t-Distributed Stochastic Neighbor Embedding or t-SNE is a popular non-linear dimensionality reduction technique that can be used for visualizing high dimensional data sets\(^1\).

t-SNE has had several criticisms over the years, which we will address here:

1. **t-SNE is slow.** This criticism likely comes from the fact that some popular packages have slow implementations of t-SNE. Until fairly recently, t-SNE did not scale well to larger data sets, however recent theoretical advances by Linderman et. al.\(^2\) have made t-SNE one of the fastest non-linear dimensionality reduction methods available, capable of scaling to millions of samples.

2. **t-SNE does not preserve global structure.** The objective of t-SNE is to preserve local structure i.e. samples close to one another in the input space remain close to each other in the embedded space. This can lead to similar clusters of data points drifting to different regions of the embedding space. Recently, Kobak and Berens\(^3\) introduced a range of tricks that address this problem and better preserve global structure.

3. **t-SNE is nonparametric therefore it is impossible to add new samples to an existing embedding.** This argument is often repeated and likely comes from the fact that most software packages simply did not take the time to implement this functionality. t-SNE is nonparametric meaning that it does not learn a function \(f\) that projects samples from the ambient space into the embedding space. However, the objective function of t-SNE is well defined and new samples can easily be added into an existing embedding by taking a data point and optimizing its position with respect to the existing embedding. This is the only available implementation we know of that allows adding new points to an existing embedding.

The goal of this project is

1. **Extensibility.** We provide efficient defaults for the typical use case. We make it very easy to use various tricks that have been introduced to improve the quality of t-SNE embeddings. The library is designed to be extensible and it is easy to implement and use your own components and makes experimentation very simple.

2. **Speed.** We provide two fast, parallel implementations of t-SNE, which are comparable to their C++ counterparts in speed. Python does incur some overhead, so if speed is your only requirement, consider using Flt-SNE. The differences are often minute and become even less apparent when utilizing multiple cores.

3. **Interactivity.** This library was built for Orange, an interactive machine learning toolkit. As such, we provide a powerful API which can control all aspects of the t-SNE algorithm and makes it suitable for interactive environments.

4. **Ease of distribution.** Flt-SNE, the reference C++ implementation for t-SNE, is not easy to install or distribute. It requires one to preinstall C libraries and requires manual compilation. This package is installable either through `pip` or `conda` with a single command, making it very easy to include in other packages.

---


Fig. 1: A visualization of 44,808 single cell transcriptomes from the mouse retina embedded using the multiscale kernel trick for preserving global structure.
1.1 Conda

openTSNE can be easily installed from conda-forge with

```
conda install --channel conda-forge opentsne
```

Conda package

1.2 PyPi

openTSNE is also available through pip and can be installed with

```
pip install opentsne
```

PyPi package

Note, however, that openTSNE requires a C/C++ compiler.

In order for openTSNE to utilize multiple threads, the C/C++ compiler must also implement OpenMP. In practice, almost all compilers implement this with the exception of older version of clang on OSX systems.

To squeeze the most out of openTSNE, you may also consider installing FFTW3 prior to installation. FFTW3 implements the Fast Fourier Transform, which is heavily used in openTSNE. If FFTW3 is not available, openTSNE will use numpy’s implementation of the FFT, which is slightly slower than FFTW. The difference is only noticeable with large data sets containing millions of data points.
2.1 Simple usage

This notebook demonstrates basic usage of the openTSNE library. This is sufficient for almost all use-cases.

2.1.1 Load data

In most of the notebooks, we will be using the Macosko 2015 mouse retina data set. This is a fairly well-known and well explored data set in the single-cell literature making it suitable as an example.

Data set contains 44808 samples with 50 features

2.1.2 Create train/test split

30021 training samples
14787 test samples

2.1.3 Run t-SNE

We'll first create an embedding on the training data.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>KL divergence</th>
<th>Iterations</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>5.7889</td>
<td>50</td>
<td>1.2277 sec</td>
</tr>
<tr>
<td>100</td>
<td>5.2496</td>
<td>50</td>
<td>1.1978 sec</td>
</tr>
<tr>
<td>150</td>
<td>5.1563</td>
<td>50</td>
<td>1.1671 sec</td>
</tr>
<tr>
<td>200</td>
<td>5.1203</td>
<td>50</td>
<td>1.3511 sec</td>
</tr>
<tr>
<td>250</td>
<td>5.1018</td>
<td>50</td>
<td>1.4377 sec</td>
</tr>
<tr>
<td>300</td>
<td>3.7958</td>
<td>50</td>
<td>1.5298 sec</td>
</tr>
<tr>
<td>350</td>
<td>3.4076</td>
<td>50</td>
<td>1.3780 sec</td>
</tr>
<tr>
<td>400</td>
<td>3.1945</td>
<td>50</td>
<td>1.3263 sec</td>
</tr>
</tbody>
</table>

(continues on next page)
## 2.1.4 Transform

openTSNE is currently the only library that allows embedding new points into an existing embedding.
2.1.5 Together

We superimpose the transformed points onto the original embedding with larger opacity.
2.2 Advanced usage

This notebook replicates what was done in the simple_usage notebooks, but this time with the advanced API. The advanced API is required if we want to use non-standard affinity methods that better preserve global structure.

