nptsne Documentation

Baldur van Lew

Dec 23, 2021

CONTENTS:

CHAPTER

ONE

INSTALLING AND USING

This release supports 3 python versions (3.6, 3.7 and 3.8) on three platforms: Windows, Ubuntu, Macos

On Windows and Linux acceleration is performed using compute shaders. On Macos, due to the lack of compute shader support, tSNE uses a rasterized implementation with a lower performance.

1.1 Installing

Windows,MacOS,Linux: install from PyPi using: **pip install nptsne**. The PyPi page.

1.2 Demo list

A number of demos have been created to help you exploit the accelerated tSNE and HSNE offered by this package. The demos are available in a single demos.zip file.

Table 1: Demos

Demo	Description
Extended HSNE demo	A complete demonstration including three different datatypes: * Image is datapoint (MNIST) * Pixel is data point (Hyperspectral solar images) * Multi-dimensional plus meta data (Genetic data)
HSNE Louvain Demo	Louvain clustering applied to levels in the HSNE hierarchy
TextureTsne	GPU accelerated t-SNE on 70000 MNIST points
TextureTsneExtended	GPU accelerated t-SNE on 70000 MNIST points with intermediate results
DocTest	Run the internal doctest examples in nptsne Can be used for install verification
Jupyter notebook for GPU accelerated tSNE	A Jupyter notebook demonstration of the tSNE API. Illustrates the following options: * a plain tSNE * a pre-loaded embedding * controlling the iteration when the exaggeration factor is removed.

CHAPTER

TWO

CITATION

When using nptsne please cite [NP2016] (HSNE) and/or [NP2019] (t-SNE)

References

CHAPTER

THREE

NPTSNE API REFERENCE

3.1 Module summary

3.1.1 API Reference

The main API classes are:

t-SNE classes

- *TextureTsne* : linear tSNE simple API
- *TextureTsneExtended* : linear tSNE advanced API wrapper with additional functionality

HSNE classes

- *HSne* : Hierarchical-SNE model builder
- *HSneScale* : Wrapper for a scale in the HSNE model

Full details are in the reference below.

3.1.2 `nptsne`: t-SNE and HSNE data embedding

nptsne.HSne	Initialize an HSne wrapper with logging state.
nptsne.HSneScale	Create a wrapper for the HSNE data scale.
nptsne.TextureTsne	Create a wrapper class for the linear tSNE implementation.
nptsne.TextureTsneExtended	Create an extended functionality wrapper for the linear tSNE implementation.
nptsne.KnnAlgorithm	Enumeration used to select the knn algorithm used. Three possibilities are

A numpy compatible python extension for GPGPU linear complexity t-SNE and HSNE

This package contains classes that wrap linear complexity *t-SNE* and classes to support *HSNE*.

Available subpackages

hsne_analysis Provides classes for selection driven navigation of the HSNE model and mapping back to the original data. The classes are intended to support visual analytics

Notes

`ndarray` types are the preferred parameters types for input and where possible internal data in the wrapped t-SNE [1] and HSNE [2] is returned without a copy in a `ndarray`.

References

class nptsne.**HSne**(*self: nptsne.libs._nptsne.HSne*, *verbose: bool = False*) → None
Bases: `pybind11_builtins.pybind11_object`

Initialize an HSne wrapper with logging state.

Parameters

verbose [bool] Enable verbose logging to standard output, default is False

Notes

HSne is a simple wrapper API for the Hierarchical SNE implementation.

Hierarchical SNE is is a GPU compute shader implementation of Hierarchical Stochastic Neighborhood Embedding described in [1].

The wrapper can be used to create a new or load an existing hSNE analysis. The hSNE analysis is then held in the HSne instance and can be accessed through the class api.

References

[1]

Examples

Create an HSNE wrapper

```
>>> import nptsne
>>> hsne = nptsne.HSne(True)
```

Attributes

num_data_points int: The number of data points in the HSne.

num_dimensions int: The number of dimensions associated with the original data.

num_scales int: The number of scales in the HSne.

create_hsne(**args*, ***kwargs*)
Overloaded function.

