*) \tag{10}$$

If  $\mathbf{H}$  is positive definite, then by definition

$$(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H} (\mathbf{w} - \mathbf{w}^*) > 0$$

when  $\mathbf{w} \neq 0$ .

That makes  $E(\mathbf{w}) > E(\mathbf{w}^*)$  for all  $\mathbf{w}$  except for  $\mathbf{w} = \mathbf{w}^*$ . Hence,  $\mathbf{w}^*$  must be the local minimum of  $E(\mathbf{w})$ .

Alternatively, if  $\mathbf{w}^*$  is the minimum of  $E(\mathbf{w})$ , then for any  $\mathbf{w}$ ,  $E(\mathbf{w}) > E(\mathbf{w}^*)$ . That can only be possible if

$$(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H} (\mathbf{w} - \mathbf{w}^*) > 0$$

which is possible if and only if  $\mathbf{H}$  is positive definite.

So either way Hessian matrix  $\mathbf{H}$  needs to be positive definite for any stationary point to be a local minimum of the error function  $E(\mathbf{w})$ , which is the necessary and sufficient condition as well. (Showed)

## • Book problem 7.4

Given,

$$y(x, w, b) = wx + b$$

$$E(w, b) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, w, b) - t_n\}^2 = \frac{1}{2} \sum_{n=1}^N \{(wx_n + b) - t_n\}^2$$

Now,

$$\begin{aligned}
\frac{\partial E}{\partial w} &= \sum_{n=1}^N \{(wx_n + b) - t_n\} x_n \\
\frac{\partial E}{\partial b} &= \sum_{n=1}^N \{(wx_n + b) - t_n\} \\
\frac{\partial^2 E}{\partial w^2} &= \sum_{n=1}^N x_n^2 \\
\frac{\partial^2 E}{\partial b \partial w} &= \frac{\partial^2 E}{\partial w \partial b} = \sum_{n=1}^N x_n = N\bar{x} \\
\frac{\partial^2 E}{\partial b^2} &= \sum_{n=1}^N 1 = N
\end{aligned} \tag{11}$$

So,

$$\mathbf{H} = \begin{bmatrix} \sum_{n=1}^N x_n^2 & N\bar{x} \\ N\bar{x} & N \end{bmatrix}$$

Here,

$$\text{Tr}(\mathbf{H}) = \sum_{n=1}^N x_n^2 + N > 0$$

And,

$$\det(\mathbf{H}) = N \sum_{n=1}^N x_n^2 - (N\bar{x})^2 \tag{12}$$

We know,

$$\begin{aligned}
\sigma^2 &= \frac{1}{N} \sum_{n=1}^N (x_n - \bar{x})^2 = \frac{1}{N} \sum_{n=1}^N (x_n^2 - 2x_n\bar{x} + \bar{x}^2) \\
&\Rightarrow \sum_{n=1}^N x_n^2 = N(\sigma^2 + \bar{x}^2)
\end{aligned} \tag{13}$$

Using eq. (13) in (12),

$$\det(\mathbf{H}) = N^2 (\sigma^2 + \bar{x}^2) - (N\bar{x})^2 = N^2 \sigma^2 > 0$$

Since, trace represents the sum of the eigenvalues and determinant corresponds to the product of the eigenvalues, both can be positive at the same time if and only if all the eigenvalues are positive. If the eigenvalues are positive, Hessian must be positive definite. If that is so, then by necessary and sufficient condition, the stationary point of the error function is minimum. (Showed)

## IV. RIDGE REGRESSION VIA SGD IN JAX

### D. Write and test the model and loss functions

1. Please refer to listing 1.
2. Please refer to listing 1
3. A function to calculate  $MSE$  is written in listing 1. To check the efficacy of the written

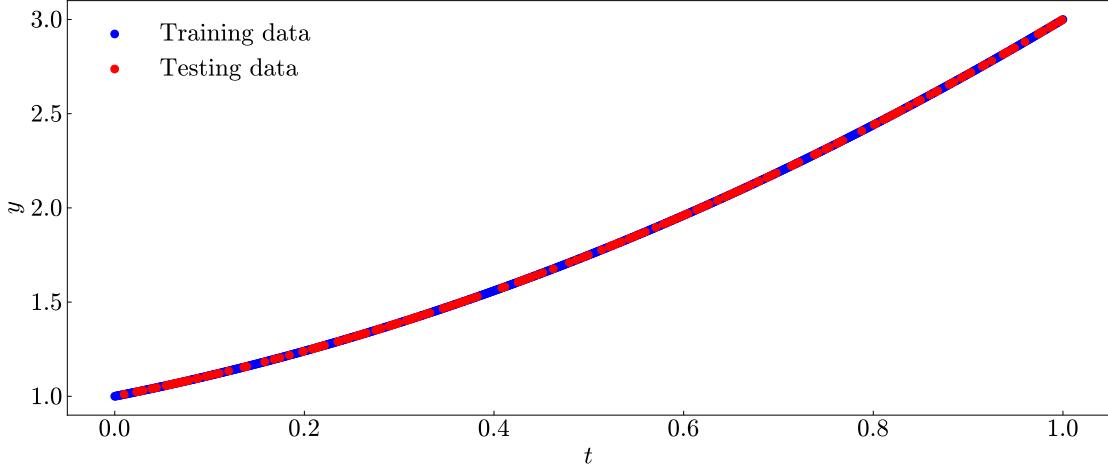


Figure 1: Training and testing data generated using `generate_data()`.

functions, two cases are considered. One is the exact polynomial weights  $\vec{\omega}_{true} = \{1, 1, 1\}$ , another is the wrong polynomial weights  $\vec{\omega}_{wrong} = \{1, 0.5, 0.5\}$ . If the functions are written correctly, the *loss* and *MSE* should come out to be in machine accuracy for case-1, and should be high for case-2.

Computed *loss* and *MSE* for case-1 (true weights):

$$loss = 0.0, \quad MSE = 0.0$$

Training and testing *MSE* are also calculated for case-1 (true weights).

$$MSE_{training} = 0.0, \quad MSE_{testing} = 0.0$$

Computed *loss* and *MSE* for case-2 (wrong weights):

$$loss = 258.57525025024205, \quad MSE = 0.25857525025024203$$

Since, with the correct weights the *loss* and *MSE* are 0.0 and for wrong weights, those are high, we can conclude that the functions are working properly.

## E. Optimizing with stochastic gradient descent

1. Given,

$$M = 3, \quad \lambda = 0, \quad \alpha = 0.2, \quad B = N_{train}$$

We start off by calculating the theoretical threshold of learning rate based on the maximum eigenvalue of the Hessian matrix, which was formed both from unscaled and scaled input data.

$$\mathbf{H} = 2A^T A$$

Since,  $t \in [0, 1]$ , its scaling was done in  $[-1, 1]$ .

$$\begin{aligned} \tilde{t} &= 2(t - 0.5) \\ \Rightarrow t &= 0.5\tilde{t} + 0.5 \end{aligned} \tag{14}$$

So, the true polynomial becomes

$$y(\tilde{t}) = 1 + (0.5\tilde{t} + 0.5) + (0.5\tilde{t} + 0.5)^2 = 1.75 + \tilde{t} + 0.25\tilde{t}^2 \tag{15}$$

So, the true weights become:

$$\vec{\omega}_{\text{true, unscaled}} = \{1, 1, 1\}, \quad \vec{\omega}_{\text{true, scaled}} = \{1.75, 1, 0.25\}$$

The closed form solutions for both the unscaled and scaled data with  $M = 3$  are:

$$\vec{\omega}_{\text{closed, unscaled}} = \{1, 1, 1, 1.948 \times 10^{-13}\}, \quad \vec{\omega}_{\text{closed, scaled}} = \{1.75, 1, 0.25, 8.394 \times 10^{-15}\}$$

The condition for the theoretical maximum learning rate:

$$\alpha_{max} = \frac{2}{\lambda_{max}(\mathbf{H})}$$

The theoretical upper bound for learning rate:

$$\alpha_{\text{theory, unscaled}} = 8.355 \times 10^{-4}, \quad \alpha_{\text{theory, scaled}} = 1.114 \times 10^{-3}$$

Both of these theoretical bound of learning rate is smaller than the learning rate prescribed for this problem. So theoretically speaking, SGD should diverge. And it actually does if we use `optax.sgd()` with only  $\alpha$  as an argument as can be seen from the  $l2$  norm of the SGD and closed form solution weights in fig. 2. Scaled data did not change the blowup of SGD as well.

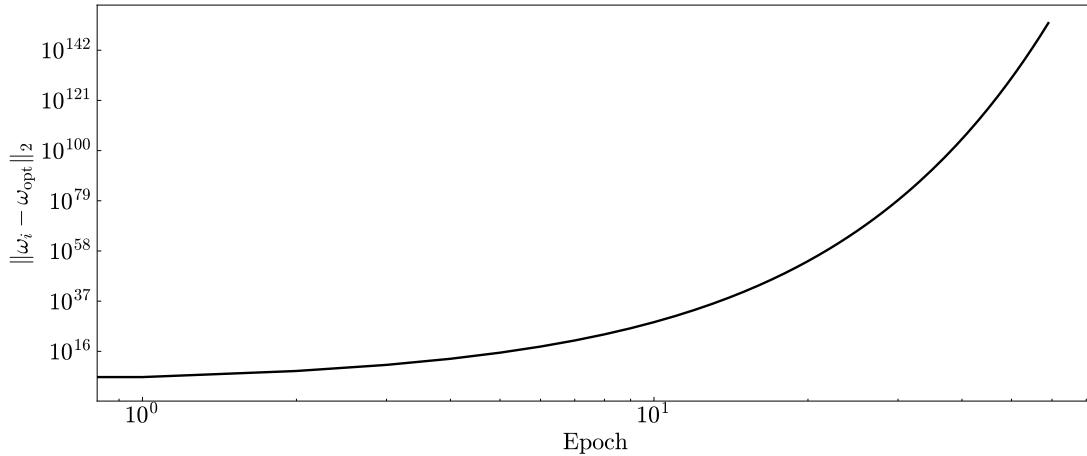


Figure 2:  $L2$  norm of the weights for unscaled data.

To remedy this, `optax.chain()` is used to make sure that the gradient does not blow up by clipping the gradient to a value of 0.5, along with `optax.sgd()` as another argument, which already had the  $\alpha$  as its argument. With this, SGD did not diverge and converges till certain point before getting stalled.

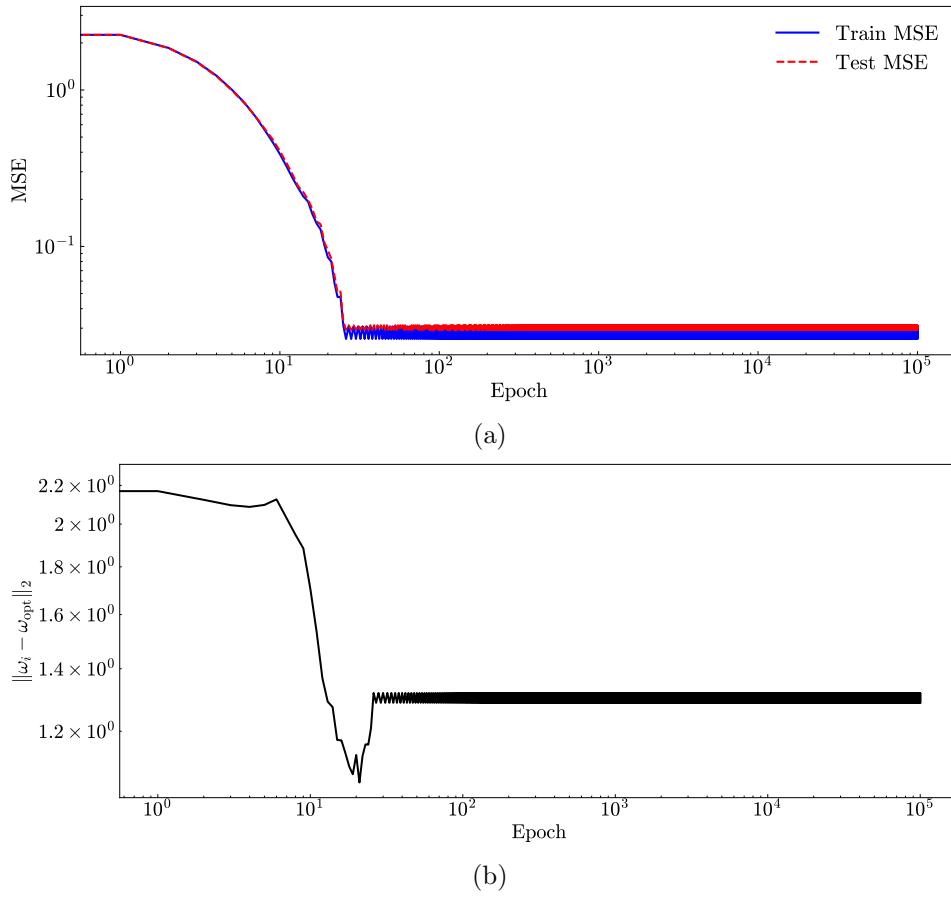


Figure 3: (a)  $MSE$  of loss and (b)  $l2$  norm of the weights for unscaled data.

Fig. 3 shows how  $MSE$  and  $l2$  norm decreases rapidly within 12 epochs, then did not have any change moving forward. Both of the quantities oscillated within a certain amplitude till 100000 epochs. We can see that the  $l2$  norm did not stalled at its minimum values, unlike  $MSE$ .

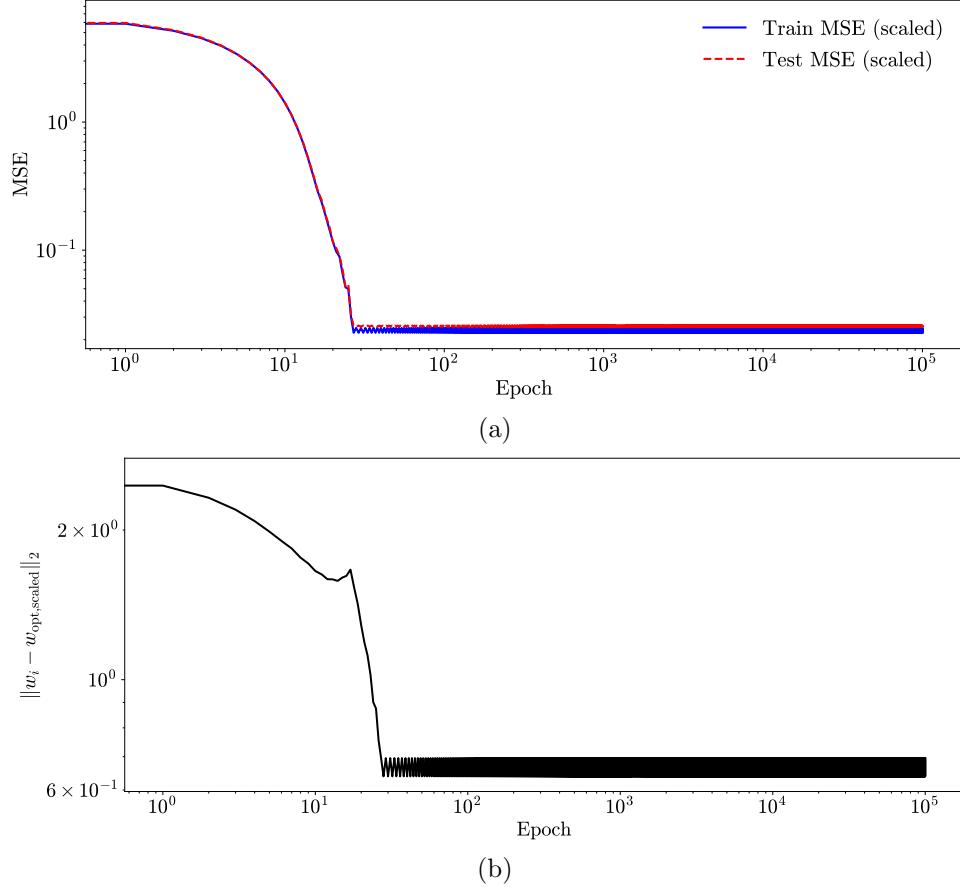


Figure 4: (a)  $MSE$  of loss and (b)  $l2$  norm of the weights for scaled data.

In comparison, scaled data showed a bit better trend than unscaled data as evident from fig. 4.  $L2$  norm stalled at its minimum value rather than shooting up as happened in fig. 3b. However, both of these  $MSE$  and  $l2$  norm showed really poor convergence, and high distance between SGD and closed form solutions.

Final computed weights via SGD:

$$\begin{aligned}\vec{\omega}_{\text{unscaled}} &= \{1.59415786, 0.41523422, 0.41608667, 0.78784401\} \\ \vec{\omega}_{\text{scaled}} &= \{1.64692162, 0.76950196, 0.65296745, 0.50743949\}\end{aligned}$$

Final computed distance:

$$\|\omega_i - \omega_{\text{opt}}\|_2_{\text{unscaled}} = 1.287, \quad \|\omega_i - \omega_{\text{opt}}\|_2_{\text{scaled}} = 0.695$$

So, scaled data gave a better closeness of the SGD weights to the closed form solution compared to unscaled data.

2. Iteration over batches of the training set for each epoch are done with a batch size of 32 for both unscaled and scaled data.

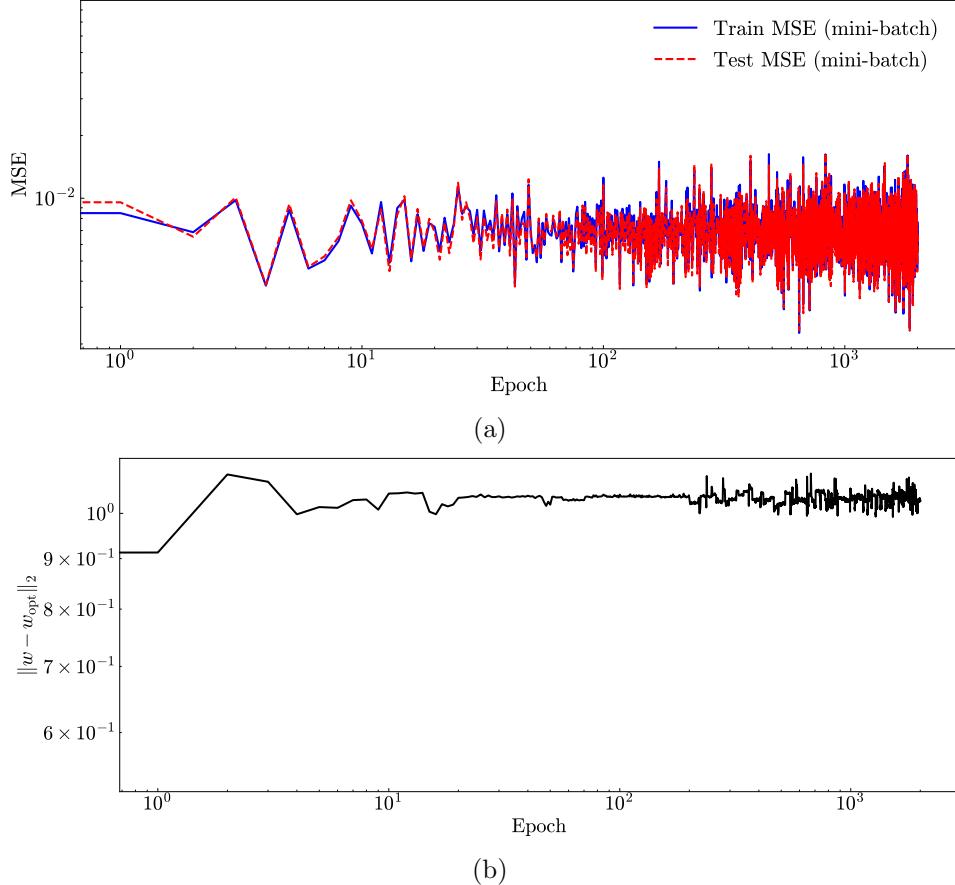


Figure 5: (a)  $MSE$  of loss and (b)  $l_2$  norm of the weights with  $B = 32$  for unscaled data.

Fig. 5a shows that  $MSE$  seems to improve marginally, but as the no. of epochs increases, the oscillation or noise in the  $MSE$  also increases, which was not the case for full batch training. Here, epochs are set to 2000 because of no real change in either  $MSE$  and  $l_2$  norm in fig. 5.

Fig. 6 more or less mimics fig. 5. Only the exception is that  $l_2$  norm is lower and has more intense oscillation than fig. 5b.  $MSE$  is also slightly lower as also was the case for full batch training. Fig. 5 and 6 largely preserve the full batch training results with the addition of spurious oscillations as the convergence somewhat gets stalled.

