DeeprAI Docs

Official DeeprAI documentation.

Getting Started

How to set up your first network

Set up your first Network

With Deepr it is quite easy to start making your own neural network model. This chapter will show you how to make your first neural network using Deepr. The goal of this project will be to sum two floating point numbers, lets get started!

First we have to install DeeprAI, navigate to a comand line and enter as follows:

```
pip install deeprai
```

Now that we have downloaded the package, we will be able to start making our neural network. For the next step make a new python file and write the following:

```
from deeprai import models

network = models.FeedForward()
```

This imports the models folder from Deepr, which is the standard practice for making a model. Then we create the network object which is initializing our FeedForward class. Our next step is to set up our network's dense layers. Because our goal is to sum two floating point numbers, we will have the layer structure of 2,5,1 neurons (1 inputs layer with 2 neurons, 1 hidden layer with 5 neurons , and 1 output layer with 1 neuron). Lets make it!

```
network.add_dense(2)

network.add_dense(5, activation='linear')

network.add_dense(1, activation='linear')
```

Great! we set the activation function to a linear function. The activation function on default will be sigmoid if nothing is specified. If you don't know what that is, don't worry about it. If you do know, there are more advanced options. Before we can train our neural network we need to set up training data it can learn from. Navigate to the top of you file and add these to the import list:

```
# ...
import numpy as np
import random
```

We are importing numpy for its array data-type, and we are importing random to generate random training data. Lets generate some training data. We need two arrays, one to be a 2D list of two floating point numbers, the second will have the summed values. For example Array 1:[[0.2, 0.6],[0.1, 0.3]], Array 2: [[.8], [.4]]. Now that we have that inplace lets build it!, Enter this below the import statements.

```
inputs = np.array([[random.random()/2 for _ in range(2)] for _ in range(5000)])
expected = np.array([[i[0] + i[1]] for i in inputs])
```

Now we have all the data, we have all the things we need to train the network. Lets train it. Navigate back down to the bottom of the file and add this line:

```
network.train_model(train_inputs=inputs,train_targets=expected,test_inputs=inputs,test_targets=expected,
epochs=20)
```

Awsome! We set up connected our inputs and expected values into the neural network, we also set the epochs to 20 (how many times the neural network will run through the training data). One last thing, if we want to test out our network we can ask it a question based on its training data. Lets quiz our network with a simple addition problem and output the results:

```
output = network.run(np.array([.3,.1]))
print(output)
```

Now we can click run and see what happens!

(Note: In PyCharm, in run config check "Emulate terminal in output console" if you want the correct loading animations.)

Here is the full code up to this point:

```
import numpy as np

import random

import deeprai.models as model


inputs = np.array([[random.random()/2 for _ in range(2)] for _ in range(3000)])

expected = np.array([[i[0] + i[1]] for i in inputs])


network = model.FeedForward()


network.add_dense(2)


network.add_dense(5, activation='linear')


network.add_dense(1, activation='linear')


network.train_model(train_inputs=inputs,train_targets=expected,test_inputs=inputs,test_targets=expected,epo
chs=20)


output = network.run(np.array([.3,.1]))
print(output)
```

This was pretty easy, in the next tutorial we will do something a bit more difficult. We will train on the mnist dataset!

Feed Forward Model

Creating a layer

Function Signature

```
def add_dense(
    neurons: int,
    activation: str = 'sigmoid',
    dropout: float = 0.,
    l1_penalty: float = 0.,
    l2_penalty: float = 0.
) -> None:
```

Parameters

- `neurons` (int): The number of neurons in the dense layer. Required parameter.
- `activation` (str, default='sigmoid'): The activation function to use in the dense layer. Valid options are 'sigmoid', 'relu', 'leaky relu' 'tanh', 'softmax', and 'linear'
- `dropout` (float, default=0.): The dropout rate to use in the dense layer. This parameter should be a float between 0 and 1, where 0 means no dropout and 1 means all neurons are dropped.
- `l1_penalty` (float, default=0.): The L1 regularization penalty to use in the dense layer. This parameter should be a float greater than or equal to 0.

`l2_penalty` (float, default=0.): The L2 regularization penalty to use in the dense layer. This parameter should be a float greater than or equal to 0.

Return Value

This function does not return anything. It modifies the `deeprai.models.FeedForward` instance by adding a dense layer with the specified parameters.

Description

The `add_dense` function adds a dense layer to the `deeprai.models.FeedForward` instance. A dense layer is a layer of fully connected neurons where each neuron is connected to every neuron in the previous layer.

If this is the first layer added to the model, then `add_dense` will automatically treat it as an input layer and ignore any arguments other than `neurons`. This is because an input layer does not have an activation function, dropout, or regularization penalties.

