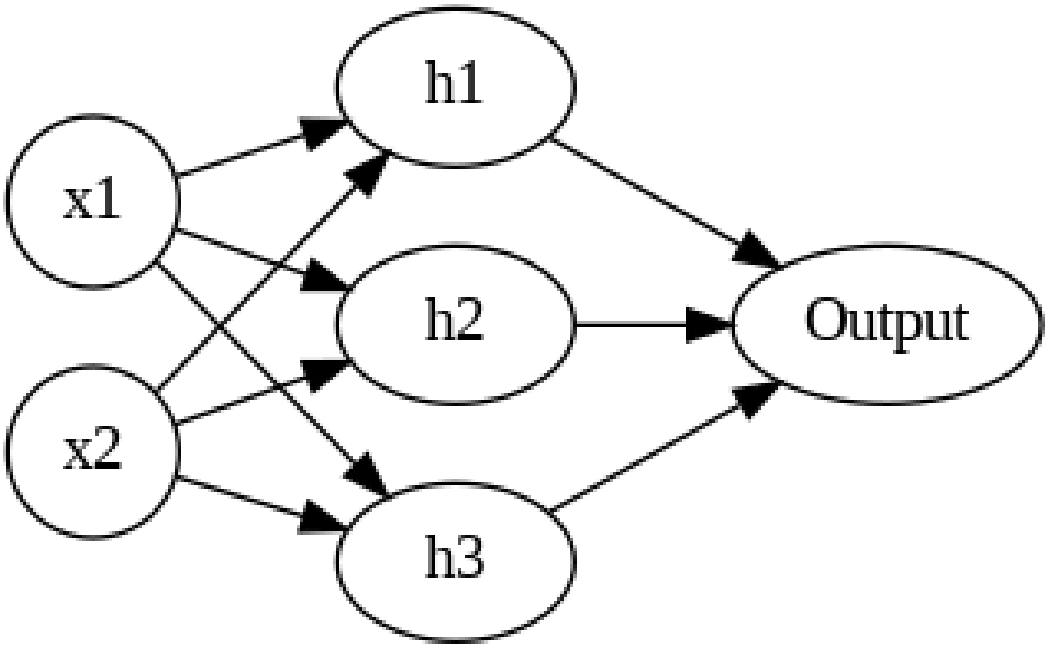
Practical-5

Aim: Write a python program to create a simple 3-layer neural network for implementation of binary function.

- 3 Layers: Input Layer, 1 Hidden Layer, 1 Output Layer
- Create neural_network class
- For input of Boolean function: use two input and 3 neurons in hidden layer
- Boolean Function: XOR
- Use Sigmoid function as activation function in all neurons
- Choose appropriate error function as Loss Function and compare
- Initialize weight and bias randomly in range of (-1, 1)
- Use Backpropagation algorithm to train neural network
- Don’t use any python package except numpy
- Print all parameters (Weight and bias) after training
- Print output of neural network after training for all possible inputs



Artificial Neural Network

Hiten M Sadani

2023-2024

Contents

1 Neural Network Initialization

Algorithm 1 Neural Network Initialization

procedure InitNN
    $n_{\text{in}} \leftarrow$ Input size
    $n_{\text{hid}} \leftarrow$ Hidden layer size
    $n_{\text{out}} \leftarrow 1$
    $W_{\text{in\_hid}} \leftarrow$ Randomly initialized weights
    $b_{\text{hid}} \leftarrow$ Randomly initialized biases
    $W_{\text{hid\_out}} \leftarrow$ Randomly initialized weights
    $b_{\text{out}} \leftarrow$ Randomly initialized biases
end procedure

Variables:

- n_{in}: Dimension of the input layer ($n_{\text{in}} \times 1$ vector).
- n_{hid}: Dimension of the hidden layer ($n_{\text{hid}} \times 1$ vector).
- n_{out}: Dimension of the output layer (fixed to 1 for this problem).
- $W_{\text{in\_hid}}$: Weight matrix connecting input layer to hidden layer ($n_{\text{in}} \times n_{\text{hid}}$ matrix).
- b_{hid}: Bias vector for the hidden layer ($n_{\text{hid}} \times 1$ vector).
- $W_{\text{hid\_out}}$: Weight matrix connecting hidden layer to output layer ($n_{\text{hid}} \times n_{\text{out}}$ matrix).
- b_{out}: Bias vector for the output layer (fixed to 1 for this problem).

Algorithm 2 Sigmoid Activation Function

procedure Sigmoid(x)
    return $1/(1 + e^{-x})$
end procedure

Algorithm 3 Sigmoid Derivative Function

procedure SigmoidDerivative(x)
    return $x \cdot (1 - x)$
end procedure

2 Forward Pass

Algorithm 4 Forward Pass

1: **procedure** FORWARD($\mathbf{X}$)
2: $H_{\text{in}} \leftarrow \mathbf{X} \cdot \mathbf{W}_{\text{in\_hid}} + \mathbf{b}_{\text{hid}}$
3: $H_{\text{out}} \leftarrow \text{Sigmoid}(H_{\text{in}})$
4: $O_{\text{in}} \leftarrow H_{\text{out}} \cdot \mathbf{W}_{\text{hid\_out}} + \mathbf{b}_{\text{out}}$
5: $O_{\text{out}} \leftarrow \text{Sigmoid}(O_{\text{in}})$
6: **return** $O_{\text{out}}, H_{\text{out}}$
7: **end procedure**

Variables:

- $\mathbf{X}$: Input data matrix ($m \times n_{\text{in}}$, where m is the number of samples).
- H_{in}: Input to the hidden layer ($m \times n_{\text{hid}}$ matrix).
- H_{out}: Output of the hidden layer ($m \times n_{\text{hid}}$ matrix).
- O_{in}: Input to the output layer ($m \times n_{\text{out}}$ matrix).
- O_{out}: Output of the output layer ($m \times n_{\text{out}}$ matrix).

3 Loss Functions

Loss functions are essential components of machine learning algorithms, particularly in supervised learning tasks such as regression and classification. They quantify the discrepancy between the predicted outputs of a model and the true targets in the training data. The choice of an appropriate loss function depends on the nature of the problem being solved and the desired properties of the model.

In regression tasks, where the goal is to predict continuous values, common loss functions include the Mean Squared Error (MSE), Mean Absolute Error (MAE), and Huber loss. These loss functions measure the difference between the predicted values and the true targets, with varying degrees of sensitivity to outliers.

In classification tasks, where the goal is to assign inputs to discrete categories, common loss functions include the Binary Cross Entropy and Categorical Cross Entropy losses. These loss functions compare the predicted class probabilities to the true class labels and penalize deviations accordingly.

The choice of a loss function can impact the performance and behavior of a machine learning model. For instance, some loss functions may be more robust to outliers or may encourage certain desirable properties in the learned model, such as sparsity or smoothness.

Overall, loss functions play a critical role in training machine learning models by quantifying the error between predictions and true targets and guiding the optimization process to minimize this error.

3.1 Mean Squared Error Loss Function

The Mean Squared Error (MSE) loss function is a common choice for regression problems. It calculates the average of the squared differences between the predicted values and the true values. For a dataset with N samples, let $\mathbf{T} = [t_1, t_2, ..., t_N]$ represent the vector of true target values and $\mathbf{P} = [p_1, p_2, ..., p_N]$ represent the vector of predicted values. Then, the MSE loss is computed as:

$$\text{MSE}(\mathbf{T}, \mathbf{P}) = \frac{1}{N} \sum_{i=1}^{N} (t_i - p_i)^2$$

where t_i is the true target value and p_i is the predicted value for the i-th sample. The MSE loss penalizes larger errors more heavily due to squaring the differences, making it sensitive to outliers. It is differentiable with respect to the predicted values, making it suitable for optimization algorithms like gradient descent.

Variables:

- $\mathbf{T}$: Target values ($N \times 1$ vector).
- $\mathbf{P}$: Predicted values ($N \times 1$ vector).
- N: Number of data points.

Algorithm 5 Mean Squared Error Loss Calculation

procedure MeanSquaredErrorLoss($\mathbf{T}, \mathbf{P}$)
    return $\frac{1}{N}\sum_{i=1}^{N}(\mathbf{T}_i - \mathbf{P}_i)^2$
end procedure

3.2 Mean Squared Error Loss Derivative

The derivative of the Mean Squared Error (MSE) loss function with respect to the predicted values is used in gradient-based optimization algorithms such as gradient descent. Given a dataset with N samples, let $\mathbf{T} = [t_1, t_2, ..., t_N]$ represent the vector of true target values and $\mathbf{P} = [p_1, p_2, ..., p_N]$ represent the vector of predicted values. Then, the derivative of MSE loss with respect to the predicted values is computed as:

$$\frac{\partial \text{MSE}}{\partial \mathbf{P}} = \frac{2}{N}(\mathbf{P} - \mathbf{T})$$

where $\frac{\partial \text{MSE}}{\partial \mathbf{P}}$ is the derivative of the MSE loss with respect to the predicted values, $\mathbf{P}$ is the vector of predicted values, and $\mathbf{T}$ is the vector of true target values. This derivative provides the direction and magnitude of the update for the predicted values during the optimization process, aiming to minimize the MSE loss.