Final computed weights via SGD:

$$\begin{aligned}\vec{\omega}_{\text{unscaled}} &= \{1.78392582, 0.95504057, 0.33473039, -0.02170426\} \\ \vec{\omega}_{\text{scaled}} &= \{1.76256173, 1.02239192, 0.36638021, 0.09804078\}\end{aligned}$$

Final distance computed:

$$\|\omega_i - \omega_{\text{opt}}\|_2_{\text{unscaled}} = 1.029, \quad \|\omega_i - \omega_{\text{opt}}\|_2_{\text{scaled}} = 0.154$$

Final distance from the closed form solution improved significantly for scaled data, while the

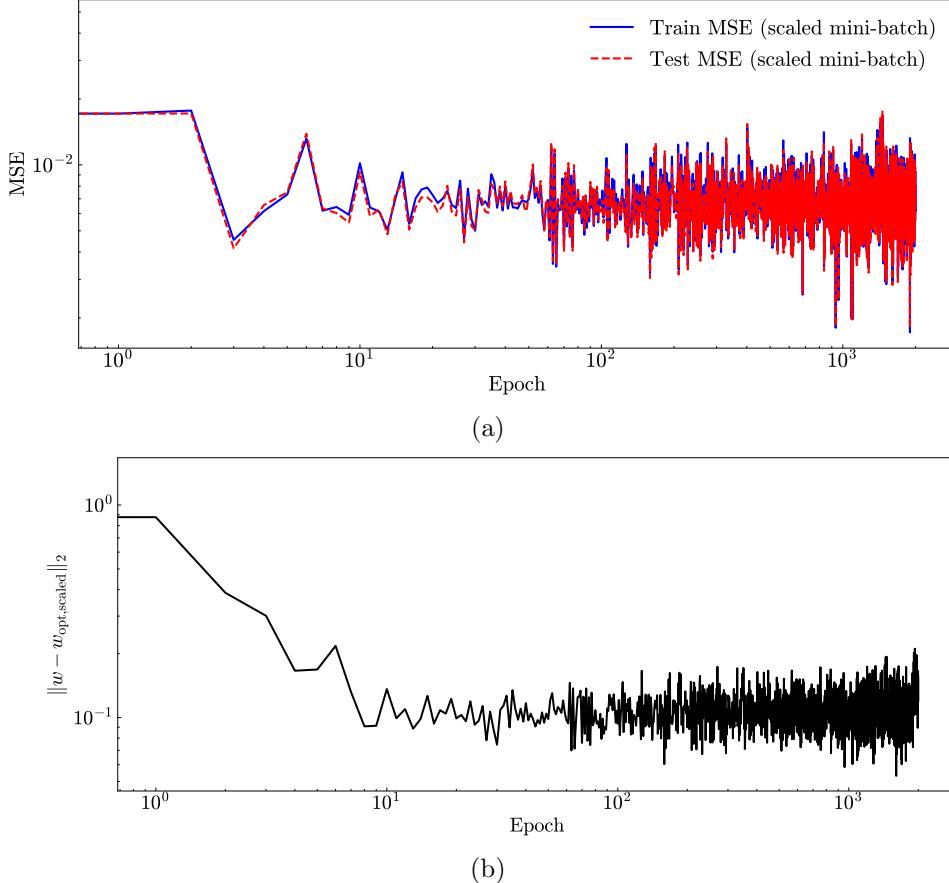


Figure 6: (a)  $MSE$  of loss and (b)  $l2$  norm of the weights with  $B = 32$  for scaled data.

improvement is negligible for unscaled data. So moving forward, scaled data will be used to perform batch size, learning rate and comparative study with other optimizers.

So, mini-batching seems to improve the overall estimation, but in return induces instability to solution.

## F. Batch size study

Given parameters,

$$M = 3, \quad \lambda = 0, \quad \alpha = 0.1, \quad B \in \{1, 16, 64, N_{train}\}, \quad \text{Epochs} = 100000$$

Criteria used in the code (listing 1):

- Minimum epochs: Each batch size case must run for this no. of epochs at initial stage. This was set at 100.
- Maximum epochs: This is the no. of epochs prescribed in the question, i.e., 100000.
- Stopping criteria: Two criteria were used to stop the training based on "target  $MSE$ " and "no improvement". "Target  $MSE$ " was set at  $10^{-4}$ , and "no improvement" criterion had two parameters in order for it to activate: change in the  $MSE$  for two successive epochs should be  $\leq 10^{-5}$  and this should persist for 5 consecutive epochs.

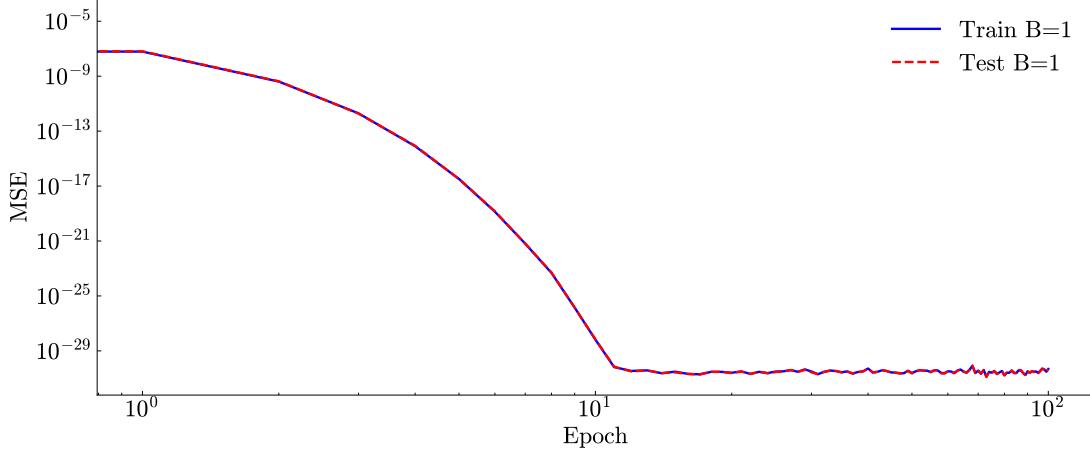


Figure 7:  $MSE$  loss for  $B = 1$ .

Fig. 7 shows the decrease in  $MSE$  for  $B = 1$  that instantly satisfies the target  $MSE$  of  $10^{-4}$ . However, since each batch must run for 100 epochs, we see that  $MSE$  flattens after 10th epoch to  $10^{-29}$ . There is minimal oscillations in the  $MSE$ . For low batch size, gradient should be noisier, but  $B = 1$  shows quite a different story. This might be attributed to the tiny gradient norm that do not get clipped by the optimizer.

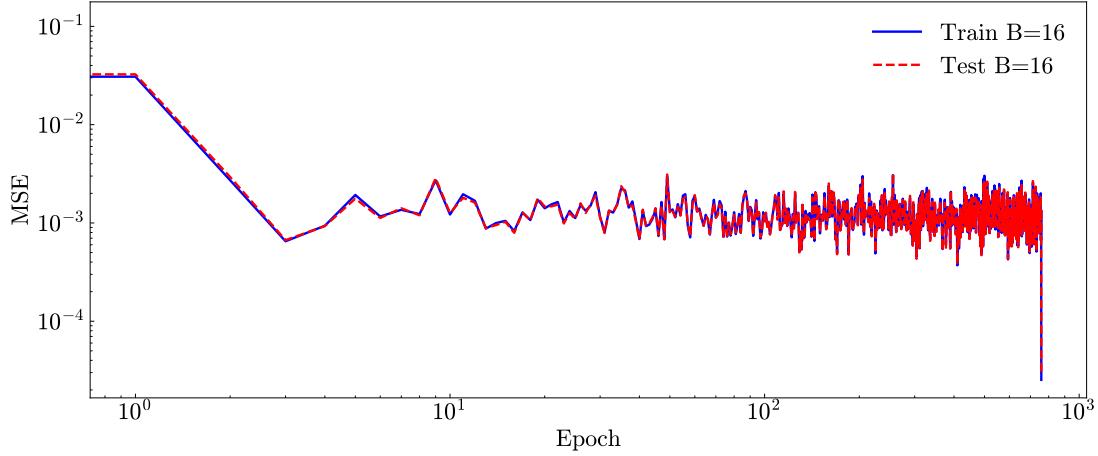


Figure 8:  $MSE$  loss for  $B = 16$ .

For  $B = 16$ , noise in  $MSE$  is present and it plateaus a bit above  $10^{-3}$ . It does not seem to reach the target  $MSE$  ( $=10^{-4}$ ). But it improves upon the  $MSE$  from previous section. There is highly occasional spike that might go below the target  $MSE$ , but it is not anything true convergence. In fig. 8, the run stopped because of that spike meeting the stopping criterion of target  $MSE$ . However, it is expected that the oscillations would continue with the mean a little above  $10^{-3}$ .

For  $B = 64$  in fig. 9, noise in  $MSE$  is more intense, which can happen due to the higher gradient norm, thus more clipping. The  $MSE$  hovers around  $10^{-3}$ , which might seem a tiny bit improved than  $B = 16$ .

Full batch training in fig. 10 shows the same trend as fig. 4a. However, now the values drops below

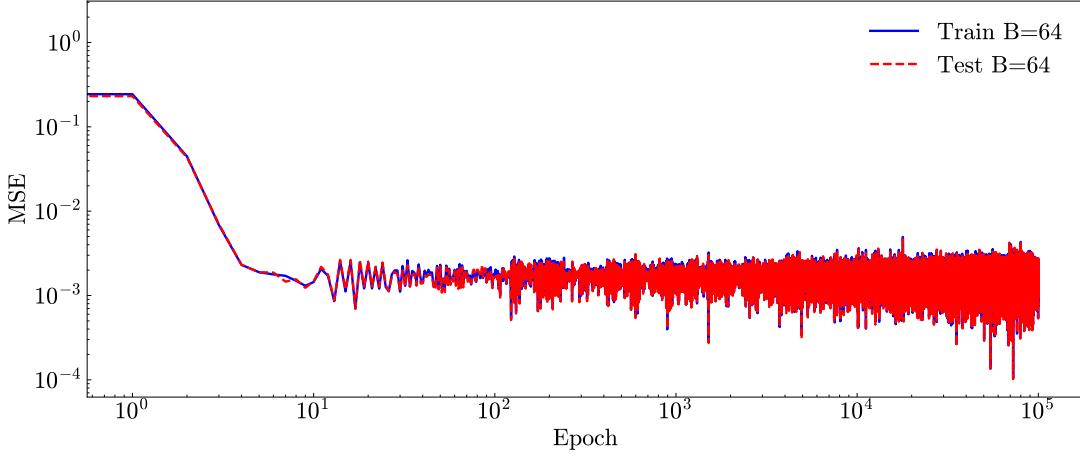


Figure 9:  $MSE$  loss for  $B = 64$ .

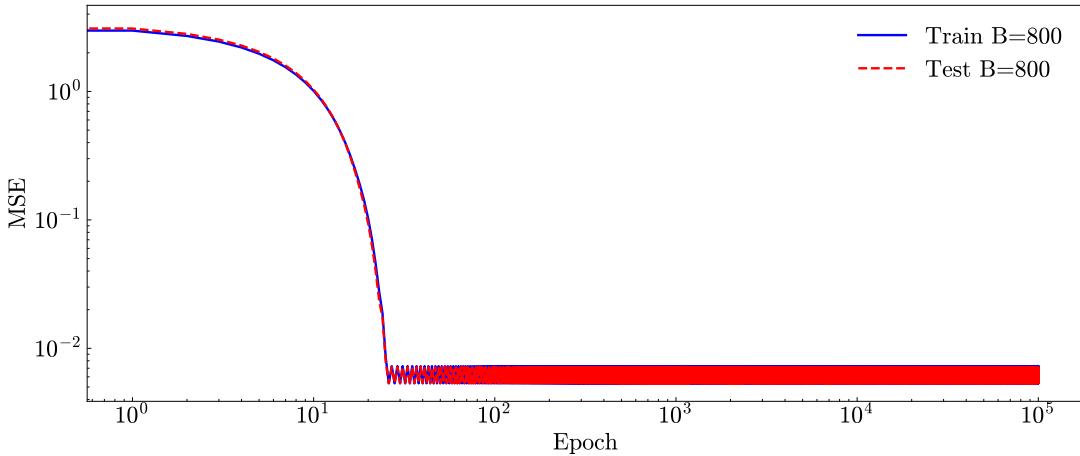


Figure 10:  $MSE$  loss for  $B = N_{train}$ .

$10^{-2}$ , which can be attributed to the difference in random initialization of the weights. This one gives the highest  $MSE$  in training and testing, but it has mild oscillations compared to  $B = 16$  and  $64$ .

Batch-size	Time to reach target $MSE$ (in sec)	Avg. time to complete each epoch (in sec)
1	1.29	1.19
16	57.53	0.076
64	N/A	0.02
800	N/A	0.002

$B = 1$  takes the least time to reach the target  $MSE$ , while  $B = 16$  does not truly reaches the target  $MSE$  but gets lucky with its big spike.  $B = 64$  and  $800$  never reaches the target  $MSE$ . From the avg. time required to complete each epoch, it is clear that  $B = 1$  is the slowest, while  $B = 800$  is the fastest.

### G. Learning-rate study

For  $B = 32$ , three initial learning rate  $\alpha = \{0.1, 0.01, 0.001\}$  were used to compare the stability and performance of three variants of SGD. Number of epochs was set to 5000 for all of them.

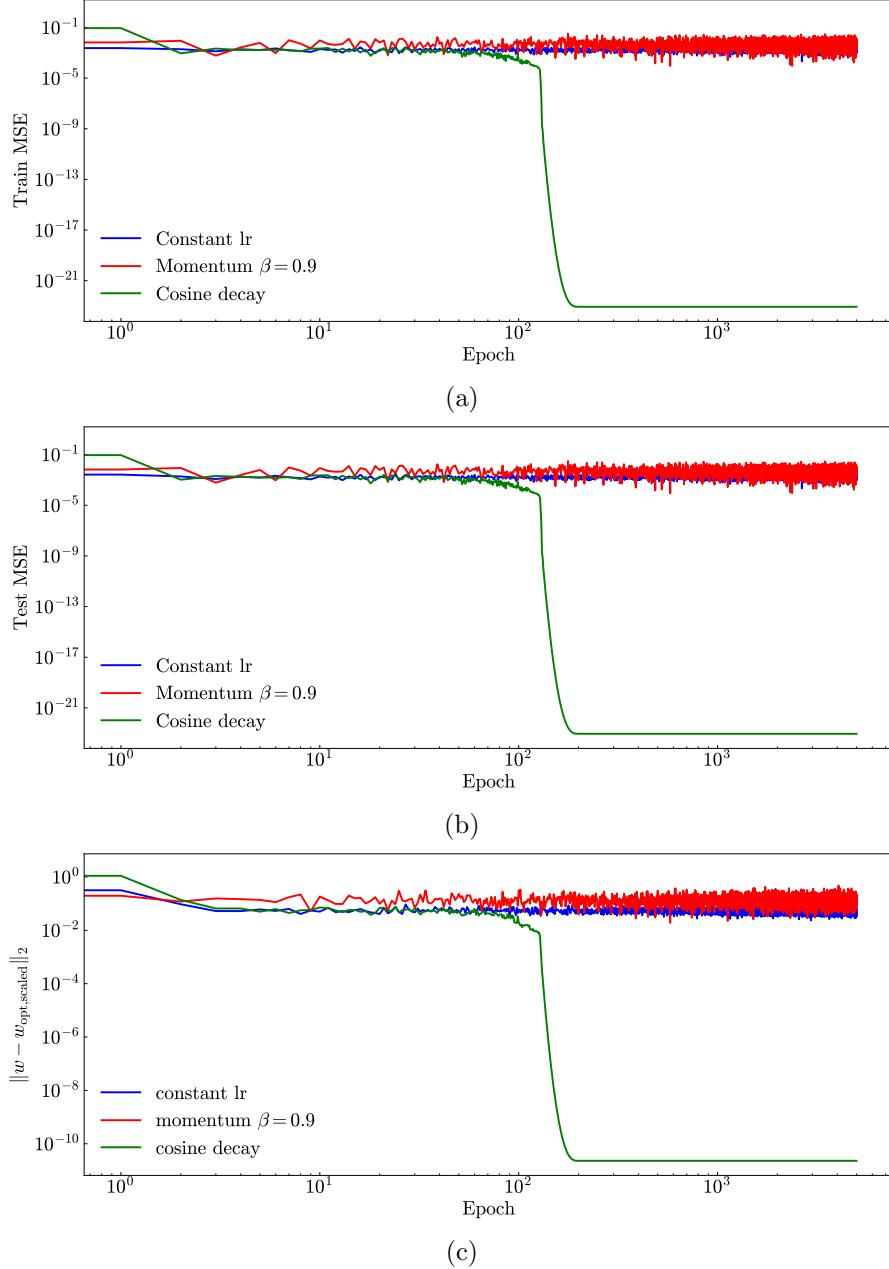


Figure 11: (a) Training  $MSE$ , (b) testing  $MSE$ , and (c)  $l_2$  norm of distance from optimizer's solution to closed-form solution for initial learning rate  $\alpha = 0.1$ .

For  $\alpha = 0.1$ , which is the highest learning rate of the three, none of constant learning rate (lr) and momentum accelerated SGD showed any improvement as can be seen from fig. 11.  $MSE$  does not change over epochs, and oscillations gets intensified as epoch progresses. However, Cosine decay performed extremely well with  $MSE$  at machine precision, and  $l_2$  norm becomes very small

( $\sim 10^{-10}$ ). The stability is great with smooth decay as well.

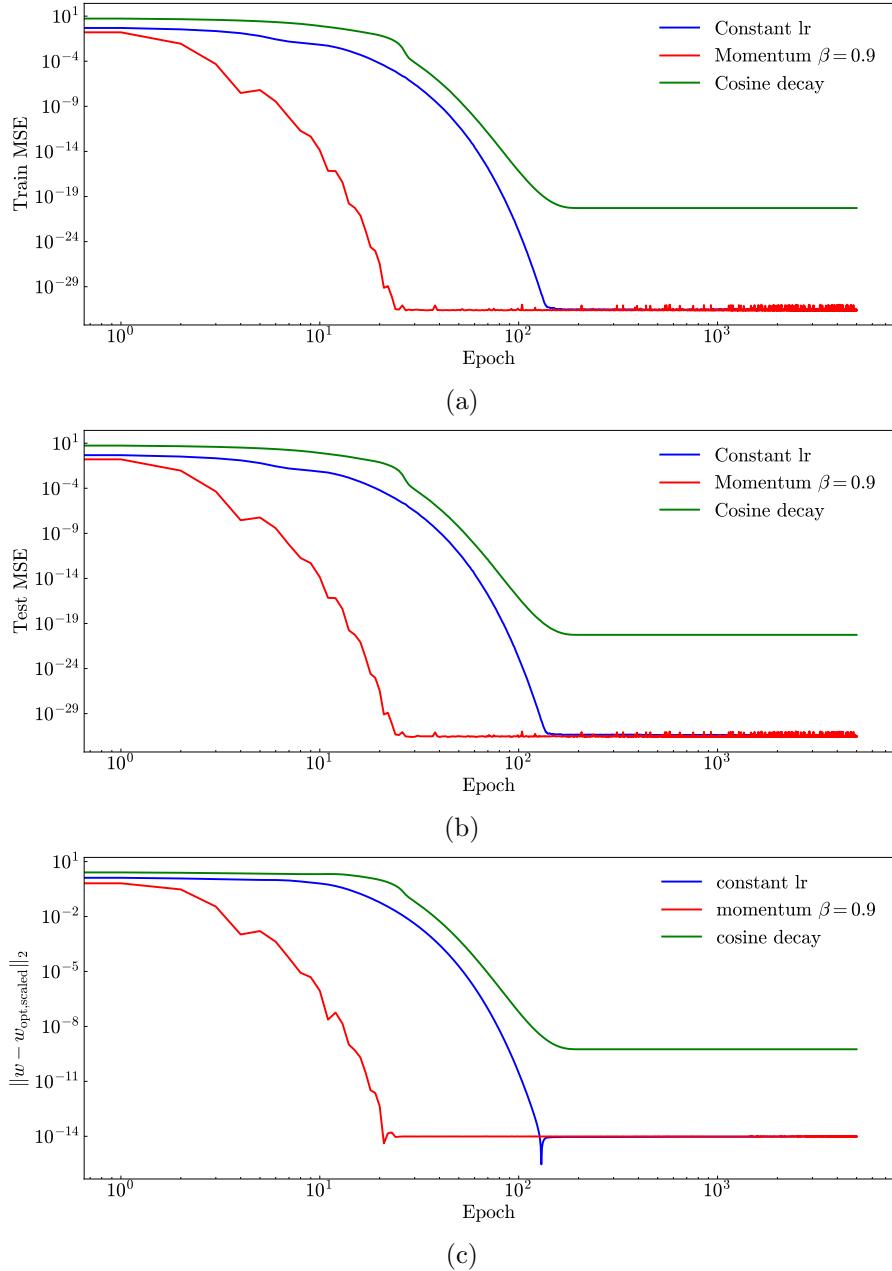


Figure 12: (a) Training  $MSE$ , (b) testing  $MSE$ , and (c)  $l_2$  norm of distance from optimizer’s solution to closed-form solution for initial learning rate  $\alpha = 0.01$ .