1. create_hsne(self: nptsne.libs._nptsne.HSne, X: numpy.ndarray[float32], num_scales: int) -> bool

2. create_hsne(self: nptsne.libs._nptsne.HSne, X: numpy.ndarray[float32], num_scales: int, point_ids: numpy.ndarray[uint64]) -> bool

   Create the hSNE analysis data hierarchy with user assigned point ids from the input data with the number of scales required.

   Parameters

   X [ndarray]

   The data used to create the saved file. Shape is : (num. data points, num. dimensions)

   num_scales [int] How many scales to create in the hsne analysis

   point_ids [ndarray, optional] Array of ids associated with the data points

Examples

```
>>> import nptsne
>>> hsne = nptsne.HSne(True)
>>> hsne.create_hsne(sample_hsne_data, 3)
True
>>> hsne.num_data_points
10000
>>> hsne.num_dimensions
16
>>> hsne.num_scales
3
```

get_scale(*self: nptsne.libs._nptsne.HSne*, *scale_number: int*) → *HSneScale*

Get the scale information at the index. 0 is the HSNE data scale.

Parameters

scale_index [int] Index of the scale to retrieve

Returns

HSneScale A numpy array contain a flatten (1D) embedding

Examples

The number of landmarks in scale 0 is the number of data points.

```
>>> scale = sample_hsne.get_scale(0)
>>> scale.num_points
10000
```

load_hsne(*self: nptsne.libs._nptsne.HSne*, *X: numpy.ndarray[float32]*, *file_path: str*) → bool

Load the HSNE analysis data hierarchy from a pre-existing HSNE file.

Parameters

X [ndarray] The data used to create the saved file. Shape is : (num. data points, num. dimensions)

file_path [str] Path to saved HSNE file

Examples

Load hsne from a file, and check that is contains the expected data

```
>>> import nptsne
>>> import doctest
>>> loaded_hsne = nptsne.HSne(True)
>>> loaded_hsne.load_hsne(sample_hsne_data, sample_hsne_file)
True
>>> loaded_hsne.num_data_points
10000
>>> loaded_hsne.num_dimensions
16
>>> loaded_hsne.num_scales
3
```

static read_num_scales (*file_path: str*) → int

Read the number of scales defined in stored hSNE data without fully loading the file.

Parameters

filename [str] The path to a saved hSNE

Returns

int The number of scales in the saved hierarchy

Examples

Read the number of scales from a saved file

```
>>> import nptsne
>>> nptsne.HSne.read_num_scales(sample_hsne_file)
3
```

save (*self: nptsne.libs._nptsne.HSne*, *file_path: str*) → None

Save the HSNE as a binary structure to a file

Parameters

filename [str] The file to save to. If it already exists it is overwritten.

Examples

Save the hsne to a file and check the number of scales was sved correctly.

```
>>> import nptsne
>>> sample_hsne.save("save_test.hsne")
>>> nptsne.HSne.read_num_scales("save_test.hsne")
3
```

property num_data_points

int: The number of data points in the HSne.

Examples

```
>>> sample_hsne.num_data_points
10000
```

property num_dimensions
int: The number of dimensions associated with the original data.

Examples

```
>>> sample_hsne.num_dimensions
16
```

property num_scales
int: The number of scales in the HSne.

Examples

```
>>> sample_hsne.num_scales
3
```

class nptsne.**HSneScale**(*self: nptsne.libs._nptsne.HSneScale*, *hsne: nptsne.libs._nptsne.HSne*, *scale_number: int*) → None
Bases: pybind11_builtins.pybind11_object

Create a wrapper for the HSNE data scale. The function *HSne.get_scale()* works more directly than calling the constructor on this class.

Parameters

hsne [*HSne*] The hierarchical SNE being explored

scale_number [int] The scale from the nsne to wrap

Examples

Using the initializer to create an HSneScale wrapper. Scale 0 contains the datapoints. (Prefer the HSne.get_scale function)

```
>>> import nptsne
>>> scale = nptsne.HSneScale(sample_hsne, 0)
>>> scale.num_points
10000
```

Attributes

num_points int: The number of landmark points in this scale

transition_matrix The transition (probability) matrix in this scale.

landmark_orig_indexes Original data indexes for each landmark in this scale.

get_landmark_weight(*self: nptsne.libs._nptsne.HSneScale*) → numpy.ndarray[float32]
The weights per landmark in the scale.