If this is not the first layer added to the model, then `add_dense` will add a dense layer with the specified parameters to the model.

Examples

Here is an example of how to use the `add_dense` function:

```
from deeprai.models import FeedForward


model = FeedForward()
model.add_dense(784)
model.add_dense(128, activation='relu', dropout=0.2)
model.add_dense(64, activation='sigmoid', l2_penalty=0.001)
```

This code creates a `FeedForward` model with an input with 784 neurons, adds a dense layer with 128 neurons, a ReLU activation function, and a 20% dropout rate, and then adds another dense layer with 64 neurons, a sigmoid activation function, and an L2 regularization penalty of 0.001.

Configuring Loss/Optimizers

Function Signature

```
def config(
    optimizer: str = 'gradient descent',
    loss: str = 'mean square error'
) -> None:
```

Parameters

- `optimizer` (str, default='gradient descent'): The optimizer to use during training. Currently, `deeprai` is in beta, so the only valid option for optimizer is 'gradient descent'.
- `loss` (str, default='mean square error'): The loss function to use during training. Valid options are 'mean square error', 'categorical cross entropy', and 'mean absolute error'.

Return Value

This function does not return anything. It modifies the `deeprai.models.FeedForward` instance by setting the optimizer and loss function.

Description

The `config` function sets the optimizer and loss function for the deeprai.models.FeedForward instance. While it is not necessary to call this function, if called, it will use the default values of 'gradient descent' for optimizer and 'mean square error' for loss function.

Currently, deeprai is in beta, so the only valid option for optimizer is 'gradient descent'. The loss parameter sets the loss function to use during training. Valid options are 'mean square error', 'categorical cross entropy', and 'mean absolute error'.

Examples

Here's an example of how to use the `config` function:

```
from deeprai.models import FeedForward


model = FeedForward()
model.add_dense(784)
model.add_dense(128, activation='relu')
model.add_dense(64, activation='relu')
model.add_dense(10, activation='softmax')
model.config(optimizer='gradient descent', loss='categorical cross entropy')
```

This code creates a `FeedForward` model with an input shape of `(784,)`, adds three dense layers with ReLU and softmax activation functions, and sets the optimizer to 'gradient descent' and the loss function to 'categorical cross entropy'.

Training a network

Function Signature

```
def train_model(
    train_inputs: np.ndarray,
    train_targets: np.ndarray,
    test_inputs: np.ndarray,
    test_targets: np.ndarray,
    batch_size: int = 36,
    epochs: int = 500,
    learning_rate: float = 0.1,
    momentum: float = 0.6,
    early_stop: bool = False,
    verbose: bool = True
) -> None:
```

Parameters

- `train_inputs` (np.ndarray): The input data to use for training. This should be a numpy array of shape `(num_samples, input_shape)`.
- `train_targets` (np.ndarray): The target data to use for training. This should be a numpy array of shape `(num_samples, output_shape)`.
- `test_inputs` (np.ndarray): The input data to use for testing. This should be a numpy array of shape `(num_samples, input_shape)`.
- `test_targets` (np.ndarray): The target data to use for testing. This should be a numpy array of shape `(num_samples, output_shape)`.
- `batch_size` (int, default=36): The batch size to use during training.
- `epochs` (int, default=500): The number of epochs to train the model for.
- `learning_rate` (float, default=0.1): The learning rate to use during training.
- `momentum` (float, default=0.6): The momentum to use during training.
- `early_stop` (bool, default=False): Whether to use early stopping during training. If True, the training will stop when the validation loss stops improving.
- `verbose` (bool, default=True): Whether to print training progress during training.

Return Value

This function does not return anything. It trains the `deeprai.models.FeedForward` instance on the given data and saves the updated weights.

The `train_model` function trains the `deeprai.models.FeedForward` instance on the given training data using the specified hyperparameters. It also evaluates the model on the test data after each epoch and prints the training progress if `verbose=True`.

The `batch_size` parameter specifies the batch size to use during training. The `epochs` parameter specifies the number of epochs to train the model for. The `learning_rate` and `momentum` parameters specify the learning rate and momentum to use during training, respectively.

The `early_stop` parameter specifies whether to use early stopping during training. If `early_stop=True`, the training will stop when the validation loss stops improving.