For  $\alpha = 0.01$ , momentum accelerated SGD performed best, followed by constant lr one. The stability is also great for both. Cosine decay dropped its performance from previous case, however, it shows stability without any noise. Fig. 12c shows that both constant lr and momentum SGD give the  $l_2$  norm of  $\sim 10^{-14}$ . However, momentum SGD reaches there almost 10x faster. Cosine decay, though showing good  $MSE$  and  $l_2$  norm, can not perform as good as the other two.

For  $\alpha = 0.001$ , constant lr and momentum SGD retain their performance even with greater stability.

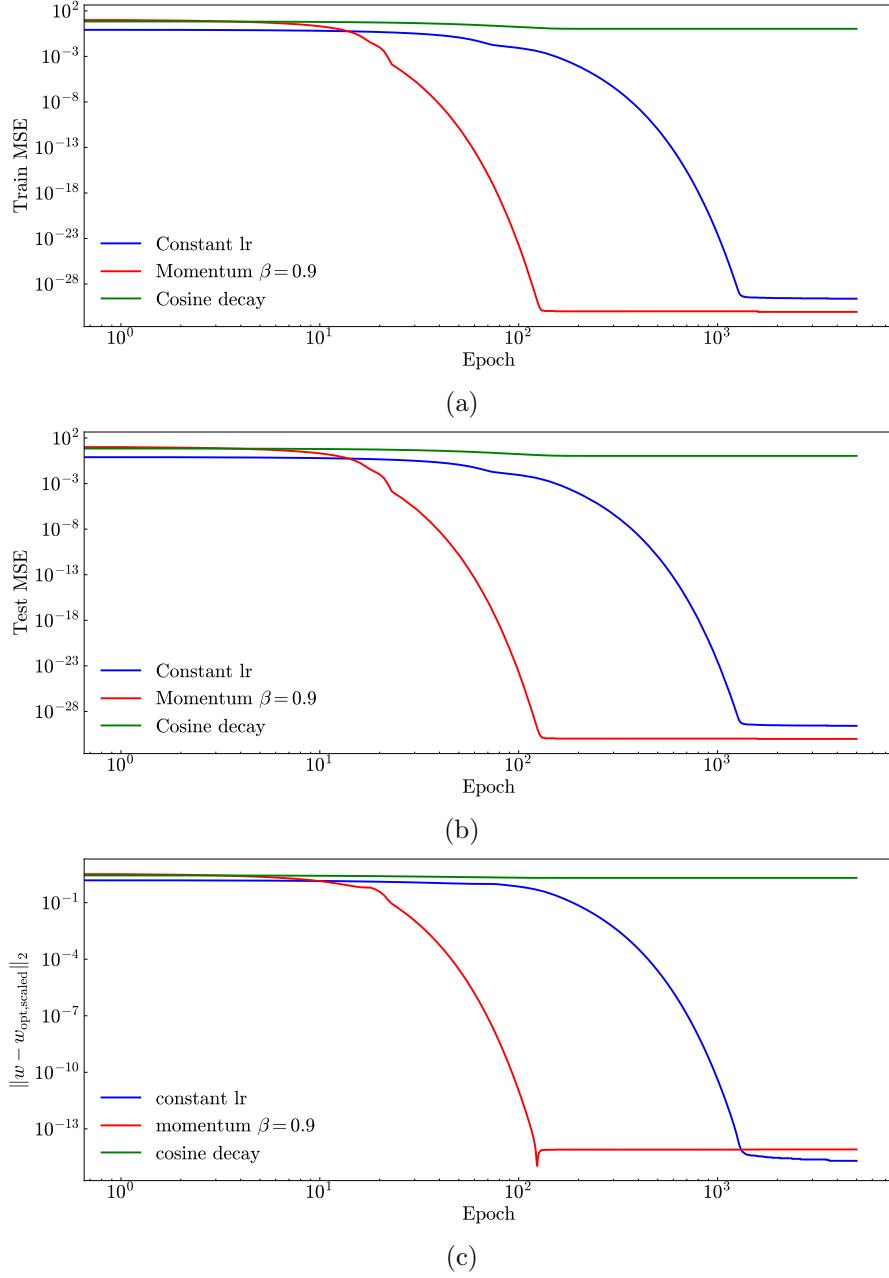


Figure 13: (a) Training  $MSE$ , (b) testing  $MSE$ , and (c)  $l_2$  norm of distance from optimizer’s solution to closed-form solution for initial learning rate  $\alpha = 0.001$ .

The small noise that was present in  $\alpha = 0.01$  vanishes for both of them. Cosine decay shows worst performance as evident in fig. 13. One thing to note is that, Cosine decay does not show instability with oscillations in any of the cases.

So, it can be concisely said that

- Constant lr and momentum accelerated SGD perform better and show better stability as learning rate is decreased. This also explains why in previous sections, with learning rate  $\alpha = 0.2$ , the constant lr performed so bad. If the learning rate is not sufficiently small, the

convergence will not happen, even if the optimizer does not diverge. It will stall.

- Cosine decay shows poor performance as learning rate is decreased. However, it shows great stability in all the learning rates considered.
- Overall, momentum accelerated SGD performs the best considering all three cases.

## H. SGD vs another optimizer

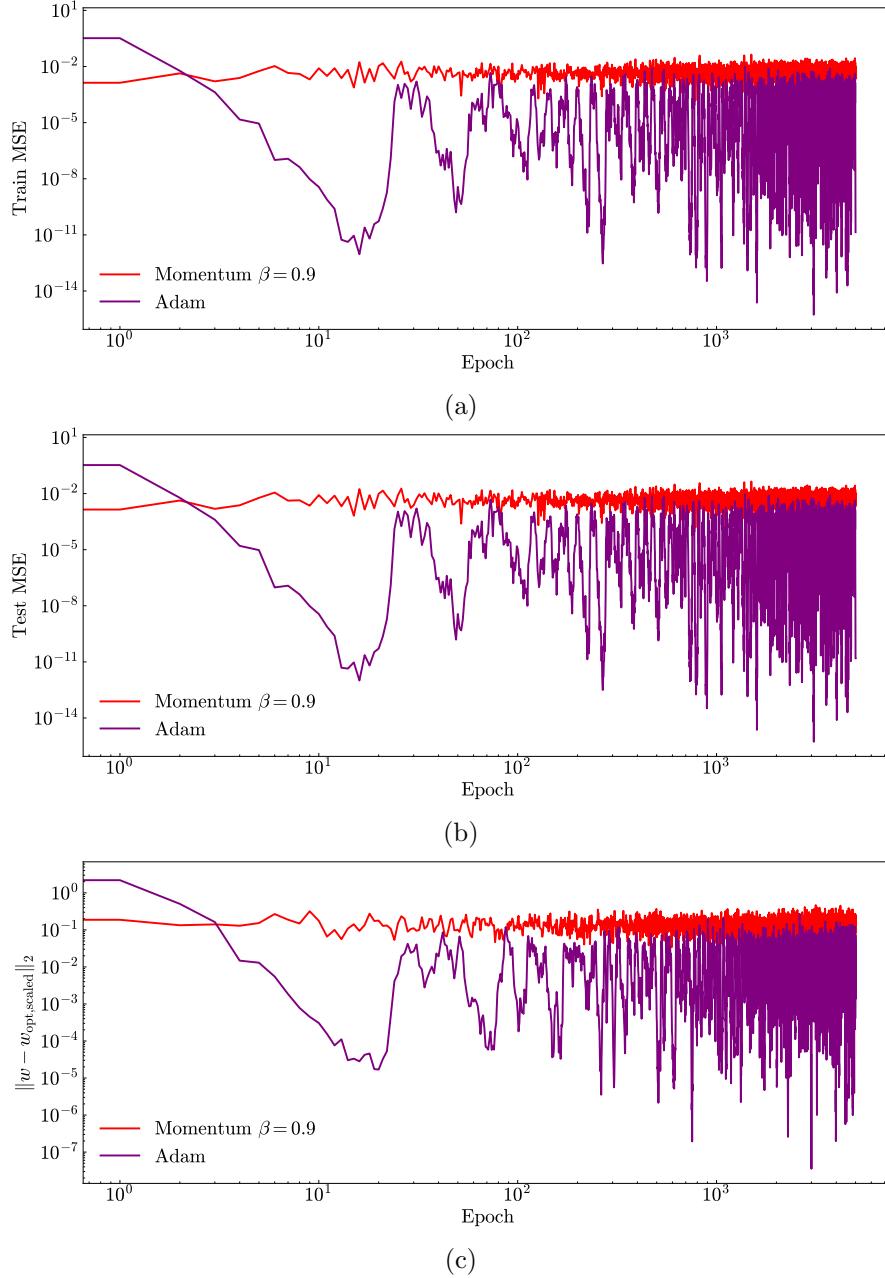


Figure 14: (a) Training  $MSE$ , (b) testing  $MSE$ , and (c)  $l2$  norm of distance from optimizer's solution to closed-form solution for initial learning rate  $\alpha = 0.1$ .

Momentum accelerated SGD is compared with `adam` for the learning rates used in the previous

section. `adam` shows instability in all the three learning rates as can be seen from fig. 14, 15 and

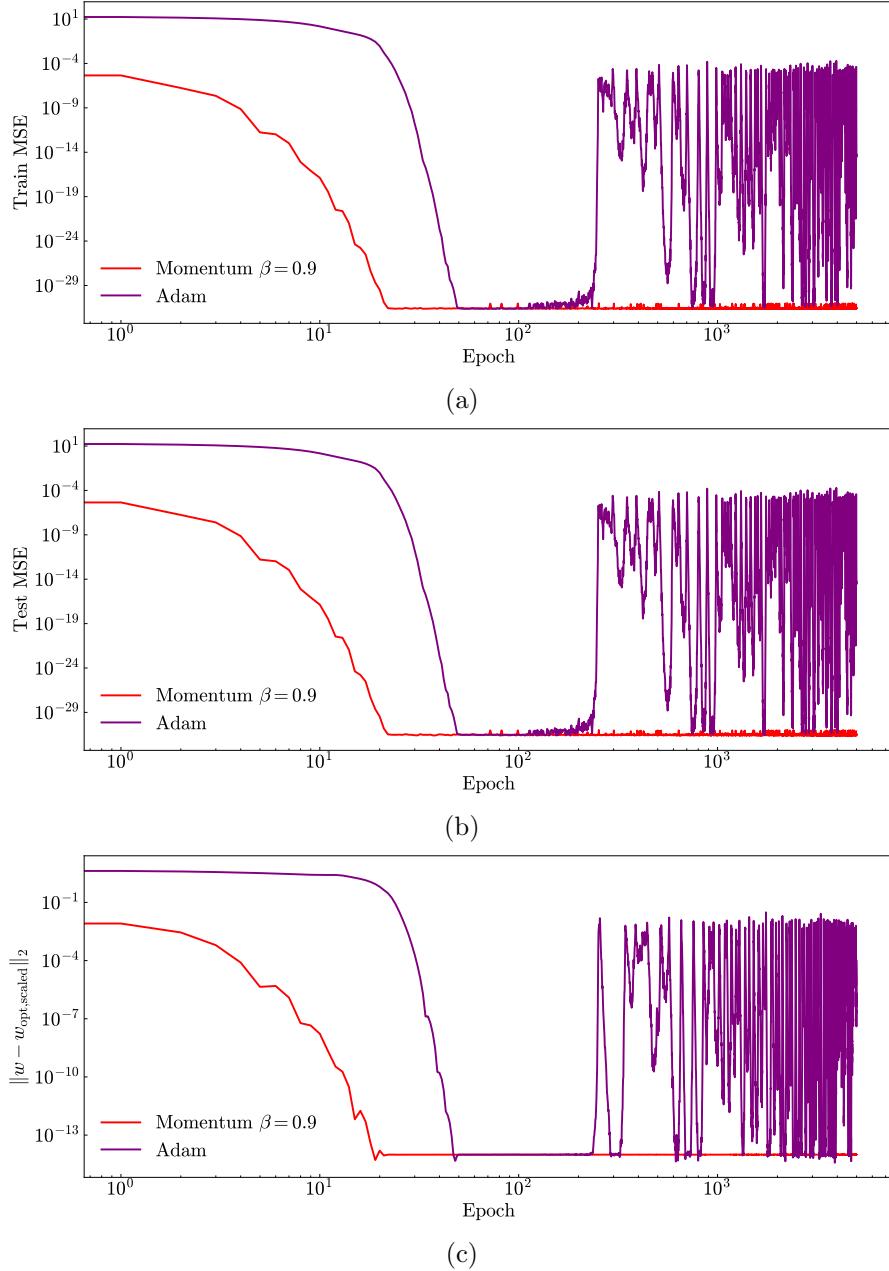


Figure 15: (a) Training  $MSE$ , (b) testing  $MSE$ , and (c)  $l2$  norm of distance from optimizer's solution to closed-form solution for initial learning rate  $\alpha = 0.01$ .

16, while momentum SGD progressively becomes stable with lower learning rate. The performance of momentum SGD also shows improvement with low learning rate. `adam`, despite achieving a very low  $MSE$  that competes well with momentum SGD momentarily, it shows instability by shooting up after the plateau. It is as if it moves back and forth near minima and away from minima after a certain number of epoch. Fig. 15 and 16 particularly show that `adam` starts to face intense instability around around 220th to 240th epoch, which suggests that `adam` has propensity to late training blowup after getting plateaued, despite retaining it for a significant number of epochs.

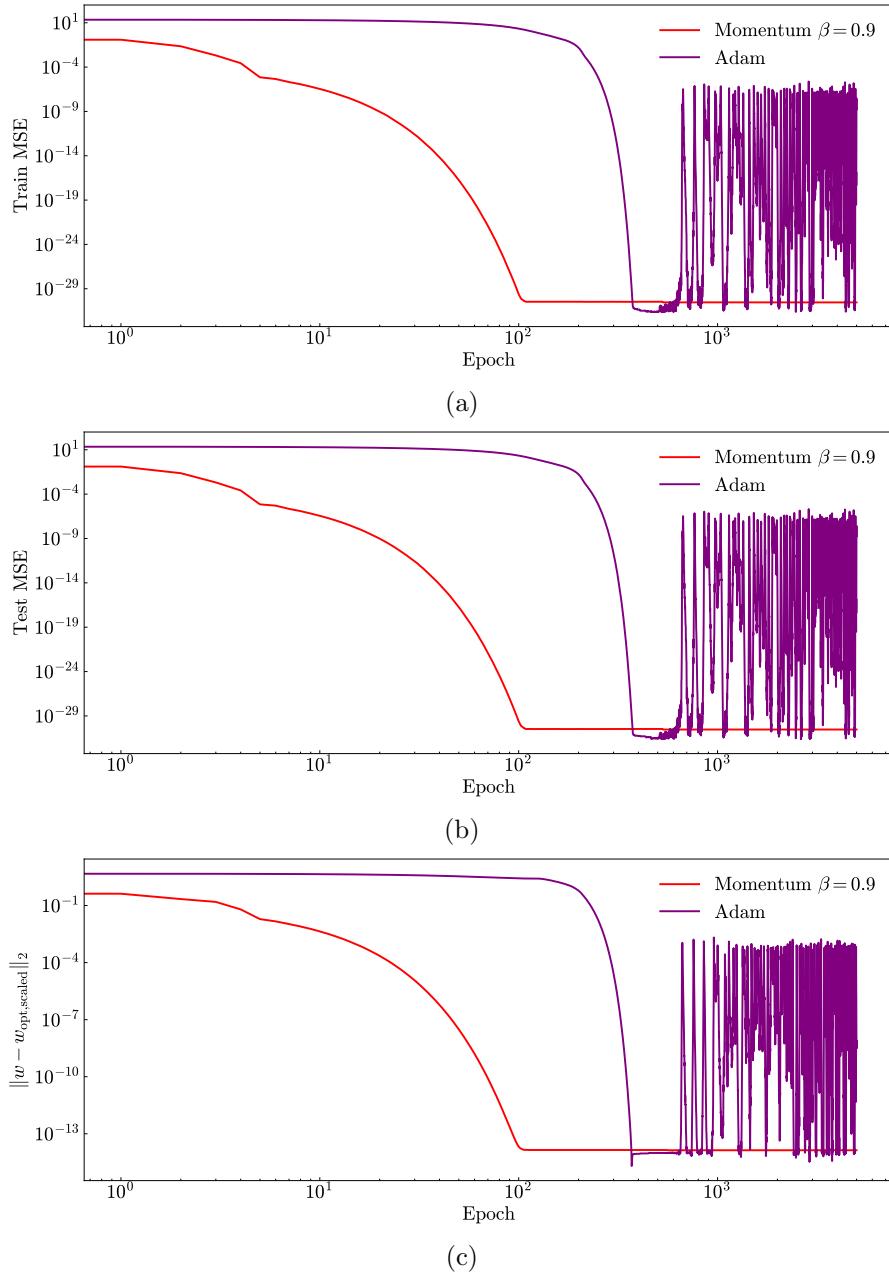


Figure 16: (a) Training  $MSE$ , (b) testing  $MSE$ , and (c)  $l2$  norm of distance from optimizer's solution to closed-form solution for initial learning rate  $\alpha = 0.001$ .

