MTH 602 Scientific Machine Learning

Homework 6

11/17/2025

S. M. Mahfuzul Hasan

02181922

UMass | Dartmouth

I. PAPER & PENCIL WORK

1. Eq. 6.11 in the book

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

- 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,

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

  where,

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

  and,

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

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

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

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

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

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

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

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

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

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,

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

Here,

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

And,

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

We know,

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

Using eq. (13) in (12),

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

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

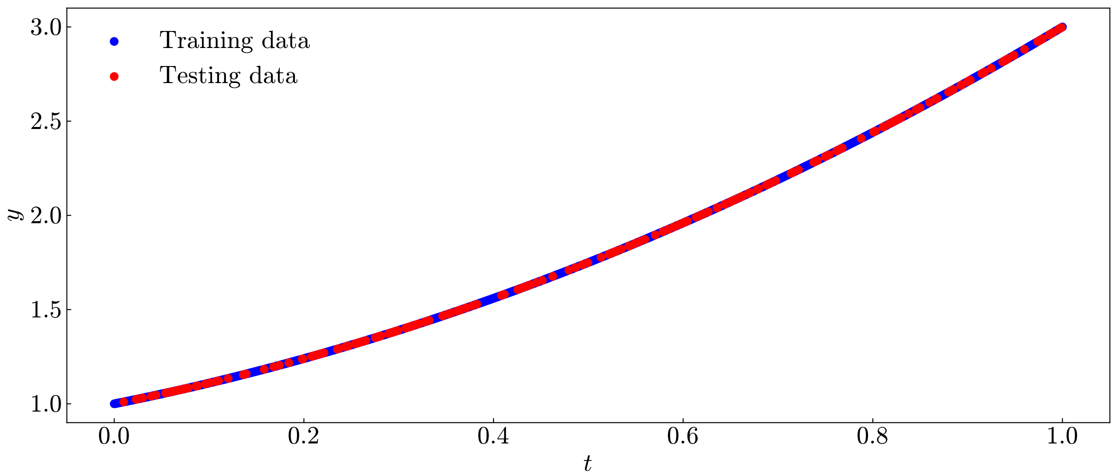


Figure 1: Training and testing data generated using `generate_data()`.

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:

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

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 α 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.

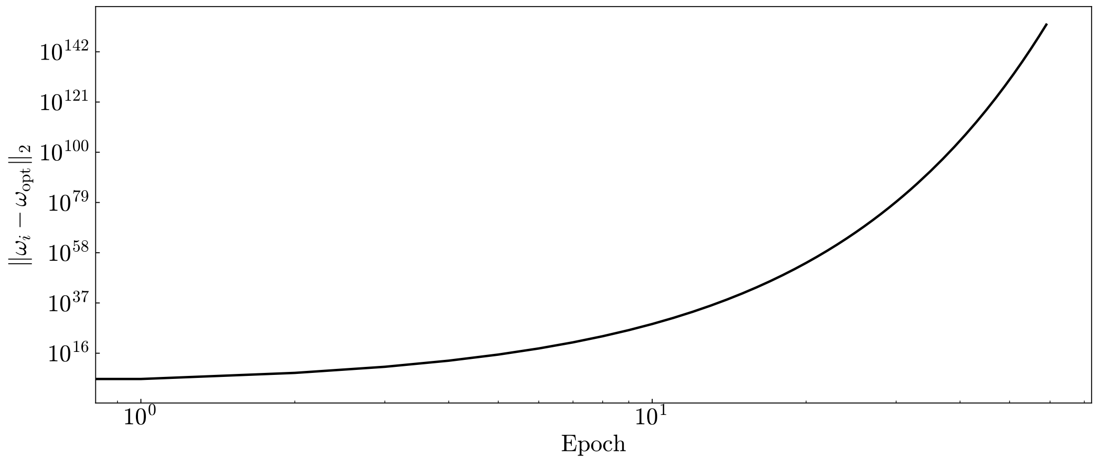
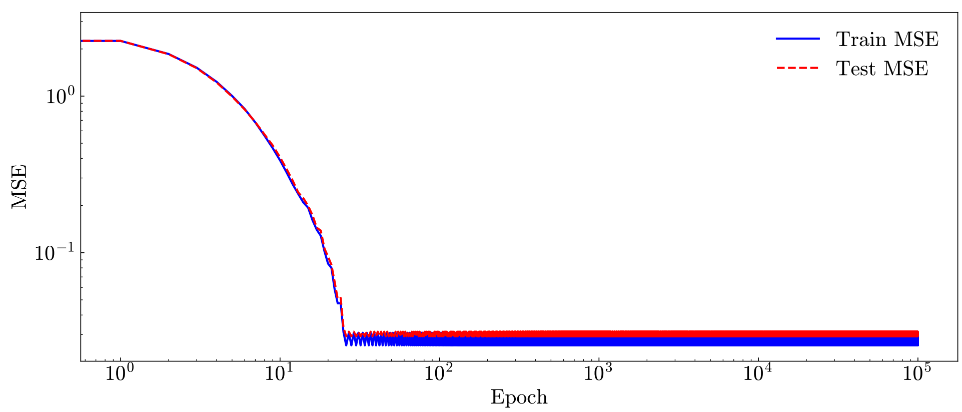
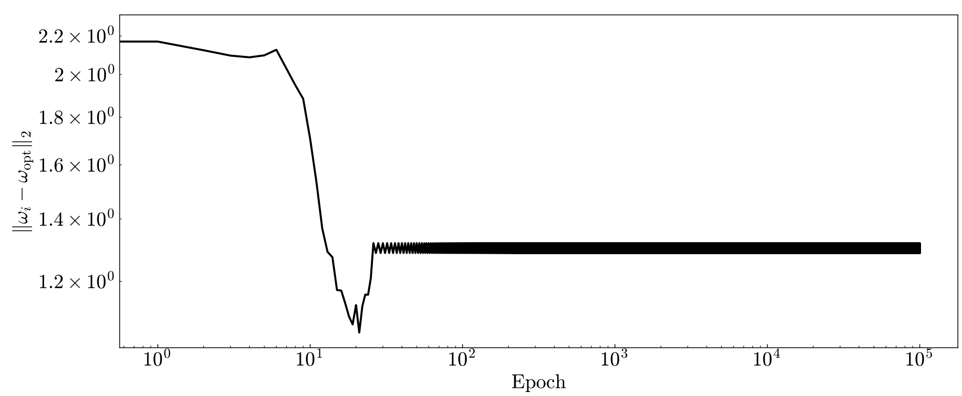


Figure 2: $L2$ norm of the weights for unscaled data.

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 α as its argument. With this, SGD did not diverge and converges till certain point before getting stalled.



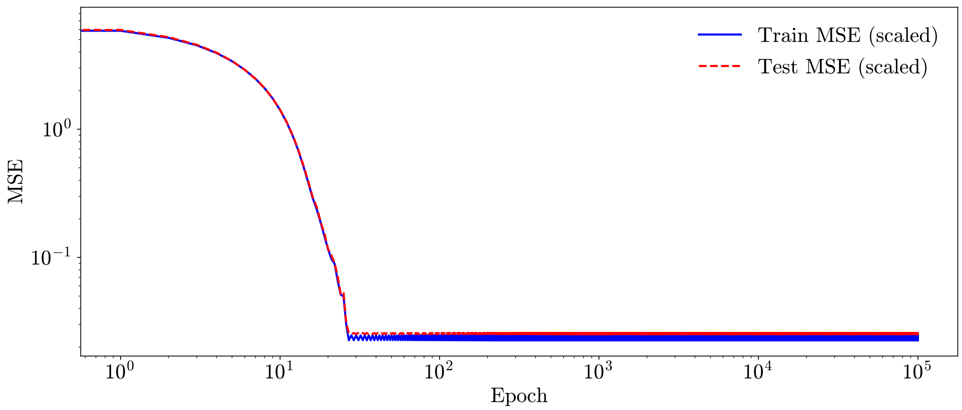
(a)



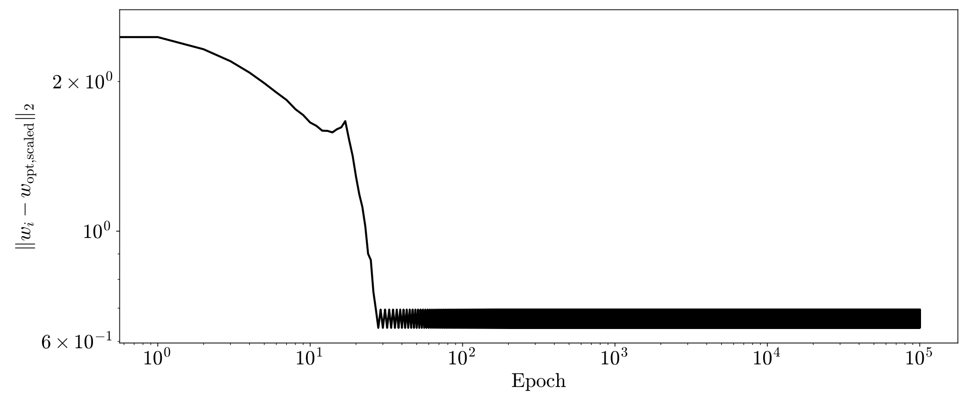
(b)

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.



(a)



(b)

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:

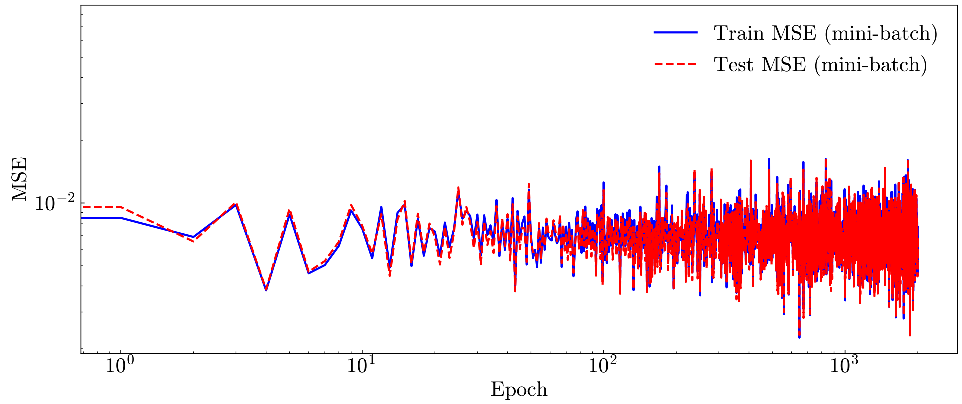
$$\vec{\omega}_{\text{unscaled}} = \{1.59415786, 0.41523422, 0.41608667, 0.78784401\}$$
$$\vec{\omega}_{\text{scaled}} = \{1.64692162, 0.76950196, 0.65296745, 0.50743949\}$$

Final computed distance:

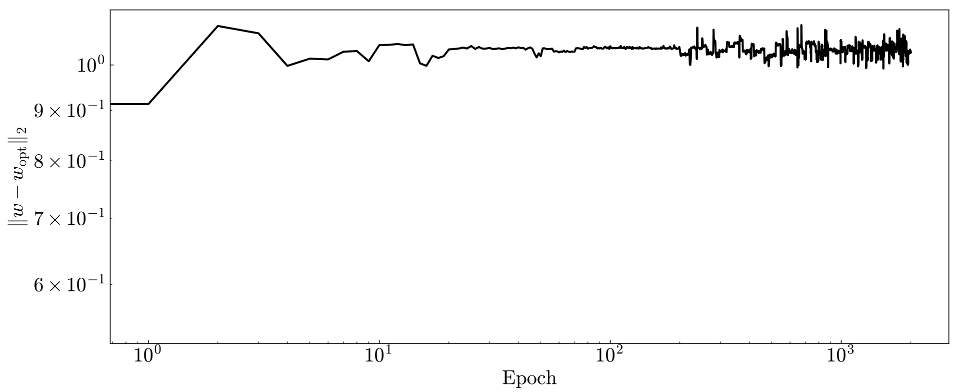
$$\|\omega_i - \omega_{\text{opt}}\|_{2_{\text{unscaled}}} = 1.287, \qquad \|\omega_i - \omega_{\text{opt}}\|_{2_{\text{scaled}}} = 0.695$$

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.



(a)



(b)

Figure 5: (a) MSE of loss and (b) $l2$ 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 $l2$ norm in fig. 5.

Fig. 6 more or less mimics fig. 5. Only the exception is that $l2$ 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.

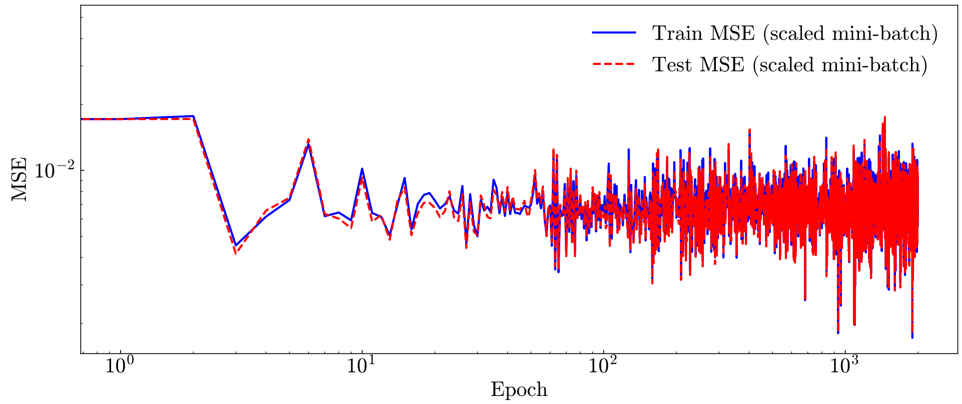
Final computed weights via SGD:

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

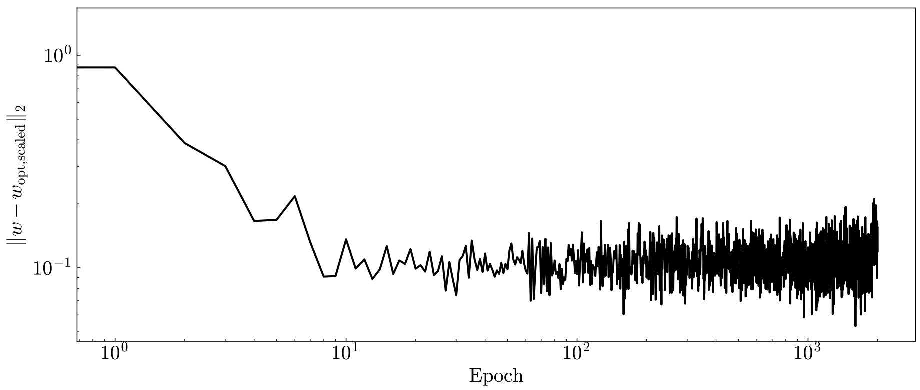
Final distance computed:

$$\|\omega_i - \omega_{\text{opt}}\|_{2_{\text{unscaled}}} = 1.029, \quad \|\omega_i - \omega_{\text{opt}}\|_{2_{\text{scaled}}} = 0.154$$

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



(a)



(b)

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.

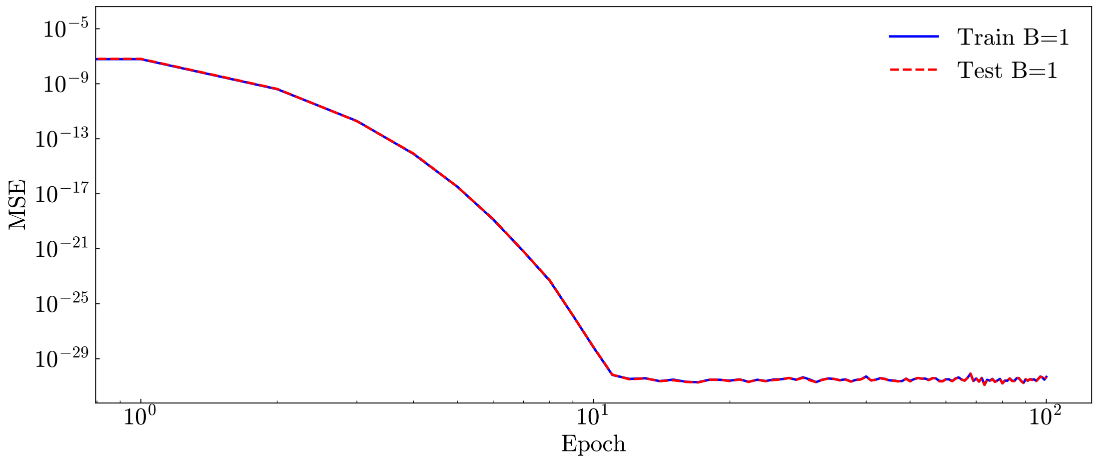


Figure 7: MSE loss for $B = 1$.

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.

