Assignment 1

Deep Learning `CS4157`

by Dipyaman Roy (SE21UCSE056) and Param Shah (SE21UCSE143)

Model and results from the first assignment of Deep Learning (CS4157), involving creating a neural network from scratch, to predict the Combined Cycle Power Plant Dataset's net hourly electrical energy output (EP) value.

Contents

> **Note** (README.md)**.**
> To find the instructions on how to execute the programs, refer to the `README.md` document provided. This document only contains model overview, results and conclusions.

0.1. Model overview

0.1.1. Features

- **Optimizer Integration:** Supports Adam, Momentum, and Gradient Descent optimizers.
- **Regularization:** Supports L2 and **Dropout** regularization.
- **Activation Functions:** Supports sigmoid, ReLU, and tanh activation functions.
- **He Initialization:** Supports He initialization for weights.
- **Early Stopping:** Supports early stopping during training.
- **R^2 Scatter Plot:** Plots predicted vs true values.
- **Modularization:** Calls activations and costs from elsewhere, making it flexible and reusable.

0.1.2. Pseudocode

Pseudocode 0.1.2.1 (network.py).

Here is the overview of the `network.py`'s `Network` class, written in CLRS[1] style.

Pseudocode

```
## Initialize-Parameters ##
INPUT: layer-dimensions, learning-rate, mini-batch-size,
       optimizer, activation, regularisation, lambd, cost-function
OUTPUT: initialized parameters
1. Initialize parameters dictionary
2. For each layer from 1 to L-1
    a. Initialize weights and biases
    b. Store weights and biases in parameters dictionary
3. Return parameters

## Initialize-Optimizer ##
INPUT: optimizer, learning-rate, beta, beta1, beta2, epsilon
OUTPUT: initialized optimizer parameters
1. If optimizer is 'adam'
    a. Initialize v and s dictionaries
2. If optimizer is 'momentum'
    a. Initialize v dictionary
3. Return optimizer parameters

## Network-Forward ##
INPUT: input-data, parameters, activation
OUTPUT: output, caches
1. Initialize output and caches
2. For each layer from 1 to L-1
    a. Compute linear-forward and activation-forward
    b. Store output and cache
3. Compute linear-forward and activation-forward for last layer
4. Store output and cache
5. Return output and caches

## Network-Backward ##
INPUT: output, true-values, caches, activation, regularisation, lambd, cost-function
OUTPUT: gradients
1. Initialize gradients
2. Compute gradients for last layer
3. For each layer from L-1 to 1
    a. Compute gradients for current layer
4. Return gradients

## Update-Parameters ##
INPUT: parameters, gradients, learning-rate
OUTPUT: updated parameters
1. For each layer from 1 to L
    a. Update weights and biases using gradients and learning-rate
2. Return updated parameters

## Update-Parameters-With-Optimizer ##
INPUT: parameters, gradients, optimizer-parameters, learning-rate, beta, beta1, beta2, epsilon
OUTPUT: updated parameters and optimizer parameters
1. If optimizer is 'adam'
    a. Update parameters using Adam optimizer
    b. Update v and s dictionaries
2. If optimizer is 'momentum'
```

```
    a. Update parameters using momentum optimizer
    b. Update v dictionary
3. Return updated parameters and optimizer parameters

## Train ##
INPUT: input-data, true-values, valid-data, valid-values,
       number-of-iterations, print-cost, early-stopping-patience
OUTPUT: trained model
1. Initialize parameters
2. Initialize optimizer parameters
3. For each iteration from 1 to number-of-iterations
    a. For each mini-batch
        i. Compute network-forward
        ii. Compute network-backward
        iii. Update parameters using optimizer
    b. Compute cost
    c. Print cost if required
    d. Check early-stopping if required
4. Return trained model

## Predict ##
INPUT: input-data
OUTPUT: predicted values
1. Compute network-forward
2. Return predicted values

## Predicted-Error ##
INPUT: input-data, true-values
OUTPUT: predicted error
1. Compute network-forward
2. Compute cost
3. Return predicted error
```

[1]Cormen, Thomas H., et al. *Introduction to Algorithms*. 3rd ed., MIT Press, 2009.

1. Toy problem: Learning the sine function

1.1. Definition

Problem 1.1.1 (ANN for sine)**.**
Develop and validate an Artificial Neural Network (ANN) to learn the mathematical function $y = \sin(x)$ over the domain $-2\pi \leq x \leq 2\pi$. The aim is to assess the performance of the ANN through training and validation using different techniques, with a final goal of generating a model whose predicted outputs overlap the true $y = \sin(x)$ values.

1.2. Architecture

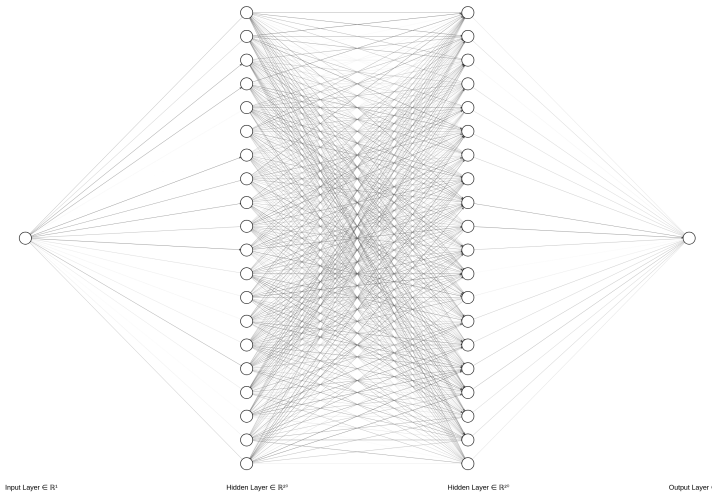
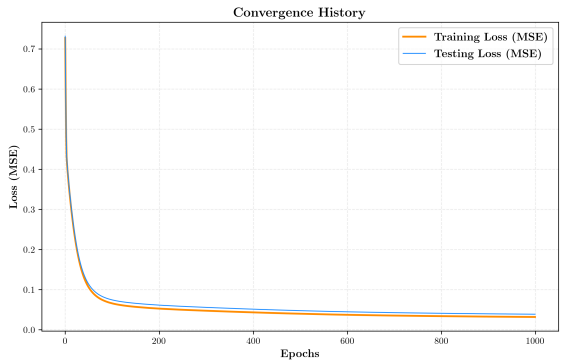