Overall, it is evident that momentum SGD performs better than `adam` for this particular problem.

```

1 import jax
2 jax.config.update("jax_enable_x64", True)
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import matplotlib as mpl
6 from sklearn.model_selection import train_test_split
7 import jax.numpy as jnp
8 import optax
9 import time
10
11 # ===== C. Dataset and setup =====
12
13 def generate_data(scale_noise, N, seed=None):
14     rng = np.random.default_rng(seed)
15     t = np.linspace(0.0, 1.0, N)
16     s = 1 + t + t*t
17     n = rng.normal(loc=0.0, scale=scale_noise, size=N)
18     y = s + n
19     return t, y, s, n
20
21 t_all, y_all, s_all, n_all      = generate_data(0.0, 1000, seed=0)
22 t_train, t_test, y_train, y_test = train_test_split(t_all,y_all,test_size=0.2,
23                                                 random_state=42)
24
25 # parameters for plotting
26 plt.rcParams['font.family'] = 'serif'
27 plt.rcParams['font.serif'] = 'cmr10'
28 plt.rcParams['mathtext.fontset'] = 'cm'
29 plt.rcParams['font.size'] = 20
30 plt.rcParams['axes.unicode_minus'] = False
31 plt.rcParams['axes.formatter.use_mathtext'] = True
32
33 fig, ax = plt.subplots(figsize=(15, 6))
34 ax.scatter(t_train, y_train, color="blue", label="Training data")
35 ax.scatter(t_test, y_test, color="red", label="Testing data")
36 plt.xlabel(r"$t$")
37 plt.ylabel(r"$y$")
38 plt.legend(loc="upper left", frameon=False)
39 plt.tick_params(axis="both", which="both", direction="in")
40 plt.savefig(f"gen_data.pdf", dpi=1080)
41 plt.show()
42
43 # ===== D. Write and test the model and loss functions =====
44 print("\n===== D. Write and test the model and loss functions =====")
45
46 # function for Vandermonde matrix A
47 def vandermonde(t, M):
48     t = jnp.asarray(t)
49     powers = jnp.arange(M + 1)
50     return t[:, None] ** powers[None, :]
51
52
53 # 1. function for model $A \vec{\omega}$
54 def model(w, A):
55     return A @ w
56
57 # 2. function for loss function ($\lambda = 0$)

```

```

58 def loss(w, A, y):
59     y_pred = model(w, A)
60     r = y_pred - y
61     return jnp.dot(r, r)
62
63 # 3. function for MSE
64 def mse(w, A, y):
65     return loss(w, A, y) / len(y)
66
67 # sanity test
68 print("\nSANITY TEST")
69
70 # noise-free data generation
71 N = 1000
72 t, y, s, _ = generate_data(scale_noise=0.0, N=N, seed=1)
73
74 # degree of polynomial and Vandermonde construction
75 M = 2
76 A = vandermonde(t, M)
77
78 # True weights for polynomial
79 w_true = jnp.array([1.0, 1.0, 1.0])
80
81 # wrong weights to check for high loss
82 w_bad = jnp.array([1.0, 0.5, 0.5])
83
84 # compute true losses and MSE
85 loss_true = loss(w_true, A, y)
86 mse_true = mse(w_true, A, y)
87
88 # compute losses and MSE for wrong weights
89 loss_bad = loss(w_bad, A, y)
90 mse_bad = mse(w_bad, A, y)
91
92 print("\nTrue weights: w_true =", w_true)
93 print("Loss(w_true) =", float(loss_true))
94 print("MSE(w_true)   =", float(mse_true))
95
96 print("\nBad weights: w_bad =", w_bad)
97 print("Loss(w_bad)  =", float(loss_bad))
98 print("MSE(w_bad)   =", float(mse_bad))
99
100 # train and test MSE
101 A_train = vandermonde(t_train, M)
102 A_test = vandermonde(t_test, M)
103 train_mse_true = mse(w_true, A_train, y_train)
104 test_mse_true = mse(w_true, A_test, y_test)
105
106 print("\nTraining MSE (true weights) =", float(train_mse_true))
107 print("Testing  MSE (true weights) =", float(test_mse_true))
108
109 # ===== E. SGD optimization =====
110
111 print("\n===== E. SGD optimization =====")
112
113 # 1. Full batch SGD
114
115 M = 3
116 lam = 0

```

```

117 alpha = 0.2
118 epochs = 100000
119 B = len(t_train)
120
121 # construct train and test Vandermonde matrices
122 A_train = vandermonde(t_train, M)
123 A_test = vandermonde(t_test, M)
124 y_train_jnp = jnp.asarray(y_train)
125 y_test_jnp = jnp.asarray(y_test)
126
127 # closed form solution
128 AtA = A_train.T @ A_train
129 Aty = A_train.T @ y_train_jnp
130 w_opt = jnp.linalg.solve(AtA + lam*jnp.eye(M+1), Aty)
131
132 print("\nClosed-form: omega_opt = ", w_opt)
133
134 def vandermonde_scaled(t, M):
135     t = jnp.asarray(t)
136     t = 2*(t - 0.5)          # scale to [-1, 1]
137     powers = jnp.arange(M + 1)
138     return t[:, None] ** powers[None, :]
139
140 # compute Hessian $H = 2A^TA$
141 hessian = 2 * (A_train.T @ A_train)
142
143 # compute the eigenvalues of the Hessian
144 eigen_hessian = np.linalg.eigvals(hessian)
145
146 # maximum eigenvalue of the Hessian
147 max_eigen_hessian = np.max(eigen_hessian)
148
149 # learning rate upper bound
150 alpha_max = 2.0 / max_eigen_hessian
151
152 print("\nMaximum eigenvalue of Hessian and upper bound of learning rate:")
153 print(f"\nlambda_max(H) = {max_eigen_hessian:.3e}")
154 print(f"alpha_max = 2 / lambda_max(H) = {alpha_max:.3e}")
155
156 # SGD diverges or converges?
157 if alpha > alpha_max:
158     print(f"\nSGD should theoretically diverge as alpha = {alpha} is greater than
159           alpha_max = {alpha_max:.3e}")
160     print(f"\nLet's try with scaled Vandermonde...")
161
162 # scaled Vandermonde
163 A_train = vandermonde_scaled(t_train, M)
164 A_test = vandermonde_scaled(t_test, M)
165
166 # compute Hessian $H = 2A^TA$
167 hessian = 2 * (A_train.T @ A_train)
168
169 # compute the eigenvalues of the Hessian
170 eigen_hessian = np.linalg.eigvals(hessian)
171
172 # maximum eigenvalue of the Hessian
173 max_eigen_hessian = np.max(eigen_hessian)
174
175 # learning rate upper bound

```

```

175     alpha_max = 2.0 / max_eigen_hessian
176
177     print("\nAfter scaling...")
178     print(f"\nlambda_max(H) = {max_eigen_hessian:.3e}")
179     print(f"alpha_max = 2 / lambda_max(H) = {alpha_max:.3e}")
180     if alpha > alpha_max:
181         print(f"\nSGD should theoretically still diverge.")
182     else:
183         print(f"\nSGD should theoretically converge")
184     else:
185         print(f"\nSGD should theoretically converge as alpha = {alpha} is less than or
186             equal to alpha_max = {alpha_max:.3e}.")
187
188 # initialization of weights
189 key = jax.random.PRNGKey(0)
190 w = jax.random.normal(key, (M+1,))
191
192 # Optax SGD with gradient clipping
193 #optimizer = optax.sgd(learning_rate=alpha)
194
195 optimizer = optax.chain(
196     optax.clip(0.5), # Lower clipping threshold for better gradient stability
197     optax.sgd(learning_rate=alpha)
198 )
199
200 opt_state = optimizer.init(w)
201
202 # update step
203 @jax.jit
204 def step(w, opt_state, A, y):
205     grads = jax.grad(loss)(w, A, y)
206     updates, opt_state = optimizer.update(grads, opt_state)
207     w = optax.apply_updates(w, updates)
208     return w, opt_state
209
210 # initialize train, test, distance arrays
211 train_mse_hist, test_mse_hist, dist_hist = [], [], []
212
213 # full batch training loop
214 for epoch in range(epochs):
215     w, opt_state = step(w, opt_state, A_train, y_train_jnp)
216
217     # compute real losses
218     train_mse_hist.append(mse(w, A_train, y_train_jnp))
219     test_mse_hist.append(mse(w, A_test, y_test_jnp))
220     dist_hist.append(jnp.linalg.norm(w - w_opt))
221
222 print("\nFinal SGD weights:", w)
223 print("Distance to closed-form =", float(dist_hist[-1]))
224
225 # parameters for plotting
226 plt.rcParams['font.family'] = 'serif'
227 plt.rcParams['font.serif'] = 'cmr10'
228 plt.rcParams['mathtext.fontset'] = 'cm'
229 plt.rcParams['font.size'] = 20
230 mpl.rcParams['axes.unicode_minus'] = False
231 plt.rcParams['axes.formatter.use_mathtext'] = True
232
233 # plot for training & testing MSE vs. epoch

```

```

233 fig, ax = plt.subplots(figsize=(15, 6))
234 ax.loglog(train_mse_hist, "b-", lw=2, label="Train MSE")
235 ax.loglog(test_mse_hist, "r--", lw=2, label="Test MSE")
236 plt.xlabel("Epoch")
237 plt.ylabel("MSE")
238 plt.legend(loc="upper right", frameon=False)
239 plt.tick_params(axis="both", which="both", direction="in")
240 plt.savefig(f"mse_wclip.pdf", dpi=1080)
241 plt.show()
242
243 # plot L2 norm of error vs. epoch
244 fig, ax = plt.subplots(figsize=(15, 6))
245 ax.loglog(dist_hist, "k-", lw=2)
246 plt.xlabel("Epoch")
247 plt.ylabel(r"$\|\omega_i - \omega_{\text{opt}}\|_2$")
248 #plt.legend(loc="upper left", frameon=False)
249 plt.tick_params(axis="both", which="both", direction="in")
250 plt.savefig(f"l2_wclip.pdf", dpi=1080)
251 plt.show()
252
253 # now try with scaled data
254
255 # scaled Vandermonde
256 A_train_s = vandermonde_scaled(t_train, M)
257 A_test_s = vandermonde_scaled(t_test, M)
258
259 # closed form solution in scaled basis
260 AtA_s = A_train_s.T @ A_train_s
261 Aty_s = A_train_s.T @ y_train_jnp
262 w_opt_s = jnp.linalg.solve(AtA_s + lam*jnp.eye(M+1), Aty_s)
263
264 print("\nClosed-form solution (scaled basis):", w_opt_s)
265
266 # new random weights
267 key2 = jax.random.PRNGKey(123)
268 w2 = jax.random.normal(key2, (M+1,))
269
270 # optimizer with clipping
271 optimizer2 = optax.chain(
272     optax.clip(0.5), # Lower clipping threshold for better gradient stability
273     optax.sgd(learning_rate=alpha)
274 )
275 opt_state2 = optimizer2.init(w2)
276
277 # new step function for scaled run
278 @jax.jit
279 def step2(w, opt_state, A, y):
280     grads = jax.grad(loss)(w, A, y)
281     updates, opt_state = optimizer2.update(grads, opt_state)
282     w = optax.apply_updates(w, updates)
283     return w, opt_state
284
285 # histories
286 train_mse_hist_2 = []
287 test_mse_hist_2 = []
288 dist_hist_2 = []
289
290 # training loop
291 for epoch in range(epochs):

```

```

292     w2, opt_state2 = step2(w2, opt_state2, A_train_s, y_train_jnp)
293
294     train_mse_hist_2.append(mse(w2, A_train_s, y_train_jnp))
295     test_mse_hist_2.append(mse(w2, A_test_s, y_test_jnp))
296     dist_hist_2.append(jnp.linalg.norm(w2 - w_opt_s))
297
298 print("\nFinal SGD weights (scaled):", w2)
299 print("Distance to scaled closed-form:", float(dist_hist_2[-1]))
300
301 # plot scaled training & testing MSE vs. epoch
302 fig, ax = plt.subplots(figsize=(15, 6))
303 ax.loglog(train_mse_hist_2, "b-", lw=2, label="Train MSE (scaled)")
304 ax.loglog(test_mse_hist_2, "r-", lw=2, label="Test MSE (scaled)")
305 plt.xlabel("Epoch")
306 plt.ylabel("MSE")
307 plt.legend(loc="upper right", frameon=False)
308 plt.savefig("mse_scaled.pdf", dpi=1080)
309 plt.show()
310
311 # plot scaled L2 norm of error vs. epoch
312 fig, ax = plt.subplots(figsize=(15, 6))
313 ax.loglog(dist_hist_2, "k-", lw=2)
314 plt.xlabel("Epoch")
315 plt.ylabel(r"$\|w_i - w_{opt,scaled}\|_2$")
316 plt.savefig("l2_scaled.pdf", dpi=1080)
317 plt.show()
318
319 # 2. Mini-batch SGD
320
321 # unscaled mini-batch
322 epochs_mb = 2000
323 batch_size = 32
324 num_batches = len(t_train)
325
326 # shuffle indices each epoch
327 def get_batches(A, y, batch_size):
328     N = len(y)
329     perm = np.random.permutation(N)
330     for i in range(0, N, batch_size):
331         idx = perm[i:i+batch_size]
332         yield A[idx], y[idx]
333
334 # fresh weights
335 key = jax.random.PRNGKey(420)
336 w_mb = jax.random.normal(key, (M+1,))
337
338 # optimizer
339 optimizer_mb = optax.chain(
340     optax.clip(0.5), # Lower clipping threshold for better gradient stability
341     optax.sgd(learning_rate=alpha)
342 )
343 opt_state_mb = optimizer_mb.init(w_mb)
344
345 # mini-batch training step
346 @jax.jit
347 def mb_step(w, opt_state, A_batch, y_batch):
348     grads = jax.grad(loss)(w, A_batch, y_batch)
349     updates, opt_state = optimizer_mb.update(grads, opt_state)
350     w = optax.apply_updates(w, updates)

```

```

351     return w, opt_state
352
353 # history arrays
354 train_mse_hist_mb = []
355 test_mse_hist_mb = []
356 dist_hist_mb = []
357
358 # mini-batch SGD loop
359 for epoch in range(epochs_mb):
360     # iterate over randomized batches
361     for A_b, y_b in get_batches(A_train, y_train_jnp, batch_size):
362         w_mb, opt_state_mb = mb_step(w_mb, opt_state_mb, A_b, y_b)
363     # record metrics each epoch
364     train_mse_hist_mb.append(mse(w_mb, A_train, y_train_jnp))
365     test_mse_hist_mb.append(mse(w_mb, A_test, y_test_jnp))
366     dist_hist_mb.append(jnp.linalg.norm(w_mb - w_opt))
367     if epoch % 100 == 0:
368         print(f"Epoch {epoch}: Train MSE = {train_mse_hist_mb[-1]:.6f}, Dist = {dist_hist_mb[-1]:.3e}")
369
370 print("\nFinal mini-batch SGD weights:", w_mb)
371 print("Distance to closed-form =", float(dist_hist_mb[-1]))
372
373 # plots
374 fig, ax = plt.subplots(figsize=(15, 6))
375 ax.loglog(train_mse_hist_mb, "b-", lw=2, label="Train MSE (mini-batch)")
376 ax.loglog(test_mse_hist_mb, "r--", lw=2, label="Test MSE (mini-batch)")
377 plt.xlabel("Epoch")
378 plt.ylabel("MSE")
379 plt.legend(loc="upper right", frameon=False)
380 plt.tick_params(axis="both", which="both", direction="in")
381 plt.savefig("mse_mb_unscaled.pdf", dpi=1080)
382 plt.show()
383
384 fig, ax = plt.subplots(figsize=(15, 6))
385 ax.loglog(dist_hist_mb, "k-", lw=2)
386 plt.xlabel("Epoch")
387 plt.ylabel(r"$\|w - w_{\text{opt}}\|_2$")
388 plt.tick_params(axis="both", which="both", direction="in")
389 plt.savefig("l2_mb_unscaled.pdf", dpi=1080)
390 plt.show()
391
392 # scaled mini-batch
393 epochs_mb_s = 2000
394 batch_size_s = 32
395
396 # build scaled Vandermonde
397 A_train_s = vandermonde_scaled(t_train, M)
398 A_test_s = vandermonde_scaled(t_test, M)
399
400 # closed form solution in scaled basis
401 AtA_s = A_train_s.T @ A_train_s
402 Aty_s = A_train_s.T @ y_train_jnp
403 w_opt_s = jnp.linalg.solve(AtA_s + lam*jnp.eye(M+1), Aty_s)
404
405 print("\nClosed-form solution (scaled basis) for mini-batch:", w_opt_s)
406
407 # fresh weights for scaled mini-batch
408 key_s = jax.random.PRNGKey(777)

```

```

409 w_mb_s = jax.random.normal(key_s, (M+1,))
410
411 # optimizer (same learning rate and clipping)
412 optimizer_mb_s = optax.chain(
413     optax.clip(0.5),
414     optax.sgd(learning_rate=alpha)
415 )
416 opt_state_mb_s = optimizer_mb_s.init(w_mb_s)
417
418 # batching function for scaled data
419 def get_batches_s(A, y, batch_size):
420     N = len(y)
421     perm = np.random.permutation(N)
422     for i in range(0, N, batch_size):
423         idx = perm[i:i+batch_size]
424         yield A[idx], y[idx]
425
426 # JIT-compiled update for scaled mini-batch
427 @jax.jit
428 def mb_step_s(w, opt_state, A_batch, y_batch):
429     grads = jax.grad(loss)(w, A_batch, y_batch)
430     updates, opt_state = optimizer_mb_s.update(grads, opt_state)
431     w = optax.apply_updates(w, updates)
432     return w, opt_state
433
434 # histories
435 train_mse_hist_mb_s = []
436 test_mse_hist_mb_s = []
437 dist_hist_mb_s = []
438
439 # training loop
440 for epoch in range(epochs_mb_s):
441     for A_b, y_b in get_batches_s(A_train_s, y_train_jnp, batch_size_s):
442         w_mb_s, opt_state_mb_s = mb_step_s(w_mb_s, opt_state_mb_s, A_b, y_b)
443
444     # record per-epoch progress
445     train_mse_hist_mb_s.append(mse(w_mb_s, A_train_s, y_train_jnp))
446     test_mse_hist_mb_s.append(mse(w_mb_s, A_test_s, y_test_jnp))
447     dist_hist_mb_s.append(jnp.linalg.norm(w_mb_s - w_opt_s))
448
449     if epoch % 100 == 0:
450         print(f"[SCALED] Epoch {epoch}: Train MSE = {train_mse_hist_mb_s[-1]:.6f},
451               Dist = {dist_hist_mb_s[-1]:.3e}")
452
453 print("\nFinal mini-batch SGD weights (scaled):", w_mb_s)
454 print("Distance to scaled closed-form =", float(dist_hist_mb_s[-1]))
455
456 # plots
457 fig, ax = plt.subplots(figsize=(15, 6))
458 ax.loglog(train_mse_hist_mb_s, "b-", lw=2, label="Train MSE (scaled mini-batch)")
459 ax.loglog(test_mse_hist_mb_s, "r--", lw=2, label="Test MSE (scaled mini-batch)")
460 plt.xlabel("Epoch")
461 plt.ylabel("MSE")
462 plt.legend(loc="upper right", frameon=False)
463 plt.tick_params(axis="both", which="both", direction="in")
464 plt.savefig("mse_mb_scaled.pdf", dpi=1080)
465 plt.show()
466
467 fig, ax = plt.subplots(figsize=(15, 6))

```

```

467 ax.loglog(dist_hist_mb_s, "k-", lw=2)
468 plt.xlabel("Epoch")
469 plt.ylabel(r"$\| w - w_{\text{opt}, \text{scaled}} \|_2$")
470 plt.tick_params(axis="both", which="both", direction="in")
471 plt.savefig("l2_mb_scaled.pdf", dpi=1080)
472 plt.show()
473
474 # F. Batch size study
475
476 print("\n===== F. Batch size study =====")
477
478 # parameters
479 batch_sizes = [1, 16, 64, len(t_train)]
480 epochs_F = 100000 # maximum epoch count
481 alpha_F = 0.1
482 M = 3
483 target_mse = 1e-4 # target
484
485 results = {}
486
487 # early stopping parameters
488 patience = 5 # no. of epochs with no significant improvement before stopping
489 min_delta = 1e-5 # minimum change in MSE to be considered as improvement
490
491 # define the mini-batch step function
492 @jax.jit
493 def mb_step_F(w, opt_state, A_batch, y_batch):
494     grads = jax.grad(loss)(w, A_batch, y_batch)
495     updates, opt_state = optimizer_F.update(grads, opt_state)
496     w = optax.apply_updates(w, updates)
497     return w, opt_state
498
499 # define the batch creation function
500 def get_batches_F(A, y, batch_size):
501     N = len(y)
502     perm = np.random.permutation(N)
503     for i in range(0, N, batch_size):
504         idx = perm[i:i+batch_size]
505         yield A[idx], y[idx]
506
507 # pre-calculated scaled data
508 A_train_scaled = A_train_s
509 A_test_scaled = A_test_s
510
511 for B in batch_sizes:
512     print(f"\n--- Running SGD for batch size B = {B} ---")
513
514     # fresh weights
515     key_F = jax.random.PRNGKey(999 + B)
516     w_F = jax.random.normal(key_F, (M + 1,))
517
518     # optimizer with gradient clipping
519     optimizer_F = optax.chain(
520         optax.clip(0.5), # clip gradients to prevent explosion
521         optax.sgd(learning_rate=alpha_F)
522     )
523     opt_state_F = optimizer_F.init(w_F)
524
525     # histories

```

```

526 train_hist, test_hist = [], []
527 t0 = time.time()
528 reached_time = None
529 previous_mse = float('inf') # initialize previous MSE as a large value
530
531 # First, ensure training runs for at least 100 epochs
532 for epoch in range(100): # first mandatory 100 epochs
533     # process batches using scaled data
534     for A_b, y_b in get_batches_F(A_train_scaled, y_train_jnp, B):
535         w_F, opt_state_F = mb_step_F(w_F, opt_state_F, A_b, y_b)
536
537     # record metrics
538     train_mse_val = mse(w_F, A_train_scaled, y_train_jnp)
539     test_mse_val = mse(w_F, A_test_scaled, y_test_jnp)
540     train_hist.append(train_mse_val)
541     test_hist.append(test_mse_val)
542
543     # time to target test MSE
544     if reached_time is None and float(test_mse_val) < target_mse:
545         reached_time = time.time() - t0
546
547     if epoch % 100 == 0:
548         print(f"Epoch {epoch}: test MSE = {float(test_mse_val):.3e}")
549
550 # after the first 100 epochs, check for target MSE and early stopping
551 for epoch in range(100, epochs_F):
552     # process batches using scaled data
553     for A_b, y_b in get_batches_F(A_train_scaled, y_train_jnp, B):
554         w_F, opt_state_F = mb_step_F(w_F, opt_state_F, A_b, y_b)
555
556     # record metrics
557     train_mse_val = mse(w_F, A_train_scaled, y_train_jnp)
558     test_mse_val = mse(w_F, A_test_scaled, y_test_jnp)
559     train_hist.append(train_mse_val)
560     test_hist.append(test_mse_val)
561
562     # time to target test MSE
563     if reached_time is None and float(test_mse_val) < target_mse:
564         reached_time = time.time() - t0
565
566     if epoch % 100 == 0:
567         print(f"Epoch {epoch}: test MSE = {float(test_mse_val):.3e}")
568
569     # Check if target MSE is reached
570     if test_mse_val <= target_mse:
571         print(f"Target MSE reached at epoch {epoch}. Stopping training.")
572         break
573
574     # Early stopping criterion after running for 100 epochs
575     if abs(previous_mse - test_mse_val) < min_delta:
576         patience -= 1
577         if patience == 0:
578             print(f"Early stopping at epoch {epoch} due to lack of improvement")
579             break
580     else:
581         patience = 5 # reset patience if there was significant improvement
582
583     previous_mse = test_mse_val # update previous MSE for the next epoch