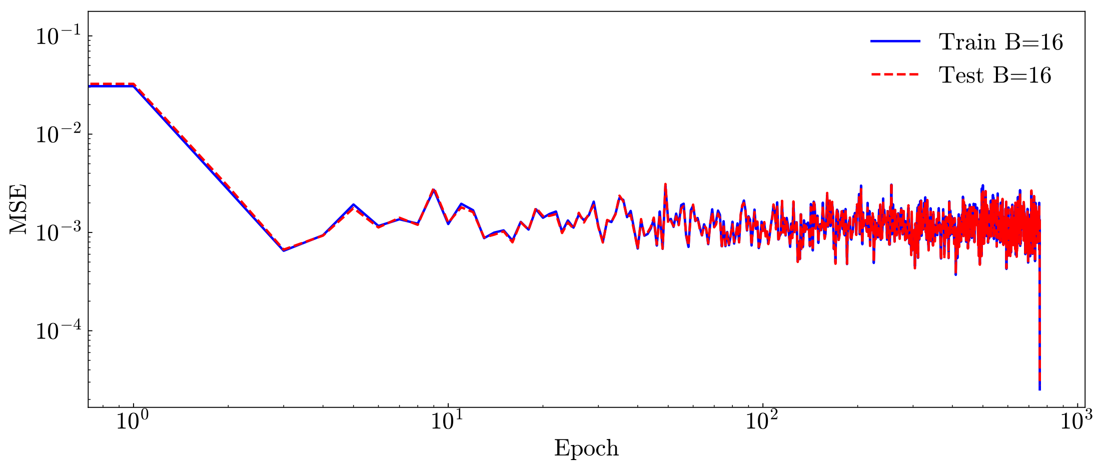


Figure 8: MSE loss for $B = 16$.

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

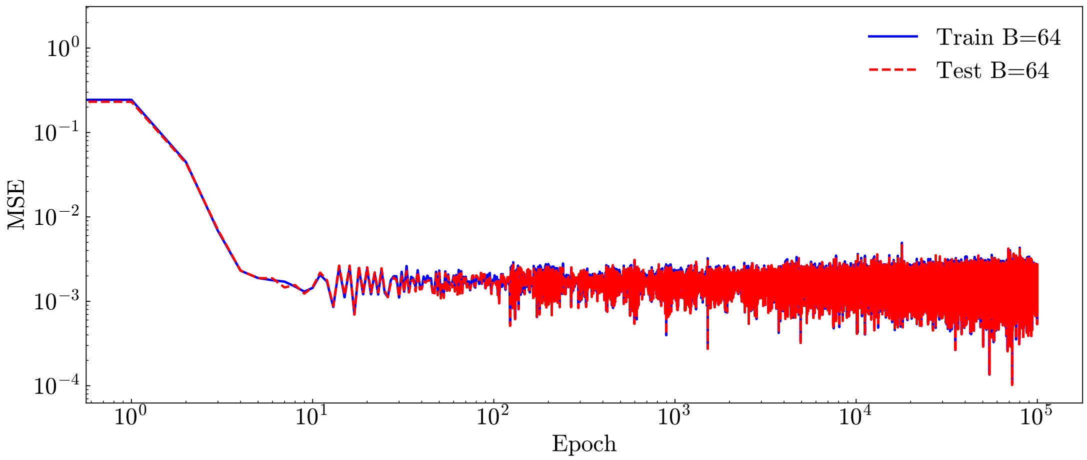


Figure 9: MSE loss for $B = 64$.

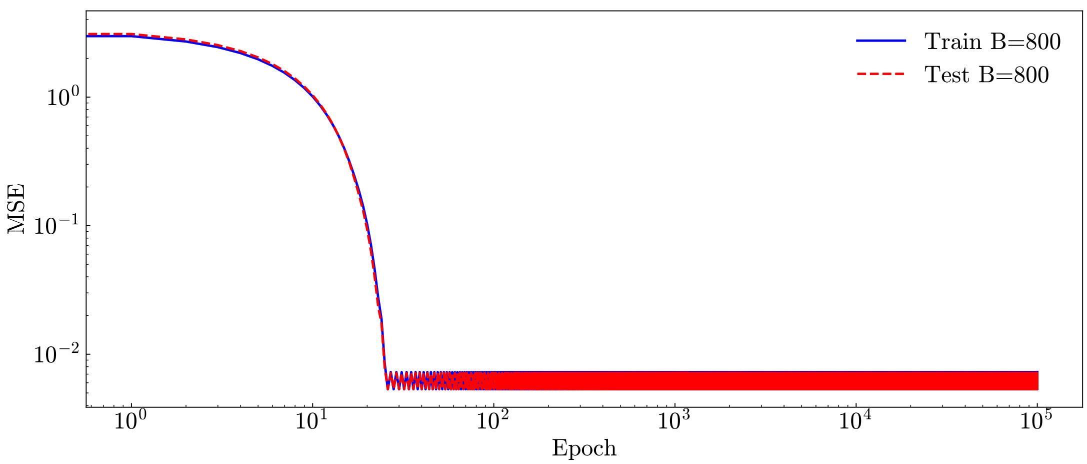


Figure 10: MSE loss for $B = N_{train}$.

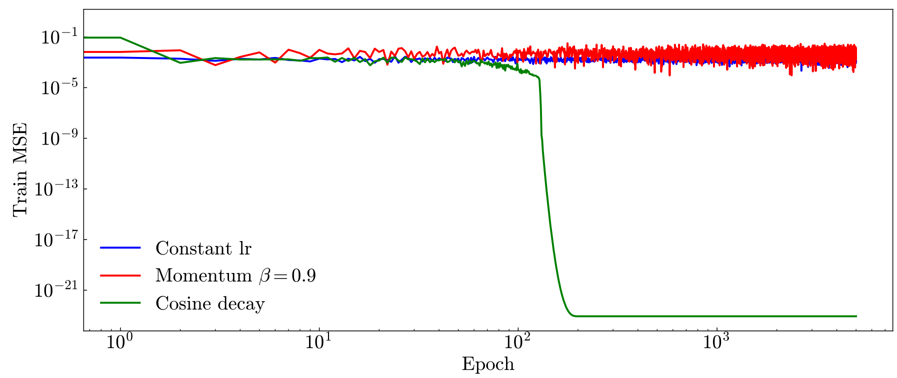
10^{-2}, which can be attributed to the difference in random initialization of the weights. This one gives the highest MSE in training and testing, but it has mild oscillations compared to $B = 16$ and 64.

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

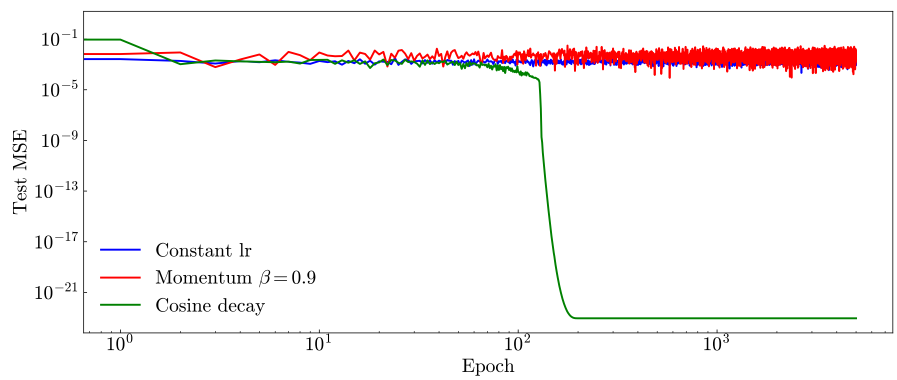
$B = 1$ takes the least time to reach the target MSE, while $B = 16$ does not truly reaches the target MSE but gets lucky with its big spike. $B = 64$ and 800 never reaches the target MSE. From the avg. time required to complete each epoch, it is clear that $B = 1$ is the slowest, while $B = 800$ is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

G. Learning-rate study

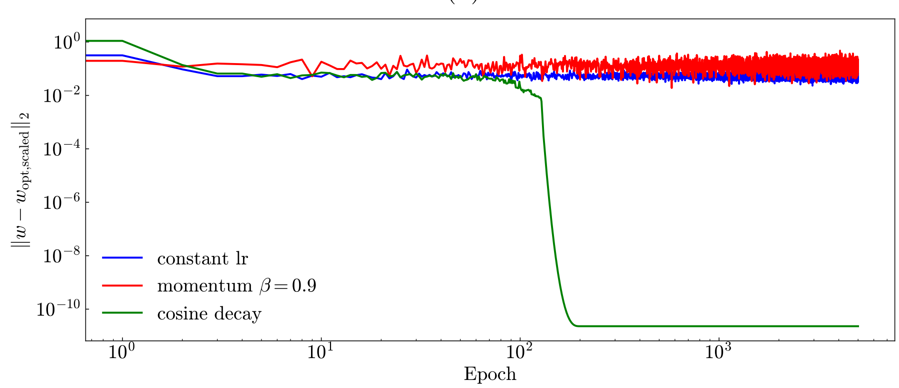
For $B = 32$, three initial learning rate $\alpha = \{0.1, 0.01, 0.001\}$ were used to compare the stability and performance of three variants of SGD. Number of epochs was set to 5000 for all of them.



(a)



(b)

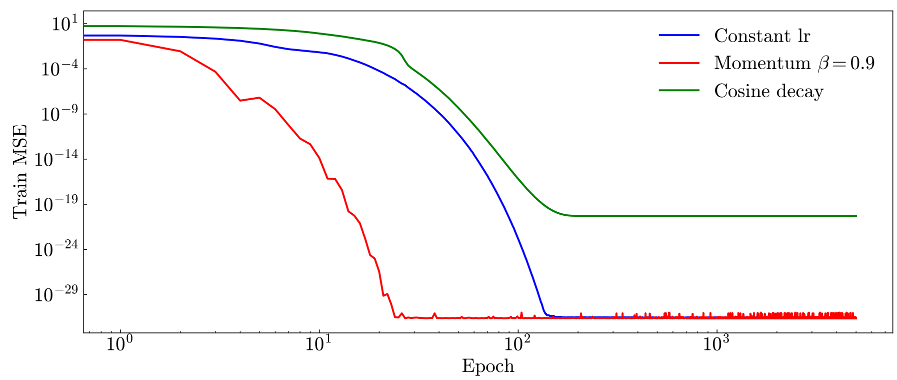


(c)

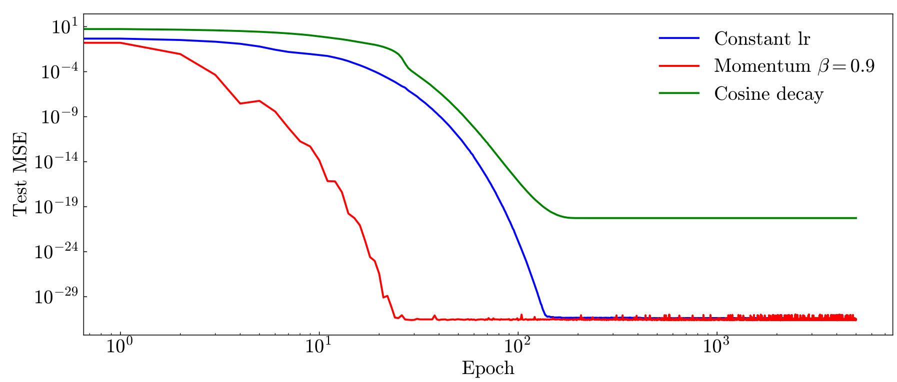
Figure 11: (a) Training MSE, (b) testing MSE, and (c) $l2$ norm of distance from optimizer's solution to closed-form solution for initial learning rate $\alpha = 0.1$.

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 $l2$ norm becomes very small

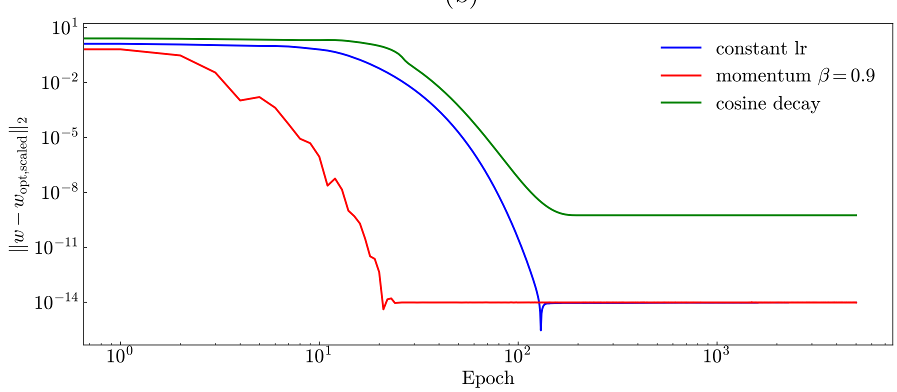
(~10^{-10}). The stability is great with smooth decay as well.



(a)



(b)

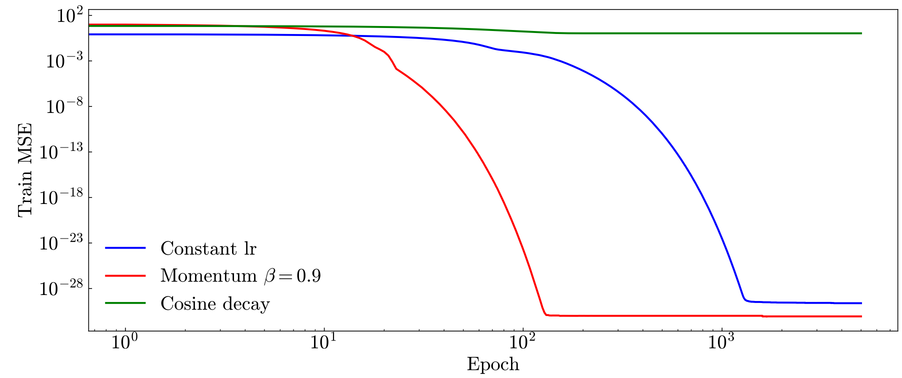


(c)

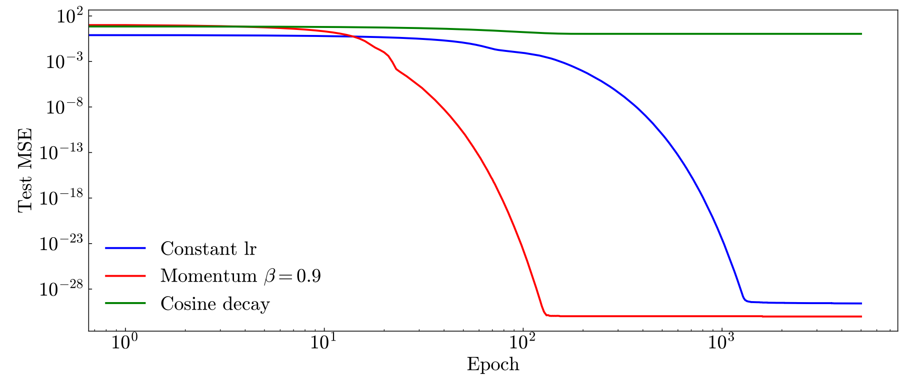
Figure 12: (a) Training MSE, (b) testing MSE, and (c) $l2$ norm of distance from optimizer's solution to closed-form solution for initial learning rate $\alpha = 0.01$.

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 $l2$ norm of ~10^{-14}. However, momentum SGD reaches there almost 10x faster. Cosine decay, though showing good MSE and $l2$ norm, can not perform as good as the other two.

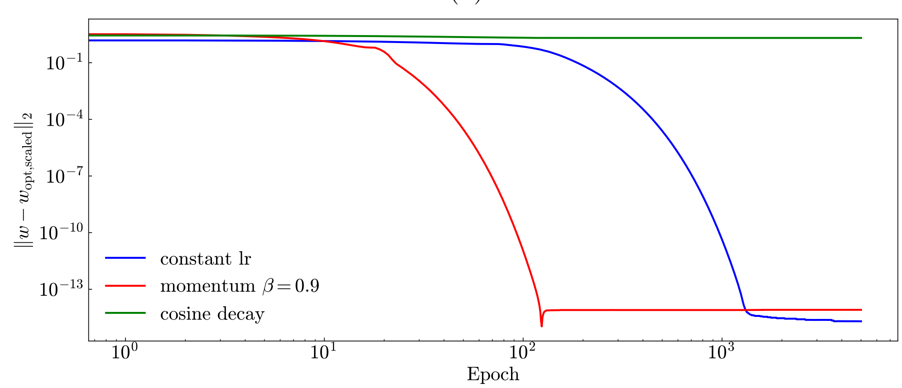
For $\alpha = 0.001$, constant lr and momentum SGD retain their performance even with greater stability.



(a)



(b)



(c)

Figure 13: (a) Training MSE, (b) testing MSE, and (c) $l2$ norm of distance from optimizer's solution to closed-form solution for initial learning rate $\alpha = 0.001$.