Examples

Here's an example of how to use the `train_model` function:

```
from deeprai.models import FeedForward


model = FeedForward()
model.add_dense(784)
model.add_dense(128, activation='relu')
model.add_dense(64, activation='relu')
model.add_dense(10, activation='sigmoid')


train_inputs = ...
train_targets = ...
test_inputs = ...
test_targets = ...


model.train_model(
    train_inputs=train_inputs,
    train_targets=train_targets,
    test_inputs=test_inputs,
    test_targets=test_targets,
    batch_size=32,
    epochs=1000,
    learning_rate=0.1,
    momentum=0.6,
    early_stop=True,
    verbose=True
)
```

This code creates a `FeedForward` model with an input shape of `(784,)`, adds three dense layers with ReLU and softmax activation functions, sets

Running data through a network

Function Signature

```
def run(
    self,
    inputs: np.ndarray
) -> np.ndarray:
```

Parameters

- `inputs` (np.ndarray): The input data to run through the network. This should be a numpy array of shape `(input_shape,)`.

Return Value

- `output` (np.ndarray): The output of the network after running the given input through it. This should be a numpy array of shape `(output_shape,)`.

Description

The `run` function takes a single input and runs it through the network, returning the output of the network.

The `inputs` parameter should be a numpy array of shape `(input_shape,)`, where `input_shape` is the shape of the input to the network.

The `output` parameter is a numpy array of shape `(output_shape,)`, where `output_shape` is the shape of the output of the network.

Examples

Here's an example of how to use the `run` function:

```
from deeprai.models import FeedForward


model = FeedForward()
model.add_dense(784)
model.add_dense(128, activation='relu')
model.add_dense(64, activation='relu')
model.add_dense(10, activation='softmax')
model.config(loss='categorical cross entropy')


input_data = np.random.rand(784)
output_data = model.run(input_data)
```

This code creates a `FeedForward` model with a single dense layer of size `784`, followed by two additional dense layers with ReLU activation functions, and a final dense layer with a softmax activation function. The `config` function sets the optimizer to `gradient descent` and the loss function to `categorical cross entropy`.

The `run` function takes a single input data of shape `(784,)` and returns the output of the network as a numpy array of shape `(10,)`.

Viewing network information

Function Signature

```
def specs(self) -> str:
```

Return Value

- `output` (str): A string representation of the network information, including the model type, optimizer, parameters, loss function, and DeeprAI version.

Description

The `specs` function returns a string representation of the network information, including the model type, optimizer, parameters, loss function, and DeeprAI version.

Examples

Here's an example of how to use the `specs` function:

```
from deeprai.models import FeedForward

model = FeedForward()
model.add_dense(784)
model.add_dense(128, activation='relu')
model.add_dense(64, activation='relu')
model.add_dense(10, activation='softmax')
model.config(optimizer='gradient descent', loss='mean square error')

model_specs = model.specs()
```

```
print(model_specs)
```

This code creates a `FeedForward` model with a single dense layer of size `784`, followed by two additional dense layers with ReLU activation functions, and a final dense layer with a softmax activation function. The `config` function sets the optimizer to `gradient descent` and the loss function to `mean square error`.

The `specs` function returns a string representation of the network information, including the model type, optimizer, parameters, loss function, and DeeprAI version, which can be printed to the console. The output should look something like this:

```
.---------------.------------------.-----------------.------------------.
|      Key      |       Val        |       Key       |       Val        |
:---------------+------------------+-----------------+------------------:
| Model         | Feed Forward     | Optimizer       | Gradient Descent |
:---------------+------------------+-----------------+------------------:
| Parameters    | 15               | Layer Model     | 2x5x1            |
:---------------+------------------+-----------------+------------------:
| Loss Function | Mean Square Error| DeeprAI Version | 0.0.12 BETA      |
'---------------'------------------'-----------------'------------------'
```

Graphing

Function Signature

```
def graph(self, metric: str = "cost") -> None:
```

Parameters

- `metric` (str, optional): The metric to plot. Default is `"cost"`. Valid metrics are `"cost"`, `"acc"`, or `"accuracy"`, and `"error"`.

Return Value

- None

Description

The `graph` function uses `matplotlib` to plot the change of the specified metric over the epochs. It should be called after training the network.

Examples

Here's an example of how to use the `graph` function:

```
from deeprai.models import FeedForward


model = FeedForward()
model.add_dense(784)
model.add_dense(128, activation='relu')
model.add_dense(64, activation='relu')
model.add_dense(10, activation='softmax')
model.config(optimizer='gradient descent', loss='mean square error')

train_inputs, train_targets, test_inputs, test_targets = # load data

model.train_model(train_inputs, train_targets, test_inputs, test_targets)

model.graph(metric='accuracy')
```

This code creates a `FeedForward` model with a single dense layer of size `784`, followed by two additional dense layers with ReLU activation functions, and a final dense layer with a softmax activation function. The `config` function sets the optimizer to `gradient descent` and the loss function to `mean square error`.

The `train_model` function trains the model on the loaded data. After training, the `graph` function is called with the `"accuracy"` metric to plot the accuracy over the epochs.

The output should be a plot of the specified metric over the epochs.