```

```

584     total_time = time.time() - t0
585
586     results[B] = {
587         "train_hist": train_hist,
588         "test_hist": test_hist,
589         "time_to_target": reached_time,
590         "final_test_mse": float(test_hist[-1]),
591         "total_time": total_time
592     }
593
594
595     print(f"Final test MSE = {results[B]['final_test_mse']:.3e}")
596     print(f"Time to reach target: {results[B]['time_to_target']} ")
597     print(f"Total time = {total_time:.2f} sec")
598
599 # plot train and test MSE
600
601 for B in batch_sizes:
602     fig, ax = plt.subplots(figsize=(15, 6))
603     ax.loglog(results[B]["train_hist"], lw=2, color='blue', label=f"Train B={B}")
604     ax.loglog(results[B]["test_hist"], lw=2, linestyle="--", color='red', label=f"Test B={B}")
605
606     plt.xlabel("Epoch")
607     plt.ylabel("MSE")
608     plt.legend(frameon=False)
609     plt.tick_params(axis="both", which="both", direction="in")
610     plt.savefig(f"batch_{B}_mse.pdf", dpi=1080)
611     plt.show()
612
613 # ===== G. Learning-rate study =====
614
615 print("\n===== G. Learning-rate study =====")
616
617 # parameters
618 b = 32
619 epoch_max = 5000
620 learning_rates = [0.1, 0.01, 0.001] # list of learning rates
621 target_mse = 1e-6
622 #min_delta = 1e-4
623 #patience_init = 5
624 #min_epochs = 1000
625
626 # reuse scaled vandermonde
627 a_train_scaled = A_train_s
628 a_test_scaled = A_test_s
629
630 # closed-form solution in scaled basis
631 ata_s = a_train_scaled.T @ a_train_scaled
632 aty_s = a_train_scaled.T @ y_train_jnp
633 w_opt_s = jnp.linalg.solve(ata_s + lam*jnp.eye(M+1), aty_s)
634
635 # function to create mini-batches
636 def get_batches_g():
637     n = len(y_train_jnp)
638     perm = np.random.permutation(n)
639     for i in range(0, n, b):
640         idx = perm[i:i+b]
641         yield a_train_scaled[idx], y_train_jnp[idx]

```

```

642
643 # general training loop for any optimizer
644 def run_optimizer(opt, label, alpha0):
645     key = jax.random.PRNGKey(999 + hash(label) % 100)
646     w = jax.random.normal(key, (M+1,))
647     opt_state = opt.init(w)
648
649     train_hist = []
650     test_hist = []
651     dist_hist = []
652
653     reached_time = None
654     t0 = time.time()
655
656     @jax.jit
657     def step(w, opt_state, A_b, y_b):
658         grads = jax.grad(loss)(w, A_b, y_b)
659         updates, opt_state = opt.update(grads, opt_state, w)
660         w = optax.apply_updates(w, updates)
661         return w, opt_state
662
663     for epoch in range(epoch_max):
664
665         # update using mini-batches
666         for A_b, y_b in get_batches_g():
667             w, opt_state = step(w, opt_state, A_b, y_b)
668
669         # compute metrics
670         train_mse = float(mse(w, a_train_scaled, y_train_jnp))
671         test_mse = float(mse(w, a_test_scaled, y_test_jnp))
672         dist_w = float(jnp.linalg.norm(w - w_opt_s))
673
674         train_hist.append(train_mse)
675         test_hist.append(test_mse)
676         dist_hist.append(dist_w)
677
678         # record time to reach target mse
679         if reached_time is None and test_mse < target_mse:
680             reached_time = time.time() - t0
681
682         # periodic debug output
683         if epoch % 1000 == 0:
684             print(f"{label}: epoch {epoch}, test mse = {test_mse:.3e}")
685
686             print(f"{label}: final mse = {train_mse:.3e}, dist = {dist_w:.3e}")
687     return train_hist, test_hist, dist_hist
688
689 # define optimizers
690 def get_optimizer(alpha0):
691     opt_const = optax.chain(
692         optax.clip(0.5),
693         optax.sgd(learning_rate=alpha0)
694     )
695
696     opt_momentum = optax.chain(
697         optax.clip(0.5),
698         optax.trace(decay=0.9, nesterov=True),
699         optax.scale(-alpha0)
700     )

```

```

701     schedule_cos = optax.cosine_decay_schedule(
702         init_value=alpha0,
703         decay_steps=epoch_max
704     )
705     opt_cosine = optax.chain(
706         optax.clip(0.5),
707         optax.sgd(learning_rate=schedule_cos)
708     )
709
710
711     return opt_const, opt_momentum, opt_cosine
712
713 # loop over all learning rates
714 for lr in learning_rates:
715     print(f"\nRunning with learning rate = {lr}")
716     # define the optimizers for the current learning rate
717     opt_const, opt_momentum, opt_cosine = get_optimizer(lr)
718
719     # run all optimizers
720     train_const, test_const, dist_const = run_optimizer(opt_const, "constant lr",
721             , lr)
721     train_mom, test_mom, dist_mom = run_optimizer(opt_momentum, "momentum
722         beta=0.9", lr)
722     train_cos, test_cos, dist_cos = run_optimizer(opt_cosine, "cosine decay
723         ", lr)
724
725     # plot train mse vs epoch
726     fig, ax = plt.subplots(figsize=(15, 6))
727
727     ax.loglog(train_const, color="blue", lw=2, label="Constant lr")
728     ax.loglog(train_mom, color="red", lw=2, label=rf"Momentum $\beta=0.9$")
729     ax.loglog(train_cos, color="green", lw=2, label="Cosine decay")
730
731     plt.xlabel("Epoch")
732     plt.ylabel("Train MSE")
733     plt.legend(frameon=False)
734     plt.tick_params(axis="both", which="both", direction="in")
735     plt.savefig(f"optimizer_comp_mse_lr{lr}.pdf", dpi=1080)
736     plt.show()
737
738     # plot test mse vs epoch
739     fig, ax = plt.subplots(figsize=(15, 6))
740
741     ax.loglog(test_const, color="blue", lw=2, label="Constant lr")
742     ax.loglog(test_mom, color="red", lw=2, label=rf"Momentum $\beta=0.9$")
743     ax.loglog(test_cos, color="green", lw=2, label="Cosine decay")
744
745     plt.xlabel("Epoch")
746     plt.ylabel("Test MSE")
747     plt.legend(frameon=False)
748     plt.tick_params(axis="both", which="both", direction="in")
749     plt.savefig(f"optimizer_comp_test_mse_lr{lr}.pdf", dpi=1080)
750     plt.show()
751
752     # plot l2 norm vs epoch
753     fig, ax = plt.subplots(figsize=(15, 6))
754
755     ax.loglog(dist_const, color="blue", lw=2, label="constant lr")
756     ax.loglog(dist_mom, color="red", lw=2, label=rf"momentum $\beta=0.9$")

```

```

757     ax.loglog(dist_cos,    color="green", lw=2, label="cosine decay")
758
759     plt.xlabel("Epoch")
760     plt.ylabel(r"$\| w - w_{\text{opt}, \text{scaled}} \|_2$")
761     plt.legend(frameon=False)
762     plt.tick_params(axis="both", which="both", direction="in")
763     plt.savefig(f"optimizer_comp_lr{lr}.pdf", dpi=1080)
764     plt.show()
765
766 # ===== H. SGD vs another optimizer =====
767
768 print("\n===== H. SGD vs another optimizer =====")
769
770 # parameters for this comparison
771 b = 32
772 epoch_max = 5000
773 learning_rates = [0.1, 0.01, 0.001] # list of learning rates
774 target_mse = 1e-6
775
776 # optimizers (momentum, adam)
777 def get_optimizer_momentum(alpha0):
778     return optax.chain(
779         optax.clip(0.5),
780         optax.trace(decay=0.9, nesterov=True),
781         optax.scale(-alpha0)
782     )
783
784 def get_optimizer_adam(alpha0):
785     return optax.chain(
786         optax.clip(0.5),
787         optax.adam(learning_rate=alpha0)
788     )
789
790 # function to create mini-batches
791 def get_batches_g(A, y, batch_size):
792     n = len(y)
793     perm = np.random.permutation(n)
794     for i in range(0, n, batch_size):
795         idx = perm[i:i+batch_size]
796         yield A[idx], y[idx]
797
798 # general training loop for any optimizer
799 def run_optimizer(opt, label, alpha0, A_train_scaled, A_test_scaled, y_train_jnp,
800                   y_test_jnp):
801     key = jax.random.PRNGKey(999 + hash(label) % 100)
802     w = jax.random.normal(key, (M+1,))
803     opt_state = opt.init(w)
804
805     train_hist = []
806     test_hist = []
807     dist_hist = []
808
809     reached_time = None
810     t0 = time.time()
811
812     @jax.jit
813     def step(w, opt_state, A_b, y_b):
814         grads = jax.grad(loss)(w, A_b, y_b)
815         updates, opt_state = opt.update(grads, opt_state, w)

```

```

815     w = optax.apply_updates(w, updates)
816     return w, opt_state
817
818 for epoch in range(epoch_max):
819     # update using mini-batches
820     for A_b, y_b in get_batches_g(A_train_scaled, y_train_jnp, b):
821         w, opt_state = step(w, opt_state, A_b, y_b)
822
823     # compute metrics
824     train_mse = float(mse(w, A_train_scaled, y_train_jnp))
825     test_mse = float(mse(w, A_test_scaled, y_test_jnp))
826     dist_w = float(jnp.linalg.norm(w - w_opt_s))
827
828     train_hist.append(train_mse)
829     test_hist.append(test_mse)
830     dist_hist.append(dist_w)
831
832     # record time to reach target mse
833     if reached_time is None and test_mse < target_mse:
834         reached_time = time.time() - t0
835
836     # periodic debug output
837     if epoch % 1000 == 0:
838         print(f"{label}: epoch {epoch}, test mse = {test_mse:.3e}")
839
840     print(f"{label}: final mse = {train_mse:.3e}, dist = {dist_w:.3e}")
841 return train_hist, test_hist, dist_hist
842
843 # loop over all learning rates for comparison of Momentum and Adam
844 for lr in learning_rates:
845     print(f"\nRunning with learning rate = {lr}")
846
847     # get optimizers for Momentum and Adam
848     opt_momentum = get_optimizer_momentum(lr)
849     opt_adam = get_optimizer_adam(lr)
850
851     # run
852     train_mom, test_mom, dist_mom = run_optimizer(opt_momentum, "Momentum beta=0.9",
853                                                 lr, A_train_s, A_test_s, y_train_jnp, y_test_jnp)
854     train_adam, test_adam, dist_adam = run_optimizer(opt_adam, "Adam optimizer",
855                                                 lr, A_train_s, A_test_s, y_train_jnp, y_test_jnp)
856
857     # plot train mse vs epoch
858     fig, ax = plt.subplots(figsize=(15, 6))
859     ax.loglog(train_mom, color="red", lw=2, label=rf"Momentum $\beta=0.9$")
860     ax.loglog(train_adam, color="purple", lw=2, label="Adam")
861     plt.xlabel("Epoch")
862     plt.ylabel("Train MSE")
863     plt.legend(frameon=False)
864     plt.tick_params(axis="both", which="both", direction="in")
865     plt.savefig(f"train_optimizer_comp_mse_lr{lr}_momentum_adam.pdf", dpi=1080)
866     plt.show()
867
868     # Pplot test mse vs epoch
869     fig, ax = plt.subplots(figsize=(15, 6))
870     ax.loglog(test_mom, color="red", lw=2, label=rf"Momentum $\beta=0.9$")
871     ax.loglog(test_adam, color="purple", lw=2, label="Adam")
872     plt.xlabel("Epoch")
873     plt.ylabel("Test MSE")

```

```

872 plt.legend(frameon=False)
873 plt.tick_params(axis="both", which="both", direction="in")
874 plt.savefig(f"test_optimizer_comp_mse_lr{lr}_momentum_adam.pdf", dpi=1080)
875 plt.show()
876
877 # plot l2 norm vs epoch
878 fig, ax = plt.subplots(figsize=(15, 6))
879 ax.loglog(dist_mom, color="red", lw=2, label=rf"Momentum $\beta=0.9$")
880 ax.loglog(dist_adam, color="purple", lw=2, label="Adam")
881 plt.xlabel("Epoch")
882 plt.ylabel(r"$\| w - w_{opt,scaled} \|_2$")
883 plt.legend(frameon=False)
884 plt.tick_params(axis="both", which="both", direction="in")
885 plt.savefig(f"norm_optimizer_comp_lr{lr}_momentum_adam.pdf", dpi=1080)
886 plt.show()

```

Listing 1: sgd.py

```

1 ===== D. Write and test the model and loss functions =====
2
3 SANITY TEST
4
5 True weights: w_true = [1. 1. 1.]
6 Loss(w_true) = 0.0
7 MSE(w_true) = 0.0
8
9 Bad weights: w_bad = [1. 0.5 0.5]
10 Loss(w_bad) = 258.57525025024205
11 MSE(w_bad) = 0.25857525025024203
12
13 Training MSE (true weights) = 0.0
14 Testing MSE (true weights) = 0.0
15
16 ===== E. SGD optimization =====
17
18 Closed-form: omega_opt = [1.0000000e+00 1.0000000e+00 1.0000000e+00 1.94831852
   e-13]
19
20 Maximum eigenvalue of Hessian and upper bound of learning rate:
21
22 lambda_max(H) = 2.394e+03
23 alpha_max = 2 / lambda_max(H) = 8.355e-04
24
25 SGD should theoretically diverge as alpha = 0.2 is greater than alpha_max = 8.355e
   -04
26
27 Let's try with scaled Vandermonde...
28
29 After scaling...
30
31 lambda_max(H) = 1.796e+03
32 alpha_max = 2 / lambda_max(H) = 1.114e-03
33
34 SGD should theoretically still diverge.
35
36 Final SGD weights: [1.59415786 0.41523422 0.41608667 0.78784401]
37 Distance to closed-form = 1.2871004385864873
38

```

```

39 | Closed-form solution (scaled basis): [1.75000000e+00 1.00000000e+00 2.50000000e-01
8.39450872e-15]
40 |
41 | Final SGD weights (scaled): [1.64692162 0.76950196 0.65296745 0.50743949]
42 | Distance to scaled closed-form: 0.6954366236666928
43 |
44 | Epoch 0: Train MSE = 0.074629, Dist = 5.416e-01
45 | Epoch 100: Train MSE = 0.012554, Dist = 1.045e+00
46 | Epoch 200: Train MSE = 0.005838, Dist = 1.037e+00
47 | Epoch 300: Train MSE = 0.008159, Dist = 1.031e+00
48 | Epoch 400: Train MSE = 0.005859, Dist = 1.031e+00
49 | Epoch 500: Train MSE = 0.005436, Dist = 1.018e+00
50 | Epoch 600: Train MSE = 0.008189, Dist = 1.029e+00
51 | Epoch 700: Train MSE = 0.005348, Dist = 1.013e+00
52 | Epoch 800: Train MSE = 0.009653, Dist = 1.051e+00
53 | Epoch 900: Train MSE = 0.007712, Dist = 1.035e+00
54 | Epoch 1000: Train MSE = 0.009104, Dist = 1.057e+00
55 | Epoch 1100: Train MSE = 0.008422, Dist = 1.034e+00
56 | Epoch 1200: Train MSE = 0.005371, Dist = 1.028e+00
57 | Epoch 1300: Train MSE = 0.006227, Dist = 1.011e+00
58 | Epoch 1400: Train MSE = 0.009272, Dist = 1.037e+00
59 | Epoch 1500: Train MSE = 0.008010, Dist = 1.033e+00
60 | Epoch 1600: Train MSE = 0.009683, Dist = 1.039e+00
61 | Epoch 1700: Train MSE = 0.007722, Dist = 1.042e+00
62 | Epoch 1800: Train MSE = 0.006759, Dist = 1.031e+00
63 | Epoch 1900: Train MSE = 0.004825, Dist = 1.021e+00
64 |
65 | Final mini-batch SGD weights: [ 1.78392582 0.95504057 0.33473039 -0.02170426]
66 | Distance to closed-form = 1.0293764006591533
67 |
68 | Closed-form solution (scaled basis) for mini-batch: [1.75000000e+00 1.00000000e+00
2.50000000e-01 8.39450872e-15]
69 | [SCALED] Epoch 0: Train MSE = 0.045927, Dist = 1.416e+00
70 | [SCALED] Epoch 100: Train MSE = 0.006505, Dist = 9.340e-02
71 | [SCALED] Epoch 200: Train MSE = 0.006073, Dist = 8.988e-02
72 | [SCALED] Epoch 300: Train MSE = 0.007051, Dist = 9.525e-02
73 | [SCALED] Epoch 400: Train MSE = 0.008163, Dist = 1.160e-01
74 | [SCALED] Epoch 500: Train MSE = 0.008366, Dist = 1.143e-01
75 | [SCALED] Epoch 600: Train MSE = 0.005401, Dist = 9.423e-02
76 | [SCALED] Epoch 700: Train MSE = 0.006311, Dist = 8.857e-02
77 | [SCALED] Epoch 800: Train MSE = 0.005444, Dist = 8.853e-02
78 | [SCALED] Epoch 900: Train MSE = 0.008220, Dist = 1.078e-01
79 | [SCALED] Epoch 1000: Train MSE = 0.005398, Dist = 9.288e-02
80 | [SCALED] Epoch 1100: Train MSE = 0.003549, Dist = 6.998e-02
81 | [SCALED] Epoch 1200: Train MSE = 0.010028, Dist = 1.276e-01
82 | [SCALED] Epoch 1300: Train MSE = 0.006572, Dist = 1.004e-01
83 | [SCALED] Epoch 1400: Train MSE = 0.007778, Dist = 1.090e-01
84 | [SCALED] Epoch 1500: Train MSE = 0.005505, Dist = 9.799e-02
85 | [SCALED] Epoch 1600: Train MSE = 0.011210, Dist = 1.334e-01
86 | [SCALED] Epoch 1700: Train MSE = 0.006529, Dist = 8.929e-02
87 | [SCALED] Epoch 1800: Train MSE = 0.008462, Dist = 1.143e-01
88 | [SCALED] Epoch 1900: Train MSE = 0.009798, Dist = 1.072e-01
89 |
90 | Final mini-batch SGD weights (scaled): [1.76256173 1.02239192 0.36638021
0.09804078]
91 | Distance to scaled closed-form = 0.15432285073856614
92 |
93 | ===== F. Batch size study =====
94 |

```

```

95 --- Running SGD for batch size B = 1 ---
96 Epoch 0: test MSE = 1.970e-05
97 Epoch 100: test MSE = 4.743e-31
98 Target MSE reached at epoch 100. Stopping training.
99 Final test MSE = 4.743e-31
100 Time to reach target: 1.2927360534667969
101 Total time = 119.36 sec
102
103 --- Running SGD for batch size B = 16 ---
104 Epoch 0: test MSE = 1.165e-01
105 Epoch 100: test MSE = 1.009e-03
106 Epoch 200: test MSE = 1.646e-03
107 Epoch 300: test MSE = 6.583e-04
108 Epoch 400: test MSE = 9.901e-04
109 Epoch 500: test MSE = 2.741e-03
110 Epoch 600: test MSE = 7.420e-04
111 Epoch 700: test MSE = 7.916e-04
112 Target MSE reached at epoch 758. Stopping training.
113 Final test MSE = 2.784e-05
114 Time to reach target: 57.53362798690796
115 Total time = 57.53 sec
116
117 --- Running SGD for batch size B = 64 ---
118 Epoch 0: test MSE = 1.894e+00
119 Epoch 100: test MSE = 1.674e-03
120 Epoch 10000: test MSE = 1.826e-03
121 Epoch 50000: test MSE = 2.094e-03
122 Final test MSE = 1.830e-03
123 Time to reach target: None
124 Total time = 2079.73 sec
125
126 --- Running SGD for batch size B = 800 ---
127 Epoch 0: test MSE = 3.392e+00
128 Epoch 100: test MSE = 5.355e-03
129 Epoch 10000: test MSE = 5.355e-03
130 Epoch 50000: test MSE = 5.355e-03
131 Final test MSE = 7.150e-03
132 Time to reach target: None
133 Total time = 209.35 sec
134
135 ===== G. Learning-rate study =====
136
137 Running with learning rate = 0.1
138 constant lr: epoch 0, test mse = 1.440e-02
139 constant lr: epoch 1000, test mse = 1.970e-03
140 constant lr: epoch 2000, test mse = 1.133e-03
141 constant lr: epoch 3000, test mse = 1.986e-03
142 constant lr: epoch 4000, test mse = 1.390e-03
143 constant lr: final mse = 2.031e-03, dist = 6.195e-02
144 momentum beta=0.9: epoch 0, test mse = 1.377e-02
145 momentum beta=0.9: epoch 1000, test mse = 9.439e-03
146 momentum beta=0.9: epoch 2000, test mse = 3.363e-03
147 momentum beta=0.9: epoch 3000, test mse = 9.066e-03
148 momentum beta=0.9: epoch 4000, test mse = 1.207e-02
149 momentum beta=0.9: final mse = 6.652e-03, dist = 1.195e-01
150 cosine decay: epoch 0, test mse = 1.091e+00
151 cosine decay: epoch 1000, test mse = 9.030e-24
152 cosine decay: epoch 2000, test mse = 9.030e-24
153 cosine decay: epoch 3000, test mse = 9.030e-24