Algorithm 6 Mean Squared Error Loss Derivative

procedure MeanSquaredErrorLossDerivative($\mathbf{P}, \mathbf{T}$)
    return $\frac{2}{N}(\mathbf{P} - \mathbf{T})$
end procedure

3.3 Cross Entropy Loss Calculation

The Cross Entropy Loss function is commonly used in classification problems, particularly in scenarios with binary classification or multiple classes. For binary classification, let $\mathbf{P} = [p_1, p_2, ..., p_N]$ represent the vector of predicted probabilities for the positive class (often referred to as the logits) and $\mathbf{T} = [t_1, t_2, ..., t_N]$ represent the vector of true binary labels (0 or 1). Then, the Cross Entropy Loss is computed as:

$$\text{CrossEntropy}(\mathbf{P}, \mathbf{T}) = -\frac{1}{N}\sum_{i=1}^{N}(t_i \cdot \log(p_i) + (1 - t_i) \cdot \log(1 - p_i))$$

where N is the number of samples, t_i is the true label for the i-th sample (0 or 1), and p_i is the predicted probability of the positive class for the i-th sample. This loss function penalizes the model based on the difference between predicted and true probabilities, with higher penalties for larger differences. It

is often used in combination with sigmoid activation for binary classification and softmax activation for multi-class classification.

Algorithm 7 Cross Entropy Loss Calculation

procedure CrossEntropyLoss($\mathbf{P}, \mathbf{T}$)
    $\epsilon \leftarrow 10^{-15}$ ▷ To prevent log(0)
    $\mathbf{P} \leftarrow \max(\min(\mathbf{P}, 1-\epsilon), \epsilon)$ ▷ Clip values to avoid log(0)
    return $-\frac{1}{N}\sum_{i=1}^{N}(\mathbf{T}_i \cdot \log(\mathbf{P}_i) + (1-\mathbf{T}_i) \cdot \log(1-\mathbf{P}_i))$
end procedure

3.4 Cross Entropy Loss Derivative

The derivative of the Cross Entropy Loss function with respect to the predicted probabilities is essential for optimizing models in classification tasks. For binary classification, let $\mathbf{P} = [p_1, p_2, ..., p_N]$ represent the vector of predicted probabilities for the positive class (logits) and $\mathbf{T} = [t_1, t_2, ..., t_N]$ represent the vector of true binary labels (0 or 1). Then, the derivative of Cross Entropy Loss with respect to the predicted probabilities is computed as:

$$\frac{\partial \text{CrossEntropy}}{\partial \mathbf{P}} = \frac{1}{N}\left(\frac{\mathbf{T}}{\mathbf{P}} - \frac{1-\mathbf{T}}{1-\mathbf{P}}\right)$$

where $\frac{\partial \text{CrossEntropy}}{\partial \mathbf{P}}$ is the derivative of the Cross Entropy Loss with respect to the predicted probabilities, $\mathbf{T}$ is the vector of true binary labels, $\mathbf{P}$ is the vector of predicted probabilities, and N is the number of samples. This derivative provides the direction and magnitude of the update for the predicted probabilities during the optimization process, aiming to minimize the Cross Entropy Loss.

Algorithm 8 Cross Entropy Loss Derivative

procedure CrossEntropyDerivative($\mathbf{P}, \mathbf{T}$)
    $\epsilon \leftarrow 10^{-15}$ ▷ To prevent division by zero
    $\mathbf{T} \leftarrow \max(\min(\mathbf{T}, 1-\epsilon), \epsilon)$ ▷ Clip values to avoid division by zero
    return $\frac{\mathbf{P}-\mathbf{T}}{\mathbf{P}\cdot(1-\mathbf{P})}$
end procedure

Variables:

- $\mathbf{P}$: Predicted probabilities.
- $\mathbf{T}$: True labels.
- ϵ: A small value to prevent division by zero.