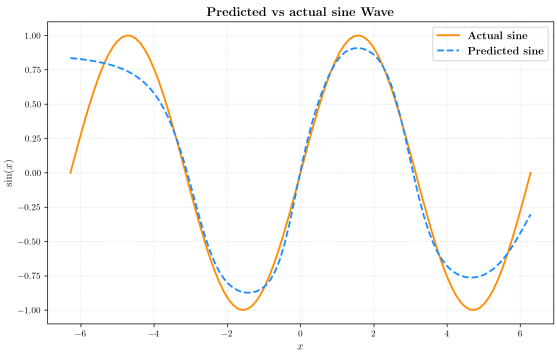
Figure 1: Architecture used for Neural network [1, 20, 20, 1][2]

1.2.1. Tested configurations

Configuration 1
Learning Rate (η) = 0.01, No. of Epochs: 1000, Mini-batch size: 64, Activation: tanh

Convergence history	Predicted vs actual $\sin(x)$



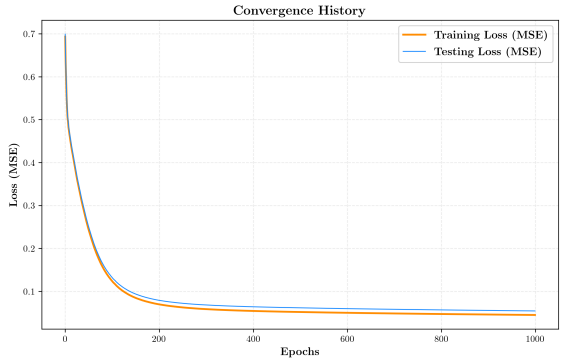


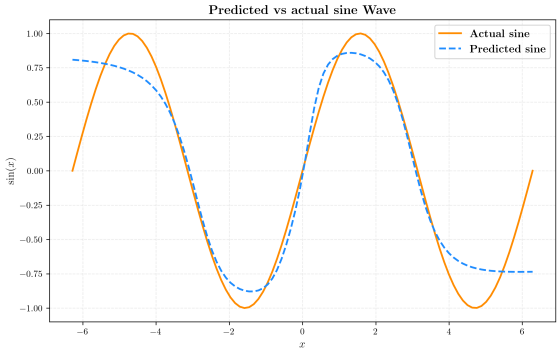
[2]LeNail, (2019). NN-SVG: Publication-Ready Neural Network Architecture Schematics. Journal of Open Source Software, 4(33), 747, https://doi.org/10.21105/joss.00747

Configuration 2

Learning Rate (η) = 0.01, No. of Epochs: 1000, Mini-batch size: 256, Activation: tanh

Convergence history	Predicted vs actual $\sin(x)$

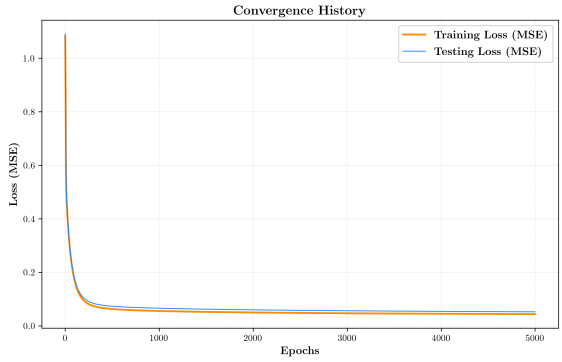


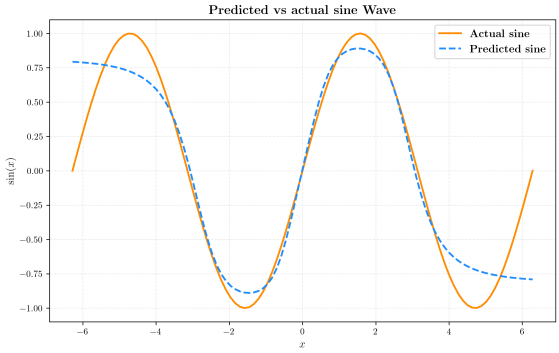


Configuration 3

Learning Rate (η) = 0.01, No. of Epochs: 5000, Mini-batch size: 256, Activation: tanh

Convergence history	Predicted vs actual $\sin(x)$

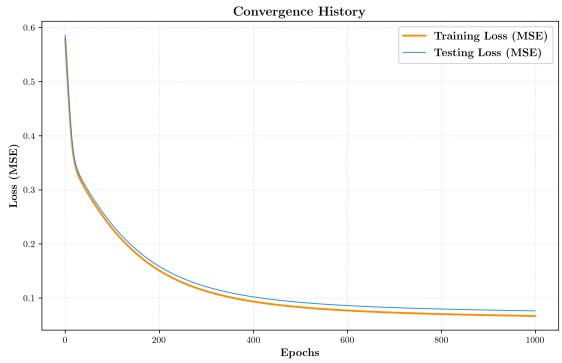


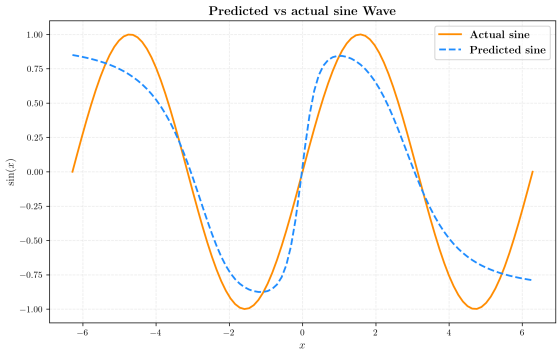


Configuration 4

Learning Rate (η) = 0.001, No. of Epochs: 1000, Mini-batch size: 64, Activation: tanh