```

```

154 cosine decay: epoch 4000, test mse = 9.030e-24
155 cosine decay: final mse = 8.731e-24, dist = 2.295e-11
156
157 Running with learning rate = 0.01
158 constant lr: epoch 0, test mse = 6.387e-01
159 constant lr: epoch 1000, test mse = 4.018e-32
160 constant lr: epoch 2000, test mse = 4.043e-32
161 constant lr: epoch 3000, test mse = 2.958e-32
162 constant lr: epoch 4000, test mse = 2.958e-32
163 constant lr: final mse = 2.459e-32, dist = 9.777e-15
164 momentum beta=0.9: epoch 0, test mse = 4.425e+00
165 momentum beta=0.9: epoch 1000, test mse = 3.032e-32
166 momentum beta=0.9: epoch 2000, test mse = 2.613e-32
167 momentum beta=0.9: epoch 3000, test mse = 3.254e-32
168 momentum beta=0.9: epoch 4000, test mse = 9.466e-32
169 momentum beta=0.9: final mse = 2.404e-32, dist = 9.786e-15
170 cosine decay: epoch 0, test mse = 6.308e+00
171 cosine decay: epoch 1000, test mse = 5.474e-21
172 cosine decay: epoch 2000, test mse = 5.474e-21
173 cosine decay: epoch 3000, test mse = 5.474e-21
174 cosine decay: epoch 4000, test mse = 5.474e-21
175 cosine decay: final mse = 5.293e-21, dist = 5.654e-10
176
177 Running with learning rate = 0.001
178 constant lr: epoch 0, test mse = 8.157e-01
179 constant lr: epoch 1000, test mse = 3.077e-23
180 constant lr: epoch 2000, test mse = 3.071e-30
181 constant lr: epoch 3000, test mse = 2.831e-30
182 constant lr: epoch 4000, test mse = 2.663e-30
183 constant lr: final mse = 2.448e-30, dist = 2.004e-15
184 momentum beta=0.9: epoch 0, test mse = 1.125e+01
185 momentum beta=0.9: epoch 1000, test mse = 1.048e-31
186 momentum beta=0.9: epoch 2000, test mse = 9.762e-32
187 momentum beta=0.9: epoch 3000, test mse = 9.688e-32
188 momentum beta=0.9: epoch 4000, test mse = 9.614e-32
189 momentum beta=0.9: final mse = 8.684e-32, dist = 8.111e-15
190 cosine decay: epoch 0, test mse = 7.158e+00
191 cosine decay: epoch 1000, test mse = 1.088e+00
192 cosine decay: epoch 2000, test mse = 1.088e+00
193 cosine decay: epoch 3000, test mse = 1.088e+00
194 cosine decay: epoch 4000, test mse = 1.088e+00
195 cosine decay: final mse = 1.072e+00, dist = 2.062e+00
196
197 ===== H. SGD vs another optimizer =====
198
199 Running with learning rate = 0.1
200 Momentum beta=0.9: epoch 0, test mse = 1.398e-03
201 Momentum beta=0.9: epoch 1000, test mse = 7.683e-03
202 Momentum beta=0.9: epoch 2000, test mse = 3.300e-03
203 Momentum beta=0.9: epoch 3000, test mse = 3.783e-03
204 Momentum beta=0.9: epoch 4000, test mse = 9.086e-03
205 Momentum beta=0.9: final mse = 2.818e-03, dist = 2.190e-01
206 Adam optimizer: epoch 0, test mse = 2.291e+00
207 Adam optimizer: epoch 1000, test mse = 1.153e-07
208 Adam optimizer: epoch 2000, test mse = 2.422e-04
209 Adam optimizer: epoch 3000, test mse = 1.552e-03
210 Adam optimizer: epoch 4000, test mse = 2.424e-07
211 Adam optimizer: final mse = 1.390e-11, dist = 1.477e-05
212

```

```

213 | Running with learning rate = 0.01
214 | Momentum beta=0.9: epoch 0, test mse = 3.438e-03
215 | Momentum beta=0.9: epoch 1000, test mse = 3.229e-32
216 | Momentum beta=0.9: epoch 2000, test mse = 3.698e-32
217 | Momentum beta=0.9: epoch 3000, test mse = 3.081e-32
218 | Momentum beta=0.9: epoch 4000, test mse = 2.761e-32
219 | Momentum beta=0.9: final mse = 2.687e-32, dist = 9.766e-15
220 | Adam optimizer: epoch 0, test mse = 1.978e+01
221 | Adam optimizer: epoch 1000, test mse = 3.550e-14
222 | Adam optimizer: epoch 2000, test mse = 1.985e-07
223 | Adam optimizer: epoch 3000, test mse = 5.544e-05
224 | Adam optimizer: epoch 4000, test mse = 5.336e-30
225 | Adam optimizer: final mse = 5.158e-15, dist = 1.750e-07
226
227 | Running with learning rate = 0.001
228 | Momentum beta=0.9: epoch 0, test mse = 3.163e-01
229 | Momentum beta=0.9: epoch 1000, test mse = 2.914e-31
230 | Momentum beta=0.9: epoch 2000, test mse = 2.911e-31
231 | Momentum beta=0.9: epoch 3000, test mse = 2.926e-31
232 | Momentum beta=0.9: epoch 4000, test mse = 2.926e-31
233 | Momentum beta=0.9: final mse = 2.879e-31, dist = 1.360e-14
234 | Adam optimizer: epoch 0, test mse = 2.253e+01
235 | Adam optimizer: epoch 1000, test mse = 8.907e-17
236 | Adam optimizer: epoch 2000, test mse = 3.355e-15
237 | Adam optimizer: epoch 3000, test mse = 9.117e-16
238 | Adam optimizer: epoch 4000, test mse = 1.030e-12
239 | Adam optimizer: final mse = 1.726e-09, dist = 8.909e-05

```

Listing 2: Output terminal (selected) for sgd.py

## V. EMPIRICAL UNIVERSAL APPROXIMATION THEOREM

### B. Data and target function

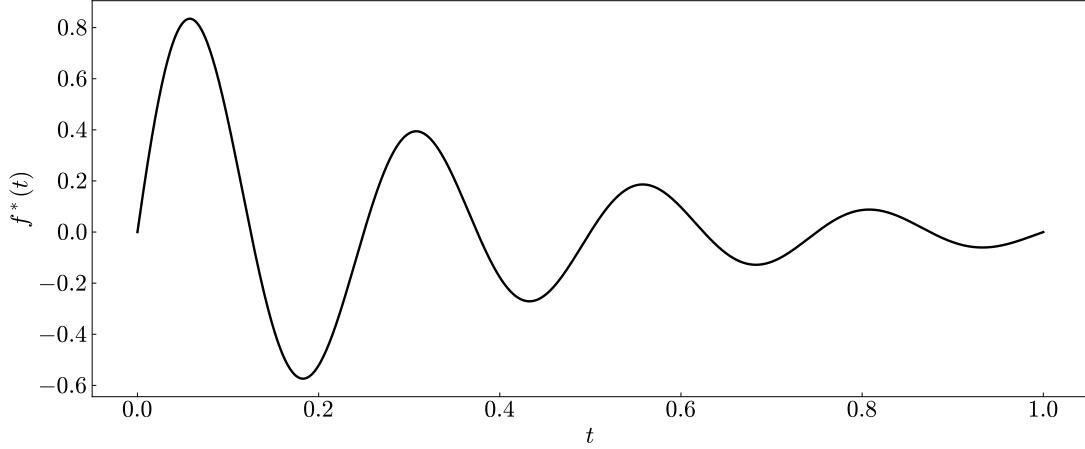


Figure 17: Target function.

### C. A two-layer network model

The two layer network is defined as,

$$\tilde{f}(t, \theta) = \sum_{j=1}^m a_j \phi(w_j t + b_j) + c$$

`adam` is used as the optimizer for this network. Biases  $b_j$  are drawn uniformly over  $[-1, 1]$ , and hidden layer weights  $w_j$  are randomly initialized from a normal distribution—scaled by 30 to have a wide distribution. Output weights  $a_j$  are initialized with small values—scaled by 0.1 to prevent large values during early stage of training. Output bias  $c$  is set to 0.

Additionally, please refer to listing 3.

#### D. Implementation & visualization

Activation function ReLU is used with network width  $m = 16$ . Number of epochs and learning rate are set at 250000 and  $10^{-3}$  respectively.

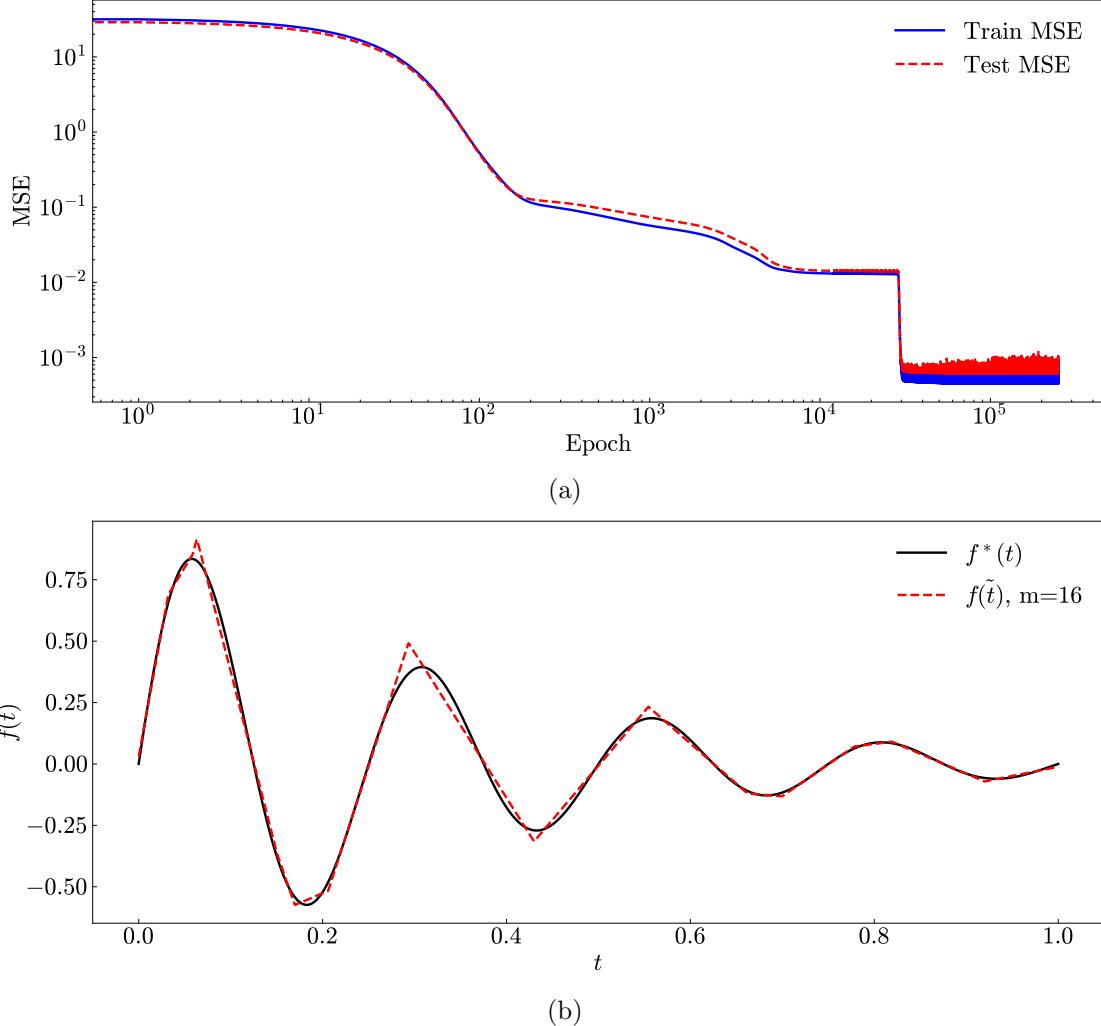


Figure 18: (a)  $MSE$  for training and testing data, and (b) Comparison of  $\tilde{f}(t; \theta)$  with  $f^*(t)$ .

We can see from fig. 18a that the training and testing  $MSE$  plateaus after  $\sim 30000$  to  $< 10^{-3}$ . Fig. 18b confirms that the learned function  $\tilde{f}(t; \theta)$  approximates the target function  $f^*(t)$  pretty well for the given width. It would approximate the target function even better with larger network width.

#### E. Empirical rate of convergence with layer width

Given,

$$m \in \{2, 4, 8, 16, 32, 64, 128\}$$

Number of epochs used is 20000 and the learning rate is set at  $10^{-3}$  as done previously. Three restarts are considered per  $m$ .

To estimate the slope  $\alpha$ , the relation between  $E(m)$  and  $m$  is linearized using  $\log$ .

$$\log E(m) \approx \log(Cm^{-\alpha}) = \log C - \alpha \log m$$

which is in the  $y = c + mx$  form.

To compute the slope, linear regression is done using `polyfit()`. Estimated rate of convergence based on  $E_2(m)$  and  $E_\infty(m)$ :

$$\alpha_{E_2} \approx 0.878, \quad \alpha_{E_\infty} \approx 0.743$$

We can see from fig. 19 that as the width increases, errors decay. Between  $m = 8$  to  $64$ , the decay

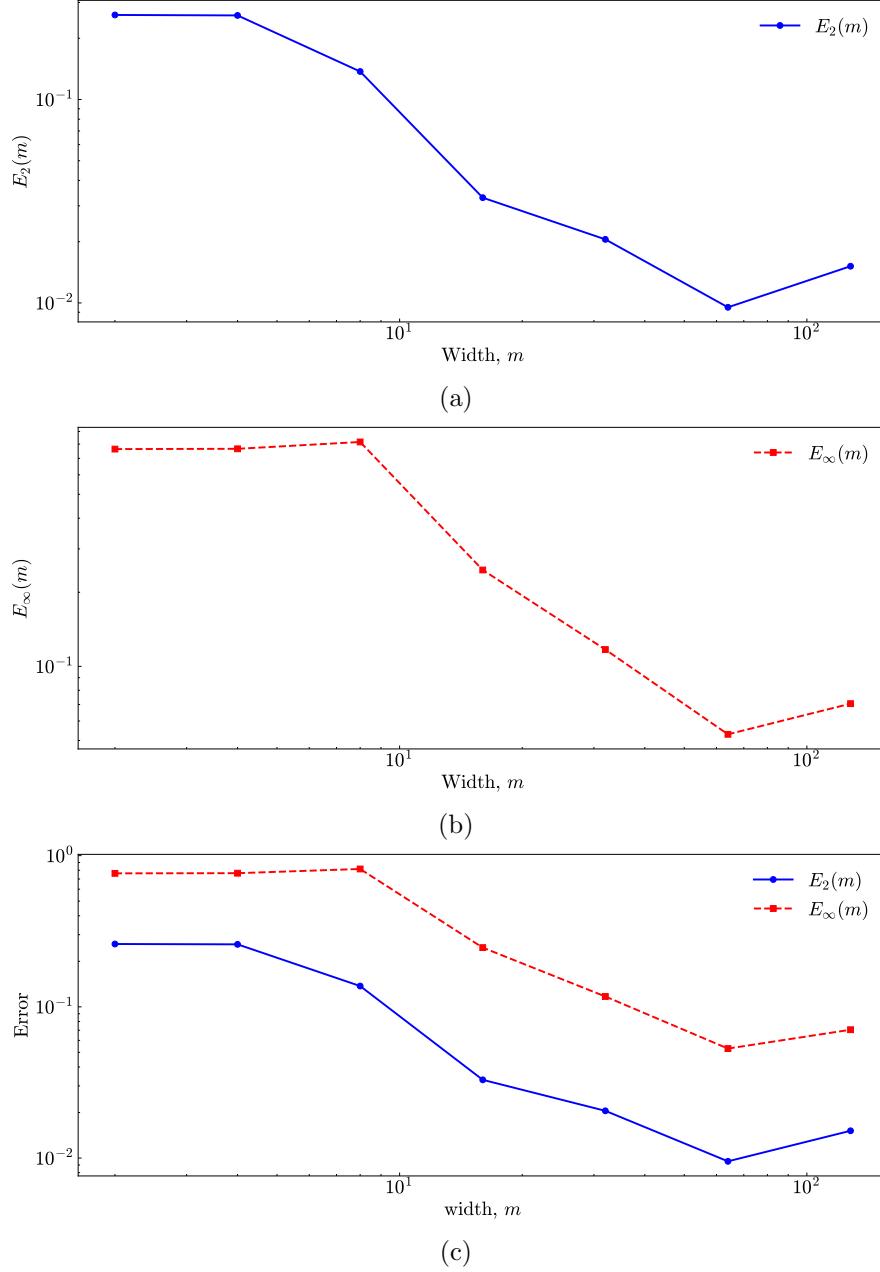


Figure 19: (a)  $E_2(m)$  vs.  $m$ , (b)  $E_\infty(m)$  vs.  $m$ , and (c) single plot of final test errors for ReLU.

in errors is almost linear. However, there is a slight increase in errors as width moves from 64 to 128.

### F. Effect of the activation function

This time the same sweeping is done using `tanh` as the activation function. All the parameters are kept the same as `ReLU` setup, so that we can make an apple-to-apple comparison.

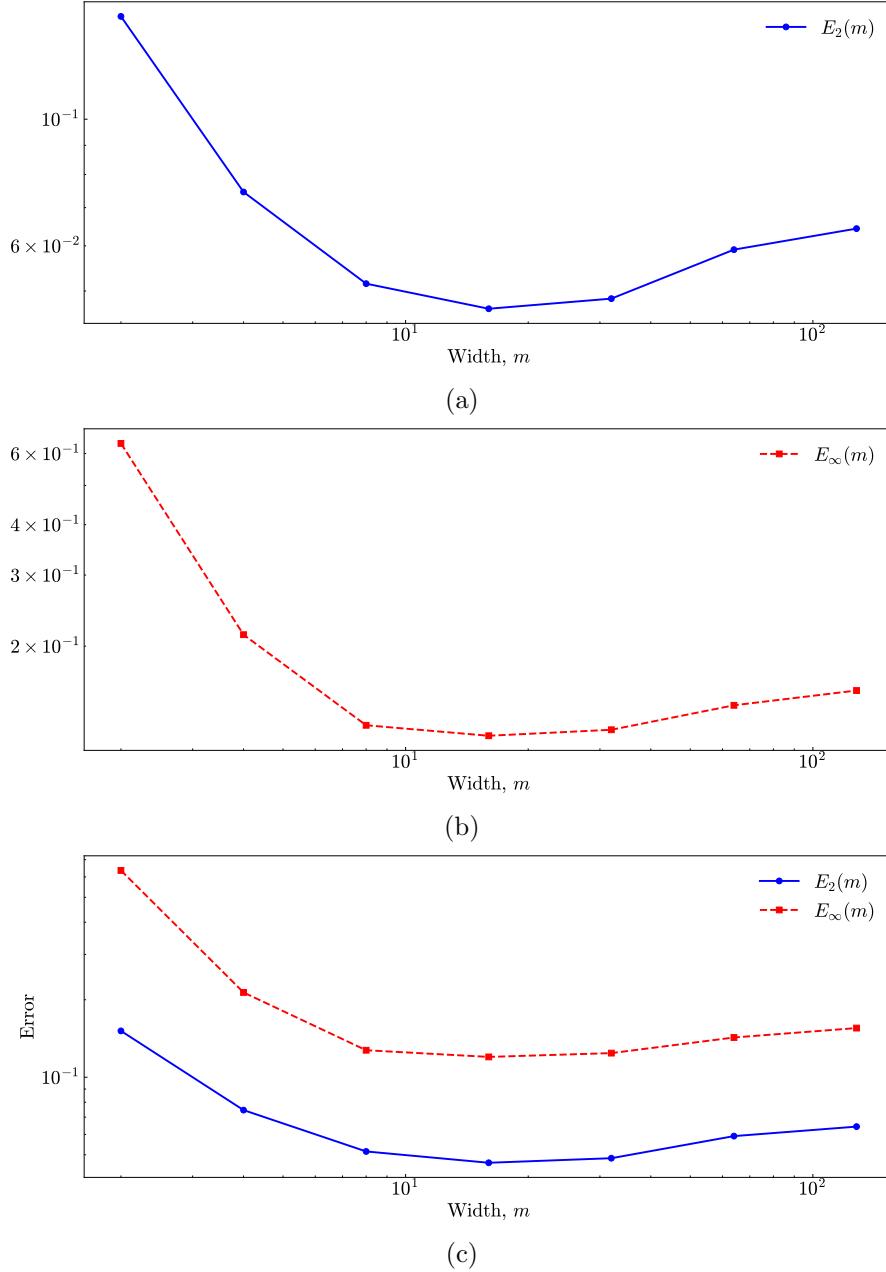


Figure 20: (a)  $E_2(m)$  vs.  $m$ , (b)  $E_\infty(m)$  vs.  $m$ , and (c) single plot of final test errors for `tanh`.