If you are comfortable with the advanced API, please refer to the preserving_global_structure notebook for a guide on how to obtain better embeddings and preserve more global structure.

2.2.1 Load data

Data set contains 44808 samples with 50 features

2.2.2 Create train/test split

30021 training samples
14787 test samples
2.2.3 Create a t-SNE embedding

Like in the `simple_usage` notebook, we will run the standard t-SNE optimization. This example shows the standard t-SNE optimization. Much can be done in order to better preserve global structure and improve embedding quality. Please refer to the `preserving_global_structure` notebook for some examples.

1. Compute the affinities between data points

   | CPU times: user 1min 39s, sys: 2.15 s, total: 1min 41s |
   | Wall time: 19.6 s |

2. Generate initial coordinates for our embedding

   | CPU times: user 3.01 s, sys: 49.6 ms, total: 3.06 s |
   | Wall time: 77.3 ms |

3. Construct the “TSNEEmbedding“ object

4. Optimize embedding
   
   1. Early exaggeration phase

   | Iteration 50, KL divergence 5.7889, 50 iterations in 1.1595 sec |
   | Iteration 100, KL divergence 5.2496, 50 iterations in 1.1852 sec |
   | Iteration 150, KL divergence 5.1563, 50 iterations in 1.1364 sec |
   | Iteration 200, KL divergence 5.1203, 50 iterations in 1.1426 sec |
   | Iteration 250, KL divergence 5.1018, 50 iterations in 1.1117 sec |

   | CPU times: user 2min 52s, sys: 3.41 s, total: 2min 55s |
   | Wall time: 5.79 s |
2. Regular optimization

<table>
<thead>
<tr>
<th>Iteration</th>
<th>KL divergence</th>
<th>Iterations</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>3.7958</td>
<td>50</td>
<td>1.3252 sec</td>
</tr>
<tr>
<td>100</td>
<td>3.4076</td>
<td>50</td>
<td>1.2355 sec</td>
</tr>
<tr>
<td>150</td>
<td>3.1945</td>
<td>50</td>
<td>1.4455 sec</td>
</tr>
<tr>
<td>200</td>
<td>3.0541</td>
<td>50</td>
<td>1.4912 sec</td>
</tr>
<tr>
<td>250</td>
<td>2.9521</td>
<td>50</td>
<td>1.9103 sec</td>
</tr>
<tr>
<td>300</td>
<td>2.8745</td>
<td>50</td>
<td>2.1101 sec</td>
</tr>
<tr>
<td>350</td>
<td>2.8131</td>
<td>50</td>
<td>2.6402 sec</td>
</tr>
<tr>
<td>400</td>
<td>2.7642</td>
<td>50</td>
<td>3.6373 sec</td>
</tr>
<tr>
<td>450</td>
<td>2.7241</td>
<td>50</td>
<td>3.8347 sec</td>
</tr>
<tr>
<td>500</td>
<td>2.6918</td>
<td>50</td>
<td>4.7176 sec</td>
</tr>
<tr>
<td>550</td>
<td>2.6655</td>
<td>50</td>
<td>6.8521 sec</td>
</tr>
<tr>
<td>600</td>
<td>2.6441</td>
<td>50</td>
<td>5.5079 sec</td>
</tr>
<tr>
<td>650</td>
<td>2.6264</td>
<td>50</td>
<td>6.5560 sec</td>
</tr>
<tr>
<td>700</td>
<td>2.6121</td>
<td>50</td>
<td>7.5798 sec</td>
</tr>
<tr>
<td>750</td>
<td>2.6002</td>
<td>50</td>
<td>9.0642 sec</td>
</tr>
</tbody>
</table>

CPU times: user 27min 24s, sys: 32.9 s, total: 27min 57s
Wall time: 1min

- Amacrine cells
- Astrocytes
- Bipolar cells
- Cones
- Fibroblasts
- Horizontal cells
- Microglia
- Muller glia
- Pericytes
- Retinal ganglion cells
- Rods
- Vascular endothelium
2.2.4 Transform