Saving a Model

Function Signature

`def save(self, file_location: str) -> None:`

Parameters

- `file_location` (str): The location/path where the model's state will be saved.

Return Value

This function does not return anything. It saves the model's state to the specified file location.

Description

The `save` function saves the current state of the `deeprai.models.FeedForward` instance to a file. This allows for easy checkpointing and restoration of trained models. The model's weights, architecture, and configurations are stored at the specified file location. It's recommended to save the model periodically during training to avoid potential data loss.

Examples

Here's an example of how to use the `save` function:

```
from deeprai.models import FeedForward


# ... [Building and training the model] ...
# Saving the model's state to a file
model.save('path/to/save/model.deepr')
```

This code initializes and trains a `FeedForward` model and then saves its state to the file located at `'path/to/save/model.deepr'`.

Loading a Model

Function Signature

`def load(self, file_location: str) -> None:`

Parameters

- `file_location` (str): The location/path from which the model's state will be loaded.

Return Value

This function does not return anything. It loads the model's state from the specified file location.

Description

The `load` function restores the `deeprai.models.FeedForward` instance from a saved state located at a file. This can be used to continue training from a checkpoint, or to deploy pre-trained models without the need to retrain them. The model's weights, architecture, and configurations are loaded from the specified file location. Before using this function, ensure the model architecture is the same as the one saved in the file.

Examples

Here's an example of how to use the `load` function:

```
from deeprai.models import FeedForward


model = FeedForward()
```

```
# Loading the model's state from a file

model.load('path/to/saved/model.deepr')
```

This code initializes a `FeedForward` model and then loads its state from the file located at `'path/to/saved/model.deepr'`.

Activation Functions

How to use the activations in your own project

Loss Functions

Set up loss functions for your own project

Embeddings

Positional Embedding

Function Signature

`def embed_position(sequence: np.ndarray) -> np.ndarray:`

Parameters

- **sequence (np.ndarray)**: A 2D numpy array where the first dimension represents the sequence length and the second represents the embedding dimension.

Return Value

Returns a 2D numpy array with positional embeddings applied to the input sequence.

Description

The `embed_position` function applies positional embedding to a given sequence. The positional embedding is computed using a Cython implementation, which is expected to be faster than a pure Python implementation. The purpose of this embedding is to provide the model with information about the position of elements in the sequence, which can be crucial for certain tasks such as sequence-to-sequence modeling.

Examples

Here's a basic example to demonstrate how to use the `embed_position` function:

```
import numpy as np
from deeprai.embedding.positional_embedding import embed_position


# Create a mock sequence
sequence = np.array([[0.1, 0.2], [0.3, 0.4], [0.5, 0.6]])


# Apply positional embedding
```

```
embedded_sequence = embed_position(sequence)


print(embedded_sequence)
```

Word Vecrotization

Module Import:

`from deeprai.embedding import word_vectorize`

Class Definition:

`class WordVectorizer:`

Initialization:

The WordVectorizer is initialized with an optional corpus, which is used for TF-IDF computations.

`def __init__(self, corpus=None):`

Parameters:

- **corpus (list of str, optional)**: List of words that forms the basis for the term frequency-inverse document frequency (TF-IDF) calculations.

Methods:

1. One-Hot Vectorization:

Converts a given word into a one-hot encoded matrix.

`def one_hot_vectorize(self, word) -> np.ndarray:`

Parameters:

- **word (str)**: The word to vectorize.

Returns:

- **numpy.ndarray**: One-hot encoded matrix representation of the word.

2. Continuous Vectorization:

Encodes a given word into continuous values for each character.

`def continuous_vectorize(self, word) -> np.ndarray:`

Parameters:

- **word (str)**: The word to vectorize.

Returns:

- **numpy.ndarray**: Continuous valued representation of the word.

3. Binary Vectorization:

Converts each character of a word into its binary ASCII representation.

`def binary_vectorize(self, word) -> np.ndarray:`

Parameters:

- **word (str)**: The word to vectorize.

Returns:

- **numpy.ndarray**: Binary ASCII representation of the word.

4. Frequency Vectorization:

Encodes the word based on the frequency of each letter normalized by word length.

`def frequency_vectorize(self, word) -> np.ndarray:`

Parameters:

- **word (str)**: The word to vectorize.

Returns:

- **numpy.ndarray**: Frequency-based representation of the word.

5. N-gram Vectorization:

Vectorizes the word by creating n-grams.

`def ngram_vectorize(self, word, n=2) -> np.ndarray:`

Parameters:

- **word (str)**: The word to vectorize.
- **n (int, default=2)**: The size of the n-grams.

Returns:

- **numpy.ndarray**: N-gram based vector representation of the word.

