DEEP LEARNING AND NEURAL NETWORKS

[AIML302]

**PRACTICAL LAB FILE**

A blue text on a white background

Description automatically generated

In partial fulfilment of the requirements for the award of the degree of

Bachelor of Technology

In

Computer Science & Technology

Department of Computer Science and Engineering

Amity University, Uttar Pradesh

Submitted by: Vindhya Sree Submitted to: Dr. Bedatri Moulik

Enrolment No.: A2305221413 ASET

Class: 7CSE4Y

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.no.** | **Program** | **Date** | **Sign** |
| 1. | To implement House Price Prediction – Example of Regression | 09/08/2024 |  |
| 2. | To implement AND, OR, NOR, NAND, XOR, and XNOR using ANN. | 23/08/2024 |  |
| 3. | To implement gradient descent for regression | 20/09/2024 |  |
| 4. | To implement a CNN model to classify CIFAR-10 dataset | 09/10/2024 |  |
| 5. | To implement sequence prediction using RNN | 18/09/2024 |  |

**EXPERIMENT 1**

**AIM:** To implement the model fw,b for linear regression with one variable (Example of House price prediction)

**THEORY:**

The lab will use a simple dataset with only two points a house with 1000 square feet(sqft) sold for $300,000 and a house with 2000 square feet sold for $500,000. These two points will constitute our data or training set. In this lab, the units of size are 1000 sqft and the units of price are 1000s of dollars.

**CODE AND RESULTS:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error

np.random.seed(42)

X = 2 np.random.rand(100, 1)

y = 4 + 3 X + np.random.randn(100, 1)

Plot the data

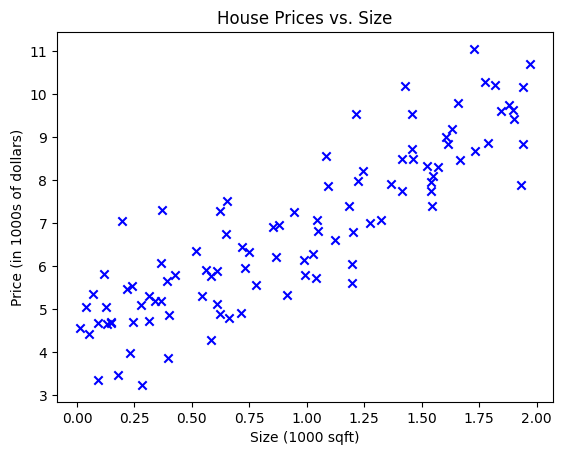
plt.scatter(X, y, c='blue', marker='x')

plt.title("House Prices vs. Size")

plt.xlabel("Size (1000 sqft)")

plt.ylabel("Price (in 1000s of dollars)")

plt.show()



X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

Training the linear regression model

model = LinearRegression()

model.fit(X\_train, y\_train)



Predict on the test set

y\_pred = model.predict(X\_test)

Calculate the mean squared error

mse = mean\_squared\_error(y\_test, y\_pred)

print(f"Mean Squared Error: {mse}")

Print the model parameters

print(f"Intercept: {model.intercept\_}")

print(f"Coefficient: {model.coef\_}")

Mean Squared Error: 0.6536995137170021

Intercept: [4.14291332]

Coefficient: [[2.79932366]]

Plot the model's predictions

plt.scatter(X\_test, y\_test, c='blue', marker='x', label='Actual')

plt.plot(X\_test, y\_pred, c='red', label='Predicted')

plt.title("House Prices vs. Size")

plt.xlabel("Size (1000 sqft)")

plt.ylabel("Price (in 1000s of dollars)")

plt.legend()

plt.show()