4 Training the Neural Network

Training a neural network involves iteratively updating the model parameters (weights and biases) to minimize a chosen loss function. This process typically involves the following steps:

1. **Forward Pass**: During each training iteration, the forward pass computes the predicted outputs of the neural network for the given input data. This involves propagating the input data through the network layers, applying activation functions, and computing the output predictions. The forward pass also computes any intermediate activations required for subsequent steps.

2. **Loss Calculation**: After the forward pass, the loss function is calculated to quantify the discrepancy between the predicted outputs and the true targets. The choice of loss function depends on the nature of the problem being solved (e.g., regression, classification) and can influence the training process.

3. **Backward Pass (Backpropagation)**: The backward pass, also known as backpropagation, computes the gradients of the loss function with respect to the model parameters (weights and biases). This involves propagating the error backward through the network layers, applying the chain rule of calculus to compute gradients, and updating the parameters in the direction that minimizes the loss.

4. **Parameter Update**: Using the gradients computed during the backward pass, the model parameters are updated using an optimization algorithm such as stochastic gradient descent (SGD) or one of its variants. The learning rate, which determines the step size of parameter updates, is an important hyperparameter that influences the convergence and stability of training.

5. **Iteration**: Steps 1-4 are repeated for a fixed number of iterations (epochs) or until a convergence criterion is met, such as reaching a certain level of loss or accuracy on a validation dataset. During each iteration, the model learns from the training data and adjusts its parameters to improve its performance on the task.

The training process continues until the model parameters converge to values that minimize the chosen loss function and achieve satisfactory performance on unseen data. Proper initialization of model parameters, appropriate choice of activation functions, regularization techniques, and hyperparameter tuning are important considerations for successful training of neural networks.

4.1 Backpropagation

Backpropagation is a key algorithm used to train neural networks by computing the gradients of the loss function with respect to the model parameters. It in-

volves propagating the error backward through the network layers and updating the parameters to minimize the loss. Below are the steps of backpropagation with detailed mathematical notation:

1. **Forward Pass**: During the forward pass, the input data $\mathbf{X}$ is propagated through the network layers to compute the predicted outputs $\mathbf{P}$ for a given input. Each layer computes its output as:

$$\mathbf{H} = \sigma(\mathbf{X} \cdot \mathbf{W}_{\text{in\_hid}} + \mathbf{b}_{\text{hid}})$$

$$\mathbf{P} = \sigma(\mathbf{H} \cdot \mathbf{W}_{\text{hid\_out}} + \mathbf{b}_{\text{out}})$$

   where $\mathbf{H}$ is the output of the hidden layer, $\mathbf{W}_{\text{in\_hid}}$ and $\mathbf{W}_{\text{hid\_out}}$ are the weight matrices, $\mathbf{b}_{\text{hid}}$ and $\mathbf{b}_{\text{out}}$ are the bias vectors, and $\sigma(\cdot)$ is the activation function (e.g., sigmoid).

2. **Loss Calculation**: The loss function $L(\mathbf{T}, \mathbf{P})$ is computed based on the predicted outputs $\mathbf{P}$ and the true targets $\mathbf{T}$. The choice of loss function depends on the task (e.g., Mean Squared Error for regression, Cross Entropy for classification).

3. **Backward Pass**: The gradients of the loss function with respect to the model parameters are computed using the chain rule of calculus. Starting from the output layer, the gradients are propagated backward through the network layers to compute the gradients for each parameter. For example, the gradient of the loss with respect to the output layer weights $\mathbf{W}_{\text{hid\_out}}$ is calculated as:

$$\frac{\partial L}{\partial \mathbf{W}_{\text{hid\_out}}} = \frac{\partial L}{\partial \mathbf{P}} \cdot \frac{\partial \mathbf{P}}{\partial \mathbf{W}_{\text{hid\_out}}}$$

   Similarly, gradients for other parameters such as biases and hidden layer weights are calculated.

4. **Parameter Update**: Using the computed gradients, the model parameters (weights and biases) are updated to minimize the loss function. This typically involves using an optimization algorithm such as stochastic gradient descent (SGD) with a specified learning rate η. For example, the update rule for the output layer weights $\mathbf{W}_{\text{hid\_out}}$ is given by:

$$\mathbf{W}_{\text{hid\_out}} \leftarrow \mathbf{W}_{\text{hid\_out}} - \eta \cdot \frac{\partial L}{\partial \mathbf{W}_{\text{hid\_out}}}$$

The backpropagation algorithm enables efficient computation of gradients for training neural networks and is a fundamental component of modern deep learning frameworks.

Variables:

- $\mathbf{X}$: Input data matrix ($N \times n_{\text{in}}$, where N is the number of samples and n_{in} is the input size).

Algorithm 9 Training the Neural Network

procedure TRAINNN($\mathbf{X}, \mathbf{T}, \eta$, epochs)
  for epoch $\leftarrow 1$ **to** epochs **do**
    $\mathbf{O}, \mathbf{H} \leftarrow$ Forward($\mathbf{X}$)
    $\mathrm{E}_{\text{out}} \leftarrow$ LossFunction($\mathbf{T}, \mathbf{O}$)
    if $\mathrm{E}_{\text{out}} \approx 0.0$ **then**
      break
    end if
    $\delta_{\text{out}} \leftarrow$ LossDerivative($\mathbf{O}, \mathbf{T}$) $\cdot$ SigmoidDerivative($\mathbf{O}$)
    $\mathrm{E}_{\text{hid}} \leftarrow \delta_{\text{out}} \cdot \mathbf{W}_{\text{hid\_out}}^{\top}$
    $\delta_{\text{hid}} \leftarrow \mathrm{E}_{\text{hid}} \cdot$ SigmoidDerivative($\mathbf{H}$)
    $\mathbf{W}_{\text{hid\_out}} \leftarrow \mathbf{W}_{\text{hid\_out}} - \eta \cdot \mathbf{H}^{\top} \cdot \delta_{\text{out}}$
    $\mathbf{b}_{\text{out}} \leftarrow \mathbf{b}_{\text{out}} - \eta \cdot \text{sum}(\delta_{\text{out}})$
    $\mathbf{W}_{\text{in\_hid}} \leftarrow \mathbf{W}_{\text{in\_hid}} - \eta \cdot \mathbf{X}^{\top} \cdot \delta_{\text{hid}}$
    $\mathbf{b}_{\text{hid}} \leftarrow \mathbf{b}_{\text{hid}} - \eta \cdot \text{sum}(\delta_{\text{hid}})$
  end for
end procedure

- $\mathbf{T}$: Target output matrix ($N \times 1$ vector).
- η: Learning rate (scalar).
- epochs: Number of training epochs (scalar).
- $\mathbf{O}$: Output predictions ($N \times 1$ vector).
- $\mathbf{H}$: Hidden layer activations ($N \times n_{\text{hid}}$ matrix, where n_{hid} is the hidden size).
- E_{out}: Current loss value (scalar).
- δ_{out}: Error signal for the output layer ($N \times 1$ vector).
- E_{hid}: Error signal for the hidden layer ($N \times n_{\text{hid}}$ matrix).
- δ_{hid}: Error signal for the hidden layer activations ($N \times n_{\text{hid}}$ matrix).

5 Testing Trained Neural Network

Algorithm 10 Testing Trained Neural Network

1: **procedure** PREDICT($\mathbf{X}$)
2: $\quad \mathbf{O}, \mathbf{H} \leftarrow \text{Forward}(\mathbf{X})$
3: $\quad$ **return O**
4: **end procedure**

Variables:

- $\mathbf{X}$: Input data matrix ($N \times D$, where N is the number of samples and D is the number of features).
- $\mathbf{O}$: Output predictions ($N \times 1$ vector).
- $\mathbf{H}$: Hidden layer activations ($N \times M$, where M is the number of neurons in the hidden layer).

6 NumPy Functions Description

6.1 np.random

The `np.random` module in NumPy provides functions for generating random numbers. It includes various distributions such as uniform, normal, binomial, etc., and also allows for setting the seed to ensure reproducibility. For more information, visit: `https://numpy.org/doc/stable/reference/random/index.html`.

6.2 np.exp

The `np.exp` function computes the element-wise exponential of an array. It returns the exponential value of each element in the input array. Mathematically, for an array $\mathbf{X}$, np.exp($\mathbf{X}$) is computed as $e^{\mathbf{X}}$. For more information, visit: `https://numpy.org/doc/stable/reference/generated/numpy.exp.html`.

6.3 np.clip

The `np.clip` function is used to limit the values in an array to a specified range. It takes an array, a minimum value, and a maximum value, and clips (limits) the values of the array to lie within the specified range. Mathematically, for an array $\mathbf{X}$, np.clip($\mathbf{X}, a, b$) ensures that all values in $\mathbf{X}$ are between a and b inclusive. For more information, visit: `https://numpy.org/doc/stable/reference/generated/numpy.clip.html`.

6.4 np.log

The `np.log` function computes the natural logarithm of the elements of an array. It returns an array where each element is the natural logarithm of the corresponding element in the input array. Mathematically, for an array $\mathbf{X}$, np.log($\mathbf{X}$) is computed as $\log(\mathbf{X})$. For more information, visit: `https://numpy.org/doc/stable/reference/generated/numpy.log.html`.

6.5 np.dot

The `np.dot` function computes the dot product of two arrays. For 1-D arrays, it computes the inner product. For 2-D arrays, it performs matrix multiplication. Mathematically, for arrays $\mathbf{A}$ and $\mathbf{B}$, np.dot($\mathbf{A}, \mathbf{B}$) is computed as $\mathbf{A} \cdot \mathbf{B}$. For more information, visit: `https://numpy.org/doc/stable/reference/generated/numpy.dot.html`.

6.6 np.sum

The `np.sum` function computes the sum of array elements over a specified axis. It returns the sum of array elements along the specified axis or axes. Mathematically, for an array $\mathbf{X}$, np.sum($\mathbf{X}$) is computed as $\sum_{\text{axis}} \mathbf{X}$. For more

information, visit: `https://numpy.org/doc/stable/reference/generated/numpy.sum.html`.

6.7 `np.where`

The `np.where` function returns elements chosen from two arrays depending on a condition. It takes a condition and two arrays as input and returns an array with elements from the second array where the condition is True, and elements from the third array where the condition is False. For more information, visit: `https://numpy.org/doc/stable/reference/generated/numpy.where.html`.

6.8 `transpose`

The `transpose` function rearranges the dimensions of an array. It returns a view of the array with the dimensions permuted as specified. For more information, visit: `https://numpy.org/doc/stable/reference/generated/numpy.transpose.html`.

6.9 `flatten`

The `flatten` method returns a copy of the array collapsed into one dimension. It returns a new one-dimensional array containing all the elements of the original array. For more information, visit: `https://numpy.org/doc/stable/reference/generated/numpy.ndarray.flatten.html`.

6.10 ndarray

The `ndarray` (n-dimensional array) is the fundamental data structure in NumPy, providing support for multi-dimensional arrays and mathematical operations on them. It represents a homogeneous collection of elements with a fixed size, stored in contiguous memory blocks. The `ndarray` object is highly efficient for numerical computations, offering a wide range of functionalities for array manipulation, indexing, slicing, and broadcasting.

Key features of the `ndarray` include its ability to handle multi-dimensional data, flexible indexing and slicing operations, and efficient element-wise arithmetic operations. It also supports various data types (e.g., integers, floats, complex numbers) and provides functions for creating, reshaping, and transforming arrays.

The `ndarray` serves as the foundation for many advanced features in NumPy, such as universal functions (ufuncs), linear algebra operations, and statistical functions. Understanding its usage and capabilities is essential for effectively utilizing NumPy's powerful array computing capabilities.

For more information about the `ndarray` object, refer to the NumPy documentation: `https://numpy.org/doc/stable/reference/arrays.ndarray.html`

6.11 Array Objects

NumPy provides a comprehensive set of array objects that extend the functionality of the `ndarray` and offer specialized features for specific use cases. These array objects include masked arrays, character arrays, and structured arrays, among others.

- **Masked Arrays**: Masked arrays extend the `ndarray` by allowing elements to be masked (i.e., marked as invalid or missing) based on certain conditions. They are useful for handling missing data or representing irregular grids in scientific computing and data analysis.

- **Character Arrays**: Character arrays provide support for storing and manipulating strings within NumPy arrays. They offer efficient storage and operations on string data, such as indexing, slicing, and string manipulation functions.

- **Structured Arrays**: Structured arrays enable the creation of arrays with heterogeneous data types, allowing each element to contain multiple fields or attributes. They are commonly used for representing structured data, such as tables with named columns in data analysis.

These array objects complement the functionality of the `ndarray` and provide specialized solutions for various data processing tasks. Understanding their usage and advantages can help in efficiently handling diverse data types and structures in NumPy.

For more information about array objects in NumPy, refer to the NumPy documentation: `https://numpy.org/doc/stable/reference/arrays.html`