Returns

ndarray Weights array in landmark index order

Examples

The size of landmark weights should match the number of points

```
>>> num_points = sample_scale2.num_points
>>> weights = sample_scale2.get_landmark_weight()
>>> weights.shape[0] == num_points
True
```

All weights at scale 0 should be 1.0

```
>>> weights = sample_scale0.get_landmark_weight()
>>> test = weights[0] == 1.0
>>> test.all()
True
```

property landmark_orig_indexes

Original data indexes for each landmark in this scale.

Returns

ndarray: An ndarray of the original data indexes.

Examples

At scale 0 the landmarks are all the data points.

```
>>> sample_scale0.landmark_orig_indexes.shape
(10000,)
>>> sample_scale0.landmark_orig_indexes[0]
0
>>> sample_scale0.landmark_orig_indexes[9999]
9999
```

property num_points

int: The number of landmark points in this scale

Examples

```
>>> sample_scale0.num_points
10000
```

property transition_matrix

The transition (probability) matrix in this scale.

Returns

list(list(tuple)): The transition (probability) matrix in this scale

Notes

The list returned has one entry for each landmark point, each entry is a list The inner list contains tuples where the first item is an integer landmark index in the scale and the second item is the transition matrix value for the two points. The resulting matrix is sparse

Examples

The size of landmark weights should match the number of points

```
>>> num_points = sample_scale2.num_points
>>> matrix = sample_scale2.transition_matrix
>>> len(matrix) == num_points
True
```

class nptsne.**KnnAlgorithm**(*self: nptsne.libs._nptsne.KnnAlgorithm*, *arg0: int*) → None

Bases: pybind11_builtins.pybind11_object

Enumeration used to select the knn algorithm used. Three possibilities are supported:

KnnAlgorithm.Flann: Knn using FLANN - Fast Library for Approximate Nearest Neighbors *KnnAlgorithm.HNSW*: Knn using Hnswlib - fast approximate nearest neighbor search *KnnAlgorithm.Annoy*: Knn using Annoy - Spotify Approximate Nearest Neighbors Oh Yeah

Members:

Flann

HNSW

Annoy

Annoy = KnnAlgorithm.Annoy

Flann = KnnAlgorithm.Flann

HNSW = KnnAlgorithm.HNSW

property name

(self: handle) -> str

class nptsne.**TextureTsne**(*self: nptsne.libs._nptsne.TextureTsne*, *verbose: bool = False*, *iterations: int = 1000*, *num_target_dimensions: int = 2*, *perplexity: int = 30*, *exaggeration_iter: int = 250*, *knn_algorithm: nptsne.libs._nptsne.KnnAlgorithm = KnnAlgorithm.Flann*) → None

Bases: pybind11_builtins.pybind11_object

Create a wrapper class for the linear tSNE implementation.

Parameters

verbose [bool] Enable verbose logging to standard output

iterations [int] The number of iterations to perform. This must be at least 1000.

num_target_dimensions [int] The number of dimensions for the output embedding. Default is 2.

perplexity [int] The tSNE parameter that defines the neighborhood size. Usually between 10 and 30. Default is 30.

exaggeration_iter [int] The iteration when force exaggeration starts to decay.

knn_algorithm [*KnnAlgorithm*] The knn algorithm used for the nearest neighbor calculation. The default is *Flann* for less than 50 dimensions *HNSW* may be faster

See also:

TextureTsneExtended

Notes

TextureTsne is a GPU compute shader implementation of the gradient descent linear tSNE. If the system does not support OpenGL 4.3 an abover the implementation falls back to the a Texture rendering approach as described in [1].

References

[1]

Examples

Create an TextureTsne wrapper

```
>>> import nptsne
>>> tsne = nptsne.TextureTsne(verbose=True)
>>> tsne.verbose
True
>>> tsne.iterations
1000
>>> tsne.num_target_dimensions
2
>>> tsne.perplexity
30
>>> tsne.exaggeration_iter
250
```

fit_transform(*self: nptsne.libs._nptsne.TextureTsne*, *X: numpy.ndarray[float32]*) → numpy.ndarray[float32]

Fit X into an embedded space and return that transformed output.

Parameters

X [ndarray] The input data with shape (num. data points, num. dimensions)

Returns

ndarray A numpy array contain a flatten (1D) embedding

property exaggeration_iter

int: The iteration where attractive force exaggeration starts to decay, set at initialization.