A graph of a house price

Description automatically generated

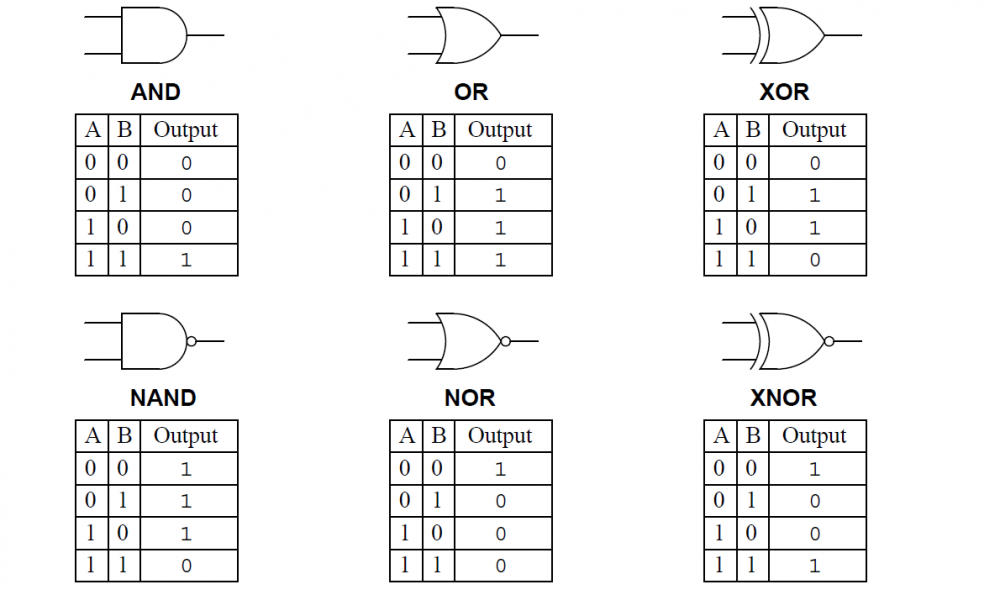
**CONCLUSION:** The regression example of house price prediction has been implemented.

**EXPERIMENT 2**

**AIM:** To implement AND, OR, NOR, NAND, XOR, and XNOR using ANN.

**THEORY:**

The following diagram shows the truth tables and the gates of AND, OR, XOR, XNOR, NAND and NOR logic functions which will be implemented in Python.



**CODE AND RESULTS:**

import numpy as np

def unitStep(v):

return 1 if v >= 0 else 0

def perceptronModel(x, w, b):

v = np.dot(w, x) + b

y = unitStep(v)

return y

# AND Logic Function

w1 = 1, w2 = 1, b = -1.5

def AND(x):

w = np.array([1, 1])

b = -1.5

return perceptronModel(x, w, b)

#OR Logic Function

w1 = 1, w2 = 1, b = -0.5

def OR(x):

w = np.array([1, 1])

b = -0.5

return perceptronModel(x, w, b)

#NOT Logic Function

wNOT = -1, bNOT = 0.5

def NOT(x):

w = np.array([-1])

b = 0.5

return perceptronModel(x, w, b)

# NAND Logic Function

#NAND is NOT of AND

def NAND(x):

output\_AND = AND(x)

return NOT(np.array([output\_AND]))

# NOR Logic Function

#NOR is NOT of OR

def NOR(x):

output\_OR = OR(x)

return NOT(np.array([output\_OR]))

# XOR Logic Function

#XOR is (A AND NOT B) OR (NOT A AND B)

def XOR(x):

y1 = AND(np.array([x[0], NOT(np.array([x[1]]))]))

y2 = AND(np.array([NOT(np.array([x[0]])), x[1]]))

finalOutput = OR(np.array([y1, y2]))

return finalOutput

# XNOR Logic Function

def XNOR(x):

output\_XOR = XOR(x)

return NOT(np.array([output\_XOR]))

# Testing functions for each logic gate

def test\_AND():

tests = [np.array([0, 0]), np.array([0, 1]), np.array([1, 0]), np.array([1, 1])]

print("\nAND Gate Results:")

for t in tests:

print(f"AND({t[0]}, {t[1]}) = {AND(t)}")

def test\_OR():

tests = [np.array([0, 0]), np.array([0, 1]), np.array([1, 0]), np.array([1, 1])]

print("\nOR Gate Results:")

for t in tests:

print(f"OR({t[0]}, {t[1]}) = {OR(t)}")

def test\_NAND():

tests = [np.array([0, 0]), np.array([0, 1]), np.array([1, 0]), np.array([1, 1])]

print("\nNAND Gate Results:")

for t in tests:

print(f"NAND({t[0]}, {t[1]}) = {NAND(t)}")

def test\_NOR():

tests = [np.array([0, 0]), np.array([0, 1]), np.array([1, 0]), np.array([1, 1])]

print("\nNOR Gate Results:")

for t in tests:

print(f"NOR({t[0]}, {t[1]}) = {NOR(t)}")

def test\_XOR():

tests = [np.array([0, 0]), np.array([0, 1]), np.array([1, 0]), np.array([1, 1])]

print("\nXOR Gate Results:")

for t in tests:

print(f"XOR({t[0]}, {t[1]}) = {XOR(t)}")

def test\_XNOR():

tests = [np.array([0, 0]), np.array([0, 1]), np.array([1, 0]), np.array([1, 1])]

print("\nXNOR Gate Results:")

for t in tests:

print(f"XNOR({t[0]}, {t[1]}) = {XNOR(t)}")

test\_AND()

test\_OR()

test\_NAND()

test\_NOR()

test\_XOR()

test\_XNOR()

**CONCLUSION:** Perceptron and its AND, OR, NAND, NOR, NOT have been implemented.

**EXPERIMENT 3**

**AIM:** To implement gradient descent for linear regression and automate the process of optimizing w and b using gradient descent.

**THEORY:**

Gradient descent is an optimization algorithm used to minimize the cost function (also known as the loss function) in machine learning models. In logistic regression, the cost function is based on the difference between the predicted probabilities and the actual labels. The aim is to minimize this difference and find the best parameters that maximize the model’s accuracy.

By using gradient descent to minimize the cost function of a machine learning model, we can find the best set of model parameters for accurate predictions. This means that it helps us find the best values for our model’s parameters so that our model can make accurate predictions.

**CODE AND RESULTS:**

import math, copy

import numpy as np

import matplotlib.pyplot as plt

from lab\_utils\_uni import plt\_house\_x, plt\_contour\_wgrad, plt\_divergence, plt\_gradients

plt.style.use('./deeplearning.mplstyle')

x\_train = np.array([1.0, 2.0]) Input feature

y\_train = np.array([300.0, 500.0]) Target output

def compute\_cost(x, y, w, b):

m = len(x) Number of training examples

total\_cost = 0

Compute the squared errors for each training example

for i in range(m):

y\_pred = w x[i] + b Predicted value

total\_cost += (y\_pred - y[i]) 2 Squared error

Return the average squared error (MSE)

return total\_cost / (2 m)

def gradient\_descent(x, y, w\_in, b\_in, alpha, n\_iterations):

m = len(x) Number of training examples

w = copy.deepcopy(w\_in) Initialize weight

b = b\_in Initialize bias

cost\_history = [] Store the cost at each iteration

for i in range(n\_iterations):

Initialize gradients

dw = 0

db = 0

Compute the gradients for each training example

for j in range(m):

y\_pred = w x[j] + b

dw += (y\_pred - y[j]) x[j] Gradient w.r.t. weight

db += (y\_pred - y[j]) Gradient w.r.t. bias

Average the gradients

dw /= m

db /= m

Update weight and bias using gradient descent rule

w -= alpha dw

b -= alpha db

Calculate and store the cost

cost = compute\_cost(x, y, w, b)

cost\_history.append(cost)

Print progress every 10 iterations

if i % 10 == 0:

print(f"Iteration {i}: Cost = {cost:.4f}, w = {w:.4f}, b = {b:.4f}")

return w, b, cost\_history

Hyperparameters

alpha = 0.01 Learning rate

n\_iterations = 100 Number of iterations

Initial values for weight and bias

w\_init = 0

b\_init = 0

Run gradient descent optimization

w\_opt, b\_opt, cost\_history = gradient\_descent(x\_train, y\_train, w\_init, b\_init, alpha, n\_iterations)

print(f"\nOptimized parameters: w = {w\_opt:.4f}, b = {b\_opt:.4f}")

Plotting cost history over iterations

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

plt.plot(range(n\_iterations), cost\_history, 'b-', label='Cost (MSE)')

plt.xlabel('Iterations')

plt.ylabel('Cost (MSE)')

plt.title('Cost over Iterations (Gradient Descent)')

plt.legend()

plt.grid(True)

plt.show()

Iteration 0: Cost = 79274.8125, w = 6.5000, b = 4.0000

Iteration 10: Cost = 39475.1597, w = 60.4384, b = 37.1642

Iteration 20: Cost = 19660.2806, w = 98.5190, b = 60.5291

Iteration 30: Cost = 9795.0827, w = 125.4104, b = 76.9798

Iteration 40: Cost = 4883.4647, w = 144.4064, b = 88.5522

Iteration 50: Cost = 2438.0524, w = 157.8315, b = 96.6828

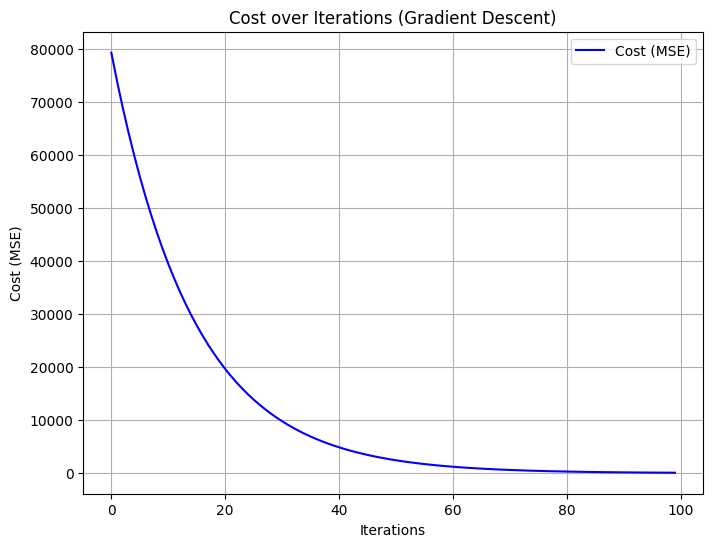
Iteration 60: Cost = 1220.4742, w = 167.3255, b = 102.3851

Iteration 70: Cost = 614.1905, w = 174.0457, b = 106.3742

Iteration 80: Cost = 312.2492, w = 178.8085, b = 109.1548

Iteration 90: Cost = 161.8301, w = 182.1900, b = 111.0829

Optimized parameters: w = 184.3911, b = 112.2986



**CONCLUSION:** The gradient descent for linear regression has been implemented with the plot of cost over iterations.

**EXPERIMENT 4**

**AIM:** To implement the working of CNN (Convolutional Neural Network)

**THEORY:**

CNNs are a class of deep neural networks primarily used for analyzing visual data. They are particularly effective for image recognition and classification tasks. Here's a breakdown of the key components and working principles of CNNs:

1. Convolutional Layers

- Convolution Operation: The core idea of CNNs is the convolution operation, which involves sliding a filter (or kernel) over the input image to produce a feature map. This operation helps in detecting local patterns such as edges, textures, and shapes.

- Filters/Kernels: These are small matrices that are applied to the input image. Different filters can detect different features. For example, one filter might detect horizontal edges, while another might detect vertical edges.

2. Activation Function

- ReLU (Rectified Linear Unit): After the convolution operation, an activation function like ReLU is applied to introduce non-linearity into the model. ReLU replaces all negative pixel values in the feature map with zero, which helps the network learn complex patterns.

3. Pooling Layers

- Max Pooling: This is a down-sampling operation that reduces the dimensionality of the feature map while retaining the most important information. Max pooling takes the maximum value from a patch of the feature map, which helps in making the network invariant to small translations of the input image.

4. Fully Connected Layers

- Flattening: After several convolutional and pooling layers, the high-level reasoning in the neural network is done via fully connected layers. The feature maps are flattened into a single vector, which is then passed through one or more fully connected layers.

- Output Layer: The final layer is typically a softmax layer for classification tasks, which outputs a probability distribution over the possible classes.

Working of CNNs:

1. Input Image: The process starts with an input image, which is passed through a series of convolutional layers.

2. Feature Extraction: Each convolutional layer applies filters to the input image, extracting features such as edges, textures, and shapes.

3. Non-Linearity: The ReLU activation function is applied to introduce non-linearity.

4. Down-Sampling: Pooling layers reduce the spatial dimensions of the feature maps, retaining the most important information.

5. Classification: The feature maps are flattened and passed through fully connected layers, culminating in an output layer that provides the final classification.

**CODE AND RESULTS:**

import tensorflow as tf

from tensorflow.keras import datasets, layers, models

import matplotlib.pyplot as plt

(train\_images, train\_labels), (test\_images, test\_labels) = datasets.cifar10.load\_data()

Normalize pixel values to be between 0 and 1

train\_images, test\_images = train\_images / 255.0, test\_images / 255.0

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step

class\_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

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

for i in range(25):

plt.subplot(5,5,i+1)

plt.xticks([])

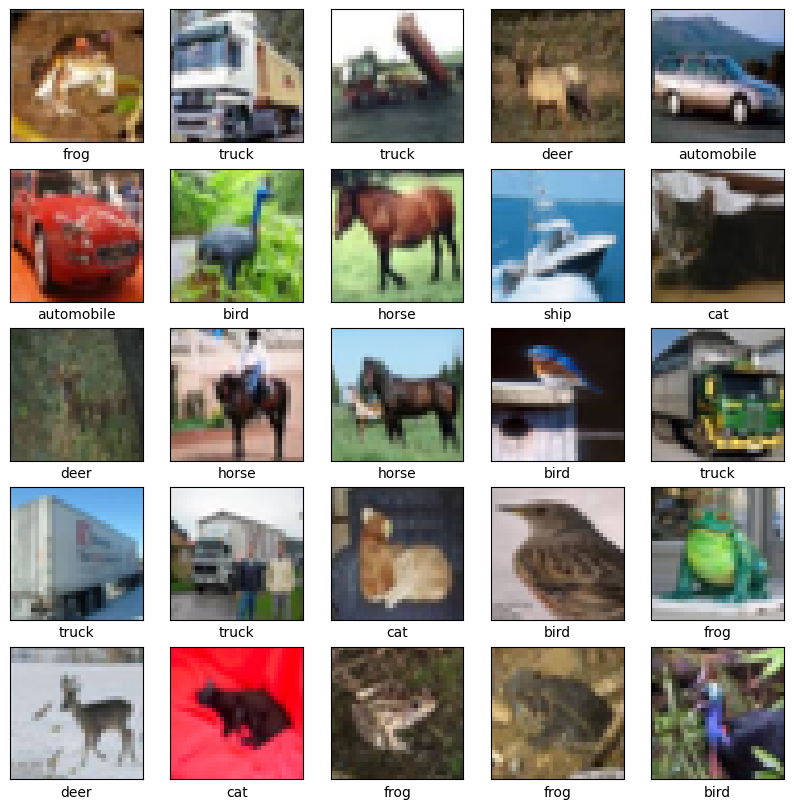
plt.yticks([])

plt.grid(False)

plt.imshow(train\_images[i])

plt.xlabel(class\_names[train\_labels[i][0]])

plt.show()



model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

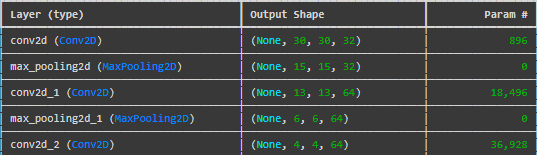
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.summary()

/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base\_conv.py:107: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().\_\_init\_\_(activity\_regularizer=activity\_regularizer, kwargs)

Model: "sequential"



Total params: 56,320 (220.00 KB)

Trainable params: 56,320 (220.00 KB)

Non-trainable params: 0 (0.00 B)

model.add(layers.Flatten())

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(10))

model.summary()

Model: "sequential"

A screenshot of a computer program

Description automatically generated

Total params: 122,570 (478.79 KB)

Trainable params: 122,570 (478.79 KB)

Non-trainable params: 0 (0.00 B)

model.compile(optimizer='adam',loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True), metrics=['accuracy'])

history = model.fit(train\_images, train\_labels, epochs=10, validation\_data=(test\_images, test\_labels))

 Epoch 1/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 76s 47ms/step - accuracy: 0.3561 - loss: 1.7349 - val\_accuracy: 0.5559 - val\_loss: 1.2353

Epoch 2/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 77s 44ms/step - accuracy: 0.5776 - loss: 1.1893 - val\_accuracy: 0.6259 - val\_loss: 1.0671

Epoch 3/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 44ms/step - accuracy: 0.6416 - loss: 1.0113 - val\_accuracy: 0.6538 - val\_loss: 0.9837

Epoch 4/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 44ms/step - accuracy: 0.6824 - loss: 0.9023 - val\_accuracy: 0.6599 - val\_loss: 0.9844

Epoch 5/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 70s 45ms/step - accuracy: 0.7113 - loss: 0.8153 - val\_accuracy: 0.6801 - val\_loss: 0.9160

Epoch 6/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 44ms/step - accuracy: 0.7278 - loss: 0.7656 - val\_accuracy: 0.7030 - val\_loss: 0.8500

Epoch 7/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 84s 45ms/step - accuracy: 0.7512 - loss: 0.7028 - val\_accuracy: 0.6998 - val\_loss: 0.8618

Epoch 8/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 45ms/step - accuracy: 0.7652 - loss: 0.6611 - val\_accuracy: 0.7090 - val\_loss: 0.8577

Epoch 9/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 79s 44ms/step - accuracy: 0.7785 - loss: 0.6321 - val\_accuracy: 0.7137 - val\_loss: 0.8609

Epoch 10/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 44ms/step - accuracy: 0.7965 - loss: 0.5813 - val\_accuracy: 0.7135 - val\_loss: 0.8727

plt.plot(history.history['accuracy'], label='accuracy')

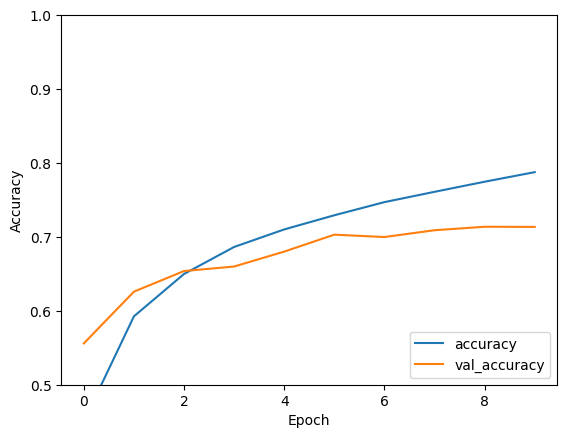
plt.plot(history.history['val\_accuracy'], label = 'val\_accuracy')

plt.xlabel('Epoch') plt.ylabel('Accuracy') plt.ylim([0.5, 1]) plt.legend(loc='lower right')

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels, verbose=2)

print(test\_acc)

313/313 - 3s - 11ms/step - accuracy: 0.7135 - loss: 0.8727



**CONCLUSION:** The working of CNNs has been implemented with CIFAR-10 dataset with 71.35 accuracy.

**EXPERIMENT 5**

**AIM:** To implement sequence prediction using RNNs (Recurrent Neural Networks)

**THEORY:**

1. Sequence prediction: It involves predicting the next item in a sequence based on the previous items. This is particularly useful in various applications such as language modelling, time series forecasting, and speech recognition. Recurrent Neural Networks (RNNs) are well-suited for this task due to their ability to maintain a memory of previous inputs through their recurrent connections.
2. Recurrent Neural Networks (RNNs): RNNs are a type of neural network designed to handle sequential data. Unlike traditional feedforward neural networks, RNNs have connections that form directed cycles, allowing information to persist. The key feature of RNNs is their hidden state, which captures information about previous inputs in the sequence. This hidden state is updated at each time step based on the current input and the previous hidden state.
3. Hidden State: The hidden state is a vector that stores information about the sequence. At each time step ( t ), the hidden state ( h\_t ) is updated using the current input ( x\_t ) and the previous hidden state ( h\_{t-1} ). The update rule can be expressed as:

ht​=σ(Wh​⋅ht−1​+Wx​⋅xt​+b)

where ( W\_h ) and ( W\_x ) are weight matrices, ( b ) is a bias vector, and ( \sigma ) is an activation function (typically tanh or ReLU).

1. Output Layer: The output at each time step is typically a function of the hidden state. For sequence prediction, the output ( y\_t ) can be computed as:

yt​=softmax(Wy​⋅ht​+c)

where ( W\_y ) is a weight matrix and ( c ) is a bias vector. The softmax function is used to produce a probability distribution over the possible next items in the sequence.

1. Training RNNs: RNNs are trained using backpropagation through time (BPTT), which is an extension of the standard backpropagation algorithm. BPTT involves unrolling the RNN through time and computing gradients for each time step. The loss function is typically the categorical cross-entropy loss for classification tasks, which measures the difference between the predicted probability distribution and the true distribution.

**CODE AND RESULTS:**

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import SimpleRNN, Dense

from tensorflow.keras.preprocessing.sequence import pad\_sequences

import matplotlib.pyplot as plt

# Sample text data

text = "hello world"

# Create a character-to-index mapping

chars = sorted(list(set(text)))

char\_to\_index = {char: idx for idx, char in enumerate(chars)}

index\_to\_char = {idx: char for idx, char in enumerate(chars)}

# Convert text to sequences of integers

sequences = [char\_to\_index[char] for char in text]

# Prepare input-output pairs

X = [] y = [] seq\_length = 3

for i in range(len(sequences) - seq\_length):

X.append(sequences[i:i + seq\_length])

y.append(sequences[i + seq\_length])

X = np.array(X)

y = np.array(y)

# Reshape X to be [samples, time steps, features] and normalize

X = np.reshape(X, (X.shape[0], X.shape[1], 1))

X = X / float(len(chars))

# One-hot encode the output

y = tf.keras.utils.to\_categorical(y, num\_classes=len(chars))

# Define and compile the RNN model

model = Sequential([SimpleRNN(50, input\_shape=(seq\_length, 1)), Dense(len(chars), activation='softmax')])

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Train the model and store the training history

history = model.fit(X, y, epochs=200, verbose=1)

Output:

Epoch 1/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.0000e+00 - loss: 2.0997

Epoch 2/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.0000e+00 - loss: 2.0782

Epoch 3/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.1250 - loss: 2.0573

Epoch 4/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - accuracy: 0.2500 - loss: 2.0370

Epoch 5/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.2500 - loss: 2.0171

.

.

Epoch 195/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - accuracy: 0.8750 - loss: 0.5795

Epoch 196/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.8750 - loss: 0.5731

Epoch 197/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.8750 - loss: 0.5667

Epoch 198/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.8750 - loss: 0.5602

Epoch 199/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 0.8750 - loss: 0.5537

Epoch 200/200

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - accuracy: 0.8750 - loss: 0.5471

# Function to predict the next character

def predict\_next\_char(model, input\_text, char\_to\_index, index\_to\_char, seq\_length):

input\_seq = [char\_to\_index[char] for char in input\_text]

input\_seq = pad\_sequences([input\_seq], maxlen=seq\_length, truncating='pre')

input\_seq = np.reshape(input\_seq, (1, seq\_length, 1))

input\_seq = input\_seq / float(len(chars))

predicted\_index = np.argmax(model.predict(input\_seq, verbose=0))

return index\_to\_char[predicted\_index]

# Test the prediction

input\_text = "hel"

predicted\_char = predict\_next\_char(model, input\_text, char\_to\_index, index\_to\_char, seq\_length)

print(f"Input: {input\_text}, Predicted next character: {predicted\_char}")

Output:

Input: hel, Predicted next character: o

# Plot training loss and accuracy

plt.figure(figsize=(12, 4))

# Plot loss

plt.subplot(1, 2, 1)

plt.plot(history.history['loss'], label='Loss')

plt.title('Training Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

# Plot accuracy

plt.subplot(1, 2, 2)

plt.plot(history.history['accuracy'], label='Accuracy')

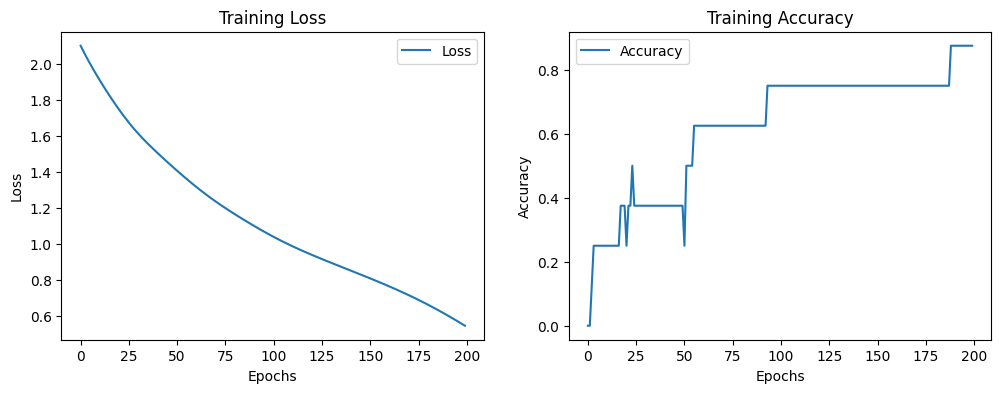
plt.title('Training Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()



**CONCLUSION:** The sequence prediction using RNNs has been implemented.