**Introduction to Neural Network & Deep Learning**

**Course Code: MEAD-654**

**Lab Practical File**

**Submitted by**

**Shantanu Shukla**

**01211805424**

**in partial fulfillment for the award of the degree**

**of**

**Master of Technology (AI & DS)**

**batch of 2024 - 26 (2nd Semester)**

****

**Center for Development of Advanced Computing, Noida**

**Affiliated to Guru Gobind Singh Indraprastha University**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| 1. **No** | **Title** | **Page No.** | **Signature** |
| 1 | **Lab-1:** Python and Deep Learning Libraries Setup | 3-5 |  |
| 2 | **Lab-2:** Introduction to NumPy for Deep Learning | 6-7 |  |
| 3 | **Lab-3:** Implementing a Perceptron | 8-9 |  |
| 4 | **Lab-4:** Adaline and Gradient Descent | 10-11 |  |
| 5 | **Lab-5:** Building a Feedforward Neural Network (from scratch) | 12-13 |  |
| 6 | **Lab-6:** Building an MLP using PyTorch | 14-15 |  |
| 7 | **Lab-7:** Visualizing Gradient Descent | 16-17 |  |
| 8 | **Lab-8:** Exploring Optimizers | 18-20 |  |
| 9 | **Lab-9:** Understanding Convolution Operations | 21-22 |  |
| 10 | **Lab-10:** Building a CNN with PyTorch | 23-25 |  |

**Lab-1**

**Objective:** Set up Python, Jupyter Notebook, and install popular deep learning libraries (NumPy, TensorFlow, PyTorch).

**Tasks:**

1. Install Anaconda, create a virtual environment.
2. Install TensorFlow, PyTorch, and Keras.
3. Write a simple program to confirm installations.

**Results:**

**a.**

*Installing Anaconda.*



*Creating a Virtual Environment.*

A screen shot of a computer

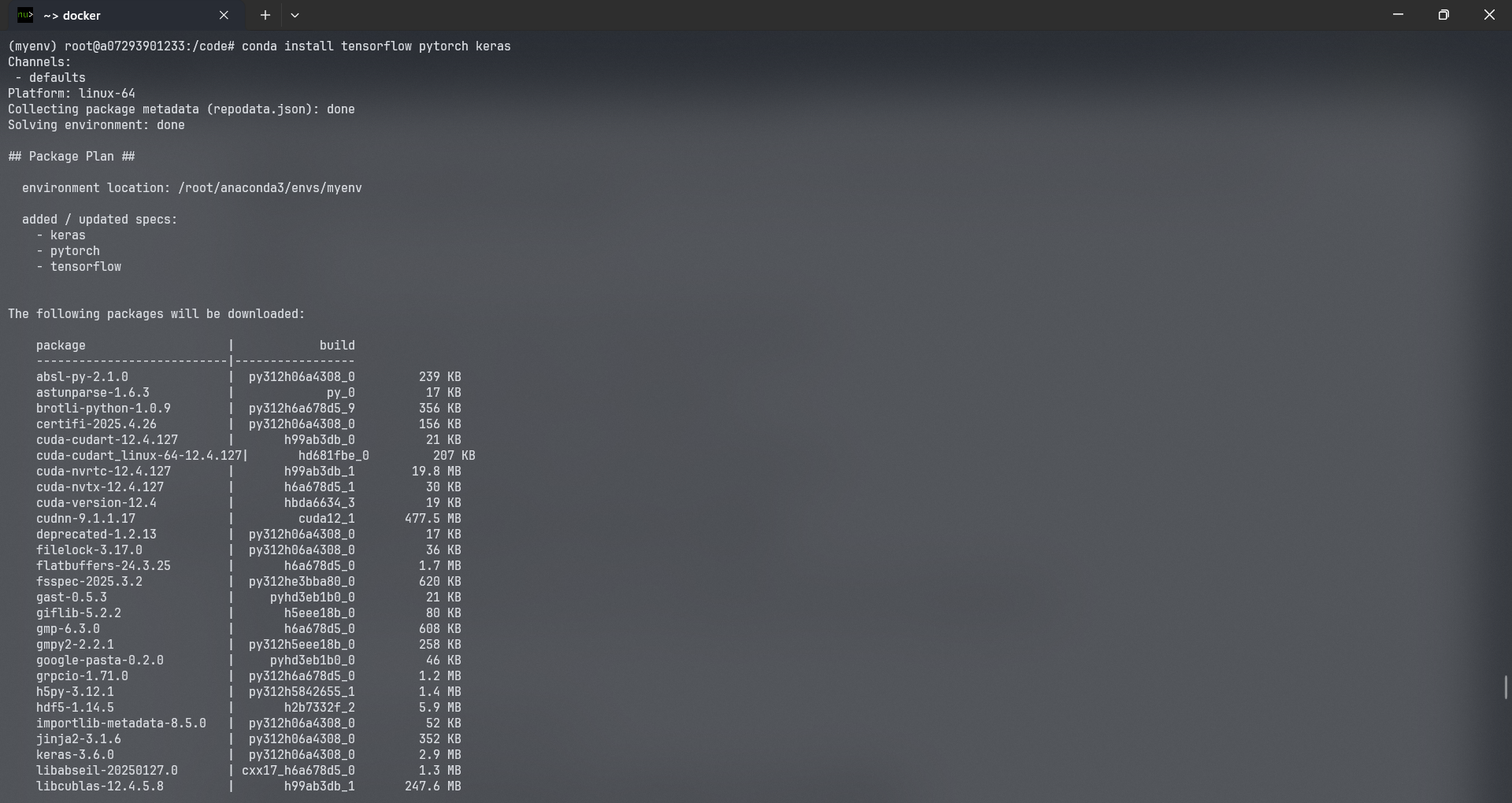
AI-generated content may be incorrect.