Estimated rate of convergence based on  $E_2(m)$  and  $E_\infty(m)$  for  $\tanh$ :

$$\alpha_{E_2} \approx 0.159, \quad \alpha_{E_\infty} \approx 0.261$$

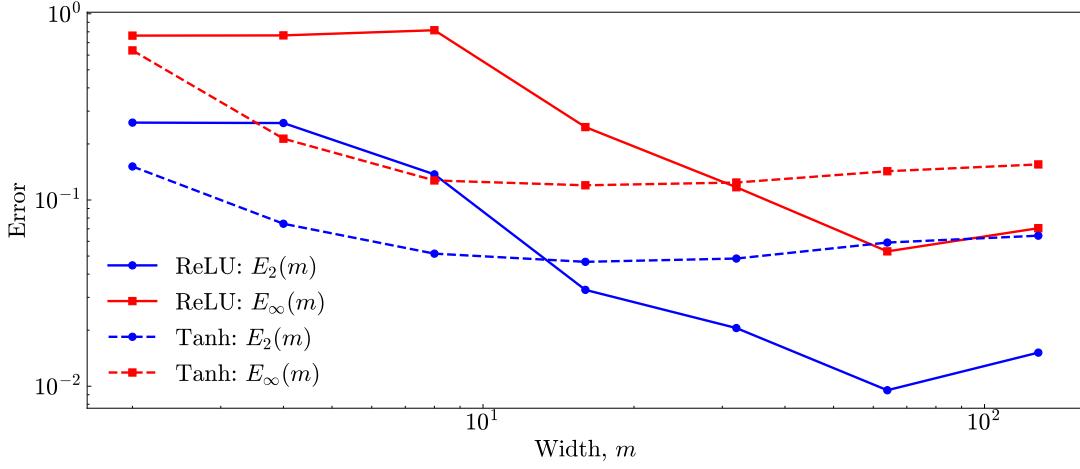


Figure 21: Comparison of final test errors between  $\text{ReLU}$  and  $\tanh$ .

Some observations from fig. 21:

- $\text{ReLU}$  has higher rate of convergence than  $\tanh$  ( $\sim 3x$  faster if we consider  $E_\infty(m)$ , and  $\sim 6x$  faster if we consider  $E_2(m)$ ). If we look at the activation function for both

$$\text{ReLU}(a) = \max(0, a) \quad \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

$\text{ReLU}$  is a simpler function, and is easy to compute. So understandably, it should be faster.

- One thing worth noting is that  $\tanh$  drops the error steeply in smaller width compared to  $\text{ReLU}$ , which decreases error steeply after  $m = 8$ . But after the initial steep decrease,  $\tanh$  becomes flattened between  $m = 8$  to  $64$ , where  $\text{ReLU}$  starts thriving in the same interval. This might indicate that  $\text{ReLU}$  has a sweet range of  $m$  which is easy to predict. However,  $\tanh$  might require a lot more experimentation to do in the initialization to find that optimum range.
- $\tanh$  can shrink high or low input value by causing the gradient to be too small, which is not the case for  $\text{ReLU}$ . This is particularly important if one does not want the gradient to vanish while training, which can potentially slow down or even stop updating training.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib as mpl
4 import jax
5 import jax.numpy as jnp
6 import optax
7
8 # ===== B. Data and target function =====
9
10 # target function f*
11 def f_star(t):
12     return np.exp(-3.0 * t) * np.sin(8 * np.pi * t)
13
14 # create train and test sets
15 rng = np.random.default_rng(0)
16 t_train = rng.uniform(0.0, 1.0, size=128)
17 y_train = f_star(t_train)
18
19 t_test = np.linspace(0.0, 1.0, 2048, endpoint=True)
20 y_test = f_star(t_test)
21
22 # plotting params
23 plt.rcParams['font.family'] = 'serif'
24 plt.rcParams['font.serif'] = 'cmr10'
25 plt.rcParams['mathtext.fontset'] = 'cm'
26 plt.rcParams['font.size'] = 20
27 mpl.rcParams['axes.unicode_minus'] = False
28 plt.rcParams['axes.formatter.use_mathtext'] = True
29
30 # plot the target function on a dense grid
31 fig, ax = plt.subplots(figsize=(15, 6))
32 ax.plot(t_test, y_test, 'k-', lw=2)
33 plt.xlabel(r"$t$")
34 plt.ylabel(r"$f^*(t)$")
35 plt.tick_params(axis="both", which="both", direction="in")
36 plt.savefig("fstar.pdf", dpi=1080)
37 plt.show()
38
39 # ===== C. Two-layer network model =====
40
41 # relu activation
42 def relu(x):
43     return jnp.maximum(0.0, x)
44
45 # tanh activation
46 def tanh(x):
47     return jnp.tanh(x)
48
49 # model initialization
50 def init_model(rng, m):
51     key_w, key_t, key_a, key_c = jax.random.split(rng, 4)
52
53     # wide input weights
54     w = 30.0 * jax.random.normal(key_w, (m,))
55
56     # kink locations uniformly across [0,1]
57     t0 = jax.random.uniform(key_t, (m,), minval=0.0, maxval=1.0)
58

```

```

59     # bias
60     b = -w * t0
61
62     # small output layer weights
63     a = 0.1 * jax.random.normal(key_a, (m,))
64
65     c = jnp.array(0.0) # output bias
66
67     return {"w": w, "b": b, "a": a, "c": c}
68
69 # forward pass
70 def model_apply(params, t, activation):
71     # compute  $w_j t + b_j$  for all hidden units
72     z = params["w"] * t[:, None] + params["b"]
73
74     # apply activation
75     h = activation(z)
76
77     # output
78     y_pred = jnp.dot(h, params["a"]) + params["c"]
79     return y_pred
80
81 # mse loss function
82 def mse_loss(params, t, y, activation):
83     y_pred = model_apply(params, t, activation)
84     r = y_pred - y
85     return jnp.mean(r * r)
86
87 # ===== D. Implementation and visualization =====
88
89 # width
90 m = 16
91
92 # number of epochs
93 epochs_d = 250000
94
95 # optimizer (adam)
96 optimizer_d = optax.adam(learning_rate=1e-3)
97
98 # initialize parameters
99 rng = jax.random.PRNGKey(299)
100 params_d = init_model(rng, m)
101
102 opt_state_d = optimizer_d.init(params_d)
103
104 # choose activation outside the jit
105 activation = relu
106
107 # jit-compiled update step
108 @jax.jit
109 def step(params, opt_state, t, y):
110     loss_val, grads = jax.value_and_grad(mse_loss)(params, t, y, activation)
111     updates, opt_state = optimizer_d.update(grads, opt_state)
112     params = optax.apply_updates(params, updates)
113     return params, opt_state, loss_val
114
115 # training history
116 train_hist_d = []
117 test_hist_d = []

```

```

118
119 # convert training data to jax arrays
120 t_train_j = jnp.asarray(t_train)
121 y_train_j = jnp.asarray(y_train)
122 t_test_j = jnp.asarray(t_test)
123 y_test_j = jnp.asarray(y_test)
124
125 # training loop
126 for epoch in range(epochs_d):
127     # shuffle each epoch
128     key_epoch = jax.random.PRNGKey(epoch)
129     perm = jax.random.permutation(key_epoch, len(t_train_j))
130     t_batch = t_train_j[perm]
131     y_batch = y_train_j[perm]
132
133     params_d, opt_state_d, train_loss = step(params_d, opt_state_d,
134                                              t_batch, y_batch)
135
136     train_hist_d.append(float(train_loss))
137
138     # compute test mse
139     test_loss = mse_loss(params_d, t_test_j, y_test_j, activation)
140     test_hist_d.append(float(test_loss))
141
142     if epoch % 500 == 0:
143         print(f"epoch {epoch}, train mse = {train_loss:.4e}")
144
145 print(f"\nfinal train mse = {train_hist_d[-1]:.4e}")
146 print(f"final test mse = {test_hist_d[-1]:.4e}")
147
148 # plot mse vs. epoch
149 fig, ax = plt.subplots(figsize=(15, 6))
150 ax.loglog(train_hist_d, "b-", lw=2, label="Train MSE")
151 ax.loglog(test_hist_d, "r--", lw=2, label="Test MSE")
152 plt.xlabel("Epoch")
153 plt.ylabel("MSE")
154 plt.legend(frameon=False)
155 plt.tick_params(axis="both", which="both", direction="in")
156 plt.savefig("mse_m16.pdf", dpi=1080)
157 plt.show()
158
159 # plot model and true function
160 y_pred_d = model_apply(params_d, t_test_j, activation)
161
162 fig, ax = plt.subplots(figsize=(15, 6))
163 ax.plot(t_test, y_test, "k-", lw=2, label=r"$f^*(t)$")
164 ax.plot(t_test, y_pred_d, "r--", lw=2, label=r"$\tilde{f}(t)$, m=16")
165 plt.xlabel(r"$t$")
166 plt.ylabel(r"$f(t)$")
167 plt.legend(frameon=False)
168 plt.tick_params(axis="both", which="both", direction="in")
169 plt.savefig("compare_m16.pdf", dpi=1080)
170 plt.show()
171
172 # ===== E. Empirical convergence with width =====
173
174 # params
175 width_list = [2, 4, 8, 16, 32, 64, 128]
176 epochs_e = 20000

```

```

177 restarts = 3
178
179 E2_vals = []
180 Einf_vals = []
181
182 # optimizer (adam)
183 optimizer_e = optax.adam(learning_rate=1e-3)
184
185 @jax.jit
186 def step_e(params, opt_state, t, y):
187     loss_val, grads = jax.value_and_grad(mse_loss)(params, t, y, relu)
188     updates, opt_state = optimizer_e.update(grads, opt_state)
189     params = optax.apply_updates(params, updates)
190     return params, opt_state, loss_val
191
192 for m in width_list:
193     print(f"\n==== width m = {m} ====")
194
195     # store errors from restarts
196     err2_list = []
197     errinf_list = []
198
199     for r in range(restarts):
200         print(f" restart {r}")
201
202         # init params
203         key = jax.random.PRNGKey(1000 + 17*m + r)
204         params = init_model(key, m)
205         opt_state = optimizer_e.init(params)
206
207         # train
208         for epoch in range(epochs_e):
209             key_epoch = jax.random.PRNGKey(epoch)
210             perm = jax.random.permutation(key_epoch, len(t_train_j))
211             t_batch = t_train_j[perm]
212             y_batch = y_train_j[perm]
213             params, opt_state, train_loss = step_e(params, opt_state, t_batch,
214                                         y_batch)
215
216             # compute test prediction
217             y_pred = model_apply(params, t_test_j, relu)
218
219             # compute errors (convert to numpy first)
220             residual = np.array(y_pred) - np.array(y_test)
221             E2 = np.sqrt(np.mean(residual**2))
222             Einf = np.max(np.abs(residual))
223
224             err2_list.append(E2)
225             errinf_list.append(Einf)
226
227             # take median across restarts
228             E2_vals.append(np.median(err2_list))
229             Einf_vals.append(np.median(errinf_list))
230
231 # convert to numpy arrays
232 width_arr = np.array(width_list)
233 E2_vals = np.array(E2_vals)
234 Einf_vals = np.array(Einf_vals)

```

```

235 # log-log slope estimation
236 logm = np.log(width_arr)
237 logE2 = np.log(E2_vals)
238 logEin = np.log(Einf_vals)
239
240 alpha_E2, _ = np.polyfit(logm, logE2, 1)
241 alpha_Einf, _ = np.polyfit(logm, logEin, 1)
242
243 print("\nestimated slopes (ReLU):")
244 print(f" alpha_E2 ~ {-alpha_E2:.3f}")
245 print(f" alpha_Einf ~ {-alpha_Einf:.3f}")
246
247 # plot E2(m)
248 fig, ax = plt.subplots(figsize=(15, 6))
249 ax.loglog(width_arr, E2_vals, "b-o", lw=2, label=r"$E_2(m)$")
250 plt.xlabel(rf"Width, $m$")
251 plt.ylabel(r"$E_2(m)$")
252 plt.legend(frameon=False)
253 plt.tick_params(axis="both", which="both", direction="in")
254 plt.savefig("E2_vs_m.pdf", dpi=1080)
255 plt.show()
256
257 # plot of Einfinity(m)
258 fig, ax = plt.subplots(figsize=(15, 6))
259 ax.loglog(width_arr, Einfinity_vals, "r--s", lw=2, label=r"$E_{\infty}(m)$")
260 plt.xlabel(rf"Width, $m$")
261 plt.ylabel(r"$E_{\infty}(m)$")
262 plt.legend(frameon=False)
263 plt.tick_params(axis="both", which="both", direction="in")
264 plt.savefig("Einfinity_vs_m.pdf", dpi=1080)
265 plt.show()
266
267 # combined plot
268 fig, ax = plt.subplots(figsize=(15, 6))
269 ax.loglog(width_arr, E2_vals, "b-o", lw=2, label=r"$E_2(m)$")
270 ax.loglog(width_arr, Einfinity_vals, "r--s", lw=2, label=r"$E_{\infty}(m)$")
271 plt.xlabel(rf"Width, $m$")
272 plt.ylabel("Error")
273 plt.legend(frameon=False)
274 plt.tick_params(axis="both", which="both", direction="in")
275 plt.savefig("convergence_width.pdf", dpi=1080)
276 plt.show()
277
278 # ===== F. Effect of Activation Function =====
279
280 # params
281 width_list = [2, 4, 8, 16, 32, 64, 128]
282 epochs_f = 20000
283 restarts_f = 3
284
285 # store results
286 E2_tanh_vals = []
287 Einfinity_tanh_vals = []
288
289 # optimizer (adam)
290 optimizer_f = optax.adam(learning_rate=1e-3)
291
292 # model initialization for tanh (smaller weights to avoid saturation)
293 def init_model_tanh(rng, m):

```

```

294     key_w, key_b, key_a, key_c = jax.random.split(rng, 4)
295
296     # smaller input weights for tanh to prevent saturation
297     w = 3.0 * jax.random.normal(key_w, (m,))
298
299     # biases uniformly across [-1, 1]
300     b = jax.random.uniform(key_b, (m,), minval=-1.0, maxval=1.0)
301
302     # small output layer weights
303     a = 0.1 * jax.random.normal(key_a, (m,))
304
305     c = jnp.array(0.0) # output bias
306
307     return {"w": w, "b": b, "a": a, "c": c}
308
309
310 # training step for tanh
311 @jax.jit
312 def step_f(params, opt_state, t, y):
313     loss_val, grads = jax.value_and_grad(mse_loss)(params, t, y, tanh)
314     updates, opt_state = optimizer_f.update(grads, opt_state)
315     params = optax.apply_updates(params, updates)
316     return params, opt_state, loss_val
317
318 # loop through each width
319 for m in width_list:
320     print(f"\n==== width m = {m} ===")
321
322     # store errors from restarts
323     err2_list = []
324     errinf_list = []
325
326     for r in range(restarts_f):
327         print(f" restart {r}")
328
329         # initialize parameters
330         key = jax.random.PRNGKey(2000 + 19 * m + r)
331         params = init_model_tanh(key, m)
332         opt_state = optimizer_f.init(params)
333
334         # training loop
335         for epoch in range(epochs_f):
336             key_epoch = jax.random.PRNGKey(epoch)
337             perm = jax.random.permutation(key_epoch, len(t_train_j))
338             t_batch = t_train_j[perm]
339             y_batch = y_train_j[perm]
340             params, opt_state, train_loss = step_f(params, opt_state, t_batch,
341                                         y_batch)
342
343             # compute test predictions
344             y_pred = model_apply(params, t_test_j, tanh)
345
346             # compute errors (convert to numpy first)
347             residual = np.array(y_pred) - np.array(y_test)
348             E2 = np.sqrt(np.mean(residual**2))
349             Einf = np.max(np.abs(residual))
350
351             err2_list.append(E2)
352             errinf_list.append(Einf)

```

```

352     # take the median across restarts
353     E2_tanh_vals.append(np.median(err2_list))
354     Einf_tanh_vals.append(np.median(errinf_list))
355
356
357     # convert to numpy arrays
358     E2_tanh_vals = np.array(E2_tanh_vals)
359     Einf_tanh_vals = np.array(Einf_tanh_vals)
360
361     # log-log slope estimation for tanh
362     logm = np.log(width_list)
363     logE2_th = np.log(E2_tanh_vals)
364     logEin_th = np.log(Einf_tanh_vals)
365
366     alpha_E2_tanh, _ = np.polyfit(logm, logE2_th, 1)
367     alpha_Einf_tanh, _ = np.polyfit(logm, logEin_th, 1)
368
369     print("\nestimated slopes (tanh):")
370     print(f"  alpha_E2_tanh ~ {-alpha_E2_tanh:.3f}")
371     print(f"  alpha_Einf_tanh ~ {-alpha_Einf_tanh:.3f}")
372
373     # plot E2(m) for tanh
374     fig, ax = plt.subplots(figsize=(15, 6))
375     ax.loglog(width_list, E2_tanh_vals, "b-o", lw=2, label=r"$E_2(m)$")
376     plt.xlabel(rf"Width, $m$")
377     plt.ylabel(r"$E_2(m)$")
378     plt.legend(frameon=False)
379     plt.tick_params(axis="both", which="both", direction="in")
380     plt.savefig("E2_tanh_vs_m.pdf", dpi=1080)
381     plt.show()
382
383     # plot Einfinity(m) for tanh
384     fig, ax = plt.subplots(figsize=(15, 6))
385     ax.loglog(width_list, Einf_tanh_vals, "r--s", lw=2, label=r"$E_{\infty}(m)$")
386     plt.xlabel(rf"Width, $m$")
387     plt.ylabel(r"$E_{\infty}(m)$")
388     plt.legend(frameon=False)
389     plt.tick_params(axis="both", which="both", direction="in")
390     plt.savefig("Einf_tanh_vs_m.pdf", dpi=1080)
391     plt.show()
392
393     # combined plot
394     fig, ax = plt.subplots(figsize=(15, 6))
395     ax.loglog(width_arr, E2_tanh_vals, "b-o", lw=2, label=r"$E_2(m)$")
396     ax.loglog(width_arr, Einf_tanh_vals, "r--s", lw=2, label=r"$E_{\infty}(m)$")
397     plt.xlabel(rf"Width, $m$")
398     plt.ylabel("Error")
399     plt.legend(frameon=False)
400     plt.tick_params(axis="both", which="both", direction="in")
401     plt.savefig("convergence_tanh.pdf", dpi=1080)
402     plt.show()
403
404     # combined comparison plot for ReLU and tanh
405     fig, ax = plt.subplots(figsize=(15, 6))
406     ax.loglog(width_arr, E2_vals, "b-o", lw=2, label=r"ReLU: $E_2(m)$")
407     ax.loglog(width_arr, Einf_vals, "r-s", lw=2, label=r"ReLU: $E_{\infty}(m)$")
408     ax.loglog(width_arr, E2_tanh_vals, "b--o", lw=2, label=r"Tanh: $E_2(m)$")
409     ax.loglog(width_arr, Einf_tanh_vals, "r--s", lw=2, label=r"Tanh: $E_{\infty}(m)$")
410     plt.xlabel(rf"Width, $m$")

```

```

411 plt.ylabel("Error")
412 plt.legend(frameon=False)
413 plt.tick_params(axis="both", which="both", direction="in")
414 plt.savefig("convergence_relu_vs_tanh.pdf", dpi=1080)
415 plt.show()

```

Listing 3: universal\_approx.py