CPU times: user 3.55 s, sys: 150 ms, total: 3.7 s
Wall time: 1.22 s
Chapter 2. How to use t-SNE

Iteration 50, KL divergence 212577.9338, 50 iterations in 8.4328 sec
Iteration 100, KL divergence 212507.1902, 50 iterations in 6.1227 sec
CPU times: user 3min 14s, sys: 3.71 s, total: 3min 18s
Wall time: 14.7 s
2.2.5 Together

We superimpose the transformed points onto the original embedding with larger opacity.
2.3 Preserving global structure

2.3.1 Load data

Data set contains 44808 samples with 50 features

To avoid constantly specifying colors in our plots, define a helper here.

2.3.2 Easy improvements

Standard t-SNE, as implemented in most software packages, can be improved in several very easy ways that require virtually no effort in openTSNE, but can drastically improve the quality of the embedding.

Standard t-SNE

First, we’ll run t-SNE as it is implemented in most software packages. This will serve as a baseline comparison.

CPU times: user 45min 48s, sys: 54.1 s, total: 46min 42s
Wall time: 1min 37s
Using PCA initialization

The first, easy improvement we can get is to “inject” some global structure into the initialization. The initialization dictates which regions points will appear in, so adding any global structure to the initialization can help.

Note that this is the default in this implementation and the parameter can be omitted.

CPU times: user 42min 9s, sys: 49.8 s, total: 42min 59s
Wall time: 1min 28s
Using cosine distance

Typically, t-SNE is used to create an embedding of high dimensional data sets. However, the notion of Euclidean distance breaks down in high dimensions and the cosine distance is far more appropriate.

We can easily use the cosine distance by setting the metric parameter.

```
CPU times: user 46min 27s, sys: 55.8 s, total: 47min 23s
Wall time: 1min 37s
```
Using PCA initialization and cosine distance

Lastly, let’s see how our embedding looks with both the changes.

CPU times: user 44min 15s, sys: 52.6 s, total: 45min 7s
Wall time: 1min 32s
Summary

We can see that we’ve made a lot of progress already. We would like points of the same color to appear close to one another.

This is not the case in standard t-SNE and t-SNE with cosine distance, because the green points appear on both the bottom and top of the embedding and the dark blue points appear on both the left and right sides.

This is improved when using PCA initialization and better still when we use both PCA initialization and cosine distance.

2.3. Preserving global structure
2.3.3 Using perplexity

Perplexity can be thought of as the trade-off parameter between preserving local and global structure. Lower values will emphasise local structure, while larger values will do a better job at preserving global structure.

Perplexity: 30

Perplexity: 500

CPU times: user 2h 27min 38s, sys: 2min 32s, total: 2h 30min 10s
Wall time: 7min 15s
2.3.4 Using different affinity models

We can take advantage of the observation above, and use combinations of perplexities to obtain better embeddings.

In this section, we describe how to use the tricks described by Kobak and Berens in “The art of using t-SNE for single-cell transcriptomics”. While the publication focuses on t-SNE applications to single-cell data, the methods shown here are applicable to any data set.

When dealing with large data sets, methods which compute large perplexities may be very slow. Please see the large_data_sets notebook for an example of how to obtain a good embedding for large data sets.

Perplexity annealing

The first trick we can use is to first optimize the embedding using a large perplexity to capture the global structure, then lower the perplexity to something smaller to emphasize the local structure.

1. Perform normal t-SNE optimization with large perplexity
2. Lower perplexity and optimize

```
CPU times: user 19.3 s, sys: 1.26 s, total: 20.6 s
Wall time: 1.54 s
```

```
CPU times: user 31min 55s, sys: 39 s, total: 32min 34s
Wall time: 49.7 s
```
Multiscale

One problem when using a high perplexity value e.g. 500 is that some of the clusters start to mix with each other, making the separation less apparent. Instead of a typical Gaussian kernel, we can use a multiscale kernel which will account for two different perplexity values. This typically results in better separation of clusters while still keeping much of the global structure.

CPU times: user 21min 11s, sys: 38.3 s, total: 21min 50s
Wall time: 4min 13s

CPU times: user 9.72 s, sys: 455 ms, total: 10.2 s
Wall time: 255 ms

Now, we just optimize just like we would standard t-SNE.