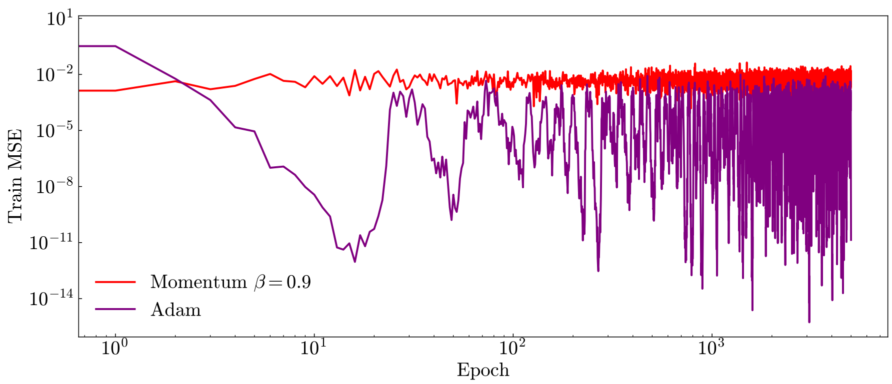
The small noise that was present in $\alpha = 0.01$ vanishes for both of them. Cosine decay shows worst performance as evident in fig. 13. One thing to note is that, Cosine decay does not show instability with oscillations in any of the cases.

So, it can be concisely said that

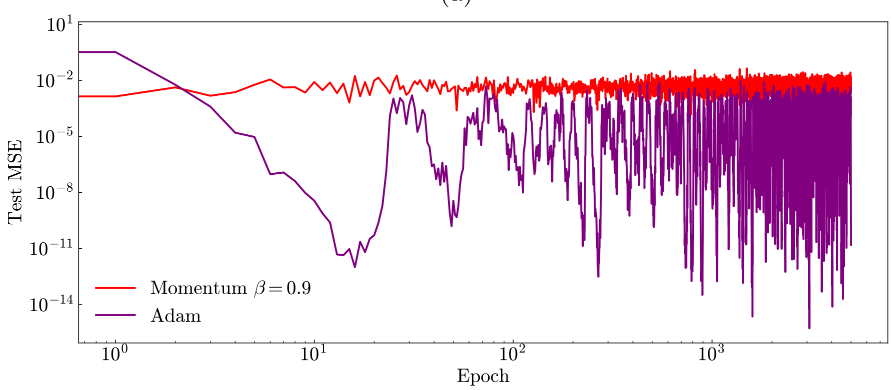
- Constant lr and momentum accelerated SGD perform better and show better stability as learning rate is decreased. This also explains why in previous sections, with learning rate $\alpha = 0.2$, the constant lr performed so bad. If the learning rate is not sufficiently small, the convergence will not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

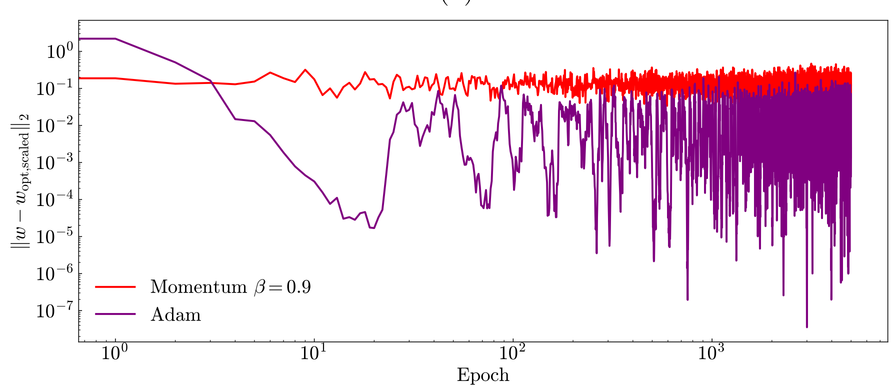
H. SGD vs another optimizer



(a)



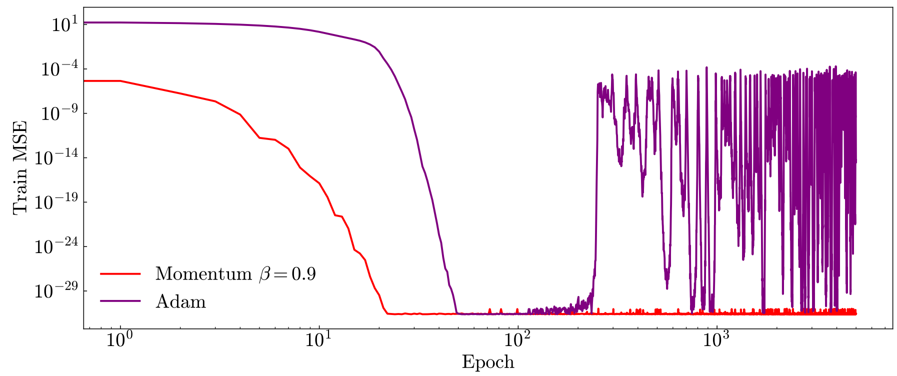
(b)



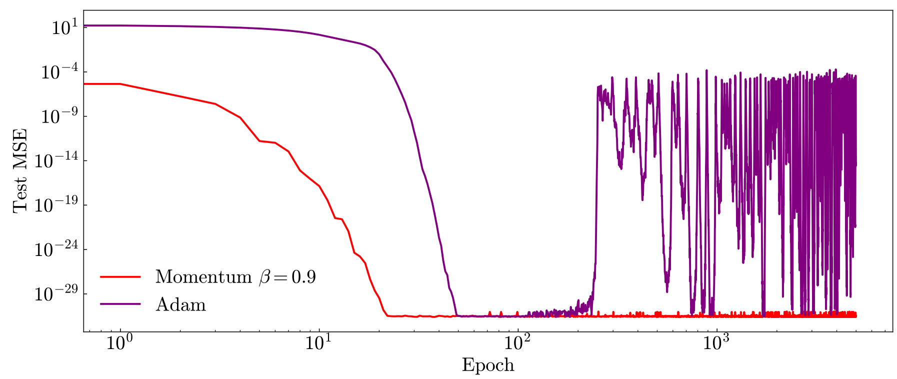
(c)

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 $\alpha = 0.1$.

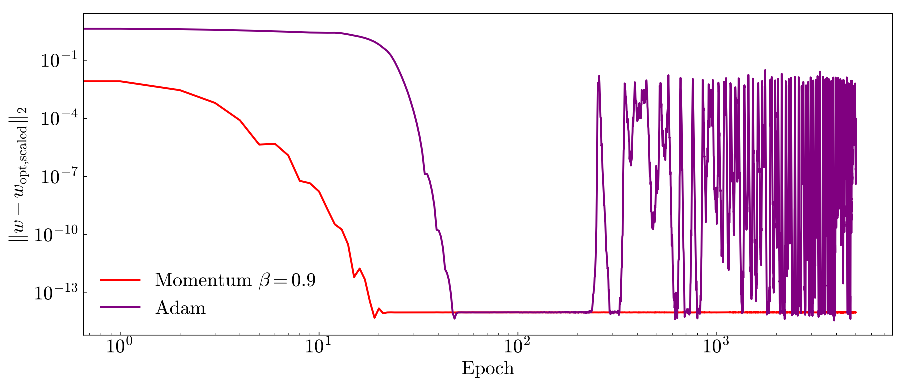
Momentum accelerated SGD is compared with `adam` for the learning rates used in the previous section. `adam` shows instability in all the three learning rates as can be seen from fig. 14, 15 and



(a)



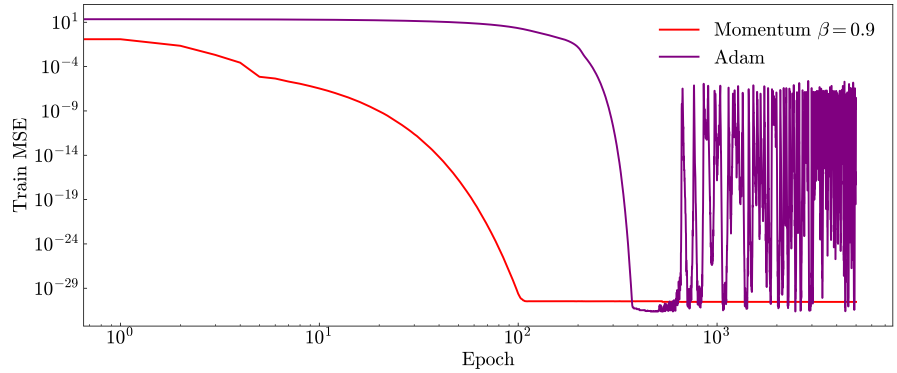
(b)



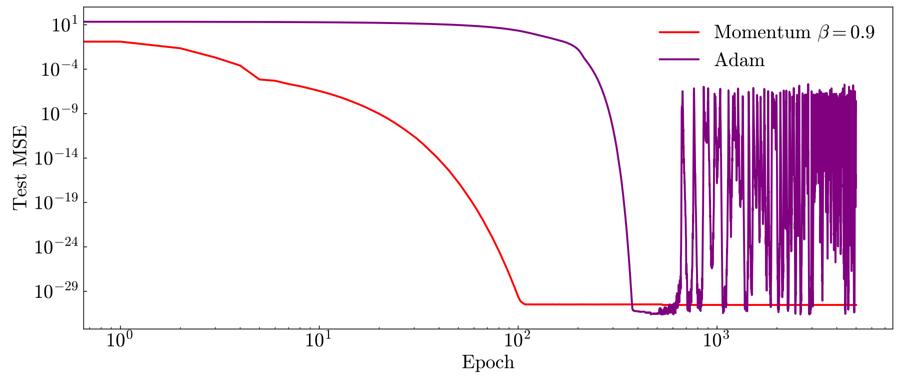
(c)

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 $\alpha = 0.01$.

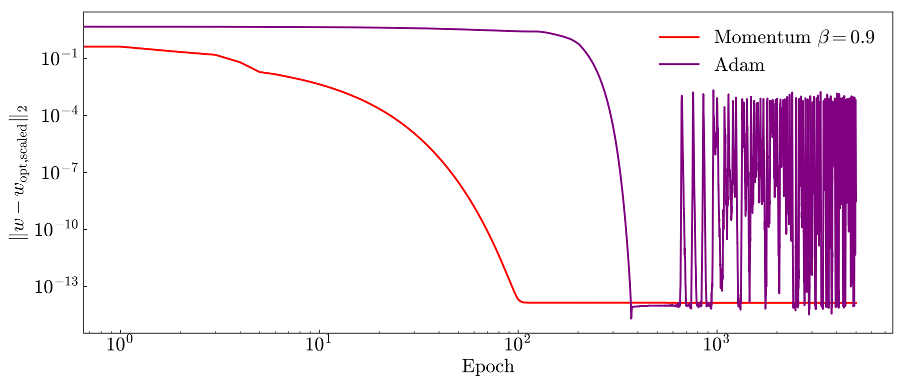
16, while momentum SGD progressively becomes stable with lower learning rate. The performance of momentum SGD also shows improvement with low learning rate. `adam`, despite achieving a very low MSE that competes well with momentum SGD momentarily, it shows instability by shooting up after the plateau. 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.



(a)



(b)



(c)