Notes

The gradient of the cost function used to iteratively optimize the embedding points y_i is a sum of an attractive and repulsive force $\frac{\delta C}{\delta y_i} = 4(\phi * F_i^{attr} - F_i^{rep})$ The iterations up to exaggeration_iter increase the F_i^{attr} term by the factor ϕ which then decays to 1.

Examples

```
>>> sample_texture_tsne.exaggeration_iter
250
```

property iterations

int: The number of iterations, set at initialization.

Examples

```
>>> sample_texture_tsne.iterations
1000
```

property num_target_dimensions

int: The number of target dimensions, set at initialization.

Examples

```
>>> sample_texture_tsne.num_target_dimensions
2
```

property perplexity

int: The tsne perplexity, set at initialization.

Examples

```
>>> sample_texture_tsne.perplexity
30
```

property verbose

bool: True if verbose logging is enabled. Set at initialization.

Examples

```
>>> sample_texture_tsne.verbose
False
```

class nptsne.**TextureTsneExtended**(*self: nptsne.libs._nptsne.TextureTsneExtended*, *verbose: bool = False*, *num_target_dimensions: int = 2*, *perplexity: int = 30*, *knn_algorithm: nptsne.libs._nptsne.KnnAlgorithm = KnnAlgorithm.Flann*) → None

Bases: pybind11_builtins.pybind11_object

Create an extended functionality wrapper for the linear tSNE implementation.

Parameters

verbose [bool] Enable verbose logging to standard output, default is False

num_target_dimensions [int] The number of dimensions for the output embedding. Default is 2.

perplexity [int] The tSNE parameter that defines the neighborhood size. Usually between 10 and 30. Default is 30.

knn_algorithm [*KnnAlgorithm*] The knn algorithm used for the nearest neighbor calculation. The default is 'Flann' for less than 50 dimensions 'HNSW' may be faster

See also:

TextureTsne

Notes

TextureTsneExtended offers additional control over the exaggeration decay compares to *TextureTsne*. Additionally it supports inputting an initial embedding. Linear tSNE is described in [1].

References

[1]

Examples

Create an TextureTsneExtended wrapper

```
>>> import nptsne
>>> tsne = nptsne.TextureTsneExtended(verbose=True, num_target_dimensions=2,␣
↪perplexity=35, knn_algorithm=nptsne.KnnAlgorithm.Annoy)
>>> tsne.verbose
True
>>> tsne.num_target_dimensions
2
>>> tsne.perplexity
35
>>> tsne.knn_algorithm
KnnAlgorithm.Annoy
```

Attributes

decay_started_at int: The iteration number when exaggeration decay started.

iteration_count int: The number of completed iterations of tSNE gradient descent.

close(*self: nptsne.libs._nptsne.TextureTsneExtended*) → None
Release GPU resources for the transform

init_transform(*self: nptsne.libs._nptsne.TextureTsneExtended*, *X: numpy.ndarray[float32]*, *initial_embedding: numpy.ndarray[float32] = array([], dtype=float32)*) → bool
Initialize the transform with given data and optional initial embedding. Fit X into an embedded space and return that transformed output.

Parameters

X [ndarray] The input data with shape (num. data points, num. dimensions)

initial_embedding [ndarray] An optional initial embedding. Shape should be (num data points, num output dimensions)

Returns

bool True if successful, False otherwise

Examples

Create an TextureTsneExtended wrapper and initialize the data. This step performs the knn.

```
>>> import nptsne
>>> tsne = nptsne.TextureTsneExtended()
>>> tsne.init_transform(sample_tsne_data)
True
```

reinitialize_transform(*self: nptsne.libs._nptsne.TextureTsneExtended*, *initial_embedding: numpy.ndarray[float32] = array([], dtype=float32)*) → None

Fit X into an embedded space and return that transformed output. Knn is not recomputed. If no initial_embedding is supplied the embedding is re-randomized.