Convergence history	Predicted vs actual $\sin(x)$

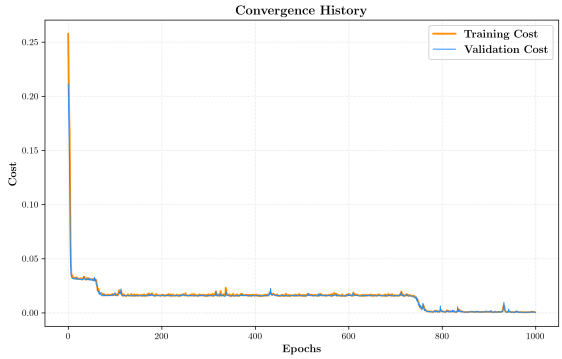


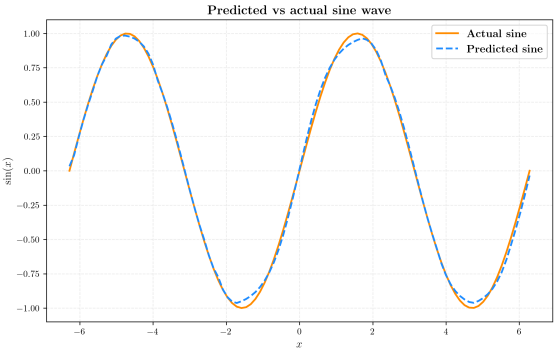


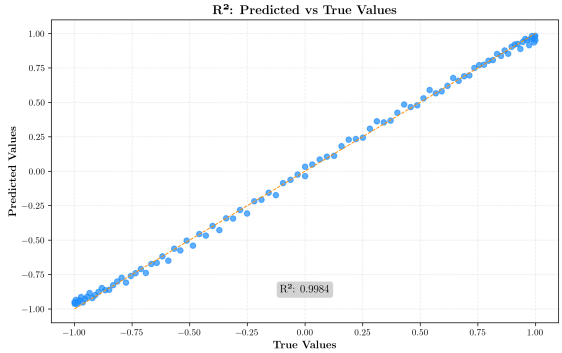
Final configuration

Learning Rate (η) = 0.01, No. of Epochs: 1000, Mini-batch size: 64,
Activation: tanh, Regularizer: L2 ($\lambda = 0.1$), Optimizer: Adam ($\beta_1 = 0.9$, $\beta_2 = 0.99$, $\varepsilon = 10^{-8}$)

Loss/Error metrics	Predicted vs actual $\sin(x)$







2. Learning the Combined Cycle Power Plant Dataset

2.1. Definition

Problem 2.1.1 (ANN for CCPP Dataset)**.**
Develop a custom Artificial Neural Network (ANN) from scratch to effectively model and predict the output variable of the Combined Cycle Power Plant dataset[3]. The dataset contains over 9,000 rows with 4 input features and 1 output feature, representing a complex relationship between them.

The objective is to demonstrate the ANN's capability to capture these intricate functional relationships, ensuring high accuracy in predictions while gaining a comprehensive understanding of the underlying mechanisms of machine learning. The solution should be efficient and robust, providing a foundation for future applications across various domains.

2.2. Results

We shall be taking a game-theoretic approach to figure out the configuration which best fits the CCPP dataset. This involves varying each of the following: architecture, mini-batch size, learning rate (η), activation function (tanh, relu, sigmoid), L2 regularization parameter (λ) and optimizer (adam, SGD with momentum or both); while keeping the others constant.

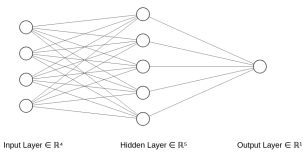
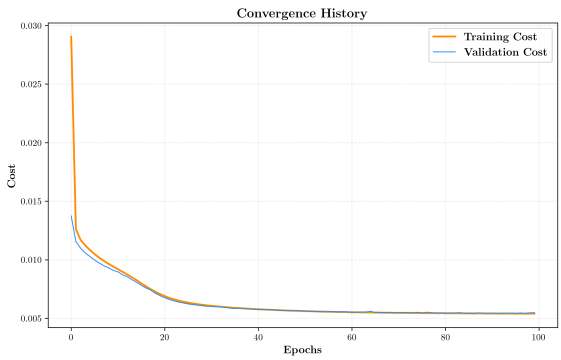
Early stopping (with a patience of 20 epochs) has been implemented. This means that the model stops iterating if the loss does not improve past 20 epochs.[4]

2.2.1. Architecture variation

First we start with varying the architecture with the following parameters constant: Learning Rate (η): 0.001, No. of Epochs: 100 (max), Activation: tanh, Minibatch Size: 1 (SGD).

Observations

Architecture and Errors	Convergence history

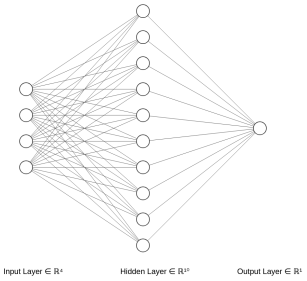
Architecture: [4, 5, 1]
`Validation MAPE: 56.40076953063576`
`Test MAPE: 65.55567087368057`
 
 	
	

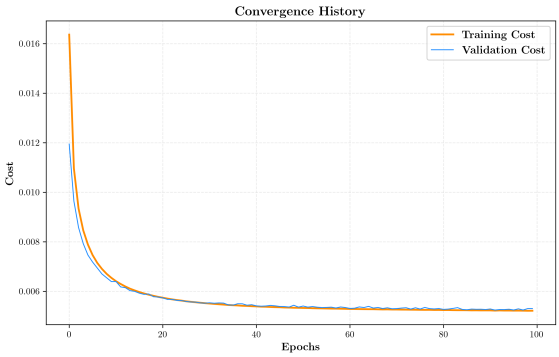
[3] Tfekci, Pnar and Heysem Kaya. 2014. Combined Cycle Power Plant. UCI Machine Learning Repository. https://doi.org/10.24432/C5002N.

[4] Thus, there may be a discrepancy in the Number of epochs (x-axis) in the charts below, but it will help us visualize which iteration converges fastest.

Architecture: [4, 10, 1]

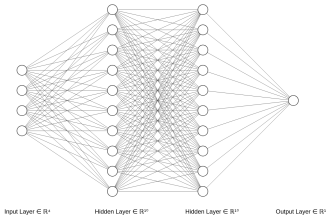
```
Validation MAPE: 62.19152555444135
Test MAPE: 59.99054926641497
```

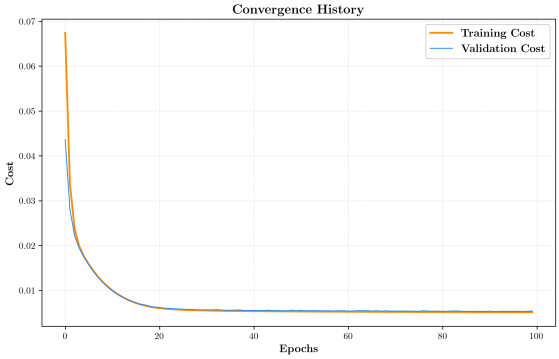




Architecture: [4, 10, 10, 1]

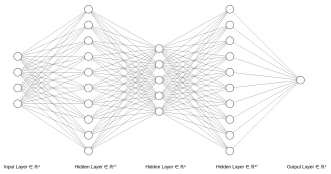
```
Validation MAPE: 62.35603080622679
Test MAPE: 74.73334953921075
```