*Activating the Virtual Environment.*

A screenshot of a computer program

AI-generated content may be incorrect.

**b.** *Installing TensorFlow, PyTorch, and Keras libraries.*

****

**A black screen with white lines

AI-generated content may be incorrect.**

**c.** *Confirming library installations.*

**A black background with white text

AI-generated content may be incorrect.**

**Lab-2**

**Objective:** Introduction to NumPy for Deep Learning.

**Tasks:**

1. Create and manipulate arrays.
2. Perform matrix multiplication, transpose, and reshaping.

**Results:**

**a.** *Create and manipulate arrays.*

1. import numpy as np

2.

3. # From a Python list

4. a = np.array([1, 2, 3])

5. print(a) # [1 2 3]

6.

7. # 2D array

8. b = np.array([[1, 2, 3], [4, 5, 6]])

9. print(b)

10.

11. # Zeros, Ones, and Identity

12. np.zeros((2, 3)) # 2x3 array of zeros

13. np.ones((3, 2)) # 3x2 array of ones

14. np.eye(3) # 3x3 identity matrix

15.

16. # Range and linspace

17. np.arange(0, 10, 2) # [0 2 4 6 8]

18. np.linspace(0, 1, 5) # [0. 0.25 0.5 0.75 1. ]

19.

20. a = np.array([[10, 20, 30], [40, 50, 60]])

21.

22. print(a[0, 1]) # 20

23. print(a[:, 1]) # Column: [20 50]

24. print(a[1, :]) # Row: [40 50 60]

**A screen shot of a computer

AI-generated content may be incorrect.**

**b.** *Perform matrix multiplication, transpose, and reshaping.*

1. import numpy as np

2.

3. # Step 1: Create two matrices for multiplication

4. A = np.array([[1, 2, 3], [4, 5, 6]])

5. B = np.array([[7, 8], [9, 10], [11, 12]])

6.

7. # Step 2: Matrix multiplication (A is 2x3, B is 3x2 -> result is 2x2)

8. C = A @ B

9. print("Matrix Multiplication (A @ B):\n", C)

10.

11. # Step 3: Transpose the result

12. C\_T = C.T

13. print("\nTranspose of the result:\n", C\_T)

14.