Parameters

initial_embedding [ndarray] An optional initial embedding. Shape should be (num data points, num output dimensions)

Examples

Create an TextureTsneExtended wrapper and initialize the data and run for 250 iterations.

```
>>> import nptsne
>>> tsne = nptsne.TextureTsneExtended()
>>> tsne.init_transform(sample_tsne_data)
True
>>> embedding = tsne.run_transform(iterations=100)
>>> tsne.iteration_count
100
>>> tsne.reinitialize_transform()
>>> tsne.iteration_count
0
```

run_transform(*self: nptsne.libs._nptsne.TextureTsneExtended*, *verbose: bool = False*, *iterations: int = 1000*) → numpy.ndarray[float32]

Run the transform gradient descent for a number of iterations with the current settings for exaggeration.

Parameters

verbose [bool] Enable verbose logging to standard output.

iterations [int] The number of iterations to run.

Returns

ndarray A numpy array contain a flatten (1D) embedding. Coordinates are arranged: x0, y0, x, y1, ...

Examples

Create an TextureTsneExtended wrapper and initialize the data and run for 250 iterations.

```
>>> import nptsne
>>> tsne = nptsne.TextureTsneExtended()
>>> tsne.init_transform(sample_tsne_data)
True
>>> embedding = tsne.run_transform(iterations=250)
>>> embedding.shape
(4000,)
>>> tsne.iteration_count
250
```

start_exaggeration_decay(*self: nptsne.libs._nptsne.TextureTsneExtended*) → None

Enable exaggeration decay. Effective on next call to run_transform. From this point exaggeration decays over the following 150 iterations, the decay this is a fixed parameter. This call is ony effective once.

Raises

RuntimeError If the decay is already active. This can be ignored.

Examples

Starting decay exaggeration is recorded in the decay_started_at property.

```
>>> import nptsne
>>> tsne = nptsne.TextureTsneExtended()
>>> tsne.init_transform(sample_tsne_data)
True
>>> tsne.decay_started_at
-1
>>> embedding = tsne.run_transform(iterations=100)
>>> tsne.start_exaggeration_decay()
>>> tsne.decay_started_at
100
```

property decay_started_at

int: The iteration number when exaggeration decay started. Is -1 if exaggeration decay has not started.

Examples

Starting decay exaggeration is recorded in the decay_started_at property.

```
>>> sample_texture_tsne_extended.decay_started_at
-1
```

property iteration_count

int: The number of completed iterations of tSNE gradient descent.

```
>>> sample_texture_tsne_extended.iteration_count
0
```

property knn_algorithm

int: The number of iterations, set at initialization.

Examples

```
>>> sample_texture_tsne_extended.knn_algorithm
KnnAlgorithm.Flann
```

property num_target_dimensions
int: The number of target dimensions, set at initialization.

Examples

```
>>> sample_texture_tsne_extended.num_target_dimensions
2
```

property perplexity
int: The tsne perplexity, set at initialization.

Examples

```
>>> sample_texture_tsne_extended.perplexity
30
```

property verbose
bool: True if verbose logging is enabled. Set at initialization.

Examples

```
>>> sample_texture_tsne_extended.verbose
False
```

3.1.3 nptsne.hsne_analysis: HSNE analysis support submodule

nptsne.hsne_analysis.Analysis	Create a new analysis as a child of an (optional) parent analysis.
nptsne.hsne_analysis.AnalysisModel	Create an analysis model with the a top level Analysis containing all landmarks at the highest scale
nptsne.hsne_analysis.EmbedderType	Enumeration used to select the embedder used. Two possibilities are
nptsne.hsne_analysis.SparseTsne	SparseTsne a wrapper for an approximating tSNE CPU implementation as described in [1].

class nptsne.hsne_analysis.**Analysis**(*self: nptsne.libs._nptsne._hsne_analysis.Analysis*, *hnse: nptsne.libs._nptsne.HSne*, *embedder_type: nptsne.libs._nptsne._hsne_analysis.EmbedderType*, *parent: nptsne.libs._nptsne._hsne_analysis.Analysis = None*, *parent_selection: List[int] = []*) → None
Bases: pybind11_builtins.pybind11_object

Create a new analysis as a child of an (optional) parent analysis.

Parameters

hsne [HSne] The hierarchical SNE being explored

embedder_type [*EmbedderType*] The tSNE to use CPU or GPU based

parent [*Analysis*, optional] The parent Analysis (where the selection was performed) if any

parent_selection [list, optional] List of selection indexes in the parent analysis.

