Deep Learning : A brief study of optimization techniques

Indian Institute of Science Education and Research

Department of Mathematics and Statistics - DMS

21MS243

MISBAH UZ ZAMAN

Internship Report
supervised by DR. Ratikanta Behera

Numerical Algorithms and Tensor Learning Laboratory(NATL)

Department of Computational and Data Sciences- CDS

IISC Bangalore

Contents

List of Figures

List of Tables

Chapter 1

Basic Structure of a Neural Network

A neural network is an interconnected network of perceptrons which take data as input and output some result based on the their training. Neural networks are used to learn from training data and make predictions on unseen seen/test data, although a neural network can be used for a lot of different applications but the most common applications are classification and regression.

1.1 Mathematical structure of a neural network

1.1.1 The percetron

A perceptron is a simplest artificial neural network expressed mathematically by :

$$\hat{y} = \phi\left(\mathbf{w}^T\mathbf{x} + b\right)$$

where $\mathbf{w} \in \mathbf{R}^n$ is the weight vector and $\mathbf{x} \in \mathbf{R}^n$ is the input vector and $b \in \mathbf{R}$
The dot product $\mathbf{w}^T\mathbf{x}$ can be calculated as

$$\mathbf{w}^T\mathbf{x} = \left(\sum_{i=1}^{n} w_i x_i\right)$$

Thus the mathematical formula of perceptron becomes:

$$\hat{y} = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

here $\hat{y}$ is the estimated value of output variable and ϕ is activation function

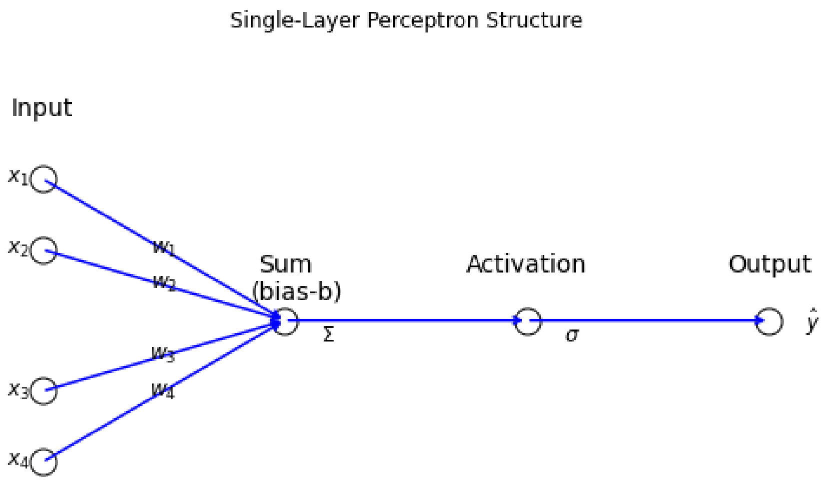


Figure 1.0.1: Basic structure of a perceptron.

1.1.2 The Activation function(ϕ)

The activation function is a non-linear function which is used to make the input data non-linear which helps in better prediction of output. some commonly used activation functions are RELU, Tanh, sigmoid etc.

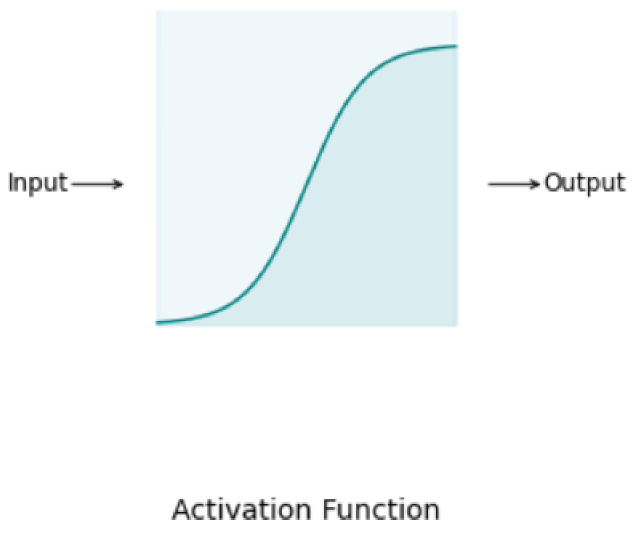


Figure 1.1.2: Example of a activation function

1.1.3 Multi layer perceptron(MLP)(ϕ)

Multi layer perceptron(MLP) is type of Artificial Neural Network (ANN) which is interconnected perceptrons with multiple layers , MLPs are used commonly to learn complex patterns in data as they use non-linear activation functions. They are commonly used for tasks such as classification, regression, and pattern recognition.

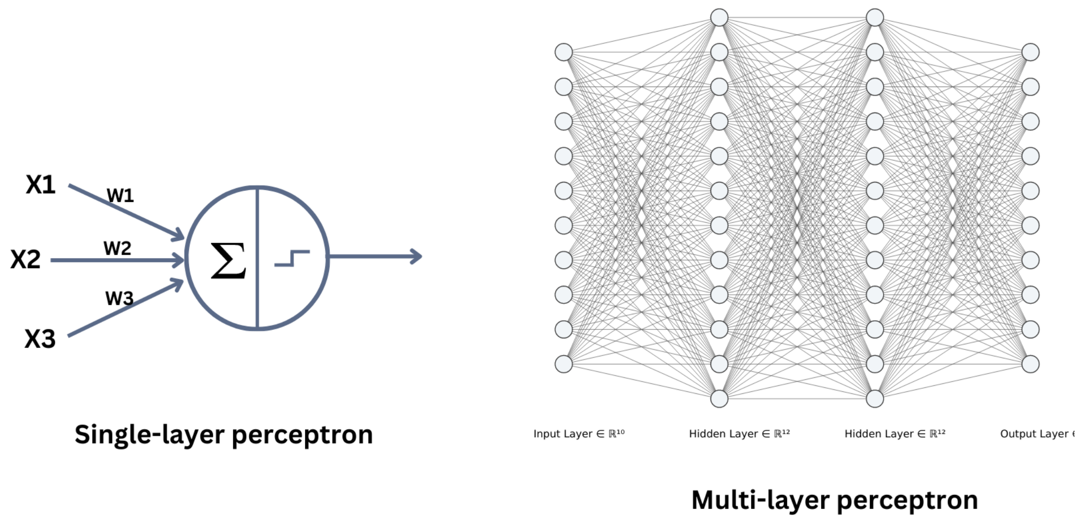


Figure 1.1.3: A schematic of Single Layer Perceptron and Multi Layer Perceptron

1.2 How does a Neural Network Learn

A Neural Network learns by optimizing the weights and biases of the model by backpropagating through the network on the basis of the calculated loss using the loss function.

1.2.1 Loss Function

The loss function is a function which quantifies how well a model behaves by calculating how much the estimated output differs from the expected output the goal of training a model is to minimize this loss function. some commonly used loss functions are MSE, cross-entropy loss etc.

Mean Squared Error (MSE)

Mean Squared Error is commonly used for regression tasks. It measures the average of the squares of the errors—that is, the average squared difference between the predicted values ($\hat{y}_i$) and the actual values (y_i).
Mathematically, it is represented as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Advantages

- Easy to compute and differentiable, which is essential for optimization algorithms like gradient descent.

Disadvantages

- Sensitive to outliers because the errors are squared, which can disproportionately affect the result.

Cross-Entropy Loss

Cross-Entropy Loss is commonly used for classification tasks, particularly in binary and multi-class classification problems. It measures the performance of a classification model whose output is a probability value between 0 and 1.

Binary Classification

For binary classification, it is represented as:

$$\text{Cross-Entropy} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

Multi-Class Classification

For multi-class classification, it is:

$$\text{Cross-Entropy} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$$

where C is the number of classes.

Advantages

- Well-suited for probabilistic classification models; penalizes confident but incorrect predictions more heavily.

Disadvantages

- Can be computationally expensive for large datasets and many classes.

1.3 Backpropagation

Backpropagation, short for "backward propagation of errors," is a fundamental algorithm used in training artificial neural networks. It is used to minimize the loss function by adjusting the weights and biases of the network through gradient descent. Here’s a detailed explanation of backpropagation, including the mathematical details and logic.

1.3.1 Steps Involved in Backpropagation

1. **Forward Pass**: Calculate the output of the network using the current weights and biases.

2. **Compute Loss**: Compute the loss using a loss function such as Mean Squared Error (MSE) or Cross-Entropy Loss.

3. **Backward Pass**: Calculate the gradients of the loss with respect to each weight using the chain rule.

4. **Update Weights**: Update the weights using gradient descent.

1.3.2 Mathematical Details

Forward Pass

For a simple neural network with one hidden layer, the forward pass can be represented as:

$$\begin{aligned}
z_1 &= W_1 x + b_1 \\
a_1 &= \sigma(z_1) \\
z_2 &= W_2 a_1 + b_2 \\
\hat{y} &= \sigma(z_2)
\end{aligned}$$

Here, W_1 and W_2 are the weight matrices, b_1 and b_2 are the bias vectors, σ is the activation function (e.g., sigmoid or ReLU), x is the input, a_1 is the activation of the hidden layer, and $\hat{y}$ is the output.

Compute Loss

For example, using Mean Squared Error (MSE) as the loss function:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Backward Pass

To perform the backward pass, we compute the gradients of the loss with respect to each weight and bias using the chain rule.
For the output layer:

$$\frac{\partial \text{Loss}}{\partial \hat{y}} = \hat{y} - y$$

For the weights and biases between the hidden layer and the output layer:

$$\begin{aligned}
\delta_2 &= (\hat{y} - y) \cdot \sigma'(z_2) \\
\frac{\partial \text{Loss}}{\partial W_2} &= \delta_2 a_1^T \\
\frac{\partial \text{Loss}}{\partial b_2} &= \delta_2
\end{aligned}$$

For the hidden layer:

$$\begin{aligned}
\delta_1 &= (W_2^T \delta_2) \cdot \sigma'(z_1) \\
\frac{\partial \text{Loss}}{\partial W_1} &= \delta_1 x^T \\
\frac{\partial \text{Loss}}{\partial b_1} &= \delta_1
\end{aligned}$$

Here, δ_2 and δ_1 are the error terms for the output and hidden layers, respectively.

Update Weights

Using gradient descent, we update the weights and biases:

$$\begin{aligned}
W_2 &\leftarrow W_2 - \eta \frac{\partial \text{Loss}}{\partial W_2} \\
b_2 &\leftarrow b_2 - \eta \frac{\partial \text{Loss}}{\partial b_2} \\
W_1 &\leftarrow W_1 - \eta \frac{\partial \text{Loss}}{\partial W_1} \\
b_1 &\leftarrow b_1 - \eta \frac{\partial \text{Loss}}{\partial b_1}
\end{aligned}$$

Here, η is the learning rate.

1.3.3 Figures

To provide a complete understanding, it's helpful to include some diagrams. Below is a simple representation of a neural network and the backpropagation process:

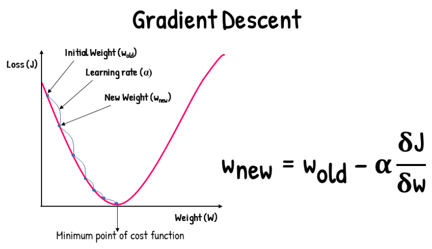


Figure 1.3.4: Gradient Descent

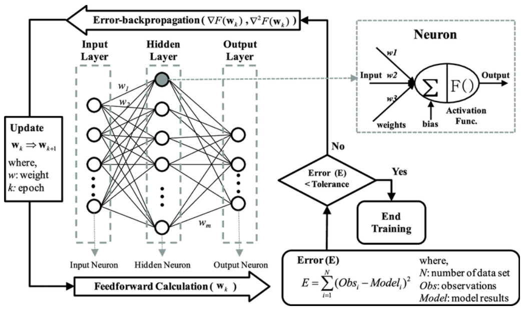


Figure 1.3.5: Backpropagation

Chapter 2

Creating an Artificial Neural Network using numpy to classify MNIST digit dataset

2.1 The MNIST dataset

The MNIST dataset is a large database of handwritten digits that is commonly used for training various image processing systems. The dataset contains a total of 70,000 images, where each image is a 28x28 pixel grayscale image of a digit from 0 to 9. These images are divided into two sets: 60,000 images for training and 10,000 images for testing.
Each image in the dataset is associated with a label, which indicates the digit represented in the image. The labels are integers ranging from 0 to 9. The MNIST dataset is widely used because it is simple to use and provides a standard benchmark for comparing the performance of different algorithms.
The dataset is used extensively in the field of machine learning for various purposes, including:

- Training and testing classification algorithms.
- Benchmarking different machine learning models.
- Research in the field of neural networks and deep learning.

Data Preparation

Before training the neural network, the MNIST data must be preprocessed. This involves normalizing the pixel values, reshaping

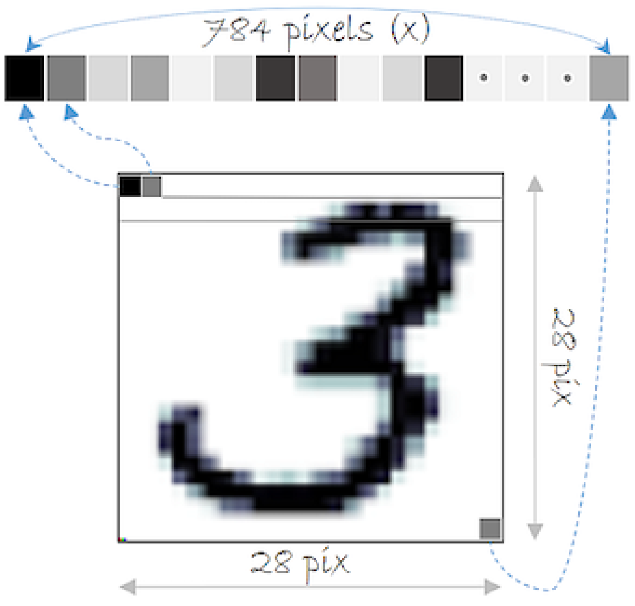


Figure 2.1.1: Sample images from the MNIST dataset

the images, and converting the labels to a one-hot encoded format. Normalizing the pixel values to the range [0, 1] helps in accelerating the training process and improving the model's performance.
The steps involved in preprocessing the MNIST data are:

1. Load the dataset.
2. Normalize the pixel values.
3. Reshape the images to 1D arrays of 784 elements (since $28 \times 28 = 784$).
4. Convert the labels to one-hot encoded vectors.

2.2 Structure of the Neural Network

The artificial neural network created to classify the MNIST digit dataset has the following structure:

- **Input Layer**: The input layer consists of 784 neurons, corresponding to the 28x28 pixel values of the MNIST images.
- **Hidden Layer**: The hidden layer has 10 neurons. The activation function used is ReLU (Rectified Linear Unit), defined as:

$$\mathrm{ReLU}(x) = \max(0, x)$$

- **Output Layer**: The output layer has 10 neurons, one for each digit class (0-9). The activation function used is Softmax, which converts the output to a probability distribution, defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

The network uses forward propagation to calculate the activations of each layer and backpropagation to update the weights based on the loss. The Mean Squared Error (MSE) is used as the loss function to measure the difference between predicted and actual labels, defined as:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

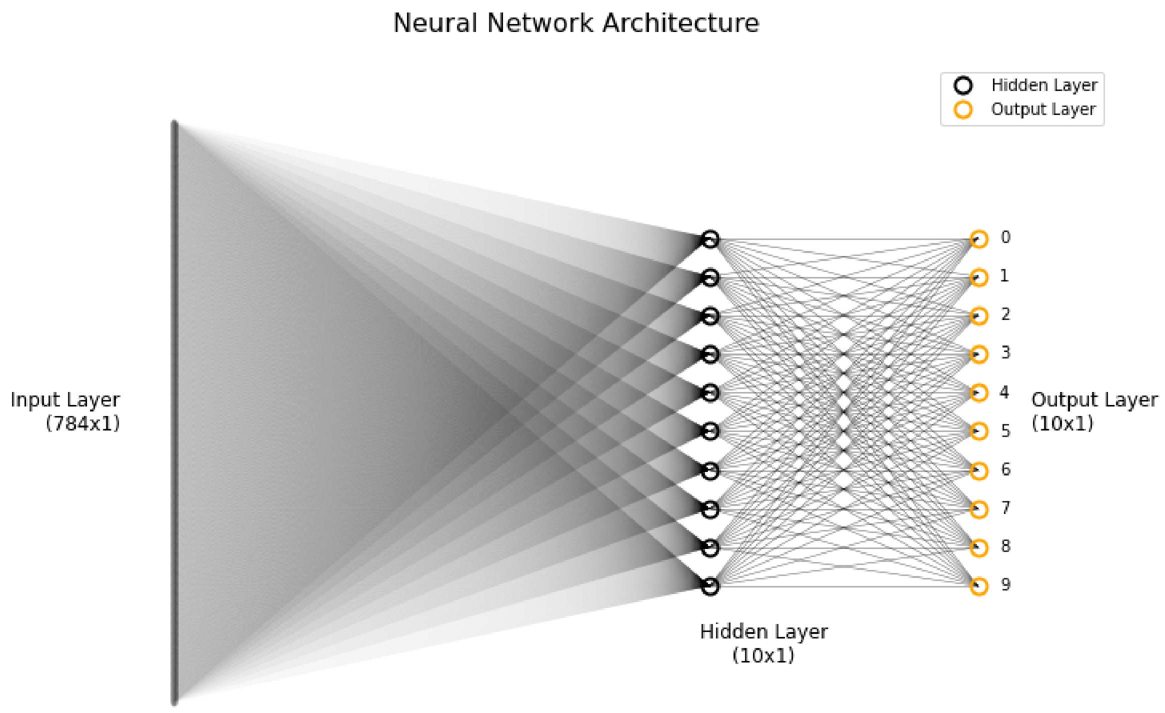


Figure 2.2.2: A schematic of Neural Network used

Forward Propagation

During forward propagation, the input data is passed through the network, applying weights and biases at each layer. The hidden layer uses the ReLU activation function, and the output layer uses the softmax activation function. The equations for forward propagation are:

$$
\begin{aligned}
h_1 &= W_1 \cdot X + b_1 \\
a_1 &= \text{ReLU}(h_1) \\
h_2 &= W_2 \cdot a_1 + b_2 \\
a_2 &= \text{Softmax}(h_2)
\end{aligned}
$$

Backpropagation

Backpropagation is used to update the weights of the network by calculating the gradient of the loss function with respect to each weight. The gradients are computed using the chain rule, and the weights are updated iteratively to minimize the loss. The equations for backpropagation are:

$$
\begin{aligned}
\delta h_2 &= a_2 - y \\
\delta W_2 &= \frac{1}{m} \delta h_2 \cdot a_1^T \\
\delta b_2 &= \frac{1}{m} \sum \delta h_2 \\
\delta h_1 &= W_2^T \cdot \delta h_2 \cdot \text{ReLU}'(h_1) \\
\delta W_1 &= \frac{1}{m} \delta h_1 \cdot X^T \\
\delta b_1 &= \frac{1}{m} \sum \delta h_1
\end{aligned}
$$

Weight Update

The weights are updated using the gradients computed during back-propagation and a learning rate α:

$$
\begin{aligned}
W_1 &= W_1 - \alpha \cdot \delta W_1 \\
b_1 &= b_1 - \alpha \cdot \delta b_1 \\
W_2 &= W_2 - \alpha \cdot \delta W_2 \\
b_2 &= b_2 - \alpha \cdot \delta b_2
\end{aligned}
$$

2.3 Learning Rate and Model Training Process

In our neural network implementation, the learning rate and model training process are crucial components. Below is an explanation of how they function based on the provided code.

Learning Rate

The learning rate, denoted by α, is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. In gradient descent algorithms, a smaller learning rate means that the steps towards the minimum of the loss function will be smaller, which can lead to more accurate convergence. However, if the learning rate is too small, the training process may become very slow. Conversely, a larger learning rate can speed up the training process, but it might overshoot the minimum, leading to instability and divergence.
In the code, the learning rate is passed as a parameter to the `train_model` function:

```
w1, b1, w2, b2 = train_model(X_train, y_train, 500, 0.001)
```

Here, `0.001` is the learning rate.

Model Training Process

The model training process involves several steps that are repeated for a number of epochs. Each epoch means one complete pass through the training dataset. The main steps in each epoch include:

1. **Forward Pass**: Compute the outputs of the neural network by passing the inputs through each layer of the network.
2. **Loss Calculation**: Measure the error by comparing the network's output with the actual labels.
3. **Backward Pass (Backpropagation)**: Compute the gradients of the loss function with respect to each weight by applying the chain rule of calculus (backpropagation).
4. **Parameter Update**: Adjust the weights using the computed gradients and the learning rate.

Forward Pass In the forward pass, the data is passed through the layers of the network to get the predicted output:

```
def forward(w1, w2, b1, b2, d):
    h1 = w1.dot(d.T) + b1
    a1 = RELU(h1)
    h2 = w2.dot(a1) + b2
    a2 = softmax(h2)
    return a1, a2, h1, h2
```

- `w1` and `w2` are the weight matrices for the input-to-hidden and hidden-to-output layers, respectively.
- `b1` and `b2` are the bias vectors for the hidden and output layers, respectively.
- `RELU` and `softmax` are activation functions used in the hidden and output layers.

Loss Calculation The loss function measures how well the network's predictions match the actual labels. In this code, Mean Squared Error (MSE) is used as the loss function:

```
def loss(out, L):
    s = np.square(out - L)
    s = np.sum(s) / len(l)
    return s
```

Backward Pass (Backpropagation) During the backward pass, the gradients of the loss with respect to the weights and biases are calculated:

```
def backprop(a1, h1, a2, h2, w2, l, d):
    m = l.size
    onehot_l = one_hot(l)
    dh2 = a2 - onehot_l
    dw2 = dh2.dot(a1.T) / m
    db2 = np.sum(dh2, axis=1, keepdims=True) / m
    dh1 = w2.T.dot(dh2) * der_RELU(h1)
    dw1 = dh1.dot(d) / m
    db1 = np.sum(dh1, axis=1, keepdims=True) / m
    return dw1, dw2, db1, db2
```

- `one_hot` converts labels to one-hot encoding.
- `der_RELU` is the derivative of the ReLU activation function.

Parameter Update The weights and biases are updated using the gradients and the learning rate:

```
def update_params(w1, b1, w2, b2, dw1, db1, db2, dw2, alpha):
    w1 = w1 - alpha * dw1
    b1 = b1 - alpha * db1
    w2 = w2 - alpha * dw2
    b2 = b2 - alpha * db2
    return w1, b1, w2, b2
```

Training Loop The training loop iterates over a specified number of epochs, performing the forward pass, backpropagation, and parameter update in each iteration:

```
def train_model(d, l, epochs, alpha):
    b1, b2, w1, w2 = init_params()

    for epoch in range(epochs):
        a1, a2, h1, h2 = forward(w1, w2, b1, b2, d)
        dw1, dw2, db1, db2 = backprop(a1, h1, a2, h2, w2, l, d)
        w1, b1, w2, b2 = update_params(w1, b1, w2, b2, dw1,
                                            db1, db2, dw2, alpha)
        if epoch % 50 == 0:
            print(’Iteration:’, epoch)
            print(’Accuracy:’, get_accuracy(get_predictions(a2), l))
            print(’Loss:’, loss(a2, one_hot(l)))
    return w1, b1, w2, b2
```

This loop runs for a specified number of epochs, updating the weights and biases to minimize the loss and improve the model’s accuracy. By following these steps, the neural network learns to classify the MNIST digit dataset more accurately over time.

2.4 Code for Neural Network

The following code implements a neural network from scratch using NumPy to classify the MNIST digit dataset.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Load data
rd = pd.read_csv('./train.csv')

l = rd['label']
d = rd.drop('label', axis=1)

print(d.shape)
print(l.shape)

d = np.array(d)
l = np.array(l)
X_train, X_test, y_train, y_test = train_test_split(d, l,
    test_size=0.3)

def RELU(x):
    return np.maximum(0, x)

def softmax(x):
    x = x - np.max(x, axis=0, keepdims=True)
    exp_x = np.exp(x)
    return exp_x / np.sum(exp_x, axis=0, keepdims=True)

def sigmoid_derivative(x):
    return x * (1 - x)

def der_RELU(x):
    return x > 0

def one_hot(l):
    onehot = np.zeros((l.size, l.max() + 1))
    onehot[np.arange(l.size), l] = 1
    onehot = onehot.T
    return onehot

def init_params():
    b1 = np.random.randn(10, 1)
    b2 = np.random.randn(10, 1)
    w1 = np.random.randn(10, 784) * 0.01
    w2 = np.random.randn(10, 10) * 0.01
    return b1, b2, w1, w2

def loss(out, L):
    s = np.square(out - L)
    s = np.sum(s) / len(l)
    return s
```

```python

def forward(w1, w2, b1, b2, d):
    h1 = w1.dot(d.T) + b1
    a1 = RELU(h1)
    h2 = w2.dot(a1) + b2
    a2 = softmax(h2)
    return a1, a2, h1, h2

def backprop(a1, h1, a2, h2, w2, l, d):
    m = l.size
    onehot_l = one_hot(l)
    dh2 = a2 - onehot_l
    dw2 = dh2.dot(a1.T) / m
    db2 = np.sum(dh2, axis=1, keepdims=True) / m
    dh1 = w2.T.dot(dh2) * der_RELU(h1)
    dw1 = dh1.dot(d) / m
    db1 = np.sum(dh1, axis=1, keepdims=True) / m
    return dw1, dw2, db1, db2

def update_params(w1, b1, w2, b2, dw1, db1, db2, dw2, alpha):
    w1 = w1 - alpha * dw1
    b1 = b1 - alpha * db1
    w2 = w2 - alpha * dw2
    b2 = b2 - alpha * db2
    return w1, b1, w2, b2

def get_predictions(a2):
    return np.argmax(a2, axis=0)

def get_accuracy(predictions, l):
    return np.sum(predictions == l) / l.size

def train_model(d, l, epochs, alpha):
    b1, b2, w1, w2 = init_params()

    for epoch in range(epochs):
        a1, a2, h1, h2 = forward(w1, w2, b1, b2, d)
        dw1, dw2, db1, db2 = backprop(a1, h1, a2, h2, w2, l, d)
        w1, b1, w2, b2 = update_params(w1, b1, w2, b2, dw1, db1,
    db2, dw2, alpha)
        if epoch % 50 == 0:
            print('Iteration:', epoch)
            print('Accuracy:', get_accuracy(get_predictions(a2),
    l))
            print('Loss:', loss(a2, one_hot(l)))
    return w1, b1, w2, b2

w1, b1, w2, b2 = train_model(X_train, y_train, 500, 0.001)

data_test = np.array(X_test)
y_test = np.array(y_test)

a1_test, a2_test, h1_test, h2_test = forward(w1, w2, b1, b2,
    data_test)

# Get predictions
```

```
predictions_test = get_predictions(a2_test)

# Calculate accuracy
accuracy_test = get_accuracy(predictions_test, y_test)
print(f"Accuracy on test set: {accuracy_test}")
```
Listing 2.1: Neural Network Code

2.5 Neural Network Training Results

2.5.1 Training Progress

During the training of the neural network, the following accuracy and loss values were observed:

Epoch	Accuracy	Loss
0	0.10850340136054422	0.7319445318205708
50	0.7456462585034014	0.260941294568 04394
100	0.8249319727891157	0.18000605716808368
150	0.8646938775510205	0.14212657567304193
200	0.8806462585034014	0.1261165682583584
250	0.8895578231292517	0.11602201499300531
300	0.8957823129251701	0.10891971590485006
350	0.9015986394557823	0.10354604918527015
400	0.9064285714285715	0.09930243064508765
450	0.9097278911564626	0.09583088306880451
500	0.9102721088435374	0.09553736761525883

Table 2.5.1: Training Accuracy and Loss over Epochs

2.5.2 Test Accuracy

After training the neural network, the model's accuracy on the test set was:

Accuracy on Test Set: *0.9057936507936508*

2.6 Results for testing with random figures

2.6.1 Custom Test Results

In addition to the standard evaluation, a custom test was performed where the model was given 20 pictures. These pictures were divided into two groups:

- **First 10 pictures:** Digits from 0 to 9.
- **Next 10 pictures:** Random figures not resembling any digits.

The results of this custom test are summarized below:

- **Digits (0 to 9):** The model correctly predicted the digits for all 10 images.
- **Random Figures:** Initially, the model predicted varying digits for the random figures. However, after some iterations, the predictions for these random figures converged to a specific digit. This behavior indicates that the model, while effective at recognizing digits, may not handle non-digit images as expected, converging to incorrect predictions for these images.

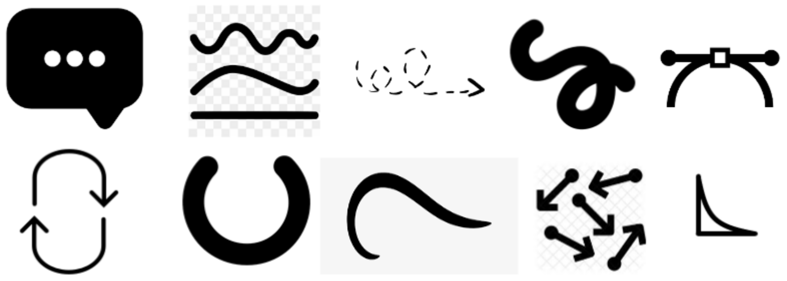


Figure 2.6.3: Random images used for testing

Epoch	Predicted Output for Digits	Predicted Output for Random Data
50	[0 1 8 3 6 5 6 7 8 8]	[4 0 3 0 2 6 2 0 6 2]
100	[0 1 8 3 6 5 6 7 8 9]	[4 0 3 7 2 6 2 0 6 2]
150	[0 1 8 3 6 5 6 7 8 9]	[4 0 0 9 2 6 2 0 6 2]
200	[0 1 8 3 4 5 6 7 8 9]	[4 0 0 9 2 6 2 0 6 2]
250	[0 1 2 3 4 5 6 7 8 9]	[4 0 0 9 2 6 2 0 6 2]
300	[0 1 2 3 4 5 6 7 8 9]	[4 0 0 9 2 6 2 0 6 2]
350	[0 1 2 3 4 5 6 7 8 9]	[4 0 0 9 2 6 2 0 6 2]
400	[0 1 2 3 4 5 6 7 8 9]	[4 0 0 9 2 6 2 0 8 2]
450	[0 1 2 3 4 5 6 7 8 9]	[4 0 0 9 2 6 2 0 8 2]
500	[0 1 2 3 4 5 6 7 8 9]	[4 0 0 9 2 6 3 0 8 2]

Table 2.6.2: Model Predictions at Different Epochs

2.7 Discussion

2.7.1 Discussion for model training and testing on MNIST data

The results show that the neural network improves in accuracy and reduces the loss over the course of training. The final accuracy on the test set is approximately 90%, indicating the model's ability to generalize well to unseen data.

2.7.2 Discussion for model training and testing on random data

The results suggest that while the neural network model successfully classifies digits from 0 to 9, it exhibits limitations when encountering images that do not resemble digits. Initially, the model attempted to classify the random figures into different digits, but eventually, it converged to specific outputs. This behavior highlights a key difference between neural networks and human cognition, where the latter can more flexibly handle and recognize non-standard inputs.

Chapter 3

Creating an Artificial Neural Network in pytorch to classify MNIST fashion dataset

3.1 Introduction to MNIST Fashion Dataset

The MNIST Fashion dataset is a collection of grayscale images of fashion items, intended to serve as a more challenging drop-in replacement for the original MNIST dataset of handwritten digits. It consists of 70,000 images divided into 60,000 training images and 10,000 test images. Each image is 28x28 pixels in size, and there are 10 different classes representing different types of fashion items.

3.1.1 Dataset Classes

The dataset includes the following classes:

- 0: T-shirt/top
- 1: Trouser
- 2: Pullover
- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker

- 8: Bag
- 9: Ankle boot

3.1.2 Dataset Format

Each image is associated with a label from 0 to 9, corresponding to the class of the fashion item depicted. The images are stored as 28x28 pixel grayscale values, ranging from 0 (black) to 255 (white).

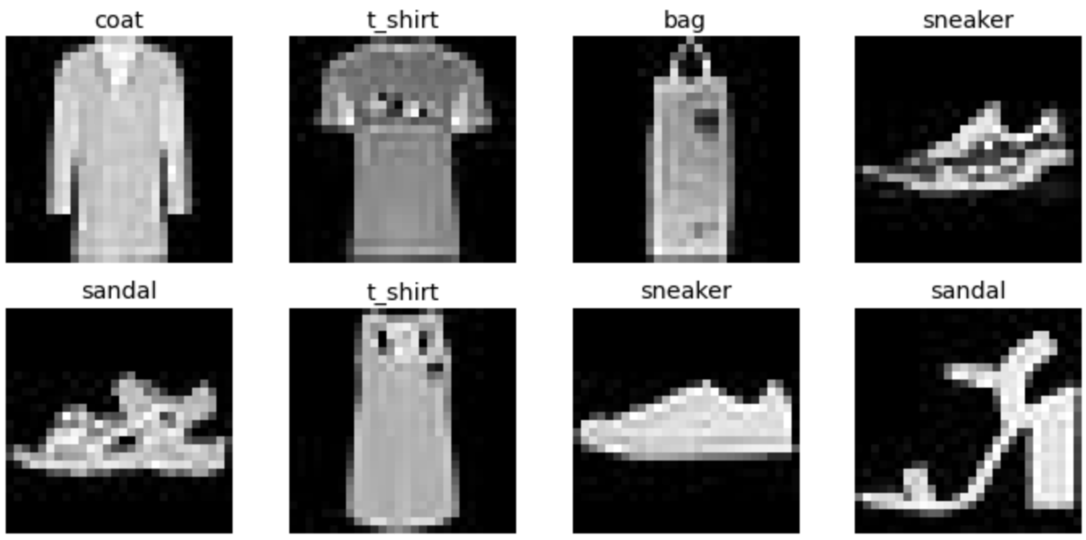


Figure 3.1.1: Sample images from the MNIST Fashion dataset

3.1.3 Use Case

The MNIST Fashion dataset is commonly used for benchmarking machine learning algorithms, particularly for image classification tasks. Due to its complexity compared to the original MNIST dataset, it provides a better challenge for algorithms, helping to assess their robustness and performance.

3.2 Structure of the Neural Network

The neural network used for classifying the MNIST Fashion dataset is implemented using PyTorch. The architecture consists of multiple fully connected (dense) layers with batch normalization and ReLU activation functions. Below is a detailed explanation of the network structure.

3.2.1 Network Architecture

- **Input Layer:** The input to the network is a flattened 28x28 pixel image, resulting in a vector of size 784.
- **Fully Connected Layer 1:**
  - **Layer:** A fully connected layer with 512 neurons.
  - **Activation Function:** ReLU (Rectified Linear Unit).
- **Fully Connected Layer 2:**
  - **Layer:** A fully connected layer with 512 neurons.
  - **Activation Function:** ReLU.
- **Fully Connected Layer 3:**
  - **Layer:** A fully connected layer with 128 neurons.
  - **Activation Function:** ReLU.
- **Fully Connected Layer 4:**
  - **Layer:** A fully connected layer with 64 neurons.
  - **Activation Function:** ReLU.
- **Fully Connected Layer 5:**
  - **Layer:** A fully connected layer with 32 neurons..
  - **Activation Function:** ReLU.
- **Output Layer:**
  - **Layer:** A fully connected layer with 10 neurons.

The following code illustrates the implementation of this neural network in PyTorch:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, 32)
        self.fc6 = nn.Linear(32, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = F.relu(self.fc5(x))
        x = self.fc6(x)
        return x
```

3.2.2 Explanation of Layers

- **Input Layer:** Accepts the 28x28 pixel image and flattens it to a vector of size 784.
- **Hidden Layers:** Consists of five fully connected layers with batch normalization and ReLU activation functions to introduce non-linearity and stabilize training.
- **Output Layer:** Outputs a vector of size 10, corresponding to the 10 classes in the MNIST Fashion dataset.

3.3 Classification of Fashion-MNIST Dataset using PyTorch

3.3.1 Introduction

In this section, we classify the Fashion-MNIST dataset using a neural network implemented in PyTorch. The Fashion-MNIST dataset contains 70,000 grayscale images of 10 fashion categories, with 7,000 images per category. Each image is 28x28 pixels.

3.3.2 Data Preparation

First, we load and preprocess the data:

```
from openml.datasets import get_dataset
fashion_mnist = get_dataset('Fashion-MNIST')
X, y, _, _ = fashion_mnist.get_data(dataset_format="dataframe")
y = X['class']
X = X.drop('class', axis=1)

import numpy as np
y = np.array(y)
X = np.array(X)
for i in range(len(y)):
    y[i] = int(y[i])

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X/255, y, test_s

import torch
import torch.nn as nn
import torch.nn.functional as F

X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.LongTensor(y_train.astype(int))
y_test = torch.LongTensor(y_test.astype(int))
```

The data is loaded from the OpenML dataset repository, and the features (X) and labels (y) are separated. The dataset is then split into training and test sets, and normalized by dividing by 255. Finally, the data is converted to PyTorch tensors.

3.3.3 Network Architecture

The neural network consists of six fully connected layers with batch normalization and ReLU activation functions.

```
class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, 32)
        self.fc6 = nn.Linear(32, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = F.relu(self.fc5(x))
        x = self.fc6(x)
        return x
```

3.3.4 Model Training

The model is trained using the Adam optimizer and CrossEntropyLoss. The training process involves forward propagation, loss calculation, backpropagation, and parameter updates.

```
model = Network()
lossf = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

epochs = 300
losses = []
accuracy_list = []

for epoch in range(epochs):
    model.train()
    y_pred = model(X_train)
    loss = lossf(y_pred, y_train)
    losses.append(loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 20 == 0:
        _, predicted = torch.max(y_pred, 1)
        correct = (predicted == y_train).sum().item()
        accuracy = correct / y_train.size(0)
        accuracy_list.append(accuracy)
        print(f'Epoch: {epoch}, Loss: {loss.item()}, Accuracy: {accu
```

In each epoch, the model performs forward propagation to predict the outputs for the training data. The loss is computed using CrossEntropyLoss, and backpropagation is performed to calculate the gradients. The optimizer updates the model parameters based on these gradients.

3.3.5 Model Evaluation

After training, the model is evaluated on the test set.

```
model.eval()
with torch.no_grad():
    test_res = model(X_test)
    test_loss = lossf(test_res, y_test)
    _, predicted = torch.max(test_res, 1)
    correct = (predicted == y_test).sum().item()
    test_accuracy = correct / y_test.size(0)
    print(f'Test Loss: {test_loss.item()}, Test Accuracy: {test_accu
```

The model's performance is measured in terms of test loss and test accuracy.

3.3.6 Mathematical Explanation

The training process can be explained mathematically as follows:

- **Forward Propagation:** The input data X is passed through the network layers to obtain the predictions $\hat{y}$.

$$\begin{aligned}
\mathbf{z}_1 &= \mathbf{W}_1\mathbf{X} + \mathbf{b}_1 \\
\mathbf{a}_1 &= \mathrm{ReLU}(\mathbf{z}_1) \\
\mathbf{z}_2 &= \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2 \\
\mathbf{a}_2 &= \mathrm{ReLU}(\mathbf{z}_2) \\
\mathbf{z}_3 &= \mathbf{W}_3\mathbf{a}_2 + \mathbf{b}_3 \\
\mathbf{a}_3 &= \mathrm{ReLU}(\mathbf{z}_3) \\
\mathbf{z}_4 &= \mathbf{W}_4\mathbf{a}_3 + \mathbf{b}_4 \\
\mathbf{a}_4 &= \mathrm{ReLU}(\mathbf{z}_4) \\
\mathbf{z}_5 &= \mathbf{W}_5\mathbf{a}_4 + \mathbf{b}_5 \\
\mathbf{a}_5 &= \mathrm{ReLU}(\mathbf{z}_5) \\
\mathbf{z}_6 &= \mathbf{W}_6\mathbf{a}_5 + \mathbf{b}_6 \\
\hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{z}_6)
\end{aligned}$$

- **Loss Calculation:** The CrossEntropyLoss is calculated between the predicted outputs $\hat{y}$ and the true labels y.

$$\mathrm{Loss} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

- **Backpropagation:** The gradients of the loss with respect to the model parameters are computed.

$$\frac{\partial \mathrm{Loss}}{\partial W_i} = \frac{\partial \mathrm{Loss}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial W_i}$$

- **Parameter Update:** The optimizer updates the model parameters using the computed gradients.

$$W_i := W_i - \eta \frac{\partial \mathrm{Loss}}{\partial W_i}$$

3.4 Code for Fashion-MNIST Classification

The following code implements a neural network using PyTorch to classify the Fashion-MNIST dataset.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from openml.datasets import get_dataset
from sklearn.model_selection import train_test_split

# Load and preprocess data
fashion_mnist = get_dataset('Fashion-MNIST')
X, y, _, _ = fashion_mnist.get_data(dataset_format="dataframe")
y = X['class']
X = X.drop('class', axis=1)
y = np.array(y)
X = np.array(X)

for i in range(len(y)):
    y[i] = int(y[i])

X_train, X_test, y_train, y_test = train_test_split(X/255, y,
    test_size=0.2)

# Convert to PyTorch tensors
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.LongTensor(y_train.astype(int))
y_test = torch.LongTensor(y_test.astype(int))

# Define the neural network model
class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, 32)
        self.fc6 = nn.Linear(32, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = F.relu(self.fc5(x))
        x = self.fc6(x)
        return x

# Initialize the model, loss function, and optimizer
model = Network()
lossf = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Training the model
epochs = 300
losses = []
accuracy_list = []
```

```

for epoch in range(epochs):
    model.train()
    y_pred = model(X_train)
    loss = lossf(y_pred, y_train)
    losses.append(loss.item())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 20 == 0:
        _, predicted = torch.max(y_pred, 1)
        correct = (predicted == y_train).sum().item()
        accuracy = correct / y_train.size(0)
        accuracy_list.append(accuracy)
        print(f'Epoch: {epoch}, Loss: {loss.item()}, Accuracy: {
    accuracy}')

# Evaluate the model
model.eval()
with torch.no_grad():
    test_res = model(X_test)
    test_loss = lossf(test_res, y_test)
    _, predicted = torch.max(test_res, 1)
    correct = (predicted == y_test).sum().item()
    test_accuracy = correct / y_test.size(0)
    print(f'Test Loss: {test_loss.item()}, Test Accuracy: {
    test_accuracy}')
```

Listing 3.1: PyTorch Code for Fashion-MNIST Classification

3.5 Results

3.5.1 Training Progress

During the training of the neural network on the Fashion-MNIST dataset, the following accuracy and loss values were observed:

Epoch	Training Accuracy	Training Loss
0	0.099625	2.305508613586426
50	0.7605	0.6287441253662109
100	0.8295892857142857	0.46974053978919983
150	0.8558035714285714	0.40529265999794006
200	0.8739464285714286	0.35431569814682007
250	0.8853214285714286	0.3200078010559082
300	0.8934642857142857	0.29551687836647034
350	0.9008571428571429	0.2724109888076782
400	0.8997678571428571	0.27761930227279663
450	0.9115535714285714	0.24491587281227112
500	0.9120892857142857	0.23957380652427673

Table 3.5.1: Training Accuracy and Loss over Epochs

3.5.2 Test Performance

After training the neural network, the model's performance on the test set was as follows:

Metric	Value
Test Accuracy	0.8825714285714286
Test Loss	0.3447877764701843

Table 3.5.2: Test Performance Metrics

3.6 Discussion

The results demonstrate that the neural network model shows steady improvement in accuracy and reduction in loss throughout the training process. The final test accuracy of approximately 88% indicates that the model performs well on unseen Fashion-MNIST data, effectively distinguishing between different fashion items.

Chapter 4

Comparative Analysis of Optimization Techniques

4.1 Introduction

Optimization is a fundamental aspect of machine learning and neural network training, aiming to find the best parameters that minimize the loss function and improve model performance. Various optimization algorithms have been developed to efficiently traverse the loss landscape and converge to an optimal solution.
In this section, we will compare three widely used optimization techniques: Gradient Descent (GD), Stochastic Gradient Descent (SGD), and Gradient Descent with Momentum (MGD).

4.2 Theory of Optimization Techniques

4.2.1 Gradient Descent (GD)

Gradient Descent is an iterative optimization algorithm used to minimize a cost function $J(\theta)$. In each iteration, the algorithm updates the parameters θ in the direction of the negative gradient of the cost function.
The update rule is given by:

$$\theta := \theta - \alpha \nabla J(\theta)$$

where:

- θ are the parameters of the model,
- α is the learning rate,

- $\nabla J(\theta)$ is the gradient of the cost function with respect to θ.

4.2.2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent is a variant of Gradient Descent where the model parameters are updated based on a randomly selected single training example rather than the full training dataset. This often leads to faster convergence in practice.
The update rule is given by:

$$\theta := \theta - \alpha \nabla J(\theta; x^{(i)}, y^{(i)})$$

where:

- θ are the parameters of the model,
- α is the learning rate,
- $\nabla J(\theta; x^{(i)}, y^{(i)})$ is the gradient of the cost function with respect to θ, evaluated using a single training example $(x^{(i)}, y^{(i)})$.

SGD is an unbiased estimator of the gradient used in GD. That is, the expected value of the gradient computed using a single example (or a small batch) is the same as the gradient computed using the entire dataset.

$$\mathbb{E}\left[\nabla J(\theta; x^{(i)}, y^{(i)})\right] = \nabla J(\theta)$$

where:

- $\mathbb{E}[\cdot]$ denotes the expectation with respect to the training examples,
- $\nabla J(\theta; x^{(i)}, y^{(i)})$ is the gradient of the cost function with respect to θ, evaluated using a single training example $(x^{(i)}, y^{(i)})$,
- $\nabla J(\theta)$ is the gradient of the cost function with respect to θ computed using the full dataset.

4.2.3 Gradient Descent with Momentum (MGD)

Gradient Descent with Momentum incorporates a velocity term to accelerate the gradient descent algorithm and help it navigate through regions with shallow gradients more effectively.

The update rules are given by:

$$v := \beta v + (1 - \beta)\nabla J(\theta)$$

$$\theta := \theta - \alpha v$$

where:

- v is the velocity term,
- β is the momentum factor (usually close to 1, e.g., 0.9),
- α is the learning rate,
- $\nabla J(\theta)$ is the gradient of the cost function with respect to θ.

4.3 Comparison of Optimization Techniques

In this section, we compare three optimization techniques: Gradient Descent (GD), Gradient Descent with Momentum (MGD), and Stochastic Gradient Descent (SGD). The goal is to visualize and analyze the performance of each technique in a simulated environment.

4.3.1 Process and Code Explanation

The code provided sets up a simulation to compare the three optimization methods. Here's a detailed explanation of the process:

- **Initialization and Setup:** The lengths of the links of the robotic arm and the initial angles (`theta`, `theta1`, `theta2`) are initialized for each optimization technique. Three robotic arms are visualized using the `vpython` library, each controlled by a different optimization method: GD, MGD, or SGD.
- **Target Points and Robotic Arms:** Functions generate random target points in the 2D plane. Each optimization technique has a specific target location. Robotic arms are created and positioned in the visual environment, with their positions updated according to the chosen optimization technique.
- **Optimization Functions:**

  - `forwardpass`, `forwardpass_momentum`, and `forwardpass_stochastic` update the robotic arm's position based on its configuration and the current angles.
  - Loss functions (`loss`, `lossv`, `losss`) compute the distance between the arm's end position and the target point.

- **Gradient Calculation:** Gradients are computed using finite differences to determine parameter updates:
  - `gradients` for GD.
  - `gradientsv` for MGD.
  - `gradientss` for SGD with stochastic updates.

- **Optimization Loop:** The main loop updates the parameters using the respective optimization technique:
  - GD updates with a constant learning rate.
  - MGD updates with momentum.
  - SGD updates with a stochastic gradient and different learning rates.

  The loop continues until each method successfully reduces its loss below a threshold.

- **Visualization and Results:** The positions of the end effectors are updated in real-time to show how each technique performs. When a robotic arm reaches the target, indicated by the ring's color changing to green, the optimization is considered successful.

The simulation provides a visual comparison of how each optimization technique converges to the target points, highlighting their effectiveness and differences in performance.

4.4 Code for Optimization Techniques Comparison

Below is the Python code used to compare the three optimization techniques: Gradient Descent (GD), Gradient Descent with Momentum (MGD), and Stochastic Gradient Descent (SGD). The code utilizes the `vpython` library to visualize the optimization process in real-time.

4.5 Code for Comparing Gradient Descent Algorithms

The following code compares the behavior of different gradient descent algorithms: Gradient Descent (GD), Stochastic Gradient Descent (SGD), and Gradient Descent with Momentum (MGD) using a simple simulation.

```python
import time
from vpython import *
import numpy as np

length1 = [2, 2, 2]
length = [2, 2, 2]
length2 = [2,2,2]
theta = np.array([np.pi/3, np.pi/4, np.pi/5])
theta1 = np.array([np.pi/4, np.pi/6, np.pi/8])
theta2 = np.array([np.pi/4, np.pi/6, np.pi/8])
n_bobs = 3
alpha = 0.1
alpham = 0.2
alphas = 0.6
momentum = 0.7
R1 = 3.5
R2 = 5.5

def samplepoints():
    theta = np.random.uniform(np.pi/6, np.pi/4)
    u = np.random.uniform(0, 1)
    r = R1 + u * (R2 - R1)
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return vector(x, y, 0)

def samplepointsm():
    theta = np.random.uniform(np.pi/6, np.pi/4)
    u = np.random.uniform(0, 1)
    r = R1 + u * (R2 - R1)
    x = r * np.cos(theta) - 10
    y = r * np.sin(theta)
    return vector(x, y, 0)

def samplepointss():
    theta = np.random.uniform(np.pi/6, np.pi/4)
    u = np.random.uniform(0, 1)
    r = R1 + u * (R2 - R1)
    x = r * np.cos(theta) + 10
    y = r * np.sin(theta)
    return vector(x, y, 0)

```

```
scene = canvas(background=color.white, pos = vector(-100,0,0) ,
    height=1000, width=1000)
target_point = samplepoints()
target_point_s = sphere(pos=target_point, radius=0.15, color=
    color.red, make_trail=False)
box1 = box(length=0.5, height=0.5, width=0.5, color=color.black)
label1 = label(pos=box1.pos + vector(0, -1, 0), text="GD", height
    =20, box=False)
bobs1 = [sphere(pos=vector(0, 0, 0), radius=0.1, color=color.
    black) for _ in range(n_bobs)]
bobs1[0].pos = vector(length[0] * np.sin(theta[0]), -length[0] *
    np.cos(theta[0]), 0)
for i in range(1, n_bobs):
     bobs1[i].pos = bobs1[i-1].pos + vector(length[i] * np.sin(
    theta[i]), -length[i] * np.cos(theta[i]), 0)

ringend1 = ring(pos=bobs1[-1].pos, color=color.black, axis=vector
    (0, 0, 1), radius=0.3, thickness=0.05)
lines1 = [cylinder(radius=0.02, pos=vector(0, 0, 0), axis=bobs1
    [0].pos - vector(0, 0, 0), color=color.black) for _ in range(
    n_bobs)]
lines1[0].axis = bobs1[0].pos - vector(0, 0, 0)

offset_x = -10
box2 = box(length=0.5, pos=vector(offset_x, 0, 0), height=0.5,
    width=0.5, color=color.black)
label2 = label(pos=box2.pos + vector(0, -1, 0), text="MGD",
    height=20, box=False)
target_pointm = samplepointsm()
target_point_sm = sphere(pos=target_pointm, radius=0.15, color=
    color.red, make_trail=False)
bobs2 = [sphere(pos=vector(0, 0, 0), radius=0.1, color=color.
    black) for _ in range(n_bobs)]
bobs2[0].pos = vector(length1[0] * np.sin(theta1[0]) + offset_x,
    -length1[0] * np.cos(theta1[0]), 0)
for i in range(1, n_bobs):
     bobs2[i].pos = bobs2[i-1].pos + vector(length1[i] * np.sin(
    theta1[i]), -length1[i] * np.cos(theta1[i]), 0)

ringend2 = ring(pos=bobs2[-1].pos, color=color.black, axis=vector
    (0, 0, 1), radius=0.3, thickness=0.05)
lines2 = [cylinder(radius=0.02, pos=vector(offset_x, 0, 0), axis=
    bobs2[0].pos - vector(offset_x, 0, 0), color=color.black) for
    _ in range(n_bobs)]
lines2[0].axis = bobs2[0].pos - vector(offset_x, 0, 0)

box3 = box(length=0.5, pos=vector(10, 0, 0), height=0.5, width
    =0.5, color=color.black)
label3 = label(pos=box3.pos + vector(0, -1, 0), text="SGD",
    height=20, box=False)
offset_x1 = 10
target_points = samplepointss()
target_point_ss = sphere(pos=target_points, radius=0.15, color=
    color.red, make_trail=False)
bobs3 = [sphere(pos=vector(0, 0, 0), radius=0.1, color=color.
    black) for _ in range(n_bobs)]
```

```
bobs3[0].pos = vector(length2[0] * np.sin(theta2[0]) + offset_x1, -length2[0] * np.cos(theta2[0]), 0)
for i in range(1, n_bobs):
    bobs3[i].pos = bobs3[i-1].pos + vector(length2[i] * np.sin(theta2[i]), -length2[i] * np.cos(theta2[i]), 0)

ringend3 = ring(pos=bobs3[-1].pos, color=color.black, axis=vector(0, 0, 1), radius=0.3, thickness=0.05)
lines3 = [cylinder(radius=0.02, pos=vector(offset_x1, 0, 0), axis=bobs3[0].pos - vector(offset_x1, 0, 0), color=color.black) for _ in range(n_bobs)]
lines3[0].axis = bobs3[0].pos - vector(offset_x1, 0, 0)

def forwardpass(bobs, lines):
    bobs[0].pos = vector(length[0] * np.sin(theta[0]), -length[0] * np.cos(theta[0]), 0)
    for i in range(1, n_bobs):
        bobs[i].pos = bobs[i-1].pos + vector(length[i] * np.sin(theta[i]), -length[i] * np.cos(theta[i]), 0)
    lines[0].axis = bobs[0].pos - vector(0, 0, 0)
    for i in range(1, n_bobs):
        lines[i].pos = bobs[i-1].pos
        lines[i].axis = bobs[i].pos - bobs[i-1].pos
    ringend1.pos = bobs[-1].pos

def forwardpass_momentum(bobs, lines, ringend, offset_x):
    bobs[0].pos = vector(length1[0] * np.sin(theta1[0]) + offset_x, -length1[0] * np.cos(theta1[0]), 0)
    for i in range(1, n_bobs):
        bobs[i].pos = bobs[i-1].pos + vector(length1[i] * np.sin(theta1[i]), -length1[i] * np.cos(theta1[i]), 0)
    lines[0].axis = bobs[0].pos - vector(offset_x, 0, 0)
    for i in range(1, n_bobs):
        lines[i].pos = bobs[i-1].pos
        lines[i].axis = bobs[i].pos - bobs[i-1].pos
    ringend.pos = bobs[-1].pos

def forwardpass_stochastic(bobs, lines, ringend, offset_x):
    bobs[0].pos = vector(length2[0] * np.sin(theta2[0]) + offset_x, -length2[0] * np.cos(theta2[0]), 0)
    for i in range(1, n_bobs):
        bobs[i].pos = bobs[i-1].pos + vector(length2[i] * np.sin(theta2[i]), -length2[i] * np.cos(theta2[i]), 0)
    lines[0].axis = bobs[0].pos - vector(offset_x, 0, 0)
    for i in range(1, n_bobs):
        lines[i].pos = bobs[i-1].pos
        lines[i].axis = bobs[i].pos - bobs[i-1].pos
    ringend.pos = bobs[-1].pos


def endpointv():
    return bobs1[-1].pos

def lossv():
    return np.linalg.norm(np.array([endpointv().x, endpointv().y, endpointv().z]) - np.array([target_point.x, target_point.y,
```

```python
target_point.z]))

def endpoint():
    return bobs2[-1].pos

def loss():
    return np.linalg.norm(np.array([endpoint().x, endpoint().y, endpoint().z]) - np.array([target_pointm.x, target_pointm.y, target_pointm.z]))

def losss():
    return np.linalg.norm(np.array([endpoints().x, endpoints().y, endpoints().z]) - np.array([target_points.x, target_points.y, target_points.z]))

def endpoints():
    return bobs3[-1].pos

def metricfunc(t1, t2, t3):
    pos1 = vector(length1[0] * np.sin(t1) + offset_x, -length1[0] * np.cos(t1), 0)
    pos2 = pos1 + vector(length1[1] * np.sin(t2), -length1[1] * np.cos(t2), 0)
    pos3 = pos2 + vector(length1[2] * np.sin(t3), -length1[2] * np.cos(t3), 0)
    return np.linalg.norm(np.array([pos3.x, pos3.y, pos3.z]) - np.array([target_pointm.x, target_pointm.y, target_pointm.z]))

def metricfuncv(t1, t2, t3):
    pos1 = vector(length[0] * np.sin(t1), -length[0] * np.cos(t1), 0)
    pos2 = pos1 + vector(length[1] * np.sin(t2), -length[1] * np.cos(t2), 0)
    pos3 = pos2 + vector(length[2] * np.sin(t3), -length[2] * np.cos(t3), 0)
    return np.linalg.norm(np.array([pos3.x, pos3.y, pos3.z]) - np.array([target_point.x, target_point.y, target_point.z]))

def metricfuncs(t1, t2, t3):
    pos1 = vector(length2[0] * np.sin(t1) + offset_x1, -length2[0] * np.cos(t1), 0)
    pos2 = pos1 + vector(length2[1] * np.sin(t2), -length2[1] * np.cos(t2), 0)
    pos3 = pos2 + vector(length2[2] * np.sin(t3), -length2[2] * np.cos(t3), 0)
    return np.linalg.norm(np.array([pos3.x, pos3.y, pos3.z]) - np.array([target_points.x, target_points.y, target_points.z]))

def gradients():
    delta = 1
    dtheta1 = (metricfunc(theta1[0] + delta, theta1[1], theta1[2]) - metricfunc(theta1[0] - delta, theta1[1], theta1[2])) / (2 * delta)
    dtheta2 = (metricfunc(theta1[0], theta1[1] + delta, theta1[2]) - metricfunc(theta1[0], theta1[1] - delta, theta1[2])) / (2 * delta)
    dtheta3 = (metricfunc(theta1[0], theta1[1], theta1[2] + delta) - metricfunc(theta1[0], theta1[1], theta1[2] - delta)) / (2 * delta)
    dtheta = np.array([dtheta1, dtheta2, dtheta3])
    return dtheta

def gradientsv():
    delta = 1
    dtheta1 = (metricfuncv(theta[0] + delta, theta[1], theta[2]) - metricfuncv(theta[0] - delta, theta[1], theta[2])) / (2 * delta)
    dtheta2 = (metricfuncv(theta[0], theta[1] + delta, theta[2]) - metricfuncv(theta[0], theta[1] - delta, theta[2])) / (2 * delta)
    dtheta3 = (metricfuncv(theta[0], theta[1], theta[2] + delta) - metricfuncv(theta[0], theta[1], theta[2] - delta)) / (2 * delta)
    grad = np.array([dtheta1, dtheta2, dtheta3])
    return grad

def gradientss():
    delta = 1
    i = np.random.choice([0, 1, 2])
    if i == 0:
        grad_theta1 = (metricfuncs(theta2[0] + delta, theta2[1], theta2[2]) - metricfuncs(theta2[0] - delta, theta2[1], theta2[2])) / (2 * delta)
        grad_theta2 = 0.0
        grad_theta3 = 0.0
    elif i == 1:
        grad_theta1 = 0.0
        grad_theta2 = (metricfuncs(theta2[0], theta2[1] + delta, theta2[2]) - metricfuncs(theta2[0], theta2[1] - delta, theta2[2])) / (2 * delta)
        grad_theta3 = 0.0
    else:
        grad_theta1 = 0.0
        grad_theta2 = 0.0
        grad_theta3 = (metricfuncs(theta2[0], theta2[1], theta2[2] + delta) - metricfuncs(theta2[0], theta2[1], theta2[2] - delta)) / (2 * delta)

    grads = np.array([grad_theta1, grad_theta2, grad_theta3])
    return grads

change = np.array([0, 0, 0])
t0 = time.time()

while True:
    target_pointm = samplepoints() + vector(-10,0,0)
    target_point_sm.pos = target_pointm
    target_point = samplepoints()
    target_point_s.pos = target_point
    target_points = samplepoints() + vector(10,0,0)
    target_point_ss.pos = target_points
    ringend1.color = color.black
    ringend2.color = color.black
    ringend3.color = color.black

    while True:
        rate(20)

        dtheta = gradients()
        grad = gradientsv()
        grads = gradientss()

        prevmov = alpham * dtheta + momentum * change

        theta1 -= prevmov
        theta -= alpha * grad
        theta2 -= alphas * grads
        change = prevmov

        theta1 = np.minimum(np.pi, np.maximum(theta1, 0))
        theta = np.minimum(np.pi, np.maximum(theta, 0))
        theta2 = np.minimum(np.pi, np.maximum(theta2, 0))
        forwardpass(bobs1, lines1)
        forwardpass_momentum(bobs2, lines2, ringend2, offset_x)
        forwardpass_stochastic(bobs3, lines3, ringend3, offset_x1)

        current_loss = loss()
        current_loss2 = lossv()
        current_loss3 = losss()

        if current_loss2 < 0.01:
            if ringend1.color != color.green: print("GD", time.time()-t0)
            ringend1.color = color.green

        if current_loss < 0.01:
            if ringend2.color != color.green: print("MGD", time.time()-t0)
            ringend2.color = color.green

        if current_loss3 < 0.01:
            if ringend3.color != color.green: print("SGD", time.time()-t0)
            ringend3.color = color.green

        if current_loss < 0.01 and current_loss2 < 0.01 and current_loss3 < 0.01:
            time.sleep(0.5)
            t0 = time.time()
            break
```

Listing 4.1: Code for Comparing Gradient Descent Algorithms

4.6 Animation Result

The following video demonstrates the animation of the gradient descent algorithms in action.

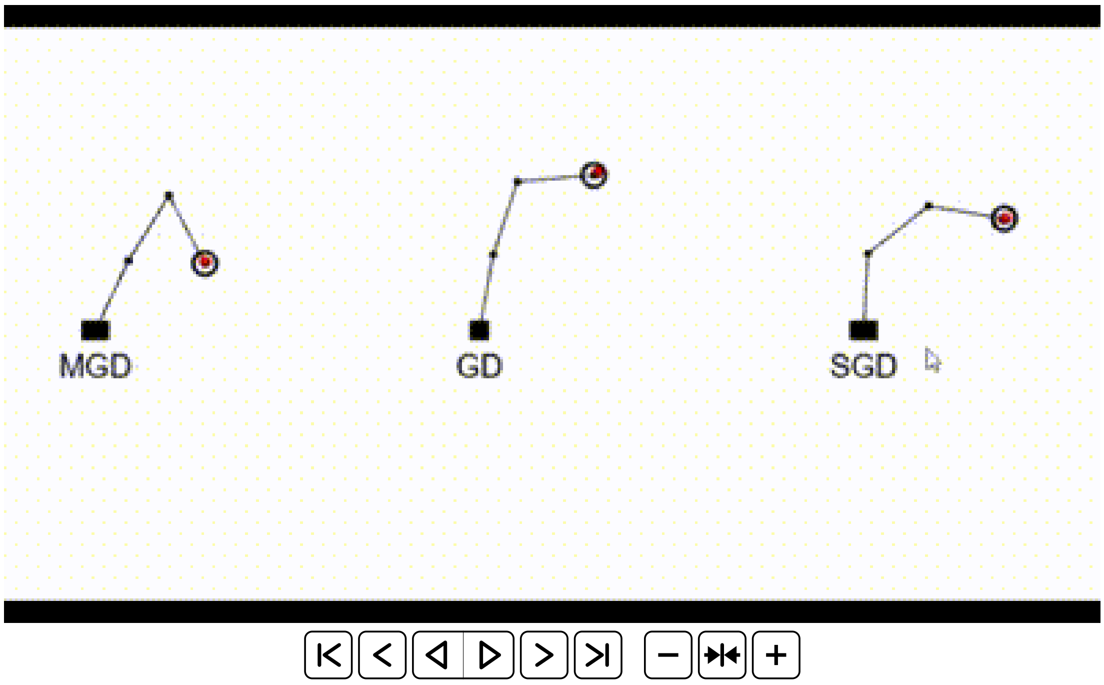


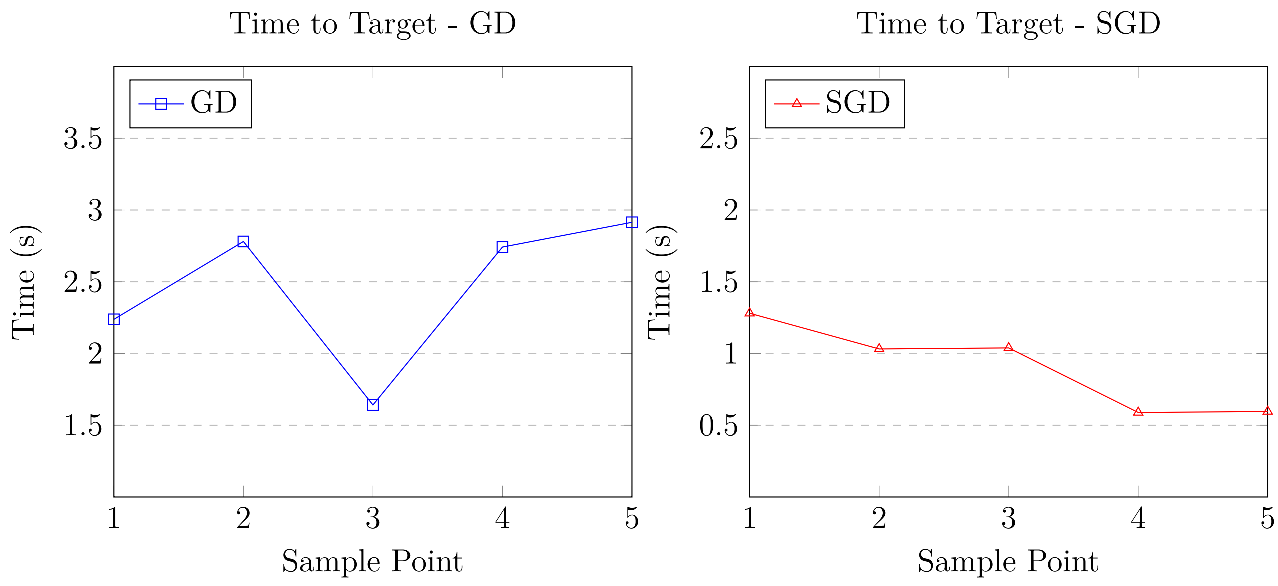
Figure 4.6.1: A visual comparison of GD , SGD , MGD for 3 parameters.

4.7 Results and Comparison of Optimization Techniques

In this section, we present the results obtained from comparing three optimization techniques: Gradient Descent (GD), Stochastic Gradient Descent (SGD), and Gradient Descent with Momentum (GDM). The performance of each technique is measured in terms of the time taken to reach the target at each sampling point.

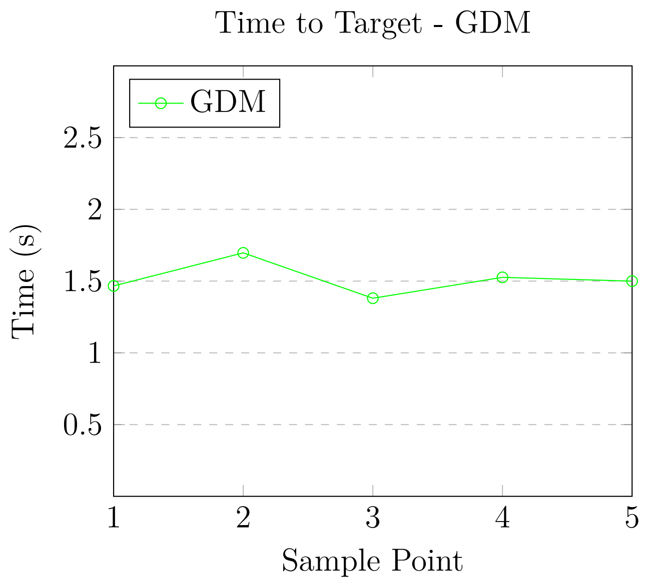
4.7.1 Graphical Representation

The following plots show the time taken by each to reach the target.



(a) Gradient Descent

(b) Stochastic Gradient Descent



(c) Gradient Descent with Momentum

Figure 4.7.2: Time to target for each optimization technique at different sampling points.

4.7.2 Time to Target for Each Technique

The following table summarizes the time taken by each optimization technique to reach the target for different sampling points.

Sample Point	**GD Time (s)**	**SGD Time (s)**	**GDM Time (s)**
Point 1	2.238104820251465	1.2803943157196045	1.4662985801696777
Point 2	2.780674695968628	1.0317022800445557	1.6968607902526855
Point 3	1.6414148330688477	1.0389554500579834	1.3805487155914307
Point 4	2.7418665885925293	0.5890228748321533	1.5261976718902588
Point 5	2.9143810272216797	0.5958073139190674	1.4997186660766602

Table 4.7.1: Time to target for each optimization technique at different sampling points.

4.7.3 Discussion

From the results presented in Table 4.7.1 and Figure 4.7.2, we observe the following:
- Gradient Descent (GD) shows a relatively consistent time to target, but it is slower compared to the other techniques.

- Stochastic Gradient Descent (SGD) is the fastest among the three techniques, showing the shortest time to target across all sample points.

- Gradient Descent with Momentum (GDM) also performs well, consistently reaching the target faster than GD but slower than SGD.

These results suggest that SGD is the most efficient optimization technique among the three, followed by GDM and then GD.

4.8 Comparison of GD and SGD for Varying Parameters

In this section, we compare the performance of Gradient Descent (GD) and Stochastic Gradient Descent (SGD) for different numbers of parameters. The range of parameters considered is from 5 to 100. For each parameter setting, 10 points were sampled, and the time taken by both optimization techniques to reach the target was recorded. The average time taken for each parameter setting was then calculated. The difference in average time taken by GD and SGD is plotted to highlight the performance variations.

4.8.1 Methodology

The experiment involves running GD and SGD for increasing parameter sizes. For each parameter size, 10 points were sampled, and the time taken to reach the target was recorded. The average time for each parameter setting was calculated as follows:

$$\overline{\text{Time}} = \frac{1}{10}\sum_{i=1}^{10} \text{Time}_i$$

The difference in average time taken by GD and SGD is then calculated as follows:

$$\Delta\overline{\text{Time}} = \overline{\text{Time}}_{\text{GD}} - \overline{\text{Time}}_{\text{SGD}}$$

A positive value of $\Delta\overline{\text{Time}}$ indicates that SGD is faster, while a negative value indicates that GD is faster.

4.8.2 Results

The following plots shows the time taken by GD ,SGD and difference in average time taken by GD and SGD for parameters ranging from 5 to 100.

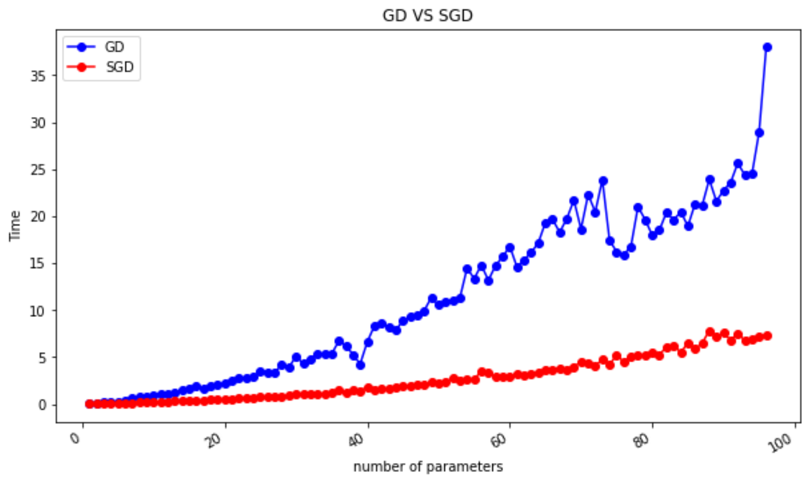


Figure 4.8.3: Time taken by GD and SGD for parameters from 5 to 100.

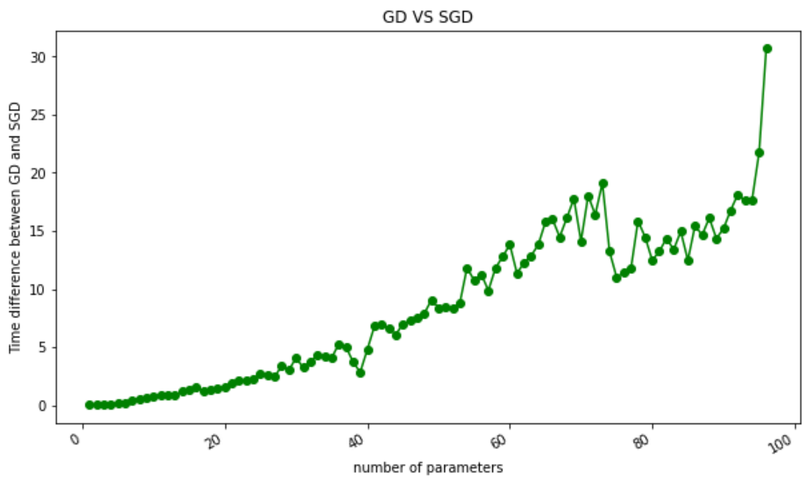


Figure 4.8.4: Difference in average time taken by GD and SGD for varying parameters. A positive value indicates SGD is faster.

4.8.3 Discussion

From the plot in Figure 4.8.4, we can make the following observations:

- Trend Analysis: The difference in average time taken ($\Delta\overline{\text{Time}}$) is consistently positive, indicating that SGD is faster than GD across all parameter settings.
- Performance Gap: The performance gap between GD and SGD increases with the number of parameters. For smaller parameter sizes, the difference is smaller, while for larger parameter sizes, the difference becomes more pronounced.
- Scalability: SGD demonstrates better scalability compared to GD, as it maintains a lower average time to target even as the number of parameters increases.

These findings further reinforce the efficiency of SGD over GD, particularly for problems with a large number of parameters.

4.9 Code used to produce the above the results

4.9.1 SGD Optimization Code

The following code snippet demonstrates the optimization process using Stochastic Gradient Descent (SGD) for parameters ranging from 5 to 100. For each parameter size, 10 points were sampled, and the average time taken to reach the target was calculated.

```python
import numpy as np
import time
import pandas as pd

alpha = 0.05
itime = []
time_n=[]

def samplepoints(R1, R2):
    theta = np.random.uniform(np.pi / 6, np.pi / 4)
    u = np.random.uniform(0, 1)
    r = np.sqrt(u * (R2**2 - R1**2) + R1**2)
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return np.array([x, y, 0])

def initialize_pendulums(n, lengths, thetas, offset_x=0):
    R1 = (n) -0.5
    R2 = (n*2) -0.5
    target_point = samplepoints(R1, R2) + np.array([offset_x,
    0.0, 0.0])
    bobs = [np.array([0.0, 0.0, 0.0]) for _ in range(n)]
    bobs[0] = np.array([lengths[0] * np.sin(thetas[0]), -lengths
    [0] * np.cos(thetas[0]), 0]) + np.array([offset_x, 0.0, 0.0])
    for i in range(1, n):
        bobs[i] = bobs[i-1] + np.array([lengths[i] * np.sin(
    thetas[i]), -lengths[i] * np.cos(thetas[i]), 0])
    return bobs, target_point

def forwardpass(bobs, lengths, thetas, n, offset_x=0):
    bobs[0] = np.array([lengths[0] * np.sin(thetas[0]), -lengths
    [0] * np.cos(thetas[0]), 0]) + np.array([offset_x, 0.0, 0.0])
    for i in range(1, n):
        bobs[i] = bobs[i-1] + np.array([lengths[i] * np.sin(
    thetas[i]), -lengths[i] * np.cos(thetas[i]), 0])

def endpoint(bobs):
    return bobs[-1]

def loss(bobs, target_point):
    return np.linalg.norm(endpoint(bobs) - target_point)

def metricfunc(thetas, lengths, target_point, n, offset_x=0):
    pos = np.array([0.0, 0.0, 0.0]) + np.array([offset_x, 0.0,
    0.0])
    for i in range(n):
        pos += np.array([lengths[i] * np.sin(thetas[i]), -lengths
    [i] * np.cos(thetas[i]), 0])
    return np.linalg.norm(pos - target_point)

def stochastic_gradients(thetas, lengths, target_point, n,
    offset_x=0, sample_size=1):
    delta = 1
    grads = np.zeros(n)
    indices = np.random.choice(n, sample_size)
    for i in indices:
        thetas[i] += delta
        loss1 = metricfunc(thetas, lengths, target_point, n,
    offset_x)
        thetas[i] -= 2 * delta
        loss2 = metricfunc(thetas, lengths, target_point, n,
    offset_x)
        thetas[i] += delta
        grads[i] = (loss1 - loss2) / (2 * delta)
    return grads

def optimize_pendulums(n, lengths, thetas, offset_x=0):
    bobs, target_point = initialize_pendulums(n, lengths, thetas,
    offset_x)

    t0 = time.time()
    while True:
        grad = stochastic_gradients(thetas, lengths, target_point
    , n, sample_size=1)
        thetas -= alpha * grad
        thetas = np.clip(thetas, 0, np.pi)

        forwardpass(bobs, lengths, thetas, n)

        current_loss = loss(bobs, target_point)

        if current_loss < 0.01:
            t1 = time.time() - t0
            itime.append(t1)

            print(f"n = {n}")
            #print(f"Sampled point: {target_point}")
            #print(f"Optimized point: {bobs[-1]}")
            #print(f"Loss: {current_loss}")
            print(f"Time taken: {t1} seconds\n")
            break

print("Starting optimization for different n values...")
for n in range(5, 101):
    for _ in range(10):
        lengths = [2] * n
        thetas = [np.pi / (i + 3) for i in range(n)]
        optimize_pendulums(n, lengths, thetas)

print("Optimization finished for all n values.")
print(itime)

for i in range(len(itime)//10):
    start_index = i * 10
    end_index = start_index + 10
    segment = itime[start_index:end_index]
    average = sum(segment) / len(segment)
    time_n.append(average)
```

```python
print(time_n)

time_n = pd.DataFrame(time_n)

csv_data = time_n.to_csv('timesgd.csv',index=False)
```

Listing 4.2: SGD Optimization Code for Parameters 5 to 100

4.9.2 GD Optimization Code

The following code snippet demonstrates the optimization process using Gradient Descent (GD) for parameters ranging from 5 to 100. For each parameter size, 10 points were sampled, and the average time taken to reach the target was calculated.

```python
import numpy as np
import time
import pandas as pd

alpha = 0.01
itime = []
time_n = []

def samplepoints(R1, R2):
    theta = np.random.uniform(np.pi / 6, np.pi / 4)
    u = np.random.uniform(0, 1)
    r = np.sqrt(u * (R2**2 - R1**2) + R1**2)
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return np.array([x, y, 0])

def initialize_pendulums(n, lengths, thetas, offset_x=0):
    R1 = (n) - 0.5
    R2 = (n * 2) - 0.5
    target_point = samplepoints(R1, R2) + np.array([offset_x,
    0.0, 0.0])
    bobs = [np.array([0.0, 0.0, 0.0]) for _ in range(n)]
    bobs[0] = np.array([lengths[0] * np.sin(thetas[0]), -lengths
    [0] * np.cos(thetas[0]), 0]) + np.array([offset_x, 0.0, 0.0])
    for i in range(1, n):
        bobs[i] = bobs[i-1] + np.array([lengths[i] * np.sin(
    thetas[i]), -lengths[i] * np.cos(thetas[i]), 0])
    return bobs, target_point

def forwardpass(bobs, lengths, thetas, n, offset_x=0):
    bobs[0] = np.array([lengths[0] * np.sin(thetas[0]), -lengths
    [0] * np.cos(thetas[0]), 0]) + np.array([offset_x, 0.0, 0.0])
    for i in range(1, n):
        bobs[i] = bobs[i-1] + np.array([lengths[i] * np.sin(
    thetas[i]), -lengths[i] * np.cos(thetas[i]), 0])

def endpoint(bobs):
    return bobs[-1]

def loss(bobs, target_point):
    return np.linalg.norm(endpoint(bobs) - target_point)

def metricfunc(thetas, lengths, target_point, n, offset_x=0):
    pos = np.array([0.0, 0.0, 0.0]) + np.array([offset_x, 0.0, 0.0])
    for i in range(n):
        pos += np.array([lengths[i] * np.sin(thetas[i]), -lengths[i] * np.cos(thetas[i]), 0])
    return np.linalg.norm(pos - target_point)

def gradients(thetas, lengths, target_point, n, offset_x=0):
    delta = 1
    grads = np.zeros(n)
    for i in range(n):
        thetas[i] += delta
        loss1 = metricfunc(thetas, lengths, target_point, n, offset_x)
        thetas[i] -= 2 * delta
        loss2 = metricfunc(thetas, lengths, target_point, n, offset_x)
        thetas[i] += delta
        grads[i] = (loss1 - loss2) / (2 * delta)
    return grads

def optimize_pendulums(n, lengths, thetas, offset_x=0):
    bobs, target_point = initialize_pendulums(n, lengths, thetas, offset_x)

    t0 = time.time()
    while True:
        grad = gradients(thetas, lengths, target_point, n, offset_x)
        thetas -= alpha * grad
        thetas = np.clip(thetas, 0, np.pi)

        forwardpass(bobs, lengths, thetas, n)

        current_loss = loss(bobs, target_point)

        if current_loss < 0.01:
            t1 = time.time() - t0
            itime.append(t1)
            print(f"n = {n}")
            #print(f"Sampled point: {target_point}")
            #print(f"Optimized point: {bobs[-1]}")
            #print(f"Loss: {current_loss}")
            print(f"Time taken: {t1} seconds\n")
            break

print("Starting optimization for different n values...")
for n in range(5, 101):
    for _ in range(10):
        lengths = [2] * n
        thetas = [np.pi / (i + 3) for i in range(n)]
        optimize_pendulums(n, lengths, thetas)

print("Optimization finished for all n values.")

for i in range(96):
    start_index = i * 10
    end_index = start_index + 10
    segment = itime[start_index:end_index]
    average = sum(segment) / len(segment)
    time_n.append(average)

print(time_n)
time_n_df = pd.DataFrame(time_n, columns=['Average Time'])
time_n_df.to_csv('time.csv', index=False)
print("Results saved to 'gdtime.csv'.")
```

Listing 4.3: SGD Optimization Code for Parameters 5 to 100

Chapter 5

Physics-Informed Neural Networks (PINNs)

5.1 Theory of PINNs

Physics-Informed Neural Networks (PINNs) are a powerful approach for solving differential equations by incorporating physical laws directly into the training process of neural networks. This method is particularly useful when dealing with complex systems where traditional numerical methods might be challenging.

Mathematical Framework

Consider a system governed by a differential equation:

$$\mathcal{N}(u(x,t), \nabla u(x,t), \nabla^2 u(x,t), x, t) = 0$$

where $u(x,t)$ is the function we want to predict, and $\mathcal{N}$ is a differential operator. The objective of PINNs is to train a neural network $\hat{u}(x,t)$ to approximate $u(x,t)$ such that:
1. Data Loss: The network's prediction fits the observed data:

$$L_{\text{data}} = \frac{1}{N_d} \sum_{i=1}^{N_d} |\hat{u}(x_i, t_i) - u_i|^2$$

2. Physics Loss: The network satisfies the differential equation:

$$L_{\text{physics}} = \frac{1}{N_p} \sum_{i=1}^{N_p} \left| \mathcal{N}(\hat{u}(x_i, t_i), \nabla \hat{u}(x_i, t_i), \nabla^2 \hat{u}(x_i, t_i), x_i, t_i) \right|^2$$

The total loss function to minimize is:

$$L = \lambda_1 L_{\text{data}} + \lambda_2 L_{\text{physics}}$$

where λ_1 and λ_2 are weights that balance the data and physics contributions.

5.1.1 Damped Harmonic Oscillator

The damped harmonic oscillator is a fundamental problem described by the following second-order differential equation:

$$\frac{d^2u}{dt^2} + 2\delta\frac{du}{dt} + \omega_0^2 u = 0$$

where: - $u(t)$ is the displacement, - δ is the damping coefficient, - ω_0 is the natural frequency of the system.
The exact solution for the displacement $u(t)$ is given by:

$$u(t) = Ae^{-\delta t}\cos(\omega t + \phi)$$

where A and ϕ are constants determined by initial conditions.

5.1.2 PINNs for Damped Harmonic Oscillator

To apply PINNs to the damped harmonic oscillator, we perform the following steps:
1. Define the Exact Solution: Calculate $u(t)$ using the exact solution formula.
2. Design the Neural Network: Implement a neural network to approximate $u(t)$.
3. Formulate the Loss Function: Include both the data loss and physics loss in the training process.
4. Training and Results: Train the network using the defined loss function and evaluate its performance.

5.1.3 Code Implementation

The Python code used for fitting the damped harmonic oscillator with a PINN is as follows:

```python
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

def exact_solution(d, w0, t):
    w = np.sqrt(w0**2 - d**2)
    phi = np.arctan(-d / w)
    A = 1 / (2 * np.cos(phi))
    cos = torch.cos(phi + w * t)
    exp = torch.exp(-d * t)
    u = exp * 2 * A * cos
    return u

d = 2
w0 = 20
mu = 2 * d
k = w0**2

t = torch.linspace(0, 1, 500).view(-1, 1)
u_exact = exact_solution(d, w0, t)

t_test = t[0:200:20]
u_test = u_exact[0:200:20]

class NODE(nn.Module):
    def __init__(self, n_input, n_output, n_hidden):
        super().__init__()
        activation = nn.Tanh
        self.fc1 = nn.Sequential(nn.Linear(n_input, n_hidden),
    activation())
        self.fc2 = nn.Sequential(nn.Linear(n_hidden, n_hidden),
    activation())
        self.fc3 = nn.Sequential(nn.Linear(n_hidden, n_output))

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x

def plot_results(t,u_exact,t_test,u_test,u_pred,xp=None):
  plt.figure(figsize=(8,4))
  plt.plot(t,u_exact, color="grey", linewidth=2, alpha=0.8, label
    ="Exact solution")
  plt.plot(t,u_pred, color="tab:blue", linewidth=4, alpha=0.8,
    label="Neural network prediction")
  plt.scatter(t_test, u_test, s=60, color="tab:orange", alpha
    =0.4, label='Training data')
  if xp is not None:
      plt.scatter(xp, -0*torch.ones_like(xp), s=60, color="tab:
    green", alpha=0.4,
                  label='Physics loss training locations')
  l = plt.legend(loc=(1.01,0.34), frameon=False, fontsize="large")
  plt.setp(l.get_texts(), color="k")
  plt.xlim(-0.05, 1.05)
  plt.ylim(-1.1, 1.1)
  plt.text(1.065,0.7,"Training step: %i"%(i+1),fontsize="xx-large",color="k")
  plt.axis("off")

torch.manual_seed(123)
model = NODE(1, 1, 32)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

t_boundary = torch.tensor([[0.0]], requires_grad=True)
u_boundary = torch.tensor([[1.0]], requires_grad=True)

t_physics = torch.linspace(0, 1, 30).view(-1, 1)
t_physics.requires_grad = True

epochs = 20000
loss_fn = nn.MSELoss()

losses = []
accuracy_list = []

for i in range(epochs):
    optimizer.zero_grad()

    u_pred = model(t_test)
    loss1 = loss_fn(u_pred,u_test)

    u_pred_physics = model(t_physics)
    u_t = torch.autograd.grad(u_pred_physics, t_physics, torch.ones_like(u_pred_physics), create_graph=True)[0]
    u_tt = torch.autograd.grad(u_t, t_physics, torch.ones_like(u_t), create_graph=True)[0]
    physics_loss = u_tt + mu * u_t + k * u_pred_physics

    physics_loss = loss_fn(physics_loss, torch.zeros_like(physics_loss))*(1e-4)

    loss =  physics_loss + loss1

    loss.backward()
    optimizer.step()
    if i % 1000 == 0:
        u_pred = model(t).detach()
        xp = t_physics.detach()

        plot_results(t,u_exact,t_test,u_test,u_pred,xp)
        plt.show()
        print(f'Epoch {i}, Loss: {loss.item()}')
```

Listing 5.1: SGD Optimization Code for Parameters 5 to 100

5.2 Results from the Physics-Informed Neural Network (PINN)

In this section, we present the results obtained from training the Physics-Informed Neural Network (PINN) to model the damped harmonic oscillator. The performance of the model is evaluated by observing the evolution of the predicted solution over different training epochs.

5.2.1 Evolution of the Model

To evaluate the progression of the model, we captured the state of the neural network after every 6000 epochs during training. The following images show the evolution of the model's prediction as it progresses through the training process.

Epoch 0

At epoch 0, the model's predictions are at the initial state, showing no resemblance to the expected behavior of the damped harmonic oscillator. This initial state is shown in Figure 5.2.1.

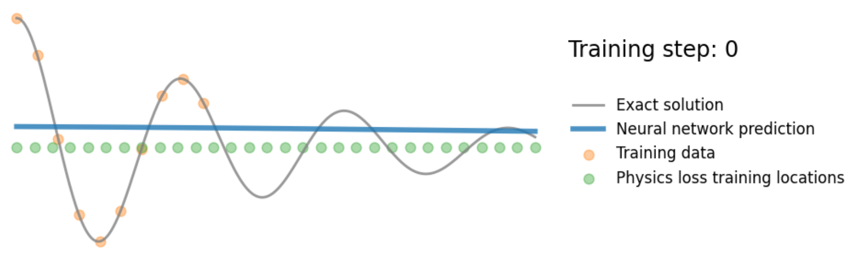


Figure 5.2.1: Model prediction at epoch 0. The neural network is at its initial state, with no learned behavior yet.

Epoch 6000

At 6000 epochs, the model has started to approximate the expected behavior of the damped harmonic oscillator. The initial predictions are shown in Figure 5.2.2.

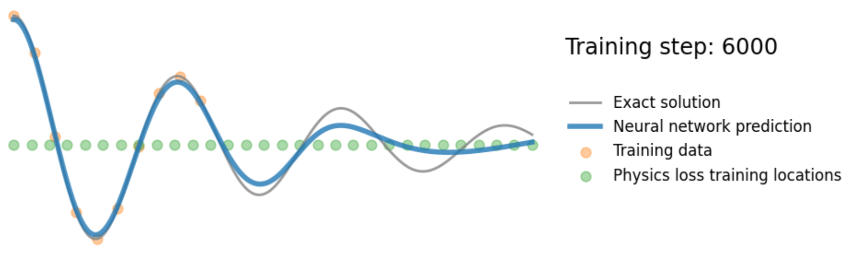


Figure 5.2.2: Model prediction after 6000 epochs. The neural network begins to capture the underlying dynamics of the damped harmonic oscillator.

Epoch 12000

By 12000 epochs, the model's predictions have improved significantly, showing a closer match to the exact solution. The progress is illustrated in Figure 5.2.3.

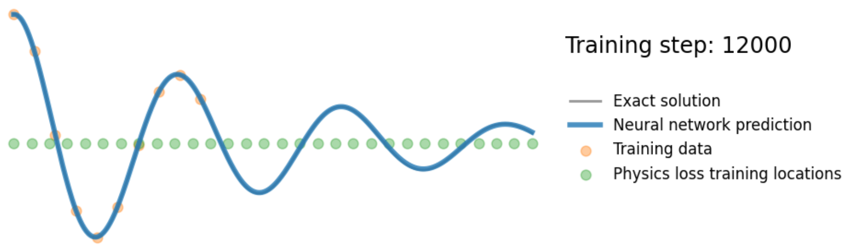


Figure 5.2.3: Model prediction after 12000 epochs. The neural network's prediction aligns more closely with the exact solution.

Epoch 18000

Finally, at the end of training, at 18000 epochs, the model has converged to a highly accurate solution. The results are depicted in Figure5.2.4.

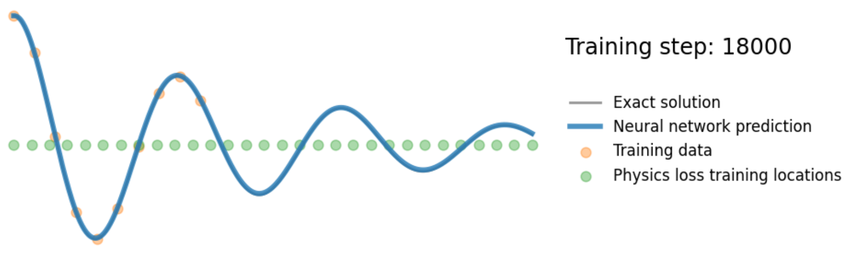


Figure 5.2.4: Model prediction after 18000 epochs. The neural network's output closely matches the exact solution, reflecting the successful training of the PINN.

5.2.2 Animation of Model Evolution

The following animation shows Model Evolution over 12000 epochs:

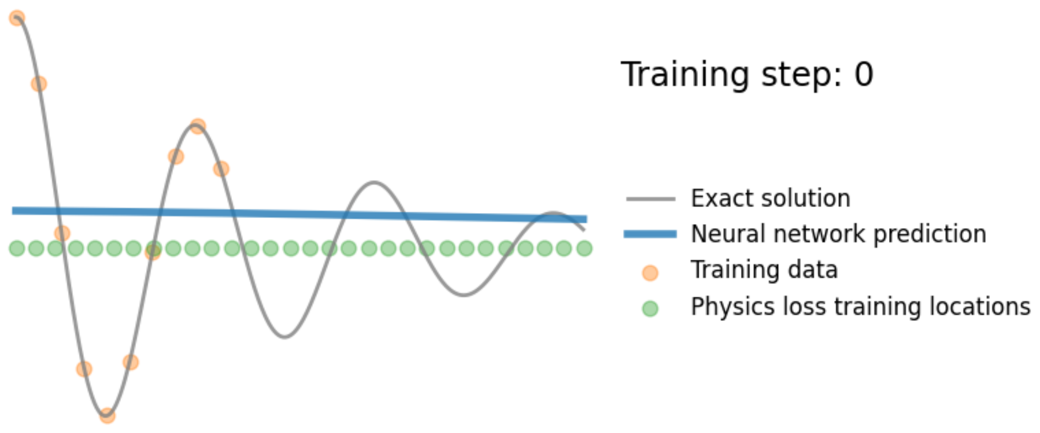


Figure 5.2.5: A visual of model evolution over 12000 epochs.

5.2.3 Discussion

The series of images demonstrates the evolution of the model's predictions throughout the training process. Starting from an initial approximation, the model progressively improves its predictions, reflecting the learning and refinement of the neural network. By the end of training, the PINN accurately models the damped harmonic oscillator, showcasing the potential of using physics-informed neural networks to solve differential equations and other physics-based

problems.
The ability of the PINN to effectively approximate the solution underscores the advantages of integrating physical laws directly into the training process. This approach not only helps in guiding the learning process but also ensures that the predictions adhere to known physical constraints, leading to more reliable and interpretable results.

5.2.4 Acknowledgements

I would like to express our gratitude to Dr. Ratikanta Behera for their support and guidance throughout this project. Additionally, I thank Himnashu Pandey for helping me throughout the Intership.

Bibliography

[1] Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378, 686-707.