6. TF-IDF Vectorization:

Vectorizes a word based on term frequency-inverse document frequency.

`def tfidf_vectorize(self, word) -> np.ndarray:`

Parameters:

- **word (str)**: The word to vectorize.

Returns:

- **numpy.ndarray**: TF-IDF representation of the word.

Raises:

- **ValueError**: If the WordVectorizer is not initialized with a corpus.

Description:

The `WordVectorizer` class from the `deeprai.embedding.word_vectorize` module provides multiple ways to represent words as vectors. These include methods like one-hot encoding, continuous encoding, binary encoding, frequency-based encoding, n-gram-based encoding, and TF-IDF encoding. The TF-IDF method requires a corpus to be passed during the initialization of the class.

Examples:

Module Import and Initialization:

First, let's import the necessary module and initialize our WordVectorizer. For methods that require a corpus (like TF-IDF), we'll provide a sample corpus.

```
from deeprai.embedding import word_vectorize


corpus = ["apple", "banana", "cherry", "date", "fig", "grape"]
vectorizer = word_vectorize.WordVectorizer(corpus=corpus)
```

1. One-Hot Vectorization:

This method will transform a word into a matrix where each row is a one-hot encoded representation of a character in the word.

```
word = "apple"
one_hot_encoded = vectorizer.one_hot_vectorize(word)
print(one_hot_encoded)
```

2. Continuous Vectorization:

This method will transform a word into a vector of continuous values.

```
word = "apple"
continuous_vector = vectorizer.continuous_vectorize(word)
print(continuous_vector)
```

3. Binary Vectorization:

This will convert each character of the word into its 8-bit ASCII representation.

```
word = "apple"
binary_vector = vectorizer.binary_vectorize(word)
print(binary_vector)
```

4. Frequency Vectorization:

This method vectorizes a word based on the normalized frequency of each letter in it.

```
word = "apple"
frequency_vector = vectorizer.frequency_vectorize(word)
print(frequency_vector)
```

5. N-gram Vectorization:

This method will break the word into n-grams and vectorize them. For this example, we'll use n=2 (bigrams).

```
word = "apple"
bigram_vector = vectorizer.ngram_vectorize(word, n=2)
print(bigram_vector)
```

6. TF-IDF Vectorization:

This method requires a corpus to compute the inverse document frequency. It will then vectorize a word based on its term frequency and the inverse document frequency from the corpus.

```
word = "apple"
tfidf_vector = vectorizer.tfidf_vectorize(word)
print(tfidf_vector)
```

Tools

Tools for helping create neural networks

Noise

Module: `deeprai.tools.noise`

This module provides a set of classes for introducing different types of noise into numpy arrays, typically used for image data augmentation or robustness testing.

---

1. GaussianNoise Class

Description:

The `GaussianNoise` class applies Gaussian noise to a list of numpy arrays (images).

Attributes:

- **mean** (`float`, default=0): Mean of the Gaussian distribution.
- **std** (`float`, default=1): Standard deviation of the Gaussian distribution.

Methods:

- **compute()**: Internal method to get a function that introduces Gaussian noise to an image.
- **noise(arrays)**: Applies Gaussian noise to a list of numpy arrays. Uses multi-threading for efficiency.

Usage:

```
from deeprai.tools.noise import GaussianNoise


gaussian_noise = GaussianNoise(mean=0, std=25)
```

```
noisy_images = gaussian_noise.noise(list_of_images)
```

---

2. SaltPepperNoise Class

Description:

The `SaltPepperNoise` class introduces salt and pepper noise to a list of numpy arrays.

Attributes:

- **s_vs_p** (`float`, default=0.5): Proportion of salt vs. pepper noise.
- **amount** (`float`, default=0.04): Overall amount of noise to introduce.

Methods:

- **compute()**: Internal method to get a function that introduces salt and pepper noise to an image.
- **noise(arrays)**: Applies salt and pepper noise to a list of numpy arrays. Uses multi-threading for efficiency.

Usage:

```
from deeprai.tools.noise import SaltPepperNoise


sp_noise = SaltPepperNoise(s_vs_p=0.5, amount=0.04)
noisy_images = sp_noise.noise(list_of_images)
```

---

3. SpeckleNoise Class

Description:

The `SpeckleNoise` class introduces speckle noise to a list of numpy arrays.

Methods:

- **compute()**: Internal method to get a function that introduces speckle noise to an image.
- **noise(arrays)**: Applies speckle noise to a list of numpy arrays. Uses multi-threading for efficiency.

Usage:

```
from deeprai.tools.noise import SpeckleNoise


speckle_noise = SpeckleNoise()
noisy_images = speckle_noise.noise(list_of_images)
```

General Note:

For all the above classes, the `noise` method is designed for efficient computation by applying noise to multiple images using multi-threading. Each image in the input list is processed in a separate thread.