Notes

Together with *AnalysisModel* provides support for visual analytics of an hSNE. The Analysis class holds both the chosen landmarks at a particular scale but also permits referencing back to the original data. Additionally a t-SNE embedder is included (a choice is provided between GPU and CPU implementations) which can be used to create an embedding of the selected landmarks.

Examples

The Analysis constructor is meant for use by the :class: *nptsne.hsne_analysis.AnalysisModel*. The example here illustrates how a top level analysis would be created from a sample hsne.

```
>>> import nptsne
>>> top_analysis = nptsne.hsne_analysis.Analysis(sample_hsne, nptsne.hsne_
↪analysis.EmbedderType.CPU)
>>> top_analysis.scale_id
2
>>> sample_hsne.get_scale(top_analysis.scale_id).num_points == top_analysis.
↪number_of_points
True
```

Attributes

number_of_points int : number of landmarks in this *Analysis*

parent_id int : Unique id of the parent analysis

transition_matrix list(dict) : The transition (probability) matrix in this *Analysis*

landmark_weights ndarray : the weights for the landmarks in this *Analysis*

landmark_indexes ndarray : the indexes for the landmarks in this *Analysis*

landmark_orig_indexes ndarray : the original data indexes for the landmarks in this *Analysis*

embedding ndarray : the tSNE embedding generated for this *Analysis*

do_iteration (*self: nptsne.libs._nptsne._hsne_analysis.Analysis*) → None
Perform one iteration of the chosen embedder

get_area_of_influence (*self: nptsne.libs._nptsne._hsne_analysis.Analysis*, *select_list: List[int]*) → numpy.ndarray[float32]
Get the area of influence of the selection in the original data.

Parameters

select_list [list] A list of selection indexes for landmarks in this analysis

Returns

ndarray The indexes for the original points represented by the selected landmarks

property embedding
`ndarray` : the tSNE embedding generated for this *Analysis*

property id
int: Internally generated unique id for the analysis.

Examples

```
>>> sample_analysis.id
0
```

property landmark_indexes
`ndarray` : the indexes for the landmarks in this *Analysis*

Examples

In a complete top level analysis all points are present in this case all the points at scale2.

```
>>> import numpy as np
>>> np.array_equal(
... np.arange(sample_scale2.num_points, dtype=np.uint32),
... sample_analysis.landmark_indexes)
True
```

property landmark_orig_indexes
`ndarray` : the original data indexes for the landmarks in this *Analysis*

property landmark_weights
`ndarray` : the weights for the landmarks in this *Analysis*

Examples

There will be a weight for every point.

```
>>> weights = sample_analysis.landmark_weights
>>> weights.shape == (sample_analysis.number_of_points,)
True
```

property number_of_points
int : number of landmarks in this *Analysis*

Examples

The sample analysis is all the top scale points

```
>>> sample_analysis.number_of_points == sample_scale2.num_points
True
```

property parent_id
int : Unique id of the parent analysis

property scale_id
int: The number of this HSNE scale where this analysis is created.

Examples

```
>>> sample_analysis.scale_id
2
```

property transition_matrix
list(dict) : The transition (probability) matrix in this *Analysis*

class nptsne.hsne_analysis.**AnalysisModel**(*hsne*, *embedder_type*)
Bases: object

Create an analysis model with the a top level Analysis containing all landmarks at the highest scale

Parameters

hsne [HSne] The python HSne wrapper class

embedder_type [hsne_analysis.EmbedderType] The embedder to be used when creating a new analysis CPU or GPU

See also:

hsne_analysis.Analysis

hsne_analysis.EmbedderType.CPU

Notes

The hsne_analysis.AnalysisModel contains the user driven selections when exploring an HSNE hierarchy. The AnalysisModel is created with a top level default hsne_analysis.Analysis containing all top level landmarks.

Examples

Initialize a model using loaded HSne data.

```
>>> import nptsne
>>> model = nptsne.hsne_analysis.AnalysisModel(sample_hsne, nptsne.hsne_analysis.
↪EmbedderType.CPU)
>>> model.top_scale_id
2
```

Attributes

top_analysis hsne_analysis.Analysis: The top level analysis

analysis_container The container for all analyses.

bottom_scale_id

top_scale_id

add_new_analysis(*parent*, *parent_selection*)
Add a new analysis based on a selection in a parent analysis

Parameters

parent: Analysis The parent analysis

parent_selection: ndarray<np.uint32> The selection indices in the parent analysis

Examples

Make a child analysis by selecting half of the points in the top analysis. The analysis is created at the next scale down is a child of the top level and contains an embedding of the right shape.

```
>>> import nptsne
>>> model = nptsne.hsne_analysis.AnalysisModel(sample_hsne, nptsne.hsne_
↪analysis.EmbedderType.CPU)
>>> sel = np.arange(int(model.top_analysis.number_of_points / 2))
>>> analysis = model.add_new_analysis(model.top_analysis, sel)
>>> analysis.scale_id
1
>>> analysis.parent_id == model.top_analysis.id
True
>>> analysis.embedding.shape == (analysis.number_of_points, 2)
True
```

get_analysis(*id*)

Get the *Analysis* for the given id

Parameters

id: int An Analysis id

Examples

```
>>> import nptsne
>>> model = nptsne.hsne_analysis.AnalysisModel(sample_hsne, nptsne.hsne_
↪analysis.EmbedderType.CPU)
>>> id = model.top_analysis.id
>>> str(model.top_analysis) == str(model.get_analysis(id))
True
```

remove_analysis(*id*)

Remove the analysis and all children Returns ——- list

list of deleted ids

Examples

```
>>> import nptsne
>>> model = nptsne.hsne_analysis.AnalysisModel(sample_hsne, nptsne.hsne_
↪analysis.EmbedderType.CPU)
>>> sel = np.arange(int(model.top_analysis.number_of_points / 2))
>>> analysis = model.add_new_analysis(model.top_analysis, sel)
>>> id = analysis.id
>>> a_list = model.remove_analysis(analysis.id)
>>> a_list == [id]
```

property analysis_container

The container for all analyses.

This is an internal property exposed for debug purposes only

property top_analysis

hsne_analysis.Analysis: The top level analysis

Examples

Retrieve the top level analysis containing all points at the top level.

```
>>> import nptsne
>>> model = nptsne.hsne_analysis.AnalysisModel(sample_hsne, nptsne.hsne_
↪analysis.EmbedderType.CPU)
>>> analysis = model.top_analysis
>>> analysis.scale_id
2
```

class nptsne.hsne_analysis.**EmbedderType**(*self: nptsne.libs._nptsne._hsne_analysis.EmbedderType*, *arg0: int*) → None

Bases: pybind11_builtins.pybind11_object

Enumeration used to select the embedder used. Two possibilities are supported:

EmbedderType.CPU: CPU tSNE *EmbedderType.CPU*: GPU tSNE

Members:

CPU

GPU

CPU = EmbedderType.CPU

GPU = EmbedderType.GPU

property name
(self: handle) -> str

class nptsne.hsne_analysis.**SparseTsne**

Bases: pybind11_builtins.pybind11_object

SparseTsne a wrapper for an approximating tSNE CPU implementation as described in [1].

Forms an alternative to *TextureTsne* when GPU acceleration for creation of the embedding is not available for internal use in the *Analysis* class

See also:

Analysis

EmbedderType

References

[1]

Attributes

embedding [ndarray] Embedding plot - shape embed dimensions x num points

do_iteration(*self: nptsne.libs._nptsne._hsne_analysis.SparseTsne*) → None
Perform a sinsle tSNE iteration on the sparse data. Once complete the embedding coordinates can be read via the embedding property

property embedding
Embedding plot - shape embed dimensions x num points

CHAPTER

FOUR

BACKGROUND INFORMATION

This page is reserved for experiments and other background material that may be of interest to the end user of nptsne.

4.1 umap to tSNE example

This short animation shows the effect of inputting a umap embedding of 7000 MNIST digits into tSNE and then relaxing the force exaggeration.

CHAPTER

FIVE

APACHE LICENSE 2.0

CHAPTER

SIX

RELEASE HISTORY

This contains the release notes for the current version (1.1.0) of nptsne.

6.1 Release notes

6.2 Version 1.1.0

Supports python 3.6, 3.7, and 3.8 on Windows, Ubuntu (using manylinux2010) and Macos.

6.2.1 Changelog 1.1.0

- Using the latest HDI Library v1.2.1 supporting the additional Annoy knn method.
- Building with cibuildwheel to give manylinux support for a wide range of linux platforms.
- This version is extended with HSNE support. This comprises the HSne class which can generate or load an hsne analysis and supporting classes (Analysis and AnalysisModel) that can be used to navigate the HSne hierarchy.
- Additionally a number of demos of GPU accelerated t-SNE and HSNE are available at *Demo list*

6.3 Previous versions

- v1.0.0
- v0.2.0
- v0.1.1a1

6.4 Version 1.0.0

Supports python 3.6 and 3.7 Windows, Ubuntu (xenial/16.04 and upward) and Macos.

6.4.1 Changelog 1.0.0

Using the latest HDI Library which supports both GPU computer shader (Windows and Linux) and GPU rasterizer (Macos) versions of the underlying tSNE algorithm.

Added reinitialize_transform to TextureTsneExtended to allow quick embedding regeneration.

6.5 Version 0.2.0

QT dependency removed. Windows release only

6.5.1 Changelog 0.2.0

This was a Windows only release without QT otherwise it is unchanged from the previous version

6.6 Version 0.1.1a1

This is an alpha release of the nptsne library. As such the API may change in future releases based on user feedback.

The initial alpha release is being distributed to a limited audience via GitHub. The current roadmap is to switch to PyPi when the API is declared stable.

6.6.1 Changelog 0.1.1a1

This is the first release of the **nptsne** package.

CHAPTER

SEVEN

ROADMAP

The following items are envisioned for future releases:

1. Investigate adding a context manager for TextureTsneExtended to automatically close the nptsne OpenGL context. This would enable the following code:

Listing 1: Possible context manager for TextureTsneExtended

```
with nptsne.TextureTsneExtended(False) as tsne:
    tsne.init_transform(mnist['data'])
    embedding = tsne.run_transform(verbose=False, iterations=step_size)

# TextureTsneExtended & OpenGL context have been freed at end of indent context
# tsne.close() is not required.
# Continue processing embedding result in parent context. e.g.:
xyembed = embedding.reshape((70000, 2))
plt.scatter(xyembed[..., 0], xyembed[..., 1])
```

Note: This is an Release candidate. For more detail see *Release notes*.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- Module Index
- search

BIBLIOGRAPHY

[NP2019] N. Pezzotti et al., "GPGPU Linear Complexity t-SNE Optimization," in IEEE Transactions on Visualization and Computer Graphics. doi: 10.1109/TVCG.2019.2934307 keywords: {Minimization;Linear programming;Computational modeling;Approximation algorithms;Complexity theory;Optimization;Data visualization;High Dimensional Data;Dimensionality Reduction;Progressive Visual Analytics;Approximate Computation;GPGPU}, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8811606&isnumber=4359476

[NP2016] Pezzotti et al., "Hierarchical Stochastic Neighbor Embedding," in Computer Graphics Forum 35, 21–30. doi: 10.1111/cgf.12878 Categories and Subject Descriptors(according to ACM CCS): I.3.0 [Computer Graphics]: General URL: https://doi.org/10.1111/cgf.12878

[1] Pezzotti, N. et al., GPGPU Linear Complexity t-SNE Optimization

[2] Pezzotti, N. et al., Hierarchical Stochastic Neighbor Embedding

[1] Hierarchical Stochastic Neighbor Embedding

[1] Pezzotti, N., Thijssen, J., Mordvintsev, A., Höllt, T., Van Lew, B., Lelieveldt, B.P.F., Eisemann, E., Vilanova, A. GPGPU Linear Complexity t-SNE Optimization IEEE Transactions on Visualization and Computer Graphics 26, 1172–1181

[1] Pezzotti, N., Thijssen, J., Mordvintsev, A., Höllt, T., Van Lew, B., Lelieveldt, B.P.F., Eisemann, E., Vilanova, A. GPGPU Linear Complexity t-SNE Optimization IEEE Transactions on Visualization and Computer Graphics 26, 1172–1181

[1] Pezzotti, N., Lelieveldt, B.P.F., Maaten, L. van der, Höllt, T., Eisemann, E., Vilanova, A., 2017. Approximated and User Steerable tSNE for Progressive Visual Analytics. IEEE Transactions on Visualization and Computer Graphics 23, 1739–1752.

PYTHON MODULE INDEX

n

INDEX

N

P

R

S

T

V