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.

```python
import jax
jax.config.update("jax_enable_x64", True)
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from sklearn.model_selection import train_test_split
import jax.numpy as jnp
import optax
import time

# ===== C. Dataset and setup =====

def generate_data(scale_noise, N, seed=None):
    rng = np.random.default_rng(seed)
    t = np.linspace(0.0, 1.0, N)
    s = 1 + t + t*t
    n = rng.normal(loc=0.0, scale=scale_noise, size=N)
    y = s + n
    return t, y, s, n

t_all, y_all, s_all, n_all       = generate_data(0.0, 1000, seed=0)
t_train, t_test, y_train, y_test = train_test_split(t_all,y_all,test_size=0.2,
    random_state=42)

# parameters for plotting
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = 'cmr10'
plt.rcParams['mathtext.fontset'] = 'cm'
plt.rcParams['font.size'] = 20
mpl.rcParams['axes.unicode_minus'] = False
plt.rcParams['axes.formatter.use_mathtext'] = True

fig, ax = plt.subplots(figsize=(15, 6))
ax.scatter(t_train, y_train, color="blue", label="Training data")
ax.scatter(t_test, y_test, color="red", label="Testing data")
plt.xlabel(r"$t$")
plt.ylabel(r"$y$")
plt.legend(loc="upper left", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig(f"gen_data.pdf", dpi=1080)
plt.show()

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

print("\n===== D. Write and test the model and loss functions =====")

# function for Vandermonde matrix A
def vandermonde(t, M):
    t = jnp.asarray(t)
    powers = jnp.arange(M + 1)
    return t[:, None] ** powers[None, :]


# 1. function for model $A\vec{\omega}$
def model(w, A):
    return A @ w

# 2. function for loss function ($\lambda = 0$)
```

```python
def loss(w, A, y):
    y_pred = model(w, A)
    r = y_pred - y
    return jnp.dot(r, r)

# 3. function for MSE
def mse(w, A, y):
    return loss(w, A, y) / len(y)

# sanity test
print("\nSANITY TEST")

# noise-free data generation
N = 1000
t, y, s, _ = generate_data(scale_noise=0.0, N=N, seed=1)

# degree of polynomial and Vandermonde construction
M = 2
A = vandermonde(t, M)

# True weights for polynomial
w_true = jnp.array([1.0, 1.0, 1.0])

# wrong weights to check for high loss
w_bad = jnp.array([1.0, 0.5, 0.5])

# compute true losses and MSE
loss_true = loss(w_true, A, y)
mse_true = mse(w_true, A, y)

# compute losses and MSE for wrong weights
loss_bad = loss(w_bad, A, y)
mse_bad = mse(w_bad, A, y)

print("\nTrue weights: w_true =", w_true)
print("Loss(w_true) =", float(loss_true))
print("MSE(w_true)  =", float(mse_true))

print("\nBad weights: w_bad =", w_bad)
print("Loss(w_bad)  =", float(loss_bad))
print("MSE(w_bad)   =", float(mse_bad))

# train and test MSE
A_train = vandermonde(t_train, M)
A_test  = vandermonde(t_test,  M)
train_mse_true = mse(w_true, A_train, y_train)
test_mse_true  = mse(w_true, A_test,  y_test)

print("\nTraining MSE (true weights) =", float(train_mse_true))
print("Testing  MSE (true weights) =", float(test_mse_true))

# ====== E. SGD optimization ======

print("\n====== E. SGD optimization ======")

# 1. Full batch SGD

M = 3
lam = 0
```

```python
alpha = 0.2
epochs = 100000
B = len(t_train)

# construct train and test Vandermonde matrices
A_train = vandermonde(t_train, M)
A_test  = vandermonde(t_test,  M)
y_train_jnp = jnp.asarray(y_train)
y_test_jnp  = jnp.asarray(y_test)

# closed form solution
AtA = A_train.T @ A_train
Aty = A_train.T @ y_train_jnp
w_opt = jnp.linalg.solve(AtA + lam*jnp.eye(M+1), Aty)

print("\nClosed-form: omega_opt = ", w_opt)

def vandermonde_scaled(t, M):
    t = jnp.asarray(t)
    t = 2*(t - 0.5)         # scale to [-1, 1]
    powers = jnp.arange(M + 1)
    return t[:, None] ** powers[None, :]

# compute Hessian $H = 2A^TA$
hessian = 2 * (A_train.T @ A_train)

# compute the eigenvalues of the Hessian
eigen_hessian = np.linalg.eigvals(hessian)

# maximum eigenvalue of the Hessian
max_eigen_hessian = np.max(eigen_hessian)

# learning rate upper bound
alpha_max = 2.0 / max_eigen_hessian

print("\nMaximum eigenvalue of Hessian and upper bound of learning rate:")
print(f"\nlambda_max(H) = {max_eigen_hessian:.3e}")
print(f"alpha_max = 2 / lambda_max(H) = {alpha_max:.3e}")

# SGD diverges or converges?
if alpha > alpha_max:
    print(f"\nSGD should theoretically diverge as alpha = {alpha} is greater than
        alpha_max = {alpha_max:.3e}")
    print(f"\nLet's try with scaled Vandermonde...")

    # scaled Vandermonde
    A_train = vandermonde_scaled(t_train, M)
    A_test  = vandermonde_scaled(t_test,  M)

    # compute Hessian $H = 2A^TA$
    hessian = 2 * (A_train.T @ A_train)

    # compute the eigenvalues of the Hessian
    eigen_hessian = np.linalg.eigvals(hessian)

    # maximum eigenvalue of the Hessian
    max_eigen_hessian = np.max(eigen_hessian)

    # learning rate upper bound
```

```
    alpha_max = 2.0 / max_eigen_hessian

    print("\nAfter scaling...")
    print(f"\nlambda_max(H) = {max_eigen_hessian:.3e}")
    print(f"alpha_max = 2 / lambda_max(H) = {alpha_max:.3e}")
    if alpha > alpha_max:
        print(f"\nSGD should theoretically still diverge.")
    else:
        print(f"\nSGD should theoretically converge")
else:
    print(f"\nSGD should theoretically converge as alpha = {alpha} is less than or
         equal to alpha_max = {alpha_max:.3e}.")

# initialization of weights
key = jax.random.PRNGKey(0)
w = jax.random.normal(key, (M+1,))

# Optax SGD with gradient clipping
#optimizer = optax.sgd(learning_rate=alpha)

optimizer = optax.chain(
    optax.clip(0.5),  # Lower clipping threshold for better gradient stability
    optax.sgd(learning_rate=alpha)
)

opt_state = optimizer.init(w)

# update step
@jax.jit
def step(w, opt_state, A, y):
    grads = jax.grad(loss)(w, A, y)
    updates, opt_state = optimizer.update(grads, opt_state)
    w = optax.apply_updates(w, updates)
    return w, opt_state

# initialize train, test, distance arrays
train_mse_hist, test_mse_hist, dist_hist = [], [], []

# full batch training loop
for epoch in range(epochs):
    w, opt_state = step(w, opt_state, A_train, y_train_jnp)

    # compute real losses
    train_mse_hist.append(mse(w, A_train, y_train_jnp))
    test_mse_hist.append(mse(w, A_test, y_test_jnp))
    dist_hist.append(jnp.linalg.norm(w - w_opt))

print("\nFinal SGD weights:", w)
print("Distance to closed-form =", float(dist_hist[-1]))

# parameters for plotting
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = 'cmr10'
plt.rcParams['mathtext.fontset'] = 'cm'
plt.rcParams['font.size'] = 20
mpl.rcParams['axes.unicode_minus'] = False
plt.rcParams['axes.formatter.use_mathtext'] = True

# plot for training & testing MSE vs. epoch
```

```
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(train_mse_hist, "b-", lw=2, label="Train MSE")
ax.loglog(test_mse_hist, "r--", lw=2, label="Test MSE")
plt.xlabel("Epoch")
plt.ylabel("MSE")
plt.legend(loc="upper right", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig(f"mse_wclip.pdf", dpi=1080)
plt.show()

# plot L2 norm of error vs. epoch
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(dist_hist, "k-", lw=2)
plt.xlabel("Epoch")
plt.ylabel(r"$\| \omega_i - \omega_{\text{opt}} \|_2$")
#plt.legend(loc="upper left", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig(f"l2_wclip.pdf", dpi=1080)
plt.show()

# now try with scaled data

# scaled Vandermonde
A_train_s = vandermonde_scaled(t_train, M)
A_test_s  = vandermonde_scaled(t_test,  M)

# closed form solution in scaled basis
AtA_s = A_train_s.T @ A_train_s
Aty_s = A_train_s.T @ y_train_jnp
w_opt_s = jnp.linalg.solve(AtA_s + lam*jnp.eye(M+1), Aty_s)

print("\nClosed-form solution (scaled basis):", w_opt_s)

# new random weights
key2 = jax.random.PRNGKey(123)
w2 = jax.random.normal(key2, (M+1,))

# optimizer with clipping
optimizer2 = optax.chain(
    optax.clip(0.5),  # Lower clipping threshold for better gradient stability
    optax.sgd(learning_rate=alpha)
)
opt_state2 = optimizer2.init(w2)

# new step function for scaled run
@jax.jit
def step2(w, opt_state, A, y):
    grads = jax.grad(loss)(w, A, y)
    updates, opt_state = optimizer2.update(grads, opt_state)
    w = optax.apply_updates(w, updates)
    return w, opt_state

# histories
train_mse_hist_2 = []
test_mse_hist_2  = []
dist_hist_2      = []

# training loop
for epoch in range(epochs):
```

```python
    w2, opt_state2 = step2(w2, opt_state2, A_train_s, y_train_jnp)

    train_mse_hist_2.append(mse(w2, A_train_s, y_train_jnp))
    test_mse_hist_2.append(mse(w2, A_test_s, y_test_jnp))
    dist_hist_2.append(jnp.linalg.norm(w2 - w_opt_s))

print("\nFinal SGD weights (scaled):", w2)
print("Distance to scaled closed-form:", float(dist_hist_2[-1]))

# plot scaled training & testing MSE vs. epoch
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(train_mse_hist_2, "b-", lw=2, label="Train MSE (scaled)")
ax.loglog(test_mse_hist_2, "r--", lw=2, label="Test MSE (scaled)")
plt.xlabel("Epoch")
plt.ylabel("MSE")
plt.legend(loc="upper right", frameon=False)
plt.savefig("mse_scaled.pdf", dpi=1080)
plt.show()

# plot scaled L2 norm of error vs. epoch
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(dist_hist_2, "k-", lw=2)
plt.xlabel("Epoch")
plt.ylabel(r"$\| w_i - w_{\text{opt,scaled}} \|_2$")
plt.savefig("l2_scaled.pdf", dpi=1080)
plt.show()

# 2. Mini-batch SGD

# unscaled mini-batch
epochs_mb = 2000
batch_size = 32
num_batches = len(t_train)

# shuffle indices each epoch
def get_batches(A, y, batch_size):
    N = len(y)
    perm = np.random.permutation(N)
    for i in range(0, N, batch_size):
        idx = perm[i:i+batch_size]
        yield A[idx], y[idx]

# fresh weights
key = jax.random.PRNGKey(420)
w_mb = jax.random.normal(key, (M+1,))

# optimizer
optimizer_mb = optax.chain(
    optax.clip(0.5),  # Lower clipping threshold for better gradient stability
    optax.sgd(learning_rate=alpha)
)
opt_state_mb = optimizer_mb.init(w_mb)

# mini-batch training step
@jax.jit
def mb_step(w, opt_state, A_batch, y_batch):
    grads = jax.grad(loss)(w, A_batch, y_batch)
    updates, opt_state = optimizer_mb.update(grads, opt_state)
    w = optax.apply_updates(w, updates)
```

```
    return w, opt_state

# history arrays
train_mse_hist_mb = []
test_mse_hist_mb  = []
dist_hist_mb      = []

# mini-batch SGD loop
for epoch in range(epochs_mb):
    # iterate over randomized batches
    for A_b, y_b in get_batches(A_train, y_train_jnp, batch_size):
        w_mb, opt_state_mb = mb_step(w_mb, opt_state_mb, A_b, y_b)
    # record metrics each epoch
    train_mse_hist_mb.append(mse(w_mb, A_train, y_train_jnp))
    test_mse_hist_mb.append(mse(w_mb, A_test,  y_test_jnp))
    dist_hist_mb.append(jnp.linalg.norm(w_mb - w_opt))
    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Train MSE = {train_mse_hist_mb[-1]:.6f}, Dist = {
            dist_hist_mb[-1]:.3e}")

print("\nFinal mini-batch SGD weights:", w_mb)
print("Distance to closed-form =", float(dist_hist_mb[-1]))

# plots
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(train_mse_hist_mb, "b-", lw=2, label="Train MSE (mini-batch)")
ax.loglog(test_mse_hist_mb,  "r--", lw=2, label="Test MSE (mini-batch)")
plt.xlabel("Epoch")
plt.ylabel("MSE")
plt.legend(loc="upper right", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("mse_mb_unscaled.pdf", dpi=1080)
plt.show()

fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(dist_hist_mb, "k-", lw=2)
plt.xlabel("Epoch")
plt.ylabel(r"$\| w - w_{\text{opt}} \|_2$")
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("l2_mb_unscaled.pdf", dpi=1080)
plt.show()

# scaled mini-batch
epochs_mb_s = 2000
batch_size_s = 32

# build scaled Vandermonde
A_train_s = vandermonde_scaled(t_train, M)
A_test_s  = vandermonde_scaled(t_test,  M)

# closed form solution in scaled basis
AtA_s = A_train_s.T @ A_train_s
Aty_s = A_train_s.T @ y_train_jnp
w_opt_s = jnp.linalg.solve(AtA_s + lam*jnp.eye(M+1), Aty_s)

print("\nClosed-form solution (scaled basis) for mini-batch:", w_opt_s)

# fresh weights for scaled mini-batch
key_s = jax.random.PRNGKey(777)
```

```
w_mb_s = jax.random.normal(key_s, (M+1,))

# optimizer (same learning rate and clipping)
optimizer_mb_s = optax.chain(
    optax.clip(0.5),
    optax.sgd(learning_rate=alpha)
)
opt_state_mb_s = optimizer_mb_s.init(w_mb_s)

# batching function for scaled data
def get_batches_s(A, y, batch_size):
    N = len(y)
    perm = np.random.permutation(N)
    for i in range(0, N, batch_size):
        idx = perm[i:i+batch_size]
        yield A[idx], y[idx]

# JIT-compiled update for scaled mini-batch
@jax.jit
def mb_step_s(w, opt_state, A_batch, y_batch):
    grads = jax.grad(loss)(w, A_batch, y_batch)
    updates, opt_state = optimizer_mb_s.update(grads, opt_state)
    w = optax.apply_updates(w, updates)
    return w, opt_state

# histories
train_mse_hist_mb_s = []
test_mse_hist_mb_s  = []
dist_hist_mb_s      = []

# training loop
for epoch in range(epochs_mb_s):
    for A_b, y_b in get_batches_s(A_train_s, y_train_jnp, batch_size_s):
        w_mb_s, opt_state_mb_s = mb_step_s(w_mb_s, opt_state_mb_s, A_b, y_b)

    # record per-epoch progress
    train_mse_hist_mb_s.append(mse(w_mb_s, A_train_s, y_train_jnp))
    test_mse_hist_mb_s.append(mse(w_mb_s, A_test_s, y_test_jnp))
    dist_hist_mb_s.append(jnp.linalg.norm(w_mb_s - w_opt_s))

    if epoch % 100 == 0:
        print(f"[SCALED] Epoch {epoch}: Train MSE = {train_mse_hist_mb_s[-1]:.6f},
             Dist = {dist_hist_mb_s[-1]:.3e}")

print("\nFinal mini-batch SGD weights (scaled):", w_mb_s)
print("Distance to scaled closed-form =", float(dist_hist_mb_s[-1]))

# plots
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(train_mse_hist_mb_s, "b-", lw=2, label="Train MSE (scaled mini-batch)")
ax.loglog(test_mse_hist_mb_s,  "r--", lw=2, label="Test MSE (scaled mini-batch)")
plt.xlabel("Epoch")
plt.ylabel("MSE")
plt.legend(loc="upper right", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("mse_mb_scaled.pdf", dpi=1080)
plt.show()

fig, ax = plt.subplots(figsize=(15, 6))
```

```python
ax.loglog(dist_hist_mb_s, "k-", lw=2)
plt.xlabel("Epoch")
plt.ylabel(r"$\| w - w_{\text{opt,scaled}} \|_2$")
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("l2_mb_scaled.pdf", dpi=1080)
plt.show()

# F. Batch size study

print("\n===== F. Batch size study =====")

# parameters
batch_sizes = [1, 16, 64, len(t_train)]
epochs_F = 100000  # maximum epoch count
alpha_F = 0.1
M = 3
target_mse = 1e-4  # target

results = {}

# early stopping parameters
patience = 5  # no. of epochs with no significant improvement before stopping
min_delta = 1e-5  # minimum change in MSE to be considered as improvement

# define the mini-batch step function
@jax.jit
def mb_step_F(w, opt_state, A_batch, y_batch):
    grads = jax.grad(loss)(w, A_batch, y_batch)
    updates, opt_state = optimizer_F.update(grads, opt_state)
    w = optax.apply_updates(w, updates)
    return w, opt_state

# define the batch creation function
def get_batches_F(A, y, batch_size):
    N = len(y)
    perm = np.random.permutation(N)
    for i in range(0, N, batch_size):
        idx = perm[i:i+batch_size]
        yield A[idx], y[idx]

# pre-calculated scaled data
A_train_scaled = A_train_s
A_test_scaled = A_test_s

for B in batch_sizes:
    print(f"\n--- Running SGD for batch size B = {B} ---")

    # fresh weights
    key_F = jax.random.PRNGKey(999 + B)
    w_F = jax.random.normal(key_F, (M + 1,))

    # optimizer with gradient clipping
    optimizer_F = optax.chain(
        optax.clip(0.5),  # clip gradients to prevent explosion
        optax.sgd(learning_rate=alpha_F)
    )
    opt_state_F = optimizer_F.init(w_F)

    # histories
```

```
    train_hist, test_hist = [], []
    t0 = time.time()
    reached_time = None
    previous_mse = float('inf')  # initialize previous MSE as a large value

    # First, ensure training runs for at least 100 epochs
    for epoch in range(100):  # first mandatory 100 epochs
        # process batches using scaled data
        for A_b, y_b in get_batches_F(A_train_scaled, y_train_jnp, B):
            w_F, opt_state_F = mb_step_F(w_F, opt_state_F, A_b, y_b)

        # record metrics
        train_mse_val = mse(w_F, A_train_scaled, y_train_jnp)
        test_mse_val  = mse(w_F, A_test_scaled, y_test_jnp)
        train_hist.append(train_mse_val)
        test_hist.append(test_mse_val)

        # time to target test MSE
        if reached_time is None and float(test_mse_val) < target_mse:
            reached_time = time.time() - t0

        if epoch % 100 == 0:
            print(f"Epoch {epoch}: test MSE = {float(test_mse_val):.3e}")

    # after the first 100 epochs, check for target MSE and early stopping
    for epoch in range(100, epochs_F):
        # process batches using scaled data
        for A_b, y_b in get_batches_F(A_train_scaled, y_train_jnp, B):
            w_F, opt_state_F = mb_step_F(w_F, opt_state_F, A_b, y_b)

        # record metrics
        train_mse_val = mse(w_F, A_train_scaled, y_train_jnp)
        test_mse_val  = mse(w_F, A_test_scaled, y_test_jnp)
        train_hist.append(train_mse_val)
        test_hist.append(test_mse_val)

        # time to target test MSE
        if reached_time is None and float(test_mse_val) < target_mse:
            reached_time = time.time() - t0

        if epoch % 100 == 0:
            print(f"Epoch {epoch}: test MSE = {float(test_mse_val):.3e}")

        # Check if target MSE is reached
        if test_mse_val <= target_mse:
            print(f"Target MSE reached at epoch {epoch}. Stopping training.")
            break

        # Early stopping criterion after running for 100 epochs
        if abs(previous_mse - test_mse_val) < min_delta:
            patience -= 1
            if patience == 0:
                print(f"Early stopping at epoch {epoch} due to lack of improvement
                    .")
                break
        else:
            patience = 5  # reset patience if there was significant improvement

        previous_mse = test_mse_val  # update previous MSE for the next epoch
```

```python

    total_time = time.time() - t0

    results[B] = {
        "train_hist": train_hist,
        "test_hist":  test_hist,
        "time_to_target": reached_time,
        "final_test_mse": float(test_hist[-1]),
        "total_time": total_time
    }

    print(f"Final test MSE = {results[B]['final_test_mse']:.3e}")
    print(f"Time to reach target: {results[B]['time_to_target']}")
    print(f"Total time = {total_time:.2f} sec")

# plot train and test MSE

for B in batch_sizes:
    fig, ax = plt.subplots(figsize=(15, 6))
    ax.loglog(results[B]["train_hist"], lw=2, color='blue', label=f"Train B={B}")
    ax.loglog(results[B]["test_hist"], lw=2, linestyle="--", color='red', label=f"Test B={B}")

    plt.xlabel("Epoch")
    plt.ylabel("MSE")
    plt.legend(frameon=False)
    plt.tick_params(axis="both", which="both", direction="in")
    plt.savefig(f"batch_{B}_mse.pdf", dpi=1080)
    plt.show()

# ===== G. Learning-rate study =====

print("\n===== G. Learning-rate study =====")

# parameters
b = 32
epoch_max = 5000
learning_rates = [0.1, 0.01, 0.001]  # list of learning rates
target_mse = 1e-6
#min_delta = 1e-4
#patience_init = 5
#min_epochs = 1000

# reuse scaled vandermonde
a_train_scaled = A_train_s
a_test_scaled  = A_test_s

# closed-form solution in scaled basis
ata_s = a_train_scaled.T @ a_train_scaled
aty_s = a_train_scaled.T @ y_train_jnp
w_opt_s = jnp.linalg.solve(ata_s + lam*jnp.eye(M+1), aty_s)

# function to create mini-batches
def get_batches_g():
    n = len(y_train_jnp)
    perm = np.random.permutation(n)
    for i in range(0, n, b):
        idx = perm[i:i+b]
        yield a_train_scaled[idx], y_train_jnp[idx]
```

```python

# general training loop for any optimizer
def run_optimizer(opt, label, alpha0):
    key = jax.random.PRNGKey(999 + hash(label) % 100)
    w = jax.random.normal(key, (M+1,))
    opt_state = opt.init(w)

    train_hist = []
    test_hist  = []
    dist_hist  = []

    reached_time = None
    t0 = time.time()

    @jax.jit
    def step(w, opt_state, A_b, y_b):
        grads = jax.grad(loss)(w, A_b, y_b)
        updates, opt_state = opt.update(grads, opt_state, w)
        w = optax.apply_updates(w, updates)
        return w, opt_state

    for epoch in range(epoch_max):

        # update using mini-batches
        for A_b, y_b in get_batches_g():
            w, opt_state = step(w, opt_state, A_b, y_b)

        # compute metrics
        train_mse = float(mse(w, a_train_scaled, y_train_jnp))
        test_mse  = float(mse(w, a_test_scaled,  y_test_jnp))
        dist_w    = float(jnp.linalg.norm(w - w_opt_s))

        train_hist.append(train_mse)
        test_hist.append(test_mse)
        dist_hist.append(dist_w)

        # record time to reach target mse
        if reached_time is None and test_mse < target_mse:
            reached_time = time.time() - t0

        # periodic debug output
        if epoch % 1000 == 0:
            print(f"{label}: epoch {epoch}, test mse = {test_mse:.3e}")

    print(f"{label}: final mse = {train_mse:.3e}, dist = {dist_w:.3e}")
    return train_hist, test_hist, dist_hist

# define optimizers
def get_optimizer(alpha0):
    opt_const = optax.chain(
        optax.clip(0.5),
        optax.sgd(learning_rate=alpha0)
    )

    opt_momentum = optax.chain(
        optax.clip(0.5),
        optax.trace(decay=0.9, nesterov=True),
        optax.scale(-alpha0)
    )
```

```python

    schedule_cos = optax.cosine_decay_schedule(
        init_value=alpha0,
        decay_steps=epoch_max
    )
    opt_cosine = optax.chain(
        optax.clip(0.5),
        optax.sgd(learning_rate=schedule_cos)
    )

    return opt_const, opt_momentum, opt_cosine

# loop over all learning rates
for lr in learning_rates:
    print(f"\nRunning with learning rate = {lr}")
    # define the optimizers for the current learning rate
    opt_const, opt_momentum, opt_cosine = get_optimizer(lr)

    # run all optimizers
    train_const, test_const, dist_const = run_optimizer(opt_const,   "constant lr", lr)
    train_mom,   test_mom,   dist_mom   = run_optimizer(opt_momentum,"momentum beta=0.9", lr)
    train_cos,   test_cos,   dist_cos   = run_optimizer(opt_cosine,  "cosine decay", lr)

    # plot train mse vs epoch
    fig, ax = plt.subplots(figsize=(15, 6))

    ax.loglog(train_const, color="blue", lw=2, label="Constant lr")
    ax.loglog(train_mom,   color="red", lw=2, label=rf"Momentum $\beta=0.9$")
    ax.loglog(train_cos,   color="green", lw=2, label="Cosine decay")

    plt.xlabel("Epoch")
    plt.ylabel("Train MSE")
    plt.legend(frameon=False)
    plt.tick_params(axis="both", which="both", direction="in")
    plt.savefig(f"optimizer_comp_mse_lr{lr}.pdf", dpi=1080)
    plt.show()

    # plot test mse vs epoch
    fig, ax = plt.subplots(figsize=(15, 6))

    ax.loglog(test_const, color="blue", lw=2, label="Constant lr")
    ax.loglog(test_mom,   color="red", lw=2, label=rf"Momentum $\beta=0.9$")
    ax.loglog(test_cos,   color="green", lw=2, label="Cosine decay")

    plt.xlabel("Epoch")
    plt.ylabel("Test MSE")
    plt.legend(frameon=False)
    plt.tick_params(axis="both", which="both", direction="in")
    plt.savefig(f"optimizer_comp_test_mse_lr{lr}.pdf", dpi=1080)
    plt.show()

    # plot l2 norm vs epoch
    fig, ax = plt.subplots(figsize=(15, 6))

    ax.loglog(dist_const, color="blue", lw=2, label="constant lr")
    ax.loglog(dist_mom,   color="red", lw=2, label=rf"momentum $\beta=0.9$")
```

```
    ax.loglog(dist_cos,   color="green", lw=2, label="cosine decay")

    plt.xlabel("Epoch")
    plt.ylabel(r"$\| w - w_{\text{opt,scaled}} \|_2$")
    plt.legend(frameon=False)
    plt.tick_params(axis="both", which="both", direction="in")
    plt.savefig(f"optimizer_comp_lr{lr}.pdf", dpi=1080)
    plt.show()

# ===== H. SGD vs another optimizer =====

print("\n===== H. SGD vs another optimizer =====")

# parameters for this comparison
b = 32
epoch_max = 5000
learning_rates = [0.1, 0.01, 0.001]  # list of learning rates
target_mse = 1e-6

# optimizers (momentum, adam)
def get_optimizer_momentum(alpha0):
    return optax.chain(
        optax.clip(0.5),
        optax.trace(decay=0.9, nesterov=True),
        optax.scale(-alpha0)
    )

def get_optimizer_adam(alpha0):
    return optax.chain(
        optax.clip(0.5),
        optax.adam(learning_rate=alpha0)
    )

# function to create mini-batches
def get_batches_g(A, y, batch_size):
    n = len(y)
    perm = np.random.permutation(n)
    for i in range(0, n, batch_size):
        idx = perm[i:i+batch_size]
        yield A[idx], y[idx]

# general training loop for any optimizer
def run_optimizer(opt, label, alpha0, A_train_scaled, A_test_scaled, y_train_jnp,
    y_test_jnp):
    key = jax.random.PRNGKey(999 + hash(label) % 100)
    w = jax.random.normal(key, (M+1,))
    opt_state = opt.init(w)

    train_hist = []
    test_hist  = []
    dist_hist  = []

    reached_time = None
    t0 = time.time()

    @jax.jit
    def step(w, opt_state, A_b, y_b):
        grads = jax.grad(loss)(w, A_b, y_b)
        updates, opt_state = opt.update(grads, opt_state, w)
```

```
        w = optax.apply_updates(w, updates)
        return w, opt_state

    for epoch in range(epoch_max):
        # update using mini-batches
        for A_b, y_b in get_batches_g(A_train_scaled, y_train_jnp, b):
            w, opt_state = step(w, opt_state, A_b, y_b)

        # compute metrics
        train_mse = float(mse(w, A_train_scaled, y_train_jnp))
        test_mse  = float(mse(w, A_test_scaled,  y_test_jnp))
        dist_w    = float(jnp.linalg.norm(w - w_opt_s))

        train_hist.append(train_mse)
        test_hist.append(test_mse)
        dist_hist.append(dist_w)

        # record time to reach target mse
        if reached_time is None and test_mse < target_mse:
            reached_time = time.time() - t0

        # periodic debug output
        if epoch % 1000 == 0:
            print(f"{label}: epoch {epoch}, test mse = {test_mse:.3e}")

    print(f"{label}: final mse = {train_mse:.3e}, dist = {dist_w:.3e}")
    return train_hist, test_hist, dist_hist

# loop over all learning rates for comparison of Momentum and Adam
for lr in learning_rates:
    print(f"\nRunning with learning rate = {lr}")

    # get optimizers for Momentum and Adam
    opt_momentum = get_optimizer_momentum(lr)
    opt_adam = get_optimizer_adam(lr)

    # run
    train_mom, test_mom, dist_mom = run_optimizer(opt_momentum, "Momentum beta=0.9
       ", lr, A_train_s, A_test_s, y_train_jnp, y_test_jnp)
    train_adam, test_adam, dist_adam = run_optimizer(opt_adam, "Adam optimizer",
       lr, A_train_s, A_test_s, y_train_jnp, y_test_jnp)

    # plot train mse vs epoch
    fig, ax = plt.subplots(figsize=(15, 6))
    ax.loglog(train_mom, color="red", lw=2, label=rf"Momentum $\beta=0.9$")
    ax.loglog(train_adam, color="purple", lw=2, label="Adam")
    plt.xlabel("Epoch")
    plt.ylabel("Train MSE")
    plt.legend(frameon=False)
    plt.tick_params(axis="both", which="both", direction="in")
    plt.savefig(f"train_optimizer_comp_mse_lr{lr}_momentum_adam.pdf", dpi=1080)
    plt.show()

    # Ppot test mse vs epoch
    fig, ax = plt.subplots(figsize=(15, 6))
    ax.loglog(test_mom, color="red", lw=2, label=rf"Momentum $\beta=0.9$")
    ax.loglog(test_adam, color="purple", lw=2, label="Adam")
    plt.xlabel("Epoch")
    plt.ylabel("Test MSE")
```

```
    plt.legend(frameon=False)
    plt.tick_params(axis="both", which="both", direction="in")
    plt.savefig(f"test_optimizer_comp_mse_lr{lr}_momentum_adam.pdf", dpi=1080)
    plt.show()

    # plot l2 norm vs epoch
    fig, ax = plt.subplots(figsize=(15, 6))
    ax.loglog(dist_mom, color="red", lw=2, label=rf"Momentum $\beta=0.9$")
    ax.loglog(dist_adam, color="purple", lw=2, label="Adam")
    plt.xlabel("Epoch")
    plt.ylabel(r"$\| w - w_{\text{opt,scaled}} \|_2$")
    plt.legend(frameon=False)
    plt.tick_params(axis="both", which="both", direction="in")
    plt.savefig(f"norm_optimizer_comp_lr{lr}_momentum_adam.pdf", dpi=1080)
    plt.show()
```

Listing 1: sgd.py

```
===== D. Write and test the model and loss functions =====

SANITY TEST

True weights: w_true = [1. 1. 1.]
Loss(w_true) = 0.0
MSE(w_true)  = 0.0

Bad weights: w_bad = [1.  0.5 0.5]
Loss(w_bad)  = 258.57525025024205
MSE(w_bad)   = 0.25857525025024203

Training MSE (true weights) = 0.0
Testing  MSE (true weights) = 0.0

====== E. SGD optimization ======

Closed-form: omega_opt =  [1.00000000e+00 1.00000000e+00 1.00000000e+00 1.94831852
    e-13]

Maximum eigenvalue of Hessian and upper bound of learning rate:

lambda_max(H) = 2.394e+03
alpha_max = 2 / lambda_max(H) = 8.355e-04

SGD should theoretically diverge as alpha = 0.2 is greater than alpha_max = 8.355e
    -04

Let''s try with scaled Vandermonde...

After scaling...

lambda_max(H) = 1.796e+03
alpha_max = 2 / lambda_max(H) = 1.114e-03

SGD should theoretically still diverge.

Final SGD weights: [1.59415786 0.41523422 0.41608667 0.78784401]
Distance to closed-form = 1.2871004385864873

```

```
Closed-form solution (scaled basis): [1.75000000e+00 1.00000000e+00 2.50000000e-01
     8.39450872e-15]

Final SGD weights (scaled): [1.64692162 0.76950196 0.65296745 0.50743949]
Distance to scaled closed-form: 0.6954366236666928

Epoch 0: Train MSE = 0.074629, Dist = 5.416e-01
Epoch 100: Train MSE = 0.012554, Dist = 1.045e+00
Epoch 200: Train MSE = 0.005838, Dist = 1.037e+00
Epoch 300: Train MSE = 0.008159, Dist = 1.031e+00
Epoch 400: Train MSE = 0.005859, Dist = 1.031e+00
Epoch 500: Train MSE = 0.005436, Dist = 1.018e+00
Epoch 600: Train MSE = 0.008189, Dist = 1.029e+00
Epoch 700: Train MSE = 0.005348, Dist = 1.013e+00
Epoch 800: Train MSE = 0.009653, Dist = 1.051e+00
Epoch 900: Train MSE = 0.007712, Dist = 1.035e+00
Epoch 1000: Train MSE = 0.009104, Dist = 1.057e+00
Epoch 1100: Train MSE = 0.008422, Dist = 1.034e+00
Epoch 1200: Train MSE = 0.005371, Dist = 1.028e+00
Epoch 1300: Train MSE = 0.006227, Dist = 1.011e+00
Epoch 1400: Train MSE = 0.009272, Dist = 1.037e+00
Epoch 1500: Train MSE = 0.008010, Dist = 1.033e+00
Epoch 1600: Train MSE = 0.009683, Dist = 1.039e+00
Epoch 1700: Train MSE = 0.007722, Dist = 1.042e+00
Epoch 1800: Train MSE = 0.006759, Dist = 1.031e+00
Epoch 1900: Train MSE = 0.004825, Dist = 1.021e+00

Final mini-batch SGD weights: [ 1.78392582  0.95504057  0.33473039 -0.02170426]
Distance to closed-form = 1.0293764006591533

Closed-form solution (scaled basis) for mini-batch: [1.75000000e+00 1.00000000e+00
     2.50000000e-01 8.39450872e-15]
[SCALED] Epoch 0: Train MSE = 0.045927, Dist = 1.416e+00
[SCALED] Epoch 100: Train MSE = 0.006505, Dist = 9.340e-02
[SCALED] Epoch 200: Train MSE = 0.006073, Dist = 8.988e-02
[SCALED] Epoch 300: Train MSE = 0.007051, Dist = 9.525e-02
[SCALED] Epoch 400: Train MSE = 0.008163, Dist = 1.160e-01
[SCALED] Epoch 500: Train MSE = 0.008366, Dist = 1.143e-01
[SCALED] Epoch 600: Train MSE = 0.005401, Dist = 9.423e-02
[SCALED] Epoch 700: Train MSE = 0.006311, Dist = 8.857e-02
[SCALED] Epoch 800: Train MSE = 0.005444, Dist = 8.853e-02
[SCALED] Epoch 900: Train MSE = 0.008220, Dist = 1.078e-01
[SCALED] Epoch 1000: Train MSE = 0.005398, Dist = 9.288e-02
[SCALED] Epoch 1100: Train MSE = 0.003549, Dist = 6.998e-02
[SCALED] Epoch 1200: Train MSE = 0.010028, Dist = 1.276e-01
[SCALED] Epoch 1300: Train MSE = 0.006572, Dist = 1.004e-01
[SCALED] Epoch 1400: Train MSE = 0.007778, Dist = 1.090e-01
[SCALED] Epoch 1500: Train MSE = 0.005505, Dist = 9.799e-02
[SCALED] Epoch 1600: Train MSE = 0.011210, Dist = 1.334e-01
[SCALED] Epoch 1700: Train MSE = 0.006529, Dist = 8.929e-02
[SCALED] Epoch 1800: Train MSE = 0.008462, Dist = 1.143e-01
[SCALED] Epoch 1900: Train MSE = 0.009798, Dist = 1.072e-01

Final mini-batch SGD weights (scaled): [1.76256173 1.02239192 0.36638021
    0.09804078]
Distance to scaled closed-form = 0.15432285073856614

===== F. Batch size study =====

```

```
--- Running SGD for batch size B = 1 ---
Epoch 0: test MSE = 1.970e-05
Epoch 100: test MSE = 4.743e-31
Target MSE reached at epoch 100. Stopping training.
Final test MSE = 4.743e-31
Time to reach target: 1.2927360534667969
Total time = 119.36 sec

--- Running SGD for batch size B = 16 ---
Epoch 0: test MSE = 1.165e-01
Epoch 100: test MSE = 1.009e-03
Epoch 200: test MSE = 1.646e-03
Epoch 300: test MSE = 6.583e-04
Epoch 400: test MSE = 9.901e-04
Epoch 500: test MSE = 2.741e-03
Epoch 600: test MSE = 7.420e-04
Epoch 700: test MSE = 7.916e-04
Target MSE reached at epoch 758. Stopping training.
Final test MSE = 2.784e-05
Time to reach target: 57.53362798690796
Total time = 57.53 sec

--- Running SGD for batch size B = 64 ---
Epoch 0: test MSE = 1.894e+00
Epoch 100: test MSE = 1.674e-03
Epoch 10000: test MSE = 1.826e-03
Epoch 50000: test MSE = 2.094e-03
Final test MSE = 1.830e-03
Time to reach target: None
Total time = 2079.73 sec

--- Running SGD for batch size B = 800 ---
Epoch 0: test MSE = 3.392e+00
Epoch 100: test MSE = 5.355e-03
Epoch 10000: test MSE = 5.355e-03
Epoch 50000: test MSE = 5.355e-03
Final test MSE = 7.150e-03
Time to reach target: None
Total time = 209.35 sec

===== G. Learning-rate study =====

Running with learning rate = 0.1
constant lr: epoch 0, test mse = 1.440e-02
constant lr: epoch 1000, test mse = 1.970e-03
constant lr: epoch 2000, test mse = 1.133e-03
constant lr: epoch 3000, test mse = 1.986e-03
constant lr: epoch 4000, test mse = 1.390e-03
constant lr: final mse = 2.031e-03, dist = 6.195e-02
momentum beta=0.9: epoch 0, test mse = 1.377e-02
momentum beta=0.9: epoch 1000, test mse = 9.439e-03
momentum beta=0.9: epoch 2000, test mse = 3.363e-03
momentum beta=0.9: epoch 3000, test mse = 9.066e-03
momentum beta=0.9: epoch 4000, test mse = 1.207e-02
momentum beta=0.9: final mse = 6.652e-03, dist = 1.195e-01
cosine decay: epoch 0, test mse = 1.091e+00
cosine decay: epoch 1000, test mse = 9.030e-24
cosine decay: epoch 2000, test mse = 9.030e-24
cosine decay: epoch 3000, test mse = 9.030e-24
```

```
cosine decay: epoch 4000, test mse = 9.030e-24
cosine decay: final mse = 8.731e-24, dist = 2.295e-11

Running with learning rate = 0.01
constant lr: epoch 0, test mse = 6.387e-01
constant lr: epoch 1000, test mse = 4.018e-32
constant lr: epoch 2000, test mse = 4.043e-32
constant lr: epoch 3000, test mse = 2.958e-32
constant lr: epoch 4000, test mse = 2.958e-32
constant lr: final mse = 2.459e-32, dist = 9.777e-15
momentum beta=0.9: epoch 0, test mse = 4.425e+00
momentum beta=0.9: epoch 1000, test mse = 3.032e-32
momentum beta=0.9: epoch 2000, test mse = 2.613e-32
momentum beta=0.9: epoch 3000, test mse = 3.254e-32
momentum beta=0.9: epoch 4000, test mse = 9.466e-32
momentum beta=0.9: final mse = 2.404e-32, dist = 9.786e-15
cosine decay: epoch 0, test mse = 6.308e+00
cosine decay: epoch 1000, test mse = 5.474e-21
cosine decay: epoch 2000, test mse = 5.474e-21
cosine decay: epoch 3000, test mse = 5.474e-21
cosine decay: epoch 4000, test mse = 5.474e-21
cosine decay: final mse = 5.293e-21, dist = 5.654e-10

Running with learning rate = 0.001
constant lr: epoch 0, test mse = 8.157e-01
constant lr: epoch 1000, test mse = 3.077e-23
constant lr: epoch 2000, test mse = 3.071e-30
constant lr: epoch 3000, test mse = 2.831e-30
constant lr: epoch 4000, test mse = 2.663e-30
constant lr: final mse = 2.448e-30, dist = 2.004e-15
momentum beta=0.9: epoch 0, test mse = 1.125e+01
momentum beta=0.9: epoch 1000, test mse = 1.048e-31
momentum beta=0.9: epoch 2000, test mse = 9.762e-32
momentum beta=0.9: epoch 3000, test mse = 9.688e-32
momentum beta=0.9: epoch 4000, test mse = 9.614e-32
momentum beta=0.9: final mse = 8.684e-32, dist = 8.111e-15
cosine decay: epoch 0, test mse = 7.158e+00
cosine decay: epoch 1000, test mse = 1.088e+00
cosine decay: epoch 2000, test mse = 1.088e+00
cosine decay: epoch 3000, test mse = 1.088e+00
cosine decay: epoch 4000, test mse = 1.088e+00
cosine decay: final mse = 1.072e+00, dist = 2.062e+00

===== H. SGD vs another optimizer =====

Running with learning rate = 0.1
Momentum beta=0.9: epoch 0, test mse = 1.398e-03
Momentum beta=0.9: epoch 1000, test mse = 7.683e-03
Momentum beta=0.9: epoch 2000, test mse = 3.300e-03
Momentum beta=0.9: epoch 3000, test mse = 3.783e-03
Momentum beta=0.9: epoch 4000, test mse = 9.086e-03
Momentum beta=0.9: final mse = 2.818e-03, dist = 2.190e-01
Adam optimizer: epoch 0, test mse = 2.291e+00
Adam optimizer: epoch 1000, test mse = 1.153e-07
Adam optimizer: epoch 2000, test mse = 2.422e-04
Adam optimizer: epoch 3000, test mse = 1.552e-03
Adam optimizer: epoch 4000, test mse = 2.424e-07
Adam optimizer: final mse = 1.390e-11, dist = 1.477e-05

```

```
Running with learning rate = 0.01
Momentum beta=0.9: epoch 0, test mse = 3.438e-03
Momentum beta=0.9: epoch 1000, test mse = 3.229e-32
Momentum beta=0.9: epoch 2000, test mse = 3.698e-32
Momentum beta=0.9: epoch 3000, test mse = 3.081e-32
Momentum beta=0.9: epoch 4000, test mse = 2.761e-32
Momentum beta=0.9: final mse = 2.687e-32, dist = 9.766e-15
Adam optimizer: epoch 0, test mse = 1.978e+01
Adam optimizer: epoch 1000, test mse = 3.550e-14
Adam optimizer: epoch 2000, test mse = 1.985e-07
Adam optimizer: epoch 3000, test mse = 5.544e-05
Adam optimizer: epoch 4000, test mse = 5.336e-30
Adam optimizer: final mse = 5.158e-15, dist = 1.750e-07

Running with learning rate = 0.001
Momentum beta=0.9: epoch 0, test mse = 3.163e-01
Momentum beta=0.9: epoch 1000, test mse = 2.914e-31
Momentum beta=0.9: epoch 2000, test mse = 2.911e-31
Momentum beta=0.9: epoch 3000, test mse = 2.926e-31
Momentum beta=0.9: epoch 4000, test mse = 2.926e-31
Momentum beta=0.9: final mse = 2.879e-31, dist = 1.360e-14
Adam optimizer: epoch 0, test mse = 2.253e+01
Adam optimizer: epoch 1000, test mse = 8.907e-17
Adam optimizer: epoch 2000, test mse = 3.355e-15
Adam optimizer: epoch 3000, test mse = 9.117e-16
Adam optimizer: epoch 4000, test mse = 1.030e-12
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

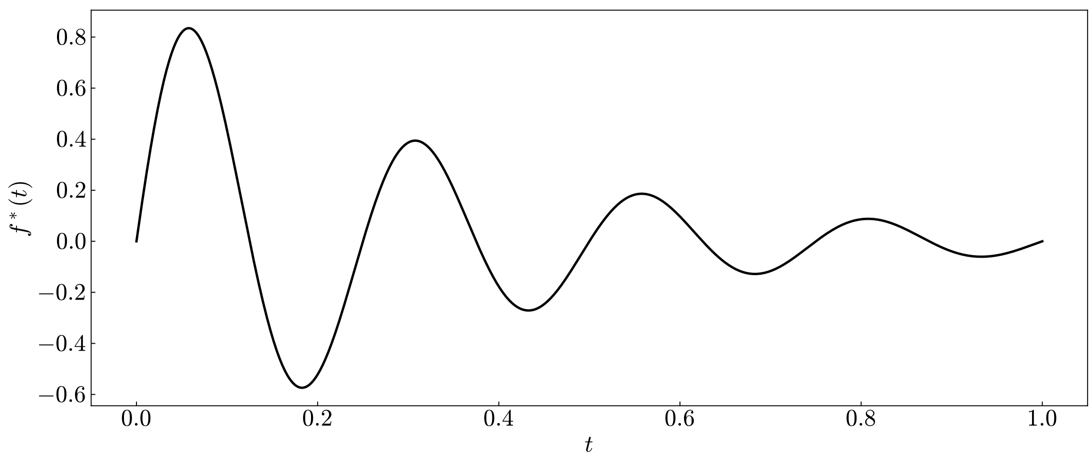


Figure 17: Target function.

C. A two-layer network model

The two layer network is defined as,

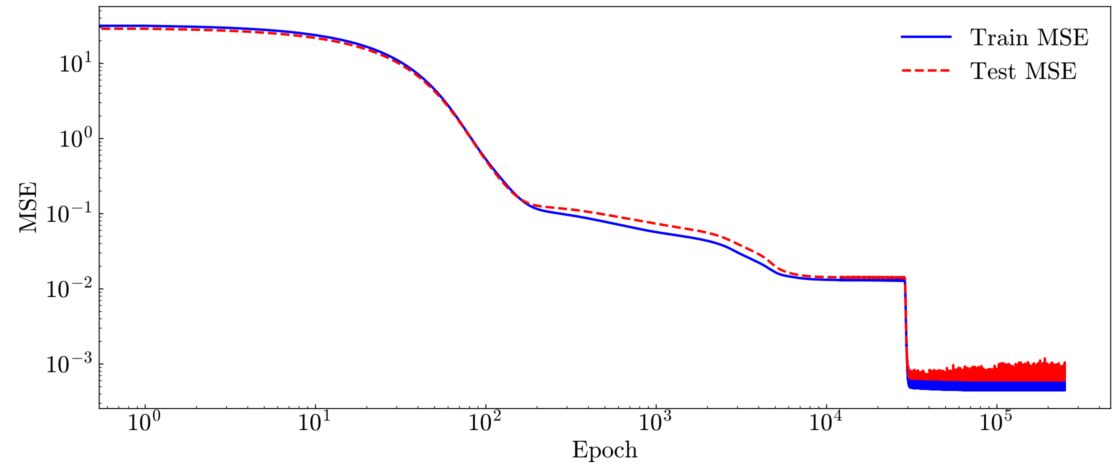
$$\tilde{f}(t, \theta) = \sum_{j=1}^{m} a_j \phi\left(w_j t + b_j\right) + c$$

`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.

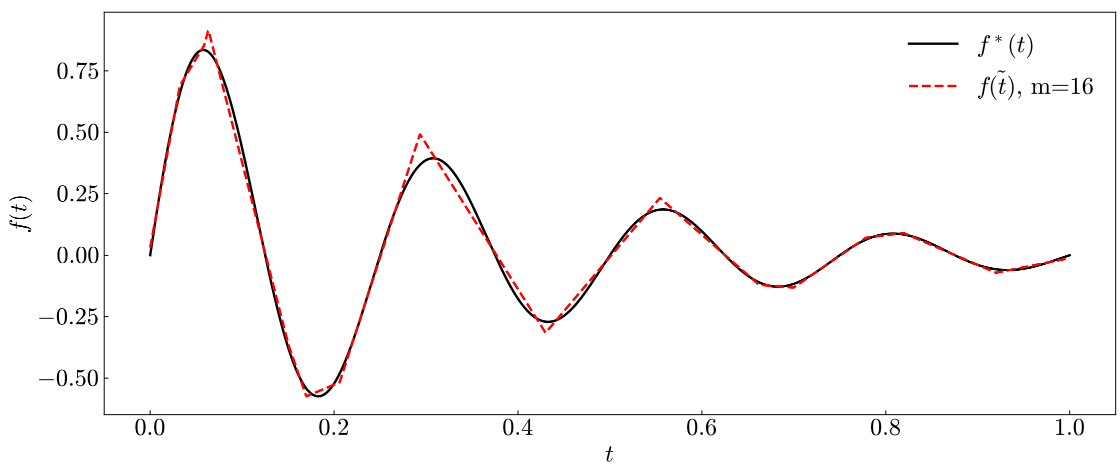
Additionally, please refer to listing 3.

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.



(a)



(b)

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 ∼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 α, the relation between $E(m)$ and m is linearized using `log`.

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

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

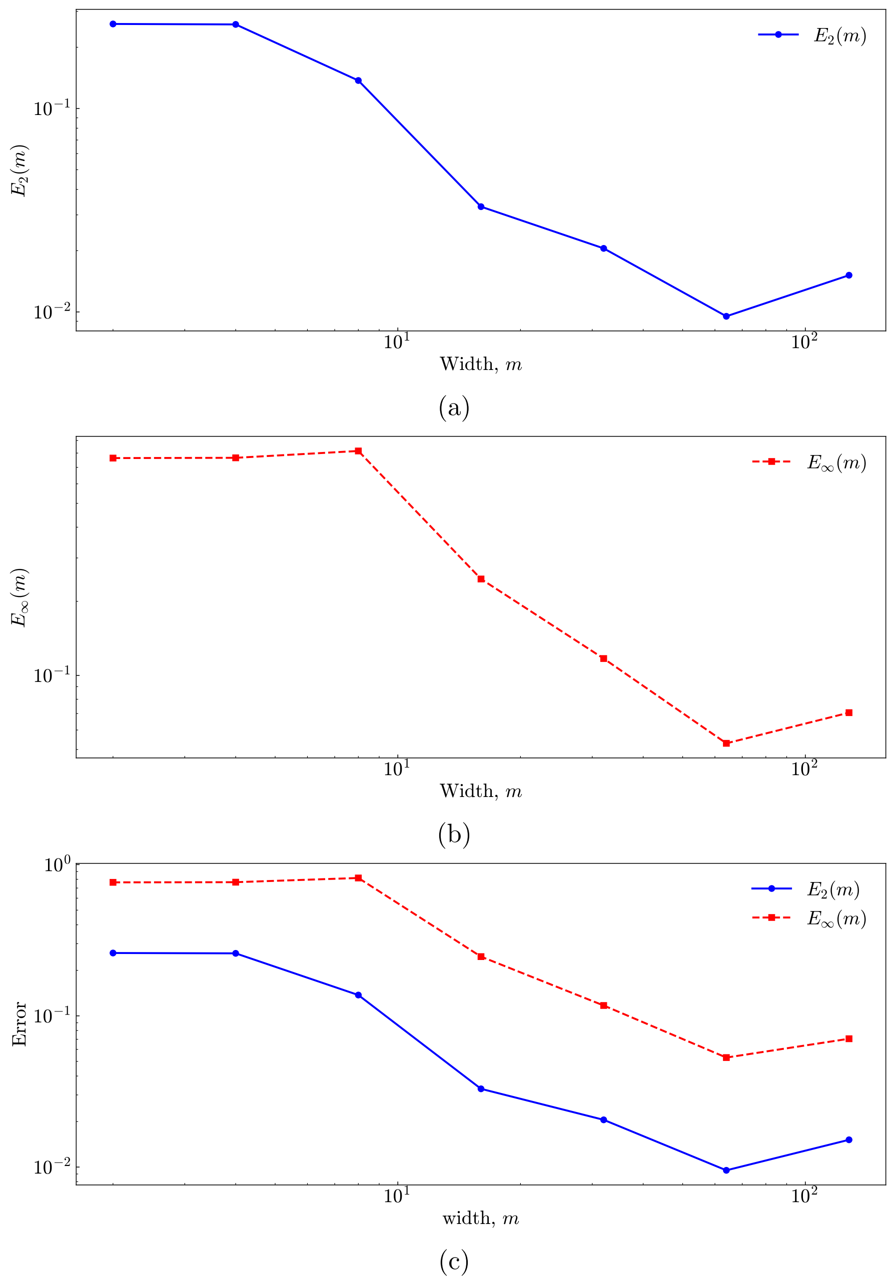
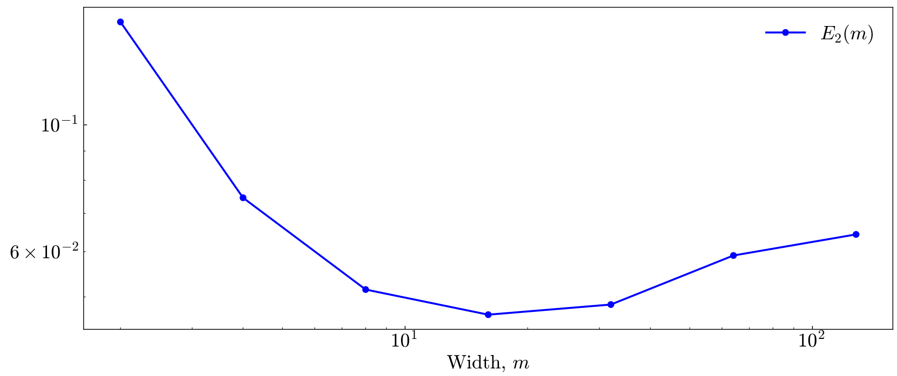


Figure 19: (a) $E_2(m)$ vs. m, (b) $E_\infty(m)$ vs. m, and (c) single plot of final test errors for `ReLU`.

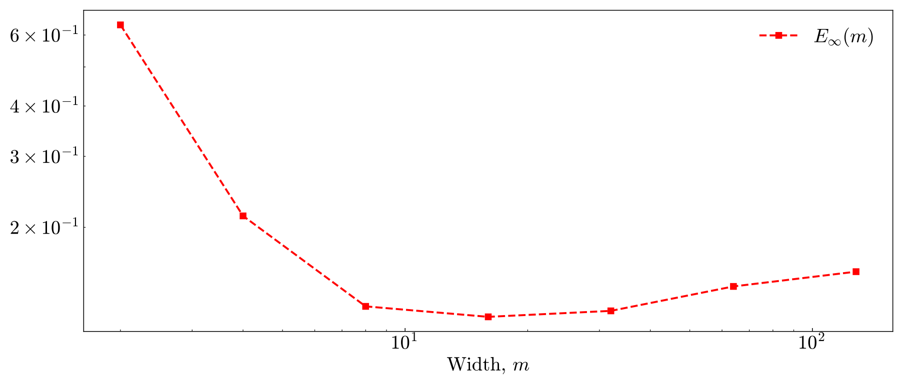
in errors is almost linear. However, there is a slight increase in errors as width moves from 64 to 128.

F. Effect of the activation function

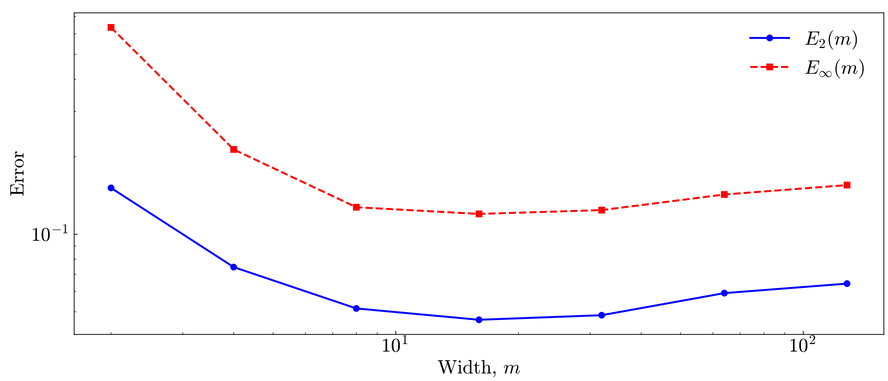
This time the same sweeping is done using `tanh` as the activation function. All the parameters are kept the same as `ReLU` setup, so that we can make an an apple-to-apple comparison.



(a)



(b)



(c)

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`:

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

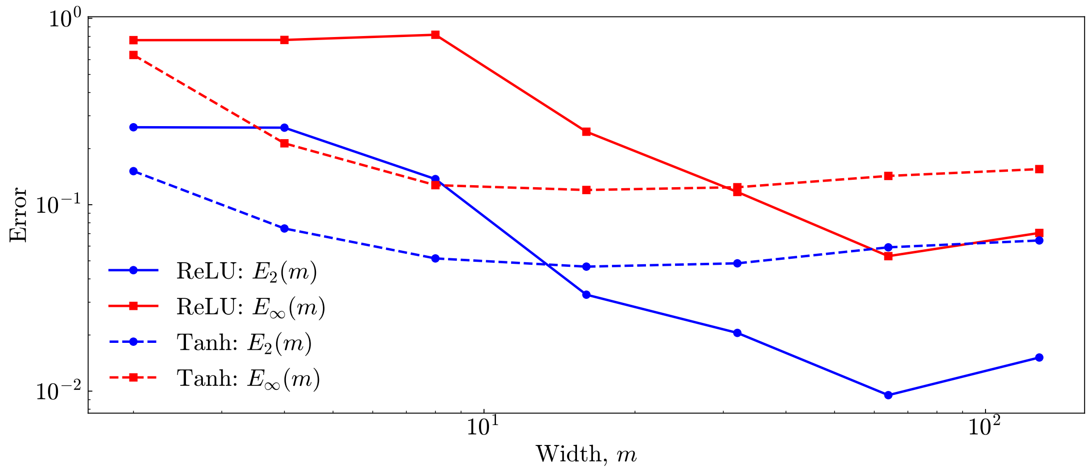


Figure 21: Comparison of final test errors between `ReLU` and `tanh`.

Some observations from fig. 21:

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

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

  `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 `ReLU`, which decreases error steeply after $m = 8$. But after the initial steep decrease, `tanh` becomes flattened between $m = 8$ to 64, where `ReLU` starts thriving in the same interval. This might indicate that `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 `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.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import jax
import jax.numpy as jnp
import optax

# ===== B. Data and target function =====

# target function f*
def f_star(t):
    return np.exp(-3.0 * t) * np.sin(8 * np.pi * t)

# create train and test sets
rng = np.random.default_rng(0)
t_train = rng.uniform(0.0, 1.0, size=128)
y_train = f_star(t_train)

t_test = np.linspace(0.0, 1.0, 2048, endpoint=True)
y_test = f_star(t_test)

# plotting params
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = 'cmr10'
plt.rcParams['mathtext.fontset'] = 'cm'
plt.rcParams['font.size'] = 20
mpl.rcParams['axes.unicode_minus'] = False
plt.rcParams['axes.formatter.use_mathtext'] = True

# plot the target function on a dense grid
fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t_test, y_test, 'k-', lw=2)
plt.xlabel(r"$t$")
plt.ylabel(r"$f^*(t)$")
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("fstar.pdf", dpi=1080)
plt.show()

# ===== C. Two-layer network model =====

# relu activation
def relu(x):
    return jnp.maximum(0.0, x)

# tanh activation
def tanh(x):
    return jnp.tanh(x)

# model initialization
def init_model(rng, m):
    key_w, key_t, key_a, key_c = jax.random.split(rng, 4)

    # wide input weights
    w = 30.0 * jax.random.normal(key_w, (m,))

    # kink locations uniformly across [0,1]
    t0 = jax.random.uniform(key_t, (m,), minval=0.0, maxval=1.0)

```

```python
    # bias
    b = -w * t0

    # small output layer weights
    a = 0.1 * jax.random.normal(key_a, (m,))

    c = jnp.array(0.0) # output bias

    return {"w": w, "b": b, "a": a, "c": c}

# forward pass
def model_apply(params, t, activation):
    # compute $w_j t + b_j$ for all hidden units
    z = params["w"] * t[:, None] + params["b"]

    # apply activation
    h = activation(z)

    # output
    y_pred = jnp.dot(h, params["a"]) + params["c"]
    return y_pred

# mse loss function
def mse_loss(params, t, y, activation):
    y_pred = model_apply(params, t, activation)
    r = y_pred - y
    return jnp.mean(r * r)

# ===== D. Implementation and visualization =====

# width
m = 16

# number of epochs
epochs_d = 250000

# optimizer (adam)
optimizer_d = optax.adam(learning_rate=1e-3)

# initialize parameters
rng = jax.random.PRNGKey(299)
params_d = init_model(rng, m)

opt_state_d = optimizer_d.init(params_d)

# choose activation outside the jit
activation = relu

# jit-compiled update step
@jax.jit
def step(params, opt_state, t, y):
    loss_val, grads = jax.value_and_grad(mse_loss)(params, t, y, activation)
    updates, opt_state = optimizer_d.update(grads, opt_state)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss_val

# training history
train_hist_d = []
test_hist_d  = []
```

```python

# convert training data to jax arrays
t_train_j = jnp.asarray(t_train)
y_train_j = jnp.asarray(y_train)
t_test_j  = jnp.asarray(t_test)
y_test_j  = jnp.asarray(y_test)

# training loop
for epoch in range(epochs_d):
    # shuffle each epoch
    key_epoch = jax.random.PRNGKey(epoch)
    perm = jax.random.permutation(key_epoch, len(t_train_j))
    t_batch = t_train_j[perm]
    y_batch = y_train_j[perm]

    params_d, opt_state_d, train_loss = step(params_d, opt_state_d,
                                             t_batch, y_batch)

    train_hist_d.append(float(train_loss))

    # compute test mse
    test_loss = mse_loss(params_d, t_test_j, y_test_j, activation)
    test_hist_d.append(float(test_loss))

    if epoch % 500 == 0:
        print(f"epoch {epoch}, train mse = {train_loss:.4e}")

print(f"\nfinal train mse = {train_hist_d[-1]:.4e}")
print(f"final test  mse = {test_hist_d[-1]:.4e}")

# plot mse vs. epoch
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(train_hist_d, "b-", lw=2, label="Train MSE")
ax.loglog(test_hist_d,  "r--", lw=2, label="Test MSE")
plt.xlabel("Epoch")
plt.ylabel("MSE")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("mse_m16.pdf", dpi=1080)
plt.show()

# plot model and true function
y_pred_d = model_apply(params_d, t_test_j, activation)

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t_test, y_test,     "k-", lw=2, label=r"$f^*(t)$")
ax.plot(t_test, y_pred_d,   "r--", lw=2, label=r"$\tilde f(t)$, m=16")
plt.xlabel(r"$t$")
plt.ylabel(r"$f(t)$")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("compare_m16.pdf", dpi=1080)
plt.show()

# ===== E. Empirical convergence with width =====

# params
width_list = [2, 4, 8, 16, 32, 64, 128]
epochs_e = 20000
```

```python
restarts = 3

E2_vals = []
Einf_vals = []

# optimizer (adam)
optimizer_e = optax.adam(learning_rate=1e-3)

@jax.jit
def step_e(params, opt_state, t, y):
    loss_val, grads = jax.value_and_grad(mse_loss)(params, t, y, relu)
    updates, opt_state = optimizer_e.update(grads, opt_state)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss_val

for m in width_list:
    print(f"\n=== width m = {m} ===")

    # store errors from restarts
    err2_list = []
    errinf_list = []

    for r in range(restarts):
        print(f" restart {r}")

        # init params
        key = jax.random.PRNGKey(1000 + 17*m + r)
        params = init_model(key, m)
        opt_state = optimizer_e.init(params)

        # train
        for epoch in range(epochs_e):
            key_epoch = jax.random.PRNGKey(epoch)
            perm = jax.random.permutation(key_epoch, len(t_train_j))
            t_batch = t_train_j[perm]
            y_batch = y_train_j[perm]
            params, opt_state, train_loss = step_e(params, opt_state, t_batch,
                y_batch)

        # compute test prediction
        y_pred = model_apply(params, t_test_j, relu)

        # compute errors (convert to numpy first)
        residual = np.array(y_pred) - np.array(y_test)
        E2   = np.sqrt(np.mean(residual**2))
        Einf = np.max(np.abs(residual))

        err2_list.append(E2)
        errinf_list.append(Einf)

    # take median across restarts
    E2_vals.append(np.median(err2_list))
    Einf_vals.append(np.median(errinf_list))

# convert to numpy arrays
width_arr = np.array(width_list)
E2_vals   = np.array(E2_vals)
Einf_vals = np.array(Einf_vals)

```

```
# log-log slope estimation
logm   = np.log(width_arr)
logE2  = np.log(E2_vals)
logEin = np.log(Einf_vals)

alpha_E2,   _ = np.polyfit(logm, logE2, 1)
alpha_Einf, _ = np.polyfit(logm, logEin, 1)

print("\nestimated slopes (ReLU):")
print(f"  alpha_E2   ~ {-alpha_E2:.3f}")
print(f"  alpha_Einf ~ {-alpha_Einf:.3f}")

# plot E2(m)
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(width_arr, E2_vals, "b-o", lw=2, label=r"$E_2(m)$")
plt.xlabel(rf"Width, $m$")
plt.ylabel(r"$E_2(m)$")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("E2_vs_m.pdf", dpi=1080)
plt.show()

# plot of Einf(m)
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(width_arr, Einf_vals, "r--s", lw=2, label=r"$E_\infty(m)$")
plt.xlabel(rf"Width, $m$")
plt.ylabel(r"$E_\infty(m)$")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("Einf_vs_m.pdf", dpi=1080)
plt.show()

# combined plot
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(width_arr, E2_vals,  "b-o",  lw=2, label=r"$E_2(m)$")
ax.loglog(width_arr, Einf_vals, "r--s", lw=2, label=r"$E_\infty(m)$")
plt.xlabel(rf"Width, $m$")
plt.ylabel("Error")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("convergence_width.pdf", dpi=1080)
plt.show()

# ===== F. Effect of Activation Function =====

# params
width_list = [2, 4, 8, 16, 32, 64, 128]
epochs_f = 20000
restarts_f = 3

# store results
E2_tanh_vals = []
Einf_tanh_vals = []

# optimizer (adam)
optimizer_f = optax.adam(learning_rate=1e-3)

# model initialization for tanh (smaller weights to avoid saturation)
def init_model_tanh(rng, m):
```

```
    key_w, key_b, key_a, key_c = jax.random.split(rng, 4)

    # smaller input weights for tanh to prevent saturation
    w = 3.0 * jax.random.normal(key_w, (m,))

    # biases uniformly across [-1, 1]
    b = jax.random.uniform(key_b, (m,), minval=-1.0, maxval=1.0)

    # small output layer weights
    a = 0.1 * jax.random.normal(key_a, (m,))

    c = jnp.array(0.0)  # output bias

    return {"w": w, "b": b, "a": a, "c": c}


# training step for tanh
@jax.jit
def step_f(params, opt_state, t, y):
    loss_val, grads = jax.value_and_grad(mse_loss)(params, t, y, tanh)
    updates, opt_state = optimizer_f.update(grads, opt_state)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss_val

# loop through each width
for m in width_list:
    print(f"\n=== width m = {m} ===")

    # store errors from restarts
    err2_list = []
    errinf_list = []

    for r in range(restarts_f):
        print(f" restart {r}")

        # initialize parameters
        key = jax.random.PRNGKey(2000 + 19 * m + r)
        params = init_model_tanh(key, m)
        opt_state = optimizer_f.init(params)

        # training loop
        for epoch in range(epochs_f):
            key_epoch = jax.random.PRNGKey(epoch)
            perm = jax.random.permutation(key_epoch, len(t_train_j))
            t_batch = t_train_j[perm]
            y_batch = y_train_j[perm]
            params, opt_state, train_loss = step_f(params, opt_state, t_batch,
                y_batch)

        # compute test predictions
        y_pred = model_apply(params, t_test_j, tanh)

        # compute errors (convert to numpy first)
        residual = np.array(y_pred) - np.array(y_test)
        E2 = np.sqrt(np.mean(residual**2))
        Einf = np.max(np.abs(residual))

        err2_list.append(E2)
        errinf_list.append(Einf)
```

```python

    # take the median across restarts
    E2_tanh_vals.append(np.median(err2_list))
    Einf_tanh_vals.append(np.median(errinf_list))

# convert to numpy arrays
E2_tanh_vals = np.array(E2_tanh_vals)
Einf_tanh_vals = np.array(Einf_tanh_vals)

# log-log slope estimation for tanh
logm = np.log(width_list)
logE2_th = np.log(E2_tanh_vals)
logEin_th = np.log(Einf_tanh_vals)

alpha_E2_tanh, _ = np.polyfit(logm, logE2_th, 1)
alpha_Einf_tanh, _ = np.polyfit(logm, logEin_th, 1)

print("\nestimated slopes (tanh):")
print(f"  alpha_E2_tanh   ~ {-alpha_E2_tanh:.3f}")
print(f"  alpha_Einf_tanh ~ {-alpha_Einf_tanh:.3f}")

# plot E2(m) for tanh
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(width_list, E2_tanh_vals, "b-o", lw=2, label=r"$E_2(m)$")
plt.xlabel(rf"Width, $m$")
plt.ylabel(r"$E_2(m)$")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("E2_tanh_vs_m.pdf", dpi=1080)
plt.show()

# plot Einf(m) for tanh
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(width_list, Einf_tanh_vals, "r--s", lw=2, label=r"$E_\infty(m)$")
plt.xlabel(rf"Width, $m$")
plt.ylabel(r"$E_\infty(m)$")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("Einf_tanh_vs_m.pdf", dpi=1080)
plt.show()

# combined plot
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(width_arr, E2_tanh_vals,  "b-o",  lw=2, label=r"$E_2(m)$")
ax.loglog(width_arr, Einf_tanh_vals, "r--s", lw=2, label=r"$E_\infty(m)$")
plt.xlabel(rf"Width, $m$")
plt.ylabel("Error")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("convergence_tanh.pdf", dpi=1080)
plt.show()

# combined comparison plot for ReLU and tanh
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(width_arr, E2_vals,      "b-o",  lw=2, label=r"ReLU: $E_2(m)$")
ax.loglog(width_arr, Einf_vals,    "r-s", lw=2, label=r"ReLU: $E_\infty(m)$")
ax.loglog(width_arr, E2_tanh_vals, "b--o", lw=2, label=r"Tanh: $E_2(m)$")
ax.loglog(width_arr, Einf_tanh_vals, "r--s", lw=2, label=r"Tanh: $E_\infty(m)$")
plt.xlabel(rf"Width, $m$")
```

```python
plt.ylabel("Error")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("convergence_relu_vs_tanh.pdf", dpi=1080)
plt.show()
```

Listing 3: universal_approx.py

```
epoch 0, train mse = 3.2565e+01
epoch 500, train mse = 7.8099e-02
epoch 1000, train mse = 5.6896e-02
epoch 1500, train mse = 4.9343e-02
epoch 2000, train mse = 4.3553e-02
epoch 2500, train mse = 3.7327e-02
epoch 3000, train mse = 3.0488e-02
epoch 3500, train mse = 2.5812e-02
epoch 4000, train mse = 2.2334e-02
epoch 4500, train mse = 1.9205e-02
epoch 5000, train mse = 1.6635e-02
epoch 10000, train mse = 1.3209e-02
epoch 25000, train mse = 1.2906e-02
epoch 25500, train mse = 1.2902e-02
epoch 26000, train mse = 1.2899e-02
epoch 26500, train mse = 1.2900e-02
epoch 27000, train mse = 1.2892e-02
epoch 27500, train mse = 1.2889e-02
epoch 28000, train mse = 1.2886e-02
epoch 28500, train mse = 1.2874e-02
epoch 29000, train mse = 9.4067e-03
epoch 29500, train mse = 1.5621e-03
epoch 30000, train mse = 6.5884e-04
epoch 40000, train mse = 4.6610e-04
epoch 50000, train mse = 4.5693e-04
epoch 100000, train mse = 4.4939e-04
epoch 150000, train mse = 4.6481e-04
epoch 200000, train mse = 4.4736e-04
final train mse = 4.5028e-04
final test  mse = 6.0630e-04

=== width m = 2 ===
 restart 0
 restart 1
 restart 2

=== width m = 4 ===
 restart 0
 restart 1
 restart 2

=== width m = 8 ===
 restart 0
 restart 1
 restart 2

=== width m = 16 ===
 restart 0
 restart 1
 restart 2

```

```
=== width m = 32 ===
 restart 0
 restart 1
 restart 2

=== width m = 64 ===
 restart 0
 restart 1
 restart 2

=== width m = 128 ===
 restart 0
 restart 1
 restart 2

estimated slopes (ReLU):
  alpha_E2   ~ 0.878
  alpha_Einf ~ 0.743

=== width m = 2 ===
 restart 0
 restart 1
 restart 2

=== width m = 4 ===
 restart 0
 restart 1
 restart 2

=== width m = 8 ===
 restart 0
 restart 1
 restart 2

=== width m = 16 ===
 restart 0
 restart 1
 restart 2

=== width m = 32 ===
 restart 0
 restart 1
 restart 2

=== width m = 64 ===
 restart 0
 restart 1
 restart 2

=== width m = 128 ===
 restart 0
 restart 1
 restart 2

estimated slopes (tanh):
  alpha_E2_tanh   ~ 0.159
  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

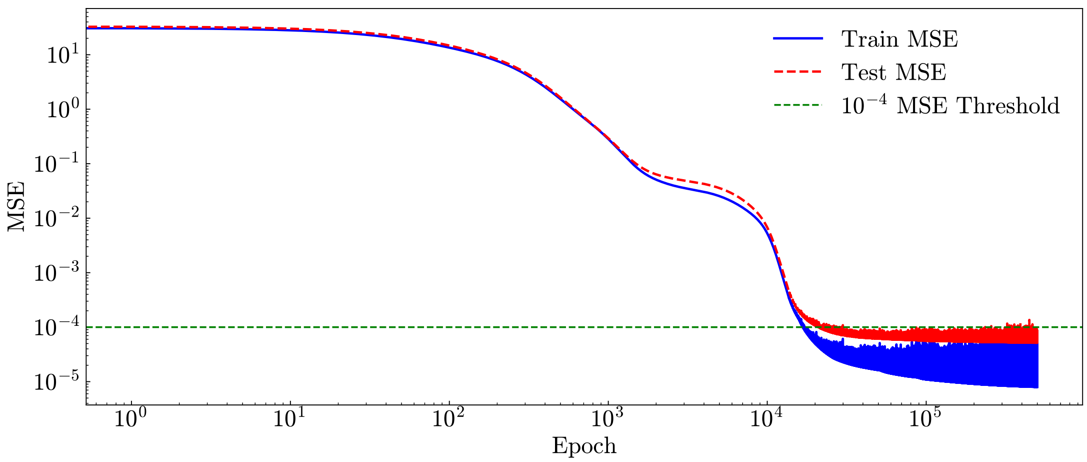


Figure 22: Training and testing MSE. Testing MSE is below 10^{-4} threshold.

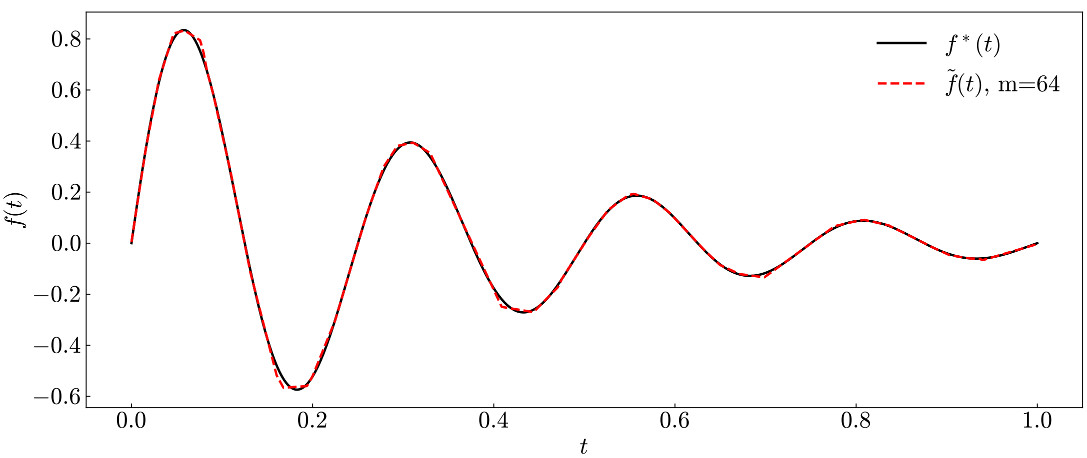


Figure 23: Learned $\tilde{f}(t;\theta)$ and target function $f^*(t)$.

Number of trainable parameters for the model: 193

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import jax
import jax.numpy as jnp
import optax

# target function f*
def f_star(t):
    return np.exp(-3.0 * t) * np.sin(8 * np.pi * t)

# train and test sets
rng = np.random.default_rng(0)
t_train = rng.uniform(0.0, 1.0, size=128)
y_train = f_star(t_train)

t_test = np.linspace(0.0, 1.0, 2048, endpoint=True)
y_test = f_star(t_test)

# plotting params
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = 'cmr10'
plt.rcParams['mathtext.fontset'] = 'cm'
plt.rcParams['font.size'] = 20
mpl.rcParams['axes.unicode_minus'] = False
plt.rcParams['axes.formatter.use_mathtext'] = True

# relu activation
def relu(x):
    return jnp.maximum(0.0, x)

# model initialization
def init_model(rng, m):
    # Initialize weights and biases for a single layer
    key_w, key_t, key_a, key_c = jax.random.split(rng, 4)
    w = 30.0 * jax.random.normal(key_w, (m,))
    t0 = jax.random.uniform(key_t, (m,), minval=0.0, maxval=1.0)  # kink locations
    b = -w * t0
    a = 0.1 * jax.random.normal(key_a, (m,))
    c = jnp.array(0.0)  # output bias
    return {"w": w, "b": b, "a": a, "c": c}

# forward pass
def model_apply(params, t, activation):
    z = params["w"] * t[:, None] + params["b"]
    h = activation(z)  # apply activation
    y_pred = jnp.dot(h, params["a"]) + params["c"]
    return y_pred

# mse loss function
def mse_loss(params, t, y, activation):
    y_pred = model_apply(params, t, activation)
    r = y_pred - y
    return jnp.mean(r * r)

# width and epochs
m = 64
epochs_d = 500000
```

```python

# optimizer (adam)
optimizer_d = optax.adam(learning_rate=1e-4)

# initialize parameters
rng = jax.random.PRNGKey(299)
params_d = init_model(rng, m)

opt_state_d = optimizer_d.init(params_d)

# activation outside the jit
activation = relu

# jit-compiled update step
@jax.jit
def step(params, opt_state, t, y):
    loss_val, grads = jax.value_and_grad(mse_loss)(params, t, y, activation)
    updates, opt_state = optimizer_d.update(grads, opt_state)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss_val

# training history
train_hist_d = []
test_hist_d  = []

# convert training data to jax arrays
t_train_j = jnp.asarray(t_train)
y_train_j = jnp.asarray(y_train)
t_test_j  = jnp.asarray(t_test)
y_test_j  = jnp.asarray(y_test)

# full batch training loop
for epoch in range(epochs_d):
    params_d, opt_state_d, train_loss = step(params_d, opt_state_d, t_train_j,
        y_train_j)

    train_hist_d.append(float(train_loss))

    # compute test mse
    test_loss = mse_loss(params_d, t_test_j, y_test_j, activation)
    test_hist_d.append(float(test_loss))

    if epoch % 5000 == 0:
        print(f"epoch {epoch}, train mse = {train_loss:.4e}, test mse = {test_loss
            :.4e}")

print(f"\nfinal train mse = {train_hist_d[-1]:.4e}")
print(f"final test  mse = {test_hist_d[-1]:.4e}")

# plot mse vs. epoch
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(train_hist_d, "b-", lw=2, label="Train MSE")
ax.loglog(test_hist_d,  "r--", lw=2, label="Test MSE")
ax.axhline(y=1e-4, color='g', linestyle='--', label=r"$10^{-4}$ MSE Threshold")

plt.xlabel("Epoch")
plt.ylabel("MSE")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
```

```python
plt.savefig(f"param_count_mse.pdf", dpi=1080)
plt.show()

# plot model and true function
y_pred_d = model_apply(params_d, t_test_j, activation)

fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t_test, y_test,     "k-", lw=2, label=r"$f^*(t)$")
ax.plot(t_test, y_pred_d,   "r--", lw=2, label=r"$\tilde f(t)$, m=64")
plt.xlabel(r"$t$")
plt.ylabel(r"$f(t)$")
plt.legend(frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig(f"model_true_bonus.pdf", dpi=1080)
plt.show()

# count the number of trainable parameters
def count_params_pytree(params):
    leaves = jax.tree_util.tree_leaves(params)  # Collect all leaf arrays from the
         nested params tree
    sizes = [int(jnp.size(x)) for x in leaves]  # Get the size of each array
    total = sum(sizes)  # Get the total number of trainable scalars
    return total

num_params = count_params_pytree(params_d)
print(f"Total number of trainable parameters: {num_params}")
```

Listing 5: bonus.py

```
epoch 0, train mse = 3.0913e+01, test mse = 3.2876e+01
epoch 5000, train mse = 2.5403e-02, test mse = 3.5700e-02
epoch 10000, train mse = 5.2675e-03, test mse = 6.7734e-03
epoch 15000, train mse = 1.8110e-04, test mse = 2.4571e-04
epoch 20000, train mse = 4.9779e-05, test mse = 1.0733e-04
epoch 25000, train mse = 2.9018e-05, test mse = 8.0896e-05
epoch 30000, train mse = 2.2734e-05, test mse = 7.1012e-05
epoch 35000, train mse = 1.9923e-05, test mse = 6.6513e-05
epoch 40000, train mse = 1.8257e-05, test mse = 6.4073e-05
epoch 45000, train mse = 3.8141e-05, test mse = 7.7431e-05
epoch 50000, train mse = 1.6273e-05, test mse = 6.1531e-05
epoch 55000, train mse = 1.4964e-05, test mse = 6.0510e-05
epoch 60000, train mse = 1.6612e-05, test mse = 6.3971e-05
epoch 65000, train mse = 1.3681e-05, test mse = 5.9280e-05
epoch 70000, train mse = 1.3034e-05, test mse = 5.8731e-05
epoch 75000, train mse = 1.2594e-05, test mse = 5.8321e-05
epoch 80000, train mse = 1.3746e-05, test mse = 5.7994e-05
epoch 85000, train mse = 1.2245e-05, test mse = 5.7767e-05
epoch 90000, train mse = 1.1703e-05, test mse = 5.7084e-05
epoch 95000, train mse = 1.1200e-05, test mse = 5.6436e-05
epoch 100000, train mse = 1.0960e-05, test mse = 5.6204e-05
epoch 105000, train mse = 1.0817e-05, test mse = 5.6085e-05
epoch 110000, train mse = 1.0642e-05, test mse = 5.5935e-05
epoch 115000, train mse = 1.0502e-05, test mse = 5.5807e-05
epoch 120000, train mse = 1.0380e-05, test mse = 5.5674e-05
epoch 125000, train mse = 1.0254e-05, test mse = 5.5560e-05
epoch 130000, train mse = 1.0141e-05, test mse = 5.5438e-05
epoch 135000, train mse = 1.2275e-05, test mse = 5.5313e-05
epoch 140000, train mse = 9.9387e-06, test mse = 5.5188e-05
epoch 145000, train mse = 9.9700e-06, test mse = 5.5180e-05
```

```
epoch 150000, train mse = 9.7648e-06, test mse = 5.4944e-05
epoch 155000, train mse = 9.6761e-06, test mse = 5.4836e-05
epoch 160000, train mse = 9.5983e-06, test mse = 5.4731e-05
epoch 165000, train mse = 9.5245e-06, test mse = 5.4629e-05
epoch 170000, train mse = 1.1408e-05, test mse = 6.1970e-05
epoch 175000, train mse = 9.3879e-06, test mse = 5.4430e-05
epoch 180000, train mse = 9.3258e-06, test mse = 5.4346e-05
epoch 185000, train mse = 9.2657e-06, test mse = 5.4255e-05
epoch 190000, train mse = 9.2100e-06, test mse = 5.4172e-05
epoch 195000, train mse = 9.1572e-06, test mse = 5.4099e-05
epoch 200000, train mse = 9.2161e-06, test mse = 5.4104e-05
epoch 205000, train mse = 9.0574e-06, test mse = 5.3947e-05
epoch 210000, train mse = 2.1596e-05, test mse = 6.0204e-05
epoch 215000, train mse = 8.9677e-06, test mse = 5.3808e-05
epoch 220000, train mse = 8.9262e-06, test mse = 5.3749e-05
epoch 225000, train mse = 8.8868e-06, test mse = 5.3693e-05
epoch 230000, train mse = 1.6735e-05, test mse = 6.4760e-05
epoch 235000, train mse = 8.8134e-06, test mse = 5.3581e-05
epoch 240000, train mse = 9.9345e-06, test mse = 5.5015e-05
epoch 245000, train mse = 8.7462e-06, test mse = 5.3480e-05
epoch 250000, train mse = 8.7548e-06, test mse = 5.3424e-05
epoch 255000, train mse = 8.6854e-06, test mse = 5.3384e-05
epoch 260000, train mse = 8.9200e-06, test mse = 5.3357e-05
epoch 265000, train mse = 9.0467e-06, test mse = 5.3305e-05
epoch 270000, train mse = 8.8819e-06, test mse = 5.3604e-05
epoch 275000, train mse = 8.5897e-06, test mse = 5.3236e-05
epoch 280000, train mse = 8.5883e-06, test mse = 5.3157e-05
epoch 285000, train mse = 8.5318e-06, test mse = 5.3125e-05
epoch 290000, train mse = 8.5077e-06, test mse = 5.3074e-05
epoch 295000, train mse = 8.7658e-06, test mse = 5.3229e-05
epoch 300000, train mse = 8.4653e-06, test mse = 5.2991e-05
epoch 305000, train mse = 8.4452e-06, test mse = 5.2938e-05
epoch 310000, train mse = 8.4256e-06, test mse = 5.2894e-05
epoch 315000, train mse = 8.4069e-06, test mse = 5.2853e-05
epoch 320000, train mse = 8.3887e-06, test mse = 5.2811e-05
epoch 325000, train mse = 8.3822e-06, test mse = 5.2769e-05
epoch 330000, train mse = 8.3547e-06, test mse = 5.2728e-05
epoch 335000, train mse = 8.3381e-06, test mse = 5.2691e-05
epoch 340000, train mse = 8.3226e-06, test mse = 5.2652e-05
epoch 345000, train mse = 8.3161e-06, test mse = 5.2611e-05
epoch 350000, train mse = 8.2919e-06, test mse = 5.2574e-05
epoch 355000, train mse = 8.2774e-06, test mse = 5.2536e-05
epoch 360000, train mse = 8.2632e-06, test mse = 5.2498e-05
epoch 365000, train mse = 8.2498e-06, test mse = 5.2462e-05
epoch 370000, train mse = 8.2370e-06, test mse = 5.2427e-05
epoch 375000, train mse = 1.1649e-05, test mse = 5.4893e-05
epoch 380000, train mse = 8.2260e-06, test mse = 5.2353e-05
epoch 385000, train mse = 8.1994e-06, test mse = 5.2323e-05
epoch 390000, train mse = 8.5365e-06, test mse = 5.2935e-05
epoch 395000, train mse = 8.1763e-06, test mse = 5.2259e-05
epoch 400000, train mse = 8.1652e-06, test mse = 5.2226e-05
epoch 405000, train mse = 8.1543e-06, test mse = 5.2194e-05
epoch 410000, train mse = 8.1439e-06, test mse = 5.2162e-05
epoch 415000, train mse = 8.1337e-06, test mse = 5.2130e-05
epoch 420000, train mse = 8.1236e-06, test mse = 5.2102e-05
epoch 425000, train mse = 8.1134e-06, test mse = 5.2073e-05
epoch 430000, train mse = 8.1042e-06, test mse = 5.2044e-05
epoch 435000, train mse = 8.0949e-06, test mse = 5.2015e-05
epoch 440000, train mse = 8.0857e-06, test mse = 5.1984e-05
```

```
epoch 445000, train mse = 8.0769e-06, test mse = 5.1956e-05
epoch 450000, train mse = 1.1278e-05, test mse = 5.6714e-05
epoch 455000, train mse = 8.9902e-06, test mse = 5.3759e-05
epoch 460000, train mse = 8.0521e-06, test mse = 5.1872e-05
epoch 465000, train mse = 8.0434e-06, test mse = 5.1846e-05
epoch 470000, train mse = 8.0364e-06, test mse = 5.1817e-05
epoch 475000, train mse = 8.0286e-06, test mse = 5.1789e-05
epoch 480000, train mse = 8.0213e-06, test mse = 5.1763e-05
epoch 485000, train mse = 8.0133e-06, test mse = 5.1738e-05
epoch 490000, train mse = 8.0065e-06, test mse = 5.1713e-05
epoch 495000, train mse = 8.3591e-06, test mse = 5.1846e-05

final train mse = 7.9922e-06
final test  mse = 5.1661e-05

Total number of trainable parameters: 193
```

Listing 6: Output terminal for `bonus.py`