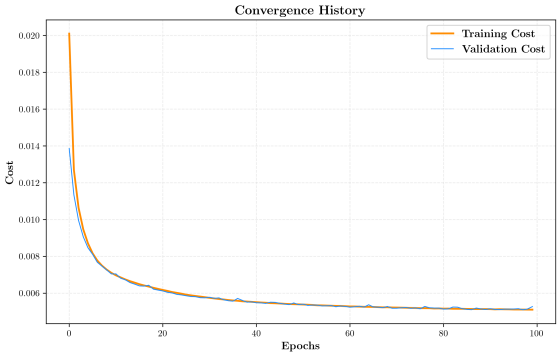




Architecture: [4, 10, 5, 10, 1]

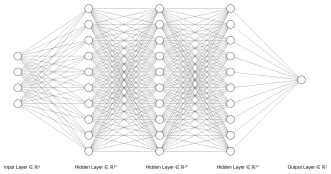
```
Validation MAPE: 59.69325897149324
Test MAPE: 64.85998603944367
```

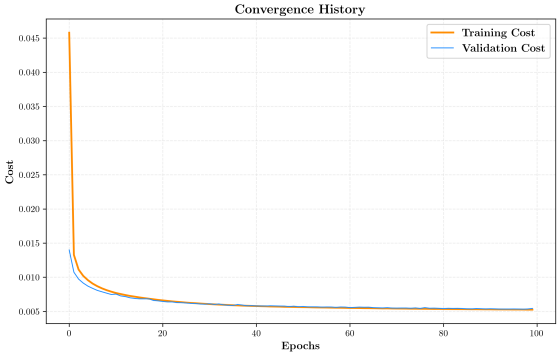




Architecture: [4, 10, 10, 10, 1]

```
Validation MAPE: 60.10262604257819
Test MAPE: 62.55688103421784
```





Conclusions

- Architecture: [4, 5, 1] had the lowest validation MAPE (56.40) and a moderate test MAPE (65.56). This simple architecture performed well, likely due to avoiding overfitting, though it lacked the flexibility to capture more complex patterns.

- Architecture: [4, 10, 1] showed an increase in validation MAPE (62.19), but its test MAPE improved to 59.99. The larger hidden layer likely allowed for better fitting, but also led to slightly worse validation performance, indicating potential overfitting.
- Architecture: [4, 10, 10, 1] resulted in a higher test MAPE (74.73) and validation MAPE (62.36). The increased model complexity did not improve generalization and likely caused overfitting, with worse performance on the test set.
- Architecture: [4, 10, 5, 10, 1] provided better validation MAPE (59.69) than some other complex models, with test MAPE (64.86) being reasonable. The introduction of an additional hidden layer offered a balance between complexity and generalization, though still not as optimal as simpler architectures.
- Architecture: [4, 10, 10, 10, 1] improved over some of the more complex models, with validation MAPE (60.10) and test MAPE (62.56), but did not outperform simpler models like [4, 5, 1]. The deeper architecture likely helped capture more intricate patterns but struggled with generalization.

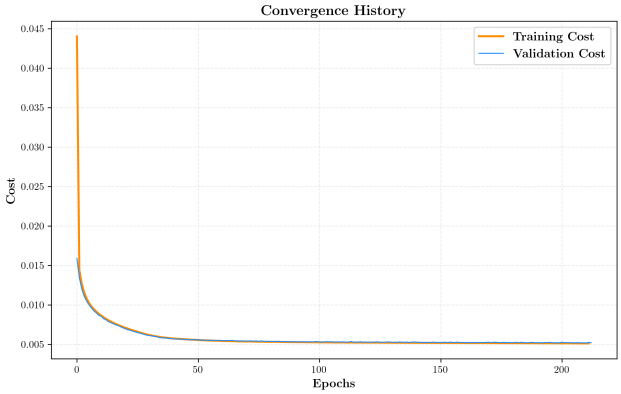
Overall, simpler architectures like [4, 5, 1] and [4, 10, 1] offered better test performance, while more complex architectures tended to overfit despite having more capacity.

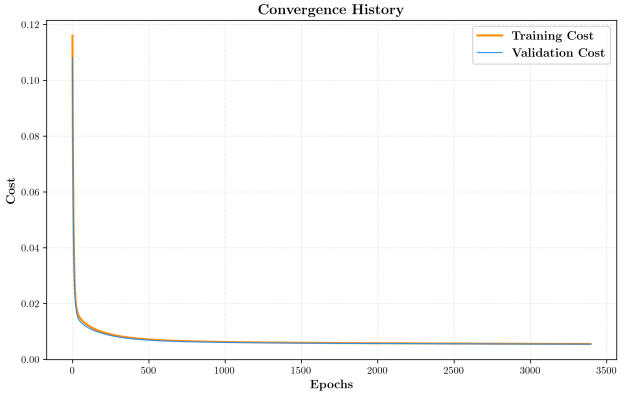
2.2.2. Mini-batch size variation

Keeping the following parameters constant: Architecture: [4, 10, 10, 1], Learning Rate (η): 0.001, No. of Epochs: 10000 (max), Activation: tanh.

Observations

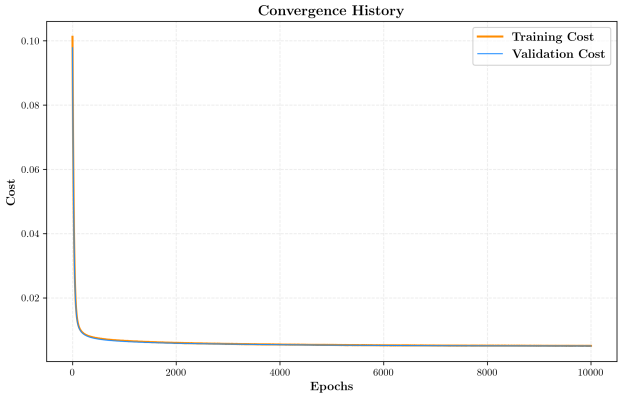
Mini-batch size variations and Errors	Convergence history

Mini-batch size: 1 (SGD)
Early-stopped after Epoch 211
`Validation MAPE: 53.693988297439596`
`Test MAPE: 68.02095593335363`
	

Mini-batch size: 64
Early-stopped after Epoch 3395
`Validation MAPE: 53.723619561686256`
`Test MAPE: 63.682885551413314`
	

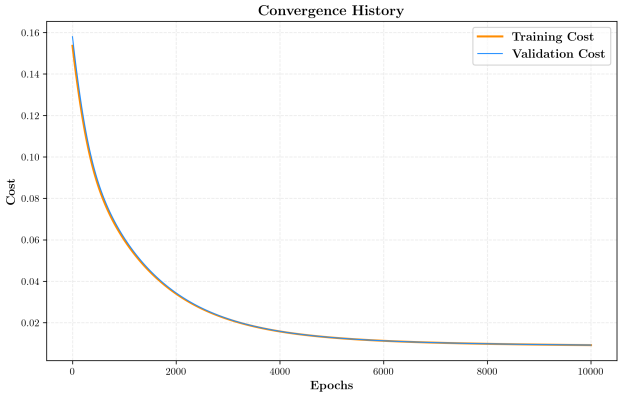
Mini-batch size: 128

```
Validation MAPE: 89.55034219372575
Test MAPE: 50.595414749224844
```



Mini-batch size: 6888 (Full-batch)

```
Validation MAPE: 61.40478683063642
Test MAPE: 63.652844872591594
```



Conclusions

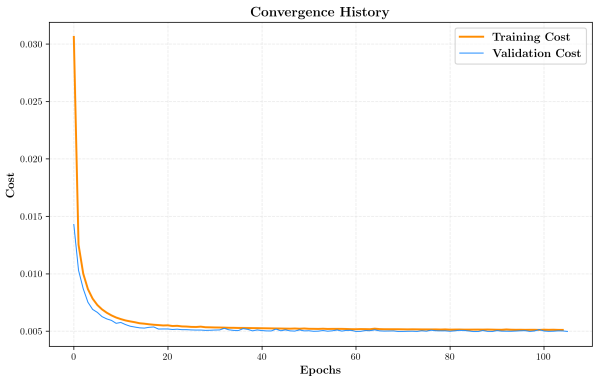
- SGD (Mini-batch size: 1) converges much faster than larger mini-batches, likely due to its ability to escape local minima and explore the loss landscape more effectively. Early stopping at epoch 211 indicates rapid learning, though the final test MAPE was slightly higher (68.02).
- As mini-batch size increases, computations per epoch become faster due to vectorization, but learning itself slows down. The mini-batch size of 64, which converged after 3395 epochs, showcases the balance between computational efficiency and slower learning, with a better test MAPE (63.68).
- For mini-batch size 128, the test MAPE was very low (50.59), but the validation MAPE was much higher, suggesting that this batch size overfit the data.
- Finally, with the full batch size (6888), convergence was extremely slow. The model's ability to generalize was average, as shown by its test MAPE (63.65), but the slow convergence indicates that averaging over the entire dataset at once reduces the stochasticity needed to push the model out of local minima, leading to slow learning.

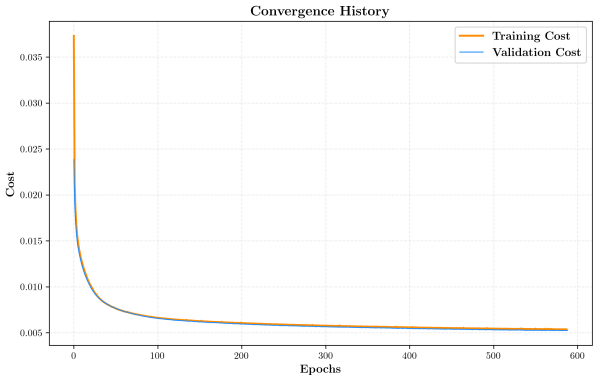
2.2.3. Learning rate (η) variation

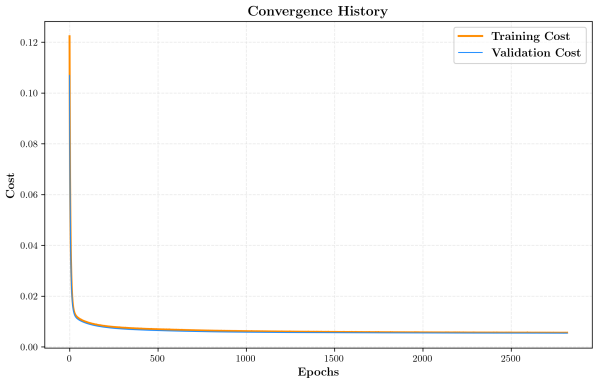
Keeping the following parameters constant: Architecture: [4, 10, 10, 1], No. of Epochs: 10000 (max), Activation: tanh, Minibatch Size: 64.

Observations

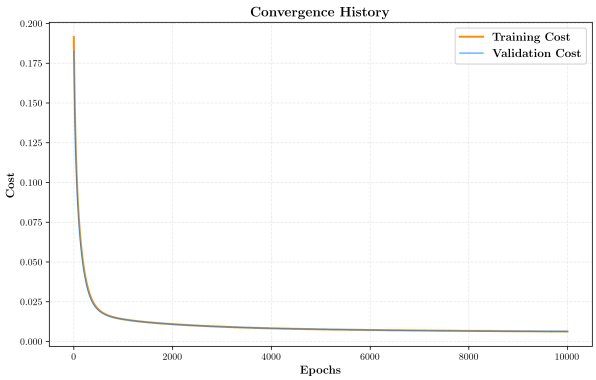
Learning rate (η) variations and Errors	Convergence history

Learning rate (η): 0.1
Early-stopped after Epoch 104
`Validation MAPE: 147.1386366518392`
`Test MAPE: 137.98690986656962`
	

Learning rate (η): 0.01
Early-stopped after Epoch 587
`Validation MAPE: 59.1783649499945`
`Test MAPE: 97.73856363362135`
	

Learning rate (η): 0.001
Early-stopped after Epoch 2817
`Validation MAPE: 57.168233365600216`
`Test MAPE: 64.77762968811736`
	

Learning rate (η): 0.0001
`Validation MAPE: 60.60312284754604`
`Test MAPE: 67.814343547627`



Conclusions

- Learning rate $\eta = 0.1$ led to very fast convergence, early-stopping after only 104 epochs. However, the model overfit badly, resulting in high validation MAPE (147.14) and test MAPE (137.99). This suggests the learning rate was too high, causing the model to miss optimal solutions and oscillate around the minima.
- Learning rate $\eta = 0.01$ improved performance, with a validation MAPE of 59.18 and test MAPE of 97.74 after early-stopping at epoch 587. Though it allowed for better convergence than $\eta = 0.1$, the test error was still relatively high.
- Learning rate $\eta = 0.001$ achieved the best balance between speed and accuracy. While it required more iterations (early-stopped at epoch 2817), the validation MAPE dropped to 57.17 and the test MAPE to 64.78, indicating improved generalization and convergence stability.
- Learning rate $\eta = 0.0001$ showed the slowest convergence. Though it avoided overfitting, both validation (60.60) and test MAPE (67.81) were higher, suggesting that the rate was too low for effective learning in a reasonable number of epochs.

Overall, $\eta = 0.001$ provided the best trade-off between convergence speed and generalization, while extremely high or low learning rates led to either overfitting or slow learning.

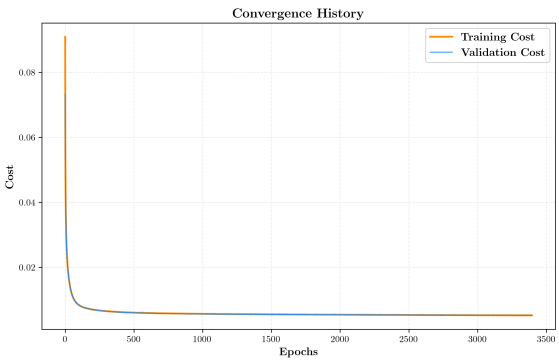
2.2.4. Activation functions' variation

Keeping the following parameters constant: Architecture: [4, 10, 10, 1], No. of Epochs: 10000 (max), Minibatch Size: 64. Varying the activation functions between tanh, relu, sigmoid

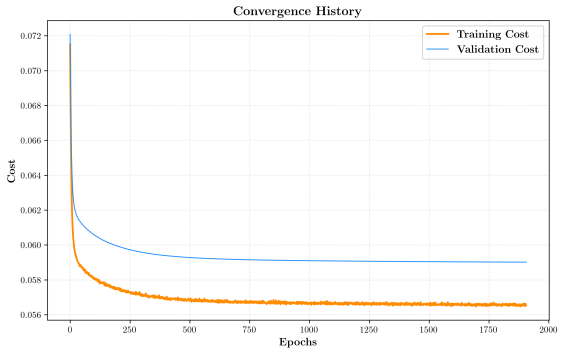
Observations

Activation functions' variations and Errors	Convergence history

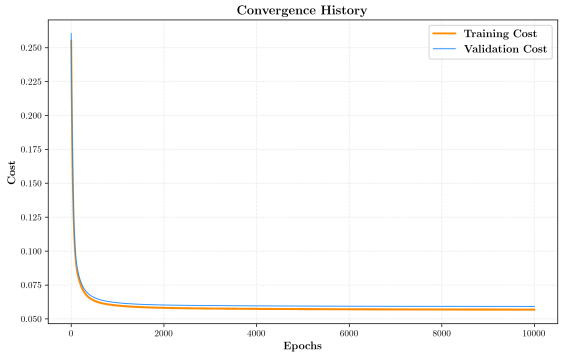
Activation function: tanh
Early-stopped after Epoch 3395
`Validation MAPE: 58.52558681866088`
`Test MAPE: 71.712402333593865`



Activation function: relu
Early-stopped after Epoch 1905
`Validation MAPE: 91.776591960264`
`Test MAPE: 105.20976090347`



Activation function: sigmoid
`Validation MAPE: 86.88217402491`
`Test MAPE: 89.613732710146`



Conclusions

- **Tanh** resulted in the best performance, with a validation MAPE of 58.53 and test MAPE of 71.71. Early stopping occurred after 3395 epochs, indicating relatively stable learning. The tanh function, known for smoothing gradients, likely helped achieve better generalization, though there is still some overfitting given the higher test MAPE.
- **ReLU** led to early stopping at epoch 1905, but the performance was significantly worse, with a validation MAPE of 91.78 and test MAPE of 105.21. While relu is generally favored for its efficiency and ability to handle larger architectures, it may have suffered from dead neurons or gradient instability in this case, leading to poorer convergence and higher errors.
- **Sigmoid** performed poorly compared to tanh, with a validation MAPE of 86.88 and test MAPE of 89.61. The slow convergence of the sigmoid function, combined with vanishing gradient issues, likely contributed to its lower accuracy, as it tends to saturate during training, making optimization harder.

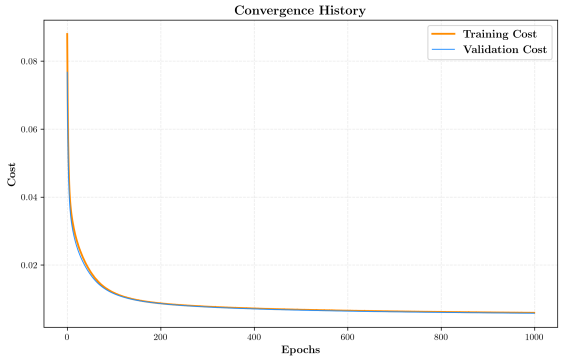
Overall, tanh was the most effective activation function, offering a balance between stable convergence and generalization, while both relu and sigmoid struggled with accuracy and convergence in this architecture.

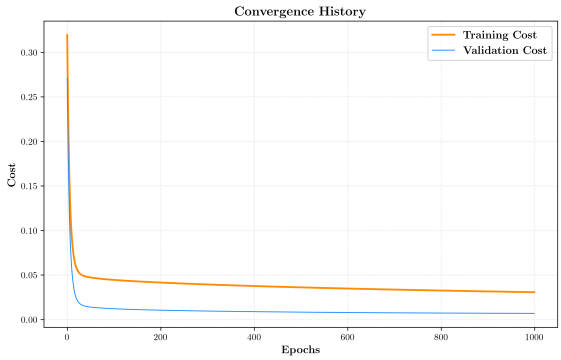
2.2.5. L2 regularization parameter (λ) variation

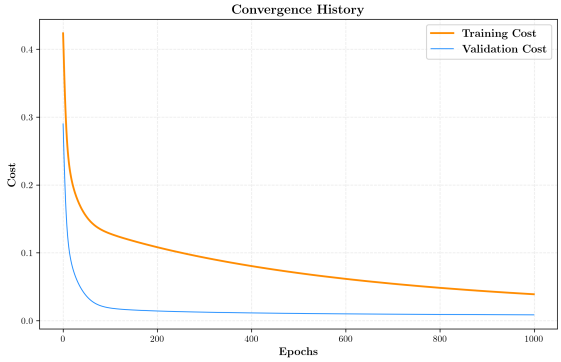
Keeping the following parameters constant: Architecture: [4, 10, 10, 1], Learning Rate (η): 0.001, No. of Epochs: 1000 (max), Activation: tanh, Minibatch Size: 64.

Observations

L2 regularization parameter (λ) variations and Errors	Convergence history

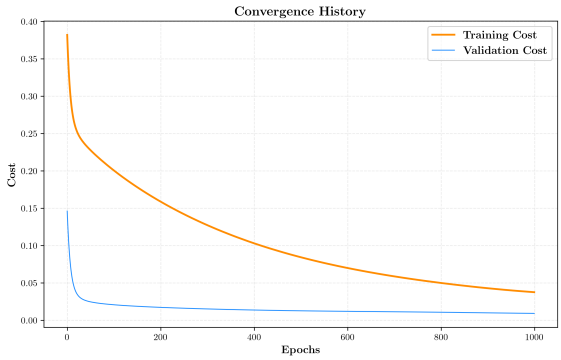
$\lambda = 0$
`Validation MAPE: 67.96135975214234`
`Test MAPE: 75.32647628768882`
	

$\lambda = 0.1$
`Validation MAPE: 64.5553469215373`
`Test MAPE: 79.30881777806314`
	

$\lambda = 0.5$
`Validation MAPE: 57.6698831018704`
`Test MAPE: 80.98407503918895`
	

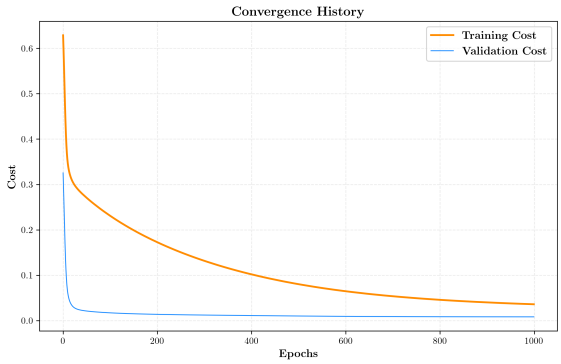
$\lambda = 0.75$

```
Validation MAPE: 52.1632371905876
Test MAPE: 76.52653655026393
```



$\lambda = 0.9$

```
Validation MAPE: 51.54916259941498
Test MAPE: 71.51602755262577
```



Conclusions

- With regularization, convergence is notably slower, as higher λ values introduce more bias into the model, which prevents overfitting but also slows down the model's learning. This is evident in the convergence history, where more iterations are required for the model to reach a minimum error compared to scenarios without regularization.
- Despite these changes, the MAPE values show only slight improvements with increasing λ, indicating that regularization primarily influences the model's ability to prevent overfitting rather than improving accuracy.

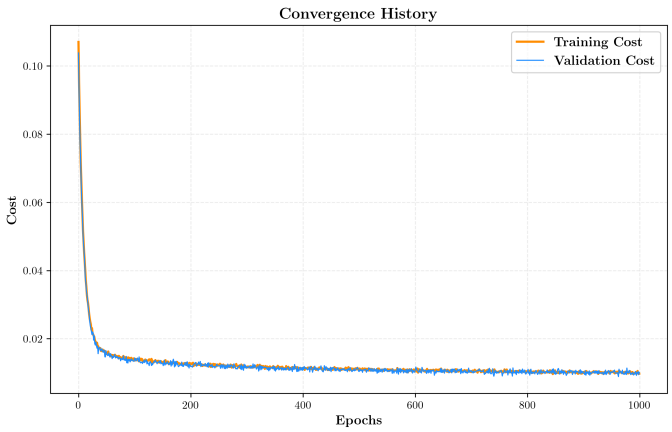
2.2.6. Dropout variation

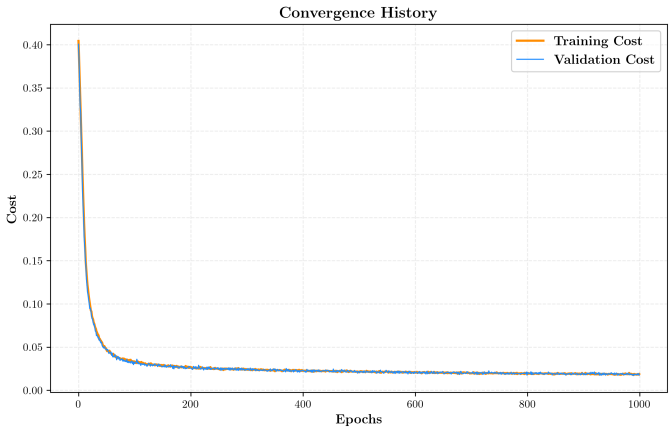
Keeping the following parameters constant: Architecture: [4, 10, 10, 1], Learning rate (η): 0.001, No. of Epochs: 1000 (max), Activation: tanh, Minibatch Size: 64.

Dropout keep means the percentage of data preserved after each epoch.

Observations

Dropout variations and Errors	Convergence history

Dropout keep = 0.95
`Validation MAPE: 90.98811435649307`
`Test MAPE: 101.25204207556962`
	

Dropout keep = 0.8
`Validation MAPE: 103.57150635827207`
`Test MAPE: 127.19708174518377`
	

Conclusions

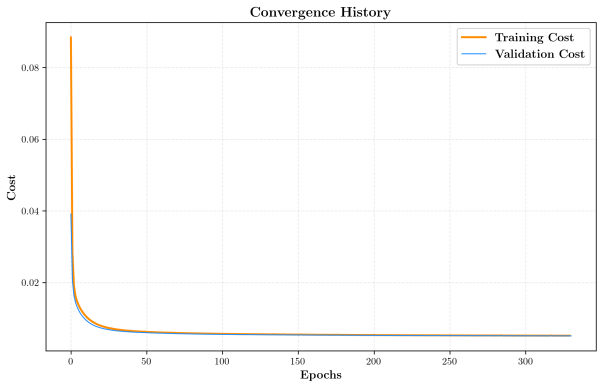
- The model's performance deteriorated with dropout, suggesting dropout can hinder learning and increase error rates.
- Training time was greatly reduced as the amount of data to iterate over become progressively lesser.

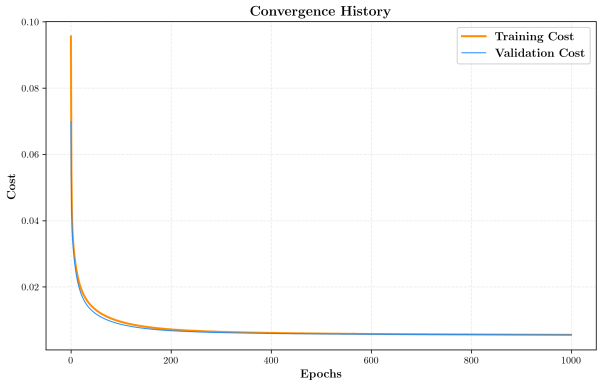
2.2.7. Performing SGD with momentum

Keeping the following parameters constant: Architecture: [4, 10, 10, 1], No. of Epochs: 1000 (max), Activation: tanh, Minibatch Size: 64, Momentum (β) = 0.9.

Observations

Learning rate (η) variations and Errors	Convergence history

$\eta = 0.01$
Early-stopped after Epoch 329
`Validation MAPE: 61.49276506021701`
`Test MAPE: 79.26198393747873`
	

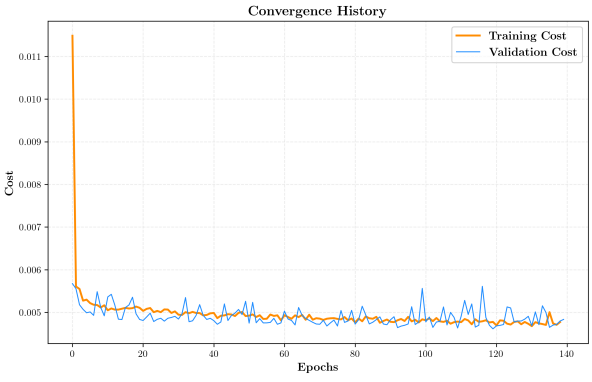
$\eta = 0.001$
`Validation MAPE: 65.54343315094347`
`Test MAPE: 79.09386599170688`
	

2.2.8. Bonus: Performing Adam + SGD with momentum

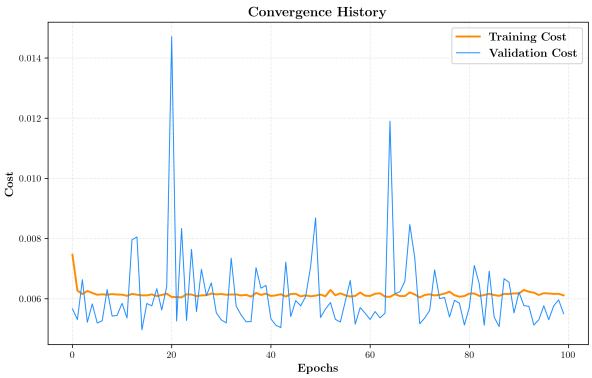
Keeping the following parameters constant: Architecture: [4, 10, 10, 1], No. of Epochs: 500 (max), Activation: tanh, Momentum (β) = 0.9, Adam parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$.

Observations

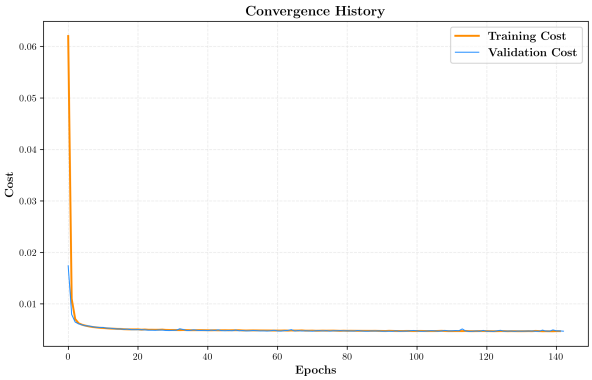
Minibatch Size, η variations and Errors	Convergence history

$\eta = 0.01$
Minibatch Size: 64
Early-stopped after Epoch 138
`Validation MAPE: 51.87322346430933`
`Test MAPE: 80.5299380249295`
	

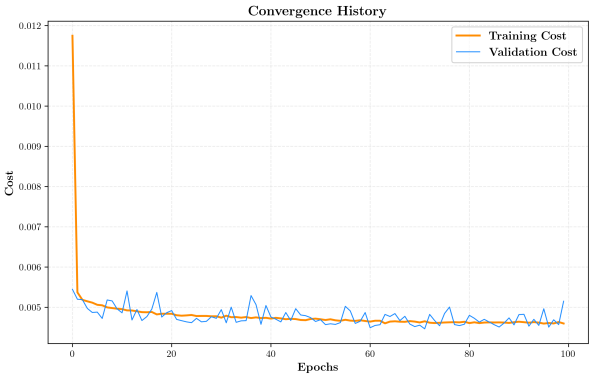
$\eta = 0.01$
Minibatch Size: 1 (SGD)
No. of epochs: 100 (Early-stopping disabled)
`Validation MAPE: 60.95508964281989`
`Test MAPE: 71.82800166722207`



$\eta = 0.001$ Minibatch Size: 64
Early-stopped after Epoch 138
`Validation MAPE: 53.67946144361848`
`Test MAPE: 66.37173096032316`



$\eta = 0.001$ Minibatch Size: 1 (SGD)
No. of epochs: 100 (Early-stopping disabled)
`Validation MAPE: 58.33988454766232`
`Test MAPE: 70.59275630746073`



Conclusions

- Learning Rate & Mini-batch Size: With a higher learning rate ($\eta = 0.01$) and minibatch size of 64, the model converged quickly but showed some signs of overfitting. Switching to SGD improved generalization but slowed convergence.
- Lower Learning Rate: Reducing the learning rate ($\eta = 0.001$) improved generalization for both mini-batch and SGD settings, though it required more iterations to converge.

2.2.9. Final configuration

The architecture used is [4, 10, 10, 10, 1] (Fig. 2), and parameters are: No. of Epochs: 500, Activation: tanh, Momentum (β) = 0.9, Adam parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$. This configuration demonstrated stable convergence and yielded reasonable accuracy on both the validation and test datasets.

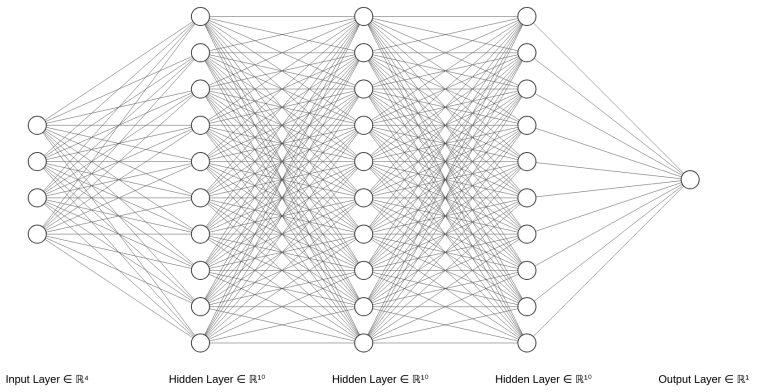


Figure 2: Architecture used for CCPP neural network [4, 10, 10, 10, 1]

The errors obtained were:
Validation MAPE: 55.81023433654681
Test MAPE: 62.588457055777184

And following are the convergence history plot and R^2 scatterplot.

Convergence history	R^2 scatterplot