```

1 epoch 0, train mse = 3.2565e+01
2 epoch 500, train mse = 7.8099e-02
3 epoch 1000, train mse = 5.6896e-02
4 epoch 1500, train mse = 4.9343e-02
5 epoch 2000, train mse = 4.3553e-02
6 epoch 2500, train mse = 3.7327e-02
7 epoch 3000, train mse = 3.0488e-02
8 epoch 3500, train mse = 2.5812e-02
9 epoch 4000, train mse = 2.2334e-02
10 epoch 4500, train mse = 1.9205e-02
11 epoch 5000, train mse = 1.6635e-02
12 epoch 10000, train mse = 1.3209e-02
13 epoch 25000, train mse = 1.2906e-02
14 epoch 25500, train mse = 1.2902e-02
15 epoch 26000, train mse = 1.2899e-02
16 epoch 26500, train mse = 1.2900e-02
17 epoch 27000, train mse = 1.2892e-02
18 epoch 27500, train mse = 1.2889e-02
19 epoch 28000, train mse = 1.2886e-02
20 epoch 28500, train mse = 1.2874e-02
21 epoch 29000, train mse = 9.4067e-03
22 epoch 29500, train mse = 1.5621e-03
23 epoch 30000, train mse = 6.5884e-04
24 epoch 40000, train mse = 4.6610e-04
25 epoch 50000, train mse = 4.5693e-04
26 epoch 100000, train mse = 4.4939e-04
27 epoch 150000, train mse = 4.6481e-04
28 epoch 200000, train mse = 4.4736e-04
29 final train mse = 4.5028e-04
30 final test mse = 6.0630e-04
31
32 === width m = 2 ===
33 restart 0
34 restart 1
35 restart 2
36
37 === width m = 4 ===
38 restart 0
39 restart 1
40 restart 2
41
42 === width m = 8 ===
43 restart 0
44 restart 1
45 restart 2
46
47 === width m = 16 ===
48 restart 0
49 restart 1
50 restart 2
51

```

```

52 === width m = 32 ===
53 restart 0
54 restart 1
55 restart 2
56
57 === width m = 64 ===
58 restart 0
59 restart 1
60 restart 2
61
62 === width m = 128 ===
63 restart 0
64 restart 1
65 restart 2
66
67 estimated slopes (ReLU):
68 alpha_E2      ~ 0.878
69 alpha_Einf   ~ 0.743
70
71 === width m = 2 ===
72 restart 0
73 restart 1
74 restart 2
75
76 === width m = 4 ===
77 restart 0
78 restart 1
79 restart 2
80
81 === width m = 8 ===
82 restart 0
83 restart 1
84 restart 2
85
86 === width m = 16 ===
87 restart 0
88 restart 1
89 restart 2
90
91 === width m = 32 ===
92 restart 0
93 restart 1
94 restart 2
95
96 === width m = 64 ===
97 restart 0
98 restart 1
99 restart 2
100
101 === width m = 128 ===
102 restart 0
103 restart 1
104 restart 2
105
106 estimated slopes (tanh):
107 alpha_E2_tanh ~ 0.159
108 alpha_Einf_tanh ~ 0.261

```

Listing 4: Output terminal (selected) for `universal_approx.py`

### Challenge bonus: Building compact networks

Setup:

Optimizer: adam

Activation function: ReLU

Width: 64 (single hidden network)

Learning rate:  $10^{-4}$

No. of epochs: 500000

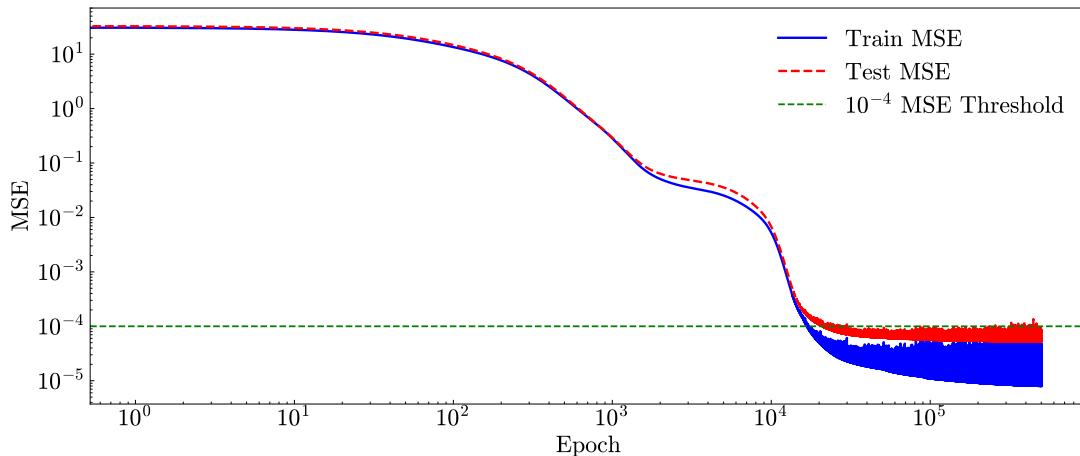


Figure 22: Training and testing  $MSE$ . Testing  $MSE$  is below  $10^{-4}$  threshold.

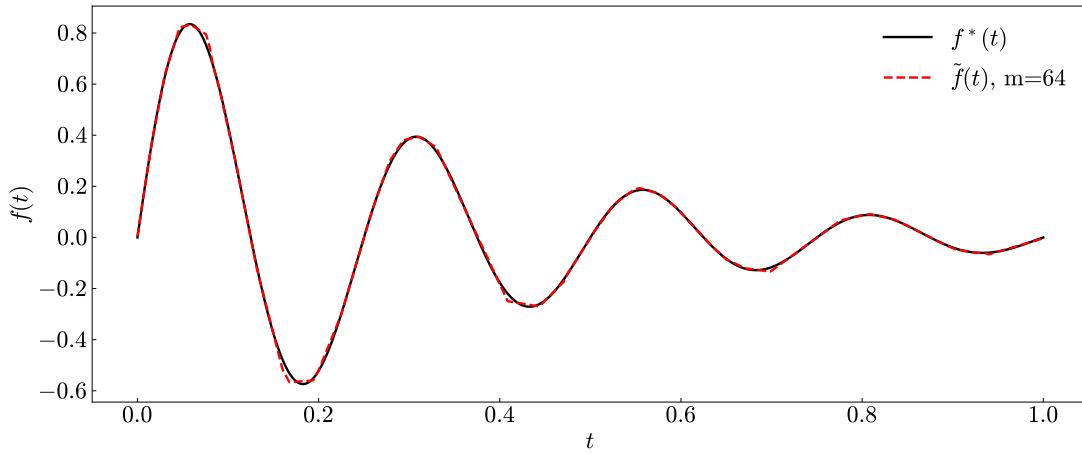


Figure 23: Learned  $\tilde{f}(t; \theta)$  and target function  $f^*(t)$ .

Number of trainable parameters for the model: 193

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib as mpl
4 import jax
5 import jax.numpy as jnp
6 import optax
7
8 # target function f*
9 def f_star(t):
10     return np.exp(-3.0 * t) * np.sin(8 * np.pi * t)
11
12 # train and test sets
13 rng = np.random.default_rng(0)
14 t_train = rng.uniform(0.0, 1.0, size=128)
15 y_train = f_star(t_train)
16
17 t_test = np.linspace(0.0, 1.0, 2048, endpoint=True)
18 y_test = f_star(t_test)
19
20 # plotting params
21 plt.rcParams['font.family'] = 'serif'
22 plt.rcParams['font.serif'] = 'cmr10'
23 plt.rcParams['mathtext.fontset'] = 'cm'
24 plt.rcParams['font.size'] = 20
25 mpl.rcParams['axes.unicode_minus'] = False
26 plt.rcParams['axes.formatter.use_mathtext'] = True
27
28 # relu activation
29 def relu(x):
30     return jnp.maximum(0.0, x)
31
32 # model initialization
33 def init_model(rng, m):
34     # Initialize weights and biases for a single layer
35     key_w, key_t, key_a, key_c = jax.random.split(rng, 4)
36     w = 30.0 * jax.random.normal(key_w, (m,))
37     t0 = jax.random.uniform(key_t, (m,), minval=0.0, maxval=1.0) # kink locations
38     b = -w * t0
39     a = 0.1 * jax.random.normal(key_a, (m,))
40     c = jnp.array(0.0) # output bias
41     return {"w": w, "b": b, "a": a, "c": c}
42
43 # forward pass
44 def model_apply(params, t, activation):
45     z = params["w"] * t[:, None] + params["b"]
46     h = activation(z) # apply activation
47     y_pred = jnp.dot(h, params["a"]) + params["c"]
48     return y_pred
49
50 # mse loss function
51 def mse_loss(params, t, y, activation):
52     y_pred = model_apply(params, t, activation)
53     r = y_pred - y
54     return jnp.mean(r * r)
55
56 # width and epochs
57 m = 64
58 epochs_d = 500000

```

```

59
60 # optimizer (adam)
61 optimizer_d = optax.adam(learning_rate=1e-4)
62
63 # initialize parameters
64 rng = jax.random.PRNGKey(299)
65 params_d = init_model(rng, m)
66
67 opt_state_d = optimizer_d.init(params_d)
68
69 # activation outside the jit
70 activation = relu
71
72 # jit-compiled update step
73 @jax.jit
74 def step(params, opt_state, t, y):
75     loss_val, grads = jax.value_and_grad(mse_loss)(params, t, y, activation)
76     updates, opt_state = optimizer_d.update(grads, opt_state)
77     params = optax.apply_updates(params, updates)
78     return params, opt_state, loss_val
79
80 # training history
81 train_hist_d = []
82 test_hist_d = []
83
84 # convert training data to jax arrays
85 t_train_j = jnp.asarray(t_train)
86 y_train_j = jnp.asarray(y_train)
87 t_test_j = jnp.asarray(t_test)
88 y_test_j = jnp.asarray(y_test)
89
90 # full batch training loop
91 for epoch in range(epochs_d):
92     params_d, opt_state_d, train_loss = step(params_d, opt_state_d, t_train_j,
93                                              y_train_j)
94
95     train_hist_d.append(float(train_loss))
96
97     # compute test mse
98     test_loss = mse_loss(params_d, t_test_j, y_test_j, activation)
99     test_hist_d.append(float(test_loss))
100
101     if epoch % 5000 == 0:
102         print(f"epoch {epoch}, train mse = {train_loss:.4e}, test mse = {test_loss
103             :.4e}")
104
105 print(f"\nfinal train mse = {train_hist_d[-1]:.4e}")
106 print(f"final test mse = {test_hist_d[-1]:.4e}")
107
108 # plot mse vs. epoch
109 fig, ax = plt.subplots(figsize=(15, 6))
110 ax.loglog(train_hist_d, "b-", lw=2, label="Train MSE")
111 ax.loglog(test_hist_d, "r--", lw=2, label="Test MSE")
112 ax.axhline(y=1e-4, color='g', linestyle='--', label=r"$10^{-4}$ MSE Threshold")
113 plt.xlabel("Epoch")
114 plt.ylabel("MSE")
115 plt.legend(frameon=False)
116 plt.tick_params(axis="both", which="both", direction="in")

```

```

116 plt.savefig(f"param_count_mse.pdf", dpi=1080)
117 plt.show()
118
119 # plot model and true function
120 y_pred_d = model_apply(params_d, t_test_j, activation)
121
122 fig, ax = plt.subplots(figsize=(15, 6))
123 ax.plot(t_test, y_test, "k-", lw=2, label=r"$f^*(t)$")
124 ax.plot(t_test, y_pred_d, "r--", lw=2, label=r"$\tilde{f}(t)$, m=64")
125 plt.xlabel(r"$t$")
126 plt.ylabel(r"$f(t)$")
127 plt.legend(frameon=False)
128 plt.tick_params(axis="both", which="both", direction="in")
129 plt.savefig(f"model_true_bonus.pdf", dpi=1080)
130 plt.show()
131
132 # count the number of trainable parameters
133 def count_params_pytree(params):
134     leaves = jax.tree_util.tree_leaves(params) # Collect all leaf arrays from the
135     nested params tree
136     sizes = [int(jnp.size(x)) for x in leaves] # Get the size of each array
137     total = sum(sizes) # Get the total number of trainable scalars
138     return total
139
140 num_params = count_params_pytree(params_d)
141 print(f"Total number of trainable parameters: {num_params}")

```

Listing 5: bonus.py

```

1 epoch 0, train mse = 3.0913e+01, test mse = 3.2876e+01
2 epoch 5000, train mse = 2.5403e-02, test mse = 3.5700e-02
3 epoch 10000, train mse = 5.2675e-03, test mse = 6.7734e-03
4 epoch 15000, train mse = 1.8110e-04, test mse = 2.4571e-04
5 epoch 20000, train mse = 4.9779e-05, test mse = 1.0733e-04
6 epoch 25000, train mse = 2.9018e-05, test mse = 8.0896e-05
7 epoch 30000, train mse = 2.2734e-05, test mse = 7.1012e-05
8 epoch 35000, train mse = 1.9923e-05, test mse = 6.6513e-05
9 epoch 40000, train mse = 1.8257e-05, test mse = 6.4073e-05
10 epoch 45000, train mse = 3.8141e-05, test mse = 7.7431e-05
11 epoch 50000, train mse = 1.6273e-05, test mse = 6.1531e-05
12 epoch 55000, train mse = 1.4964e-05, test mse = 6.0510e-05
13 epoch 60000, train mse = 1.6612e-05, test mse = 6.3971e-05
14 epoch 65000, train mse = 1.3681e-05, test mse = 5.9280e-05
15 epoch 70000, train mse = 1.3034e-05, test mse = 5.8731e-05
16 epoch 75000, train mse = 1.2594e-05, test mse = 5.8321e-05
17 epoch 80000, train mse = 1.3746e-05, test mse = 5.7994e-05
18 epoch 85000, train mse = 1.2245e-05, test mse = 5.7767e-05
19 epoch 90000, train mse = 1.1703e-05, test mse = 5.7084e-05
20 epoch 95000, train mse = 1.1200e-05, test mse = 5.6436e-05
21 epoch 100000, train mse = 1.0960e-05, test mse = 5.6204e-05
22 epoch 105000, train mse = 1.0817e-05, test mse = 5.6085e-05
23 epoch 110000, train mse = 1.0642e-05, test mse = 5.5935e-05
24 epoch 115000, train mse = 1.0502e-05, test mse = 5.5807e-05
25 epoch 120000, train mse = 1.0380e-05, test mse = 5.5674e-05
26 epoch 125000, train mse = 1.0254e-05, test mse = 5.5560e-05
27 epoch 130000, train mse = 1.0141e-05, test mse = 5.5438e-05
28 epoch 135000, train mse = 1.2275e-05, test mse = 5.5313e-05
29 epoch 140000, train mse = 9.9387e-06, test mse = 5.5188e-05
30 epoch 145000, train mse = 9.9700e-06, test mse = 5.5180e-05

```

```

31 epoch 150000, train mse = 9.7648e-06, test mse = 5.4944e-05
32 epoch 155000, train mse = 9.6761e-06, test mse = 5.4836e-05
33 epoch 160000, train mse = 9.5983e-06, test mse = 5.4731e-05
34 epoch 165000, train mse = 9.5245e-06, test mse = 5.4629e-05
35 epoch 170000, train mse = 1.1408e-05, test mse = 6.1970e-05
36 epoch 175000, train mse = 9.3879e-06, test mse = 5.4430e-05
37 epoch 180000, train mse = 9.3258e-06, test mse = 5.4346e-05
38 epoch 185000, train mse = 9.2657e-06, test mse = 5.4255e-05
39 epoch 190000, train mse = 9.2100e-06, test mse = 5.4172e-05
40 epoch 195000, train mse = 9.1572e-06, test mse = 5.4099e-05
41 epoch 200000, train mse = 9.2161e-06, test mse = 5.4104e-05
42 epoch 205000, train mse = 9.0574e-06, test mse = 5.3947e-05
43 epoch 210000, train mse = 2.1596e-05, test mse = 6.0204e-05
44 epoch 215000, train mse = 8.9677e-06, test mse = 5.3808e-05
45 epoch 220000, train mse = 8.9262e-06, test mse = 5.3749e-05
46 epoch 225000, train mse = 8.8868e-06, test mse = 5.3693e-05
47 epoch 230000, train mse = 1.6735e-05, test mse = 6.4760e-05
48 epoch 235000, train mse = 8.8134e-06, test mse = 5.3581e-05
49 epoch 240000, train mse = 9.9345e-06, test mse = 5.5015e-05
50 epoch 245000, train mse = 8.7462e-06, test mse = 5.3480e-05
51 epoch 250000, train mse = 8.7548e-06, test mse = 5.3424e-05
52 epoch 255000, train mse = 8.6854e-06, test mse = 5.3384e-05
53 epoch 260000, train mse = 8.9200e-06, test mse = 5.3357e-05
54 epoch 265000, train mse = 9.0467e-06, test mse = 5.3305e-05
55 epoch 270000, train mse = 8.8819e-06, test mse = 5.3604e-05
56 epoch 275000, train mse = 8.5897e-06, test mse = 5.3236e-05
57 epoch 280000, train mse = 8.5883e-06, test mse = 5.3157e-05
58 epoch 285000, train mse = 8.5318e-06, test mse = 5.3125e-05
59 epoch 290000, train mse = 8.5077e-06, test mse = 5.3074e-05
60 epoch 295000, train mse = 8.7658e-06, test mse = 5.3229e-05
61 epoch 300000, train mse = 8.4653e-06, test mse = 5.2991e-05
62 epoch 305000, train mse = 8.4452e-06, test mse = 5.2938e-05
63 epoch 310000, train mse = 8.4256e-06, test mse = 5.2894e-05
64 epoch 315000, train mse = 8.4069e-06, test mse = 5.2853e-05
65 epoch 320000, train mse = 8.3887e-06, test mse = 5.2811e-05
66 epoch 325000, train mse = 8.3822e-06, test mse = 5.2769e-05
67 epoch 330000, train mse = 8.3547e-06, test mse = 5.2728e-05
68 epoch 335000, train mse = 8.3381e-06, test mse = 5.2691e-05
69 epoch 340000, train mse = 8.3226e-06, test mse = 5.2652e-05
70 epoch 345000, train mse = 8.3161e-06, test mse = 5.2611e-05
71 epoch 350000, train mse = 8.2919e-06, test mse = 5.2574e-05
72 epoch 355000, train mse = 8.2774e-06, test mse = 5.2536e-05
73 epoch 360000, train mse = 8.2632e-06, test mse = 5.2498e-05
74 epoch 365000, train mse = 8.2498e-06, test mse = 5.2462e-05
75 epoch 370000, train mse = 8.2370e-06, test mse = 5.2427e-05
76 epoch 375000, train mse = 1.1649e-05, test mse = 5.4893e-05
77 epoch 380000, train mse = 8.2260e-06, test mse = 5.2353e-05
78 epoch 385000, train mse = 8.1994e-06, test mse = 5.2323e-05
79 epoch 390000, train mse = 8.5365e-06, test mse = 5.2935e-05
80 epoch 395000, train mse = 8.1763e-06, test mse = 5.2259e-05
81 epoch 400000, train mse = 8.1652e-06, test mse = 5.2226e-05
82 epoch 405000, train mse = 8.1543e-06, test mse = 5.2194e-05
83 epoch 410000, train mse = 8.1439e-06, test mse = 5.2162e-05
84 epoch 415000, train mse = 8.1337e-06, test mse = 5.2130e-05
85 epoch 420000, train mse = 8.1236e-06, test mse = 5.2102e-05
86 epoch 425000, train mse = 8.1134e-06, test mse = 5.2073e-05
87 epoch 430000, train mse = 8.1042e-06, test mse = 5.2044e-05
88 epoch 435000, train mse = 8.0949e-06, test mse = 5.2015e-05
89 epoch 440000, train mse = 8.0857e-06, test mse = 5.1984e-05

```

```
90 epoch 445000, train mse = 8.0769e-06, test mse = 5.1956e-05
91 epoch 450000, train mse = 1.1278e-05, test mse = 5.6714e-05
92 epoch 455000, train mse = 8.9902e-06, test mse = 5.3759e-05
93 epoch 460000, train mse = 8.0521e-06, test mse = 5.1872e-05
94 epoch 465000, train mse = 8.0434e-06, test mse = 5.1846e-05
95 epoch 470000, train mse = 8.0364e-06, test mse = 5.1817e-05
96 epoch 475000, train mse = 8.0286e-06, test mse = 5.1789e-05
97 epoch 480000, train mse = 8.0213e-06, test mse = 5.1763e-05
98 epoch 485000, train mse = 8.0133e-06, test mse = 5.1738e-05
99 epoch 490000, train mse = 8.0065e-06, test mse = 5.1713e-05
100 epoch 495000, train mse = 8.3591e-06, test mse = 5.1846e-05
101
102 final train mse = 7.9922e-06
103 final test mse = 5.1661e-05
104
105 Total number of trainable parameters: 193
```

Listing 6: Output terminal for `bonus.py`