CPU times: user 31min 24s, sys: 32.7 s, total: 31min 56s
Wall time: 48.6 s
2.3. Preserving global structure

CPU times: user 1h 36min 12s, sys: 1min 38s, total: 1h 37min 51s
Wall time: 2min 38s
2.3.5 Comparison to UMAP

/home/ppolicar/local/miniconda3/envs/tsne/lib/python3.7/site-packages/scikit_learn-0.21.0-py3.7-linux-x86_64.egg/sklearn/externals/joblib/__init__.py:15:
  DeprecationWarning: sklearn.externals.joblib is deprecated in 0.21 and will be removed in 0.23. Please import this functionality directly from joblib, which can be installed with: pip install joblib. If this warning is raised when loading pickled models, you may need to re-serialize those models with scikit-learn 0.21+.
  warnings.warn(msg, category=DeprecationWarning)

2.3. Preserving global structure
2.4 Embedding large data sets

Embedding large data sets typically requires more care. Using various tricks described in `preserving_global_structure` can become quite slow to run. Instead, we can take a smaller, manageable sample of our data set, obtain a good visualization of that. Then, we can add the remaining points to the embedding and use that as our initialization.

Remember that the initialization largely affects the structure of the embedding. This way, our initialization provides the global structure for the embedding, and the subsequent optimization can focus on preserving local structure.
2.4.1 Load data

Data set contains 1306127 samples with 50 features

We’ll also precompute the full affinities, since we’ll be needing it in several places throughout the notebook, and can take a long time to run.

CPU times: user 1h 2min 57s, sys: 1min 47s, total: 1h 4min 44s
Wall time: 37min 40s

2.4.2 Standard t-SNE

First, let’s see what standard t-SNE does.
This doesn’t look too great. The cluster separation is quite poor and the visualization is visually not very appealing.

### 2.4.3 Using exaggeration

Exaggeration can be used in order to get better separation between clusters. Let’s see if that helps.
The separation has improved quite a bit, but many clusters are still intertwined with others.

### 2.4.4 With downsampling

We now perform the sample-transform trick we described above.

Create train/test split
Create sample embedding

<p>| Iteration | KL divergence | 50 iterations in 5.7042 sec | 100 iterations in 6.0842 sec | 150 iterations in 5.6341 sec | 200 iterations in 6.5530 sec | 250 iterations in 5.5155 sec | CPU times: user 12min, sys: 10 s, total: 12min 10s | Wall time: 29.9 s |</p>
<table>
<thead>
<tr>
<th>Iteration</th>
<th>KL divergence</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1.5264</td>
<td>6.6760 sec</td>
</tr>
<tr>
<td>100</td>
<td>1.3347</td>
<td>6.5701 sec</td>
</tr>
<tr>
<td>150</td>
<td>1.2537</td>
<td>5.7738 sec</td>
</tr>
<tr>
<td>200</td>
<td>1.2109</td>
<td>6.0972 sec</td>
</tr>
<tr>
<td>250</td>
<td>1.1694</td>
<td>6.1227 sec</td>
</tr>
<tr>
<td>300</td>
<td>1.1599</td>
<td>6.1803 sec</td>
</tr>
<tr>
<td>350</td>
<td>1.1536</td>
<td>5.6595 sec</td>
</tr>
<tr>
<td>400</td>
<td>1.1500</td>
<td>6.1716 sec</td>
</tr>
<tr>
<td>450</td>
<td>1.1465</td>
<td>6.5309 sec</td>
</tr>
<tr>
<td>500</td>
<td>1.1440</td>
<td>6.5737 sec</td>
</tr>
<tr>
<td>550</td>
<td>1.1418</td>
<td>6.4968 sec</td>
</tr>
<tr>
<td>600</td>
<td>1.1402</td>
<td>5.2163 sec</td>
</tr>
<tr>
<td>650</td>
<td>1.1388</td>
<td>5.6052 sec</td>
</tr>
<tr>
<td>700</td>
<td>1.1373</td>
<td>6.2774 sec</td>
</tr>
<tr>
<td>750</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU times: user 36min 50s, sys: 30.9 s, total: 37min 21s
Wall time: 1min 32s
Learn the full embedding

CPU times: user 2min 37s, sys: 652 ms, total: 2min 38s
Wall time: 1min 18s
array([1.00000000e-04, 1.14353061e-04])

Iteration 50, KL divergence 10.2843, 50 iterations in 48.5307 sec
Iteration 100, KL divergence 10.2803, 50 iterations in 48.2069 sec
Iteration 150, KL divergence 9.3181, 50 iterations in 47.9606 sec
Iteration 200, KL divergence 8.7442, 50 iterations in 48.3420 sec
Iteration 250, KL divergence 8.5049, 50 iterations in 48.1888 sec
Iteration 300, KL divergence 8.3797, 50 iterations in 48.1925 sec
Iteration 350, KL divergence 8.3061, 50 iterations in 48.5544 sec
Iteration 400, KL divergence 8.2591, 50 iterations in 48.4718 sec
Iteration 450, KL divergence 8.2273, 50 iterations in 48.4514 sec
Iteration 500, KL divergence 8.2049, 50 iterations in 48.4425 sec
CPU times: user 2h 15min 27s, sys: 2min 28s, total: 2h 17min 56s
Wall time: 8min 5s

2.4. Embedding large data sets
Iteration 50, KL divergence 7.6311, 50 iterations in 54.7087 sec
Iteration 100, KL divergence 7.4543, 50 iterations in 48.5959 sec
Iteration 150, KL divergence 7.3482, 50 iterations in 48.6267 sec
Iteration 200, KL divergence 7.2778, 50 iterations in 48.8663 sec
Iteration 250, KL divergence 7.2278, 50 iterations in 48.9803 sec
CPU times: user 1h 7min 26s, sys: 1min 16s, total: 1h 8min 43s
Wall time: 4min 12s
<table>
<thead>
<tr>
<th>Iteration</th>
<th>KL divergence</th>
<th>50 iterations in</th>
<th>CPU time: user 1h 6min 21s, sys: 1min 12s, total: 1h 7min 33s</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>7.1927</td>
<td>47.5765 sec</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>7.1623</td>
<td>47.2527 sec</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>7.1375</td>
<td>47.1718 sec</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>7.1169</td>
<td>47.1572 sec</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>7.0995</td>
<td>47.5738 sec</td>
<td></td>
</tr>
</tbody>
</table>

Wall time: 3min 59s

2.4. Embedding large data sets
Iteration  50, KL divergence  7.0854,  50 iterations in  47.2708 sec
Iteration 100, KL divergence  7.0716,  50 iterations in  47.4487 sec
Iteration 150, KL divergence  7.0592,  50 iterations in  47.5619 sec
Iteration 200, KL divergence  7.0478,  50 iterations in  47.1930 sec
Iteration 250, KL divergence  7.0381,  50 iterations in  47.6492 sec
CPU times: user 1h 6min 45s, sys: 1min 14s, total: 1h 7min 59s
Wall time:  3min 59s
2.4.5 Comparison to UMAP

```
/home/ppolicar/local/miniconda3/envs/tsne/lib/python3.7/site-packages/scikit_learn-0.21.0-py3.7-linux-x86_64.egg/sklearn/externals/joblib/__init__.py:15:
  -DeprecationWarning: sklearn.externals.joblib is deprecated in 0.21 and will be removed in 0.23. Please import this functionality directly from joblib, which can be installed with pip install joblib. If this warning is raised when loading pickled models, you may need to re-serialize those models with scikit-learn 0.21+.
warnings.warn(msg, category=DeprecationWarning)

CPU times: user 11h 1min 37s, sys: 18min 41s, total: 11h 20min 19s
Wall time: 1h 24min 29s
```
t-Distributed Stochastic Neighbor Embedding\(^1\) or t-SNE is a popular non-linear dimensionality reduction technique used for visualizing high dimensional data sets.

In this section, we describe the algorithm in a way that will hopefully be accessible to most audiences. We skip much of the mathematical rigour but provide references where necessary. This way, we hope to bring intuition to how t-SNE works, where it can shine and when and why it can fail.

### 3.1 t-SNE

Given a \(D\)-dimensional data set \(X \in \mathbb{R}^D\), t-SNE aims to produce a low dimensional embedding \(Y \in \mathbb{R}^d\) where \(d\) is much smaller than \(D\), typically 2, such that if two points \(x_i\) and \(x_j\) are close to one another in the input space \(X\), then their corresponding lower dimensional points \(y_i\) and \(y_j\) are also close.

In order to achieve this, t-SNE models similarities in the input and embedding space as probability densities. In the input space, the similarities are given by a Gaussian distribution.

\[
p_{ji|i} = \frac{\exp \left( -\frac{||x_i - x_j||^2}{2\sigma_i^2} \right)}{\sum_{k \neq i} \exp \left( -\frac{||x_i - x_k||^2}{2\sigma_i^2} \right)}
\]

These conditional probabilities are then typically symmetrized to obtain joint probabilities \(p_{ij}\).

\[
p_{ij} = \frac{p_{j|i} + p_{i|j}}{2}
\]

In the embedding space, we replace the Gaussian distribution with the Student’s t-distribution is used, hence the name t-SNE. The t-distribution has fatter tails, allowing some distances to be less faithfully preserved in the embedding.

\[
g_{ij} = \left( \frac{1 + