Artificial Neural Networks(UNIT1)

Artificial Neural Networks (ANNs) are a class of machine learning models inspired by the structure and functioning of the human brain. They are designed to simulate the way biological neurons process information. ANNs have gained immense popularity and success in various fields due to their ability to learn complex patterns and make predictions based on input data.

Key components of an artificial neural network include:

- **Neurons (Nodes):** Neurons are the fundamental units of an ANN. They receive input, perform computations on that input, and produce an output. Neurons are organized into layers: input layer, hidden layers, and an output layer.
- **Connections (Synapses)**: Connections between neurons carry information and have associated weights that determine the strength of the connection. The weights are learned during the training process.
- **Activation Functions:** Activation functions introduce non-linearity into the network, allowing it to learn complex relationships in the data. Common activation functions include the sigmoid, tanh, ReLU (Rectified Linear Unit), and their variants.
- **Layers:** ANNs are typically organized into layers: input, hidden, and output layers. The input layer receives raw data, the hidden layers process and transform the data, and the output layer produces the final prediction or output.
- **Feedforward Propagation:** During the forward pass, input data is passed through the network layer by layer, and computations are performed using the weights and activation functions to produce an output.
- **Backpropagation**: Backpropagation is a learning algorithm used to train ANNs. It involves calculating the error between the predicted output and the actual target, propagating this error backward through the network, and adjusting the weights using gradient descent to minimize the error.
- **Loss Function:** The loss function quantifies the difference between predicted outputs and actual targets. During training, the goal is to minimize this loss by adjusting the network's parameters (weights and biases).
- **Optimization Algorithms:** Gradient descent optimization algorithms, such as stochastic gradient descent (SGD) and its variants (e.g., Adam, RMSprop), are used to update the weights of the network during training.
- **Architecture**: The architecture of an ANN refers to the arrangement of layers, the number of neurons in each layer, and their connections. The choice of architecture depends on the nature of the problem being solved.
- **Deep Learning**: Deep Neural Networks (DNNs) refer to ANNs with multiple hidden layers. Deep learning has shown remarkable success in tasks like image and speech recognition, natural language processing, and more.
- **Regularization Techniques:** Techniques like dropout and L2 regularization help prevent overfitting, where the network learns to perform well on the training data but fails to generalize to new, unseen data.

Artificial Neural Networks have been applied to a wide range of tasks, including image and speech recognition, natural language processing, autonomous driving, game playing, drug discovery, and more. They are a foundational concept in the field of machine learning and have paved the way for more advanced architectures like Convolutional Neural Networks (CNNs) for images and Recurrent Neural Networks (RNNs) for sequences.

Linear Algebra- Creating matrices add vectors using NumPy

Linear algebra is a fundamental mathematical concept that plays a crucial role in machine learning and neural networks. In Python, the NumPy library provides powerful tools for working with matrices, vectors, and various linear algebra operations. Here's how you can create matrices, add vectors, and perform basic operations using NumPy:

1. Installing NumPy:

If you haven't already, you can install NumPy using the following command in your terminal or command prompt:

```
➢ pip install numpy
```

2.Creating Matrices and Vectors:

To create matrices and vectors using NumPy, you can use the numpy.array() function. Here's an example:

```
➢ import numpy as np
# Creating a matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Creating a vector
vector = np.array([10, 11, 12])
```

3.Adding Vectors:

You can add two vectors element-wise using NumPy's element-wise addition:

```
vector1 = np.array([1, 2, 3])
vector2 = np.array([4, 5, 6])
sum_vector = vector1 + vector2
print(sum_vector)  # Output: [5 7 9]
```

4.Matrix Addition:

Matrices can also be added element-wise in the same manner as vectors:

```
matrix1 = np.array([[1, 2],
                    [3, 4]])
matrix2 = np.array([[5, 6],
                    [7, 8]])
sum_matrix = matrix1 + matrix2
print(sum_matrix)
# Output:
# [[ 6  8]
#  [10 12]]
```

Keep in mind that for addition to be valid, the dimensions of the matrices or vectors must match. For example, to add two matrices, they must have the same number of rows and columns.

NumPy provides a wide range of functions for performing various other linear algebra operations, such as matrix multiplication, dot product, eigenvalue decomposition, and more. It's a powerful library that forms the backbone of numerical computations in Python, especially in machine learning and scientific computing.

Remember to consult the official NumPy documentation for more in-depth information and examples: NumPy Documentation.

implementation of operations on matrices-addition, subtraction, multiplication, transpose, inverse, determinant

Python:

```
import numpy as np

# Creating matrices
matrix_a = np.array([[1, 2, 3],
             [4, 5, 6],
             [7, 8, 9]])

matrix_b = np.array([[9, 8, 7],
             [6, 5, 4],
             [3, 2, 1]])

# Matrix Addition
matrix_sum = matrix_a + matrix_b
print("Matrix Addition:")
print(matrix_sum)

# Matrix Subtraction
matrix_diff = matrix_a - matrix_b
print("Matrix Subtraction:")
print(matrix_diff)

# Matrix Multiplication
matrix_prod = np.dot(matrix_a, matrix_b)
print("Matrix Multiplication:")
print(matrix_prod)
```

```
# Matrix Transpose
matrix_a_transpose = np.transpose(matrix_a)
print("Matrix A Transpose:")
print(matrix_a_transpose)

# Matrix Inverse
matrix_a_inverse = np.linalg.inv(matrix_a)
print("Matrix A Inverse:")
print(matrix_a_inverse)

# Matrix Determinant
matrix_a_determinant = np.linalg.det(matrix_a)
print("Matrix A Determinant:")
print(matrix_a_determinant)
```

Remember to handle cases where matrix multiplication or inversion might not be possible due to matrix dimensions or singularity. Also, keep in mind that not all matrices have inverses or determinants.

Vectors- addition, subtraction, dot product, various norms.

Python:

```
import numpy as np

# Creating vectors
vector_a = np.array([1, 2, 3])
vector_b = np.array([4, 5, 6])

# Vector Addition
vector_sum = vector_a + vector_b
print("Vector Addition:")
print(vector_sum)

# Vector Subtraction
vector_diff = vector_a - vector_b
print("Vector Subtraction:")
print(vector_diff)

# Dot Product
dot_product = np.dot(vector_a, vector_b)
print("Dot Product:")
print(dot_product)

# L2 Norm (Euclidean Norm)
l2_norm_a = np.linalg.norm(vector_a)
l2_norm_b = np.linalg.norm(vector_b)
print("L2 Norm (Euclidean Norm) of Vector A:", l2_norm_a)
print("L2 Norm (Euclidean Norm) of Vector B:", l2_norm_b)

# L1 Norm (Manhattan Norm)
l1_norm_a = np.linalg.norm(vector_a, ord=1)
l1_norm_b = np.linalg.norm(vector_b, ord=1)
print("L1 Norm (Manhattan Norm) of Vector A:", l1_norm_a)
print("L1 Norm (Manhattan Norm) of Vector B:", l1_norm_b)

# Infinity Norm
inf_norm_a = np.linalg.norm(vector_a, ord=np.inf)
inf_norm_b = np.linalg.norm(vector_b, ord=np.inf)
```

```
print("Infinity Norm of Vector A:", inf_norm_a)
print("Infinity Norm of Vector B:", inf_norm_b)
```

In this code, we perform the following vector operations:
- Vector Addition: Element-wise addition of two vectors.
- Vector Subtraction: Element-wise subtraction of one vector from another.
- Dot Product: Calculation of the dot product (also known as the inner product) of two vectors.
- L2 Norm (Euclidean Norm): Calculation of the Euclidean norm (length) of a vector.
- L1 Norm (Manhattan Norm): Calculation of the Manhattan norm (sum of absolute values) of a vector.
- Infinity Norm: Calculation of the maximum absolute value in a vector.

The np.linalg.norm() function is used to compute various norms. The ord parameter allows you to specify the order of the norm (e.g., ord=2 for L2 norm, ord=1 for L1 norm, and ord=np.inf for infinity norm).

Linear transformations, pre-processing data using pandas. Scikit Learn-data processing, creating model using scikit-learn

Linear Transformations:
Linear transformations are operations that preserve the structure of vectors and can be represented by matrices. In the context of machine learning, they are used for various purposes such as feature scaling, rotation, and projection.

Here's an example of a simple linear transformation using NumPy:

```
import numpy as np
# Creating a matrix for transformation
transformation_matrix = np.array([[2, 0],
                                  [0, 0.5]])
# Original vector
original_vector = np.array([3, 4])

# Applying the linear transformation
transformed_vector = np.dot(transformation_matrix, original_vector)
print("Original Vector:", original_vector)
print("Transformed Vector:", transformed_vector)
```

Data Preprocessing using Pandas:
Pandas is a popular library for data manipulation and analysis. It's often used to preprocess and clean data before feeding it into machine learning models.

Here's an example of basic data preprocessing using Pandas:

```
import pandas as pd
# Creating a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 22],
        'Salary': [50000, 60000, 45000]}

df = pd.DataFrame(data)

# Removing duplicates
df = df.drop_duplicates()

# Handling missing values
df = df.dropna()

# Encoding categorical variables
df = pd.get_dummies(df, columns=['Name'])
```

```
print(df)
```

Data Processing and Model Creation using Scikit-Learn:
Scikit-Learn is a widely used library for machine learning tasks. It provides tools for preprocessing data, building models, and evaluating them.

Here's an example of how you might preprocess data and create a simple linear regression model using Scikit-Learn:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Load dataset
data = pd.read_csv('data.csv')

# Preprocessing
X = data.drop('target_column', axis=1)
y = data['target_column']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Model creation
model = LinearRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluation
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

In the above example, replace 'data.csv' with the path to your dataset file, and 'target_column' with the actual column name you're trying to predict.