The results are then compiled and returned as a list of numpy arrays.

Toolkit

Module: `deeprai.tools.toolkit`

This module provides a collection of utility functions designed for numpy arrays. These functions offer various operations like verification, rounding, normalization, reshaping, and others, enhancing usability and information retrieval from numpy arrays.

---

1. `verify_inputs(array)`

Description:

Verify if the given input is a numpy array.

Parameters:

- **array**: The input to be checked.

Returns:

- **bool**: True if the input is a numpy array, otherwise False.

Example:

```
from deeprai.tools.toolkit import verify_inputs


result = verify_inputs(np.array([1, 2, 3]))
print(result)  # True
```

---

2. `round_out(array, a=2)`

Description:

Round the elements of a numpy array and set specific print options.

Parameters:

- **array** (`np.ndarray`): The input numpy array.
- **a** (`int`, optional): Decimal places to round to. Defaults to 2.

Returns:

- **np.ndarray**: The rounded numpy array.

Example:

```
from deeprai.tools.toolkit import round_out


rounded_array = round_out(np.array([1.12345, 2.6789]))
print(rounded_array)  # [1.12, 2.68]
```

---

3. `normalize(array)`

Description:

Normalize the elements of the numpy array to the range [0, 1].

Parameters:

- **array** (`np.ndarray`): The input array.

Returns:

- **np.ndarray**: The normalized array.

Example:

```
from deeprai.tools.toolkit import normalize


norm_array = normalize(np.array([10, 20, 30, 40]))
print(norm_array)
```

---

4. `reshape_to_2d(array)`

Description:

Reshape the numpy array to a 2D format if it's not already in that shape.

Parameters:

- **array** (`np.ndarray`): The input array.

Returns:

- **np.ndarray**: The reshaped 2D array.

Example:

```
from deeprai.tools.toolkit import reshape_to_2d

reshaped_array = reshape_to_2d(np.array([1, 2, 3, 4]))
print(reshaped_array)
```

---

5. `is_square_matrix(array)`

Description:

Check if the given numpy array is a square matrix.

Parameters:

- **array** (`np.ndarray`): The input array.

Returns:

- **bool**: True if the array is a square matrix, otherwise False.

Example:

```
from deeprai.tools.toolkit import is_square_matrix


result = is_square_matrix(np.array([[1, 2], [3, 4]]))
print(result)  # True
```

---

6. `sum_along_axis(array, axis=0)`

Description:

Compute the sum of elements of the numpy array along a specified axis.

Parameters:

- **array** (`np.ndarray`): The input array.
- **axis** (`int`, optional): Axis along which the sum is computed. Defaults to 0.

Returns:

- **np.ndarray**: The sum along the specified axis.

Example:

```
from deeprai.tools.toolkit import sum_along_axis


summed_array = sum_along_axis(np.array([[1, 2], [3, 4]]))
print(summed_array)  # [4, 6]
```

---

7. `array_info(array)`

Description:

Retrieve essential information about the numpy array.

Parameters:

- **array** (`np.ndarray`): The input array.

Returns:

- **dict**: A dictionary containing shape, data type, minimum and maximum values.

Example:

```
from deeprai.tools.toolkit import array_info


info = array_info(np.array([[1, 2], [3, 4]]))
print(info)
```

Regression

Regression Models

Linear Regression

Module: `deeprai.models.regression.linear_regression`

This module introduces a simple Linear Regression model. Linear Regression is a statistical technique commonly used for modeling and analyzing relationships between two variables.

Class: `LinearRegression`

A class representation of the linear regression model.

---

1. Initializer: `__init__(self)`

Description:

Initializes the `LinearRegression` class.

Attributes:

- **fitted_vals** (`list`): A list to store the results after the model has been fitted. This is primarily the coefficients of the linear equation.

Example:

```
from deeprai.models.regression.linear_regression import LinearRegression


model = LinearRegression()
```

---

2. Method: `fit(self, x_vals, y_vals)`

Description:

Fit the model to the given `x_vals` and `y_vals` using linear regression.

Parameters:

- **x_vals** (`list` or `np.ndarray`): The input values or features.
- **y_vals** (`list` or `np.ndarray`): The output values or labels.

Returns:

- `list`: Coefficients of the linear equation.

Example:

`model.fit(x_vals=[1, 2, 3], y_vals=[2, 4, 6])`

---

3. Method: `run(self, x_val)`

Description:

Use the previously fitted model to predict the output for a given `x_val`.

Parameters:

- **x_val** (`float`): The input value for which the prediction is desired.

Returns:

- `float`: Predicted value based on the linear regression equation.

Example:

```
predicted_val = model.run(4)

print(predicted_val)
```

Poly Regression

Module: `deeprai.models.regression.poly_regression`

Class: `PolyRegression`

A class representation of the polynomial regression model.

---

1. Initializer: `__init__(self)`

Description:

Initializes the `PolyRegression` class.

Attributes:

- **fitted_vals** (`list`): A list to store the results after the model has been fitted. These values represent the coefficients of the polynomial equation.

Example:

```
from deeprai.models.regression import PolyRegression


model = PolyRegression()
```

---

2. Method: `fit(self, x_vals, y_vals)`

Description:

Fit the model to the given `x_vals` and `y_vals` using polynomial regression.

Parameters:

- **x_vals** (`list` or `np.ndarray`): The input values or features.
- **y_vals** (`list` or `np.ndarray`): The output values or labels.

Returns:

- `list`: Coefficients of the polynomial equation, starting from the coefficient of the highest degree term.

Example:

`model.fit(x_vals=[1, 2, 3], y_vals=[2, 5, 10])`

---

3. Method: `run(self, x_val)`

Description:

Use the previously fitted model to predict the output for a given `x_val` based on the polynomial equation.

Parameters:

- **x_val** (`float`): The input value for which the prediction is desired.

Returns:

- `float`: Predicted value based on the polynomial regression equation.

Example:

```
predicted_val = model.run(4)

print(predicted_val)
```

Sine Regression

Module: `deeprai.models.regression.sine_regression`

Class: `SineRegression`

A class representation of the sine regression model.

1. Initializer: `__init__(self)`

Description:

Initializes the `SineRegression` class.

Attributes:

- **fitted_vals** (`list`): A list to store the results after the model has been fitted. These values represent the parameters of the sine equation.

Example:

```
from deeprai.models.regression import SineRegression


model = SineRegression()
```

2. Method: `fit(self, x_vals, y_vals)`

Description:

Fit the model to the given `x_vals` and `y_vals` using sine regression.

Parameters:

- **x_vals** (`list` or `np.ndarray`): The input values or features.
- **y_vals** (`list` or `np.ndarray`): The output values or labels.

Returns:

- `list`: Parameters of the sine equation, which includes amplitude, frequency, phase shift, and vertical shift.

Example:

`model.fit(x_vals=[1, 2, 3], y_vals=[2, 1.5, 2.5])`

---

3. Method: `run(self, x_val)`

Description:

Use the previously fitted model to predict the output for a given `x_val` based on the sine equation.

Parameters:

- **x_val** (`float`): The input value for which the prediction is desired.

Returns:

- `float`: Predicted value based on the sine regression equation.

Example:

```
predicted_val = model.run(4)

print(predicted_val)
```

K-Nearest Neighbors

K-nearest neighbors (KNN) is a supervised machine learning algorithm that classifies a data point based on how its neighbors are classified.

Instant classifier

Instant Classifier Function

Function Signature

```
def instant_classifier(self, x_vals, y_vals, query_point, p=3, k=2):
```

Parameters

- **x_vals**: The input data points.
- **y_vals** (must be converted to int32): The labels corresponding to the input data points.
- **query_point**: The point for which classification is to be determined.
- **p** (int, default=3): The power parameter for the Minkowski distance metric.
- **k** (int, default=2): The number of nearest neighbors to consider for classification.

Return Value

Returns the classification result for the `query_point` based on the `k` nearest neighbors in the `x_vals` dataset.

Description

The `instant_classifier` function classifies a given `query_point` based on the `k` nearest neighbors in the `x_vals` dataset. The labels of the `k` nearest neighbors are then used to determine the classification of the `query_point`.

Examples

```
from deeprai.models import KNN

# Sample data
x_vals = [[1, 2], [2, 3], [3, 4]]
y_vals = [0, 1, 0]
query_point = [2, 2]

# Create an instance of the classifier
classifier = KNN()

# Classify the query_point
result = classifier.instant_classifier(x_vals, y_vals, query_point, p=3, k=2)
print(result)  # This will print the classification result for the query_point
```

Note: Ensure that `y_vals` is converted to `int32` before passing it to the function.

Store Data in KNN

Storing Values in KNN Classifier

Function Signature

```
def store_vals(self, x_values, y_values, p=3, k=2):
```

Parameters

- **x_values**: The input data points to be stored in the classifier.
- **y_values** (must be converted to int32): The labels corresponding to the input data points to be stored in the classifier.
- **p** (int, default=3): The power parameter for the Minkowski distance metric, to be stored for future use.
- **k** (int, default=2): The number of nearest neighbors to consider for classification, to be stored for future use.

Return Value

This function does not return anything. It modifies the KNN instance by storing the provided values.

Description

The `store_vals` function stores the provided data points, labels, power parameter, and number of neighbors in the KNN classifier instance. This allows for the classifier to use these values in subsequent classification tasks without needing them to be provided again.

Examples

```
from deeprai.models import KNN


# Sample data
x_vals = [[1, 2], [2, 3], [3, 4]]
y_vals = [0, 1, 0]


# Create an instance of the classifier
classifier = KNN()


# Store the values in the classifier
classifier.store_vals(x_vals, y_vals, p=3, k=2)
```

Note: Ensure that `y_vals` is converted to `int32` before storing.

Classifying a Query Point

Classifying a Query Point with KNN

Function Signature

```
def classify(self, query_point):
```

Parameters

- **query_point**: The point for which classification is to be determined.

Return Value

Returns the classification result for the `query_point` based on the stored values in the KNN instance.

Description

The `classify` function classifies a given `query_point` based on the stored values in the KNN instance. The distance between the points is calculated using the Minkowski distance metric with the stored power parameter. The labels of the stored data points are then used to determine the classification of the `query_point`.

It's important to note that the `store_vals` function must be called prior to using the `classify` function to ensure that the necessary values are stored in the KNN instance.

Examples

```
from deeprai.models import KNN


# Sample data
x_vals = [[1, 2], [2, 3], [3, 4]]
y_vals = [0, 1, 0]
query_point = [2, 2]


# Create an instance of the classifier
classifier = KNN()


# Store the values in the classifier
classifier.store_vals(x_vals, y_vals, p=3, k=2)


# Classify the query_point
result = classifier.classify(query_point)
print(result)  # This will print the classification result for the query_point
```

Calculating Classification Probability

Calculating Classification Probability with KNN

Function Signature

```
def classify_probability(self, query_point, expected_val):
```

Parameters

- **query_point**: The point for which classification probability is to be determined.
- **expected_val**: The label value for which the probability is to be calculated.

Return Value

Returns the probability (in percentage) that the `query_point` belongs to the class specified by `expected_val` based on the stored values in the KNN instance.

Description

The `classify_probability` function calculates the probability that a given `query_point` belongs to the class specified by `expected_val`. It first retrieves the nearest neighbors of the `query_point` using the `classify_neighbors` function. It then counts how many of these neighbors have the label `expected_val` and calculates the probability based on this count.

It's important to note that the `store_vals` function must be called prior to using the `classify_probability` function to ensure that the necessary values are stored in the KNN instance.

Examples

```
from deeprai.models import KNN


# Sample data
x_vals = [[1, 2], [2, 3], [3, 4]]
y_vals = [0, 1, 0]
query_point = [2, 2]


# Create an instance of the classifier
classifier = KNN()


# Store the values in the classifier
classifier.store_vals(x_vals, y_vals, p=3, k=2)


# Calculate the probability that the query_point belongs to class 1
probability = classifier.classify_probability(query_point, 1)
print(f"The probability that the query point belongs to class 1 is {probability}%")
```

Retrieving Nearest Neighbors

Retrieving Nearest Neighbors with KNN

Function Signature

```
def classify_neighbors(self, query_point):
```

Parameters

- **query_point**: The point for which the nearest neighbors are to be determined.

Return Value

Returns the indices of the `k` nearest neighbors to the `query_point` based on the stored values in the KNN instance.

Description

The `classify_neighbors` function retrieves the indices of the `k` nearest neighbors for a given `query_point` based on the stored values in the KNN instance.

It's important to note that the `store_vals` function must be called prior to using the `classify_neighbors` function to ensure that the necessary values are stored in the KNN instance.

Examples

```
from deeprai.models import KNN


# Sample data
x_vals = [[1, 2], [2, 3], [3, 4]]
y_vals = [0, 1, 0]
query_point = [2, 2]

# Create an instance of the classifier
classifier = KNN()

# Store the values in the classifier
classifier.store_vals(x_vals, y_vals, p=3, k=2)

# Retrieve the nearest neighbors of the query_point
neighbors = classifier.classify_neighbors(query_point)
print(f"The indices of the nearest neighbors to the query point are: {neighbors}")
```

Configuring Distance Metric

Configuring Distance Metric for KNN

Function Signature

```
def config_distance(self, distance):
```

Parameters

- **distance**: The name of the distance metric to be configured for the KNN instance.

Return Value

This function does not return anything. It modifies the KNN instance.

Description

The `config_distance` function sets the distance metric for the KNN instance.

The valid distance metrics that can be passed to this function are:

- "hamming distance"
- "minkowski distance"
- "manhattan distance"

- "euclidean distance"

Examples

```
from deeprai.models import KNN


# Create an instance of the classifier
classifier = KNN()


# Configure the distance metric to be used
classifier.config_distance("euclidean distance")
```