15. # Step 4: Reshape the transposed matrix (from 2x2 to 1x4)

16. C\_reshaped = C\_T.reshape((1, 4))

17. print("\nReshaped Transposed Matrix (1x4):\n", C\_reshaped)

**A black screen with white text

AI-generated content may be incorrect.**

**Lab-3**

**Objective:** Understand and implement a simple Perceptron model from scratch.

**Tasks:**

1. Initialize weights and bias.
2. Implement the activation function (sign function).
3. Train the Perceptron on a linearly separable dataset.

**Results:**

1. import numpy as np

2.

3.

4. class Perceptron:

5. def \_\_init\_\_(self, input\_size, learning\_rate=0.1, epochs=10):

6. # a. Initialize weights and bias

7. self.weights = np.zeros(input\_size)

8. self.bias = 0.0

9. self.lr = learning\_rate

10. self.epochs = epochs

11.

12. # b. Activation function: Sign

13. def activation(self, x):

14. return 1 if x >= 0 else -1

15.

16. def predict(self, x):

17. linear\_output = np.dot(self.weights, x) + self.bias

18. return self.activation(linear\_output)

19.

20. # c. Training

21. def train(self, X, y):

22. for epoch in range(1, self.epochs + 1):

23. print(f"\nEpoch {epoch}")

24. for xi, target in zip(X, y):

25. pred = self.predict(xi)

26. error = target - pred

27. self.weights += self.lr \* error \* xi

28. self.bias += self.lr \* error

29. print(f" Weights: {self.weights}, Bias: {self.bias}")

30.

31.

32. # Linearly separable dataset

33. X = np.array([[2, 1], [1, -1], [-1, -2], [-2, -1]])

34. y = np.array([1, 1, -1, -1]) # Labels: +1 or -1

35.

36. # Train

37. p = Perceptron(input\_size=2, learning\_rate=0.1, epochs=10)

38. p.train(X, y)

39.

40. # Final predictions

41. print("\nFinal predictions:")

42. for xi in X:

43. print(f"{xi} => {p.predict(xi)}")

A screenshot of a computer program

AI-generated content may be incorrect.

**Lab-4**

**Objective**: Implement Adaptive Linear Neuron (Adaline) using gradient descent.

**Tasks:**

1. Initialize weights and bias.
2. Use a continuous loss function (MSE).
3. Implement gradient descent for weight updates.

**Results:**

1. import numpy as np

2.

3.

4. class Adaline:

5. def \_\_init\_\_(self, input\_size, learning\_rate=0.01, epochs=10):

6. # a. Initialize weights and bias

7. self.weights = np.zeros(input\_size)

8. self.bias = 0.0

9. self.lr = learning\_rate

10. self.epochs = epochs

11.

12. def net\_input(self, X):

13. return np.dot(X, self.weights) + self.bias

14.

15. def activation(self, X):

16. # For Adaline, activation is linear (identity function)

17. return self.net\_input(X)

18.

19. def train(self, X, y):

20. for epoch in range(1, self.epochs + 1):

21. # c. Compute predictions and errors

22. output = self.activation(X)

23. errors = y - output

24.

25. # b. Compute Mean Squared Error (MSE)

26. mse = np.mean(errors\*\*2)

27.

28. # c. Gradient descent: update weights and bias

29. self.weights += self.lr \* np.dot(X.T, errors)

30. self.bias += self.lr \* errors.sum()

31.

32. print(

33. f"Epoch {epoch}: MSE = {mse:.4f}, Weights = {self.weights}, Bias = {self.bias}"

34. )

35.

36. def predict(self, X):

37. return np.where(self.activation(X) >= 0.0, 1, -1)

38.

39.

40. if \_\_name\_\_ == "\_\_main\_\_":

41. # Simple dataset: linearly separable

42. X = np.array([[1, 1], [2, 1], [1, -1], [-1, -2], [-2, -1]])

43. y = np.array([1, 1, 1, -1, -1]) # Targets in {-1, +1}

44.

45. model = Adaline(input\_size=2, learning\_rate=0.01, epochs=10)

46. model.train(X, y)

47.

48. # Predictions

49. print("\nFinal predictions:")

50. for xi in X:

51. print(f"{xi} => {model.predict(xi)}")

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Lab-5**

**Objective:** Manually build an MLP with one hidden layer.

**Tasks:**

1. Initialize weights and biases.
2. Implement forward propagation.
3. Apply activation functions (ReLU, Sigmoid).
4. Implement backpropagation for weight updates.

**Results:**

1. import numpy as np

2.

3.

4. def sigmoid(x):

5. return 1 / (1 + np.exp(-x))

6.

7.

8. def sigmoid\_derivative(x):

9. sx = sigmoid(x)

10. return sx \* (1 - sx)

11.

12.

13. def relu(x):

14. return np.maximum(0, x)

15.

16.

17. def relu\_derivative(x):

18. return (x > 0).astype(float)

19.

20.

21. class MLP:

22. def \_\_init\_\_(self, input\_size, hidden\_size, learning\_rate=0.1):

23. # a. Initialize weights and biases

24. self.w1 = np.random.randn(input\_size, hidden\_size) # (2, 2)

25. self.b1 = np.zeros((1, hidden\_size)) # (1, 2)

26. self.w2 = np.random.randn(hidden\_size, 1) # (2, 1)

27. self.b2 = np.zeros((1, 1)) # (1, 1)

28. self.lr = learning\_rate

29.

30. def forward(self, X):

31. # b. Forward propagation

32. self.z1 = np.dot(X, self.w1) + self.b1

33. self.a1 = relu(self.z1) # c. ReLU activation in hidden layer

34.

35. self.z2 = np.dot(self.a1, self.w2) + self.b2

36. self.a2 = sigmoid(self.z2) # c. Sigmoid activation in output

37.

38. return self.a2

39.

40. def backward(self, X, y):

41. # d. Backpropagation

42.

43. # Output layer error

44. output\_error = self.a2 - y # dL/da2

45. output\_delta = output\_error \* sigmoid\_derivative(self.z2)

46.

47. # Hidden layer error

48. hidden\_error = np.dot(output\_delta, self.w2.T)

49. hidden\_delta = hidden\_error \* relu\_derivative(self.z1)

50.

51. # Gradient descent updates

52. self.w2 -= self.lr \* np.dot(self.a1.T, output\_delta)

53. self.b2 -= self.lr \* np.sum(output\_delta, axis=0, keepdims=True)

54.

55. self.w1 -= self.lr \* np.dot(X.T, hidden\_delta)

56. self.b1 -= self.lr \* np.sum(hidden\_delta, axis=0, keepdims=True)

57.

58. def train(self, X, y, epochs=1000):

59. for epoch in range(1, epochs + 1):

60. output = self.forward(X)

61. self.backward(X, y)

62. if epoch % 100 == 0:

63. loss = np.mean((y - output) \*\* 2)

64. print(f"Epoch {epoch}: Loss = {loss:.4f}")

65.

66. def predict(self, X):

67. output = self.forward(X)

68. return (output > 0.5).astype(int)

69.

70.

71. if \_\_name\_\_ == "\_\_main\_\_":

72. # Input features

73. X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

74.

75. # XOR-like targets

76. y = np.array([[0], [1], [1], [0]])

77.

78. mlp = MLP(input\_size=2, hidden\_size=2, learning\_rate=0.1)

79. mlp.train(X, y, epochs=1000)

80.

81. print("\nPredictions:")

82. for xi in X:

83. print(f"{xi} => {mlp.predict(np.array([xi]))[0][0]}")

**A black rectangle with white dots

AI-generated content may be incorrect.**

**Lab-6**

**Objective:** Use PyTorch to quickly build an MLP.

**Tasks:**

1. Use nn.Module to create a model.
2. Train the model using an optimizer (SGD).
3. Evaluate model performance.

**Results:**

1. import torch

2. import torch.nn as nn

3. import torch.optim as optim

4.

5.

6. # a. Define MLP using nn.Module

7. class MLP(nn.Module):

8. def \_\_init\_\_(self, input\_size=2, hidden\_size=4):

9. super(MLP, self).\_\_init\_\_()

10. self.model = nn.Sequential(

11. nn.Linear(input\_size, hidden\_size),

12. nn.ReLU(),

13. nn.Linear(hidden\_size, 1),

14. nn.Sigmoid(), # for binary classification

15. )

16.

17. def forward(self, x):

18. return self.model(x)

19.

20.

21. # Example XOR data

22. X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)

23. y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)

24.

25. # Model, loss, optimizer

26. model = MLP()

27. criterion = nn.BCELoss() # Binary Cross Entropy Loss

28. optimizer = optim.SGD(model.parameters(), lr=0.1)

29.

30. # b. Training loop

31. for epoch in range(1000):

32. # Forward pass

33. output = model(X)

34. loss = criterion(output, y)

35.

36. # Backward pass

37. optimizer.zero\_grad()

38. loss.backward()

39. optimizer.step()

40.

41. if (epoch + 1) % 100 == 0:

42. print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")

43.

44. # c. Evaluation

45. with torch.no\_grad():

46. predictions = model(X)

47. predicted\_classes = (predictions >= 0.5).float()

48. accuracy = (predicted\_classes == y).float().mean()

49. print("\nPredictions:", predicted\_classes.view(-1).tolist())

50. print(f"Accuracy: {accuracy.item() \* 100:.2f}%")

**A computer screen with text on it

AI-generated content may be incorrect.**

**Lab-7**

**Objective:** Understand gradient descent with visualization.

**Tasks:**

1. Plot a 2D loss surface.
2. Visualize gradient descent steps.

**Results:**

1. import numpy as np

2. import matplotlib.pyplot as plt

3.

4.

5. # a. Define 2D quadratic loss surface: f(w) = w1² + w2²

6. def loss(w):

7. return w[0] \*\* 2 + w[1] \*\* 2

8.

9.

10. def grad(w):

11. return 2 \* w # gradient of the loss

12.

13.

14. # Gradient Descent function

15. def gradient\_descent(start, lr=0.1, steps=20):

16. path = [start]

17. w = start.copy()

18.

19. for \_ in range(steps):

20. g = grad(w)

21. w = w - lr \* g

22. path.append(w.copy())

23.

24. return np.array(path)

25.

26.

27. # Generate path

28. start = np.array([2.0, 1.5]) # starting point

29. trajectory = gradient\_descent(start, lr=0.1, steps=20)

30.

31. # b. Plot the loss surface and the path

32. w1 = np.linspace(-3, 3, 100)

33. w2 = np.linspace(-3, 3, 100)

34. W1, W2 = np.meshgrid(w1, w2)

35. Z = W1\*\*2 + W2\*\*2

36.

37. plt.figure(figsize=(8, 6))

38. plt.contour(W1, W2, Z, levels=30, cmap="viridis")

39. plt.plot(

40. trajectory[:, 0], trajectory[:, 1], "o-", color="red", label="Gradient Descent Path"

41. )

42. plt.scatter(0, 0, c="black", marker="x", s=100, label="Minimum")

43. plt.title("Gradient Descent on 2D Loss Surface")

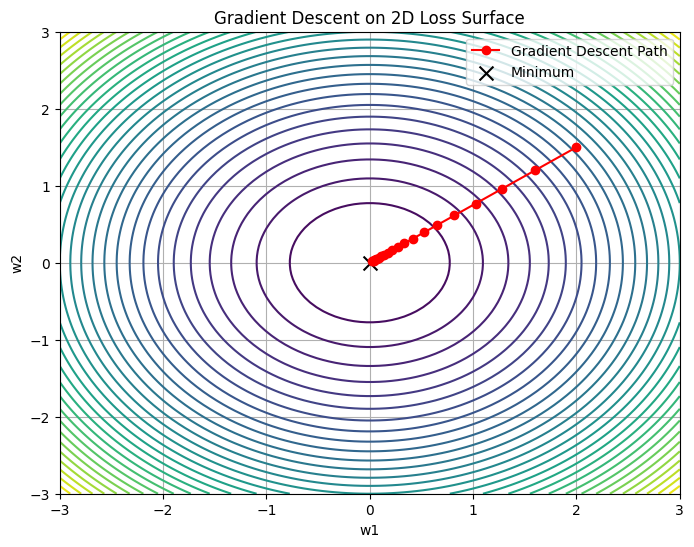
44. plt.xlabel("w1")

45. plt.ylabel("w2")

46. plt.legend()

47. plt.grid(True)

48. plt.show()

****

**Lab-8**

**Objective:** Explore different optimization algorithms (SGD, Adam, RMSprop).

**Tasks:**

1. Implement each optimizer manually.
2. Compare their convergence on a sample problem.

**Results:**

1. import numpy as np

2. import matplotlib.pyplot as plt

3.

4.

5. # Loss and gradient

6. def loss(w):

7. return w[0] \*\* 2 + w[1] \*\* 2

8.

9.

10. def grad(w):

11. return 2 \* w

12.

13.

14. # SGD optimizer

15. def sgd(w, lr):

16. return w - lr \* grad(w)

17.

18.

19. # RMSprop optimizer

20. def rmsprop(w, lr, grad\_func, cache, beta=0.9, epsilon=1e-8):

21. g = grad\_func(w)

22. cache = beta \* cache + (1 - beta) \* g\*\*2

23. w -= lr \* g / (np.sqrt(cache) + epsilon)

24. return w, cache

25.

26.

27. # Adam optimizer

28. def adam(w, lr, grad\_func, m, v, t, beta1=0.9, beta2=0.999, epsilon=1e-8):

29. g = grad\_func(w)

30. m = beta1 \* m + (1 - beta1) \* g

31. v = beta2 \* v + (1 - beta2) \* g\*\*2

32.

33. m\_hat = m / (1 - beta1\*\*t)

34. v\_hat = v / (1 - beta2\*\*t)

35.

36. w -= lr \* m\_hat / (np.sqrt(v\_hat) + epsilon)

37. return w, m, v

38.

39.

40. def run\_optimizer(name, steps=50, lr=0.1):

41. w = np.array([2.0, 1.5])

42. trajectory = [w.copy()]

43.

44. # optimizer-specific variables

45. cache = np.zeros\_like(w)

46. m, v = np.zeros\_like(w), np.zeros\_like(w)

47.

48. for t in range(1, steps + 1):

49. if name == "sgd":

50. w = sgd(w, lr)

51. elif name == "rmsprop":

52. w, cache = rmsprop(w, lr, grad, cache)

53. elif name == "adam":

54. w, m, v = adam(w, lr, grad, m, v, t)

55. trajectory.append(w.copy())

56.

57. return np.array(trajectory)

58.

59.

60. # Generate trajectories

61. traj\_sgd = run\_optimizer("sgd", lr=0.1)

62. traj\_rmsprop = run\_optimizer("rmsprop", lr=0.01)

63. traj\_adam = run\_optimizer("adam", lr=0.05)

64.

65. # Plot loss surface

66. w1 = np.linspace(-3, 3, 100)

67. w2 = np.linspace(-3, 3, 100)

68. W1, W2 = np.meshgrid(w1, w2)

69. Z = W1\*\*2 + W2\*\*2

70.

71. plt.figure(figsize=(10, 7))

72. plt.contour(W1, W2, Z, levels=30, cmap="viridis")

73. plt.plot(traj\_sgd[:, 0], traj\_sgd[:, 1], "o-", label="SGD", color="red")

74. plt.plot(traj\_rmsprop[:, 0], traj\_rmsprop[:, 1], "o-", label="RMSprop", color="blue")

75. plt.plot(traj\_adam[:, 0], traj\_adam[:, 1], "o-", label="Adam", color="green")

76. plt.scatter(0, 0, c="black", marker="x", s=100, label="Minimum")

77. plt.title("Optimizer Trajectories on f(w) = w1² + w2²")

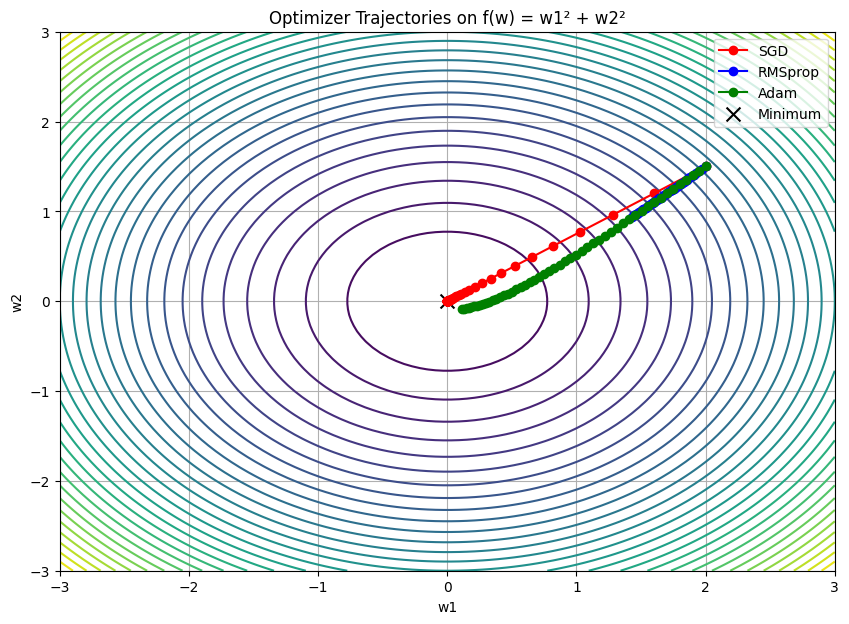
78. plt.xlabel("w1")

79. plt.ylabel("w2")

80. plt.legend()

81. plt.grid(True)

82. plt.show()

****

**Lab-9**

**Objective:** Manually implement convolution and pooling.

**Tasks:**

1. Create a 2D image matrix.
2. Apply convolution with a kernel.

**Results:**

1. import numpy as np

2.

3. # a. Sample 5x5 grayscale image (values from 0 to 255)

4. image = np.array(

5. [

6. [10, 50, 80, 60, 20],

7. [30, 100, 200, 150, 30],

8. [50, 120, 255, 180, 40],

9. [20, 90, 160, 140, 20],

10. [10, 60, 70, 50, 10],

11. ]

12. )

13.

14. # b. Example 3x3 kernel (edge detector / sharpening filter)

15. kernel = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])

16.

17.

18. def convolve2d(image, kernel):

19. """Manual 2D convolution without padding"""

20. image\_h, image\_w = image.shape

21. kernel\_h, kernel\_w = kernel.shape

22. output\_h = image\_h - kernel\_h + 1

23. output\_w = image\_w - kernel\_w + 1

24.

25. output = np.zeros((output\_h, output\_w))

26.

27. for i in range(output\_h):

28. for j in range(output\_w):

29. region = image[i : i + kernel\_h, j : j + kernel\_w]

30. output[i, j] = np.sum(region \* kernel)

31.

32. return output

33.

34.

35. # Perform convolution

36. convolved = convolve2d(image, kernel)

37.

38. # Display results

39. print("Original Image:\n", image)

40. print("\nKernel:\n", kernel)

41. print("\nConvolved Output:\n", convolved.astype(int))

**A screen shot of a computer

AI-generated content may be incorrect.**

**Lab-10**

**Objective:** Implement a simple CNN model.

**Tasks:**

1. Use nn.Conv2d for convolution.
2. Train on the MNIST dataset.

**Results:**

1. import matplotlib.pyplot as plt

2. import numpy as np

3. import torch

4. import torch.nn as nn

5. import torch.optim as optim

6. import torchvision

7. import torchvision.transforms as transforms

8. from torch.utils.data import DataLoader

9.

10. # Transform to Tensor and normalize to [0, 1]

11. transform = transforms.Compose(

12. [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]

13. )

14.

15. # Download and load training and test sets

16. train\_set = torchvision.datasets.MNIST(

17. root="./data", train=True, download=True, transform=transform

18. )

19. test\_set = torchvision.datasets.MNIST(

20. root="./data", train=False, download=True, transform=transform

21. )

22.

23. train\_loader = DataLoader(train\_set, batch\_size=64, shuffle=True)

24. test\_loader = DataLoader(test\_set, batch\_size=64, shuffle=False)

25.

26.

27. class SimpleCNN(nn.Module):

28. def \_\_init\_\_(self):

29. super(SimpleCNN, self).\_\_init\_\_()

30. self.conv1 = nn.Conv2d(

31. in\_channels=1, out\_channels=8, kernel\_size=3

32. ) # 1x28x28 -> 8x26x26

33. self.pool = nn.MaxPool2d(kernel\_size=2, stride=2) # 8x26x26 -> 8x13x13

34. self.fc1 = nn.Linear(8 \* 13 \* 13, 10) # Fully connected layer for 10 classes

35.

36. def forward(self, x):

37. x = self.pool(torch.relu(self.conv1(x)))

38. x = x.view(-1, 8 \* 13 \* 13) # Flatten

39. x = self.fc1(x)

40. return x

41.

42.

43. device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

44. model = SimpleCNN().to(device)

45. criterion = nn.CrossEntropyLoss()

46. optimizer = optim.Adam(model.parameters(), lr=0.001)

47.

48. # Training loop

49. for epoch in range(5):

50. running\_loss = 0.0

51. for images, labels in train\_loader:

52. images, labels = images.to(device), labels.to(device)

53.

54. optimizer.zero\_grad()

55. outputs = model(images)

56. loss = criterion(outputs, labels)

57. loss.backward()

58. optimizer.step()

59.

60. running\_loss += loss.item()

61.

62. print(f"Epoch {epoch+1}, Loss: {running\_loss/len(train\_loader):.4f}")

63.

64. correct = 0

65. total = 0

66. model.eval()

67.

68. with torch.no\_grad():

69. for images, labels in test\_loader:

70. images, labels = images.to(device), labels.to(device)

71. outputs = model(images)

72. \_, predicted = torch.max(outputs, 1)

73. total += labels.size(0)

74. correct += (predicted == labels).sum().item()

75.

76. accuracy = 100 \* correct / total

77. print(f"Test Accuracy: {accuracy:.2f}%")

78.

79.

80. # Helper: Unnormalize for plotting

81. def imshow(img):

82. img = img \* 0.5 + 0.5 # unnormalize from [-1, 1] to [0, 1]

83. npimg = img.numpy()

84. plt.imshow(np.transpose(npimg, (1, 2, 0)), cmap="gray")

85. plt.axis("off")

86.

87.

88. # Get a batch of test images

89. dataiter = iter(test\_loader)

90. images, labels = next(dataiter)

91. images, labels = images.to(device), labels.to(device)

92.

93. # Predict

94. model.eval()

95. with torch.no\_grad():

96. outputs = model(images)

97. \_, predicted = torch.max(outputs, 1)

98.

99. # Show images

100. plt.figure(figsize=(10, 5))

101. imshow(torchvision.utils.make\_grid(images.cpu()[:8], nrow=8))

102. plt.title("Sample Predictions")

103. plt.show()

104.

105. # Print labels below the images

106. print("Ground Truth: ", " ".join(f"{labels[j].item()}" for j in range(8)))

107. print("Predicted: ", " ".join(f"{predicted[j].item()}" for j in range(8)))

**A black rectangle with white text

AI-generated content may be incorrect.**

**A black and white image of numbers

AI-generated content may be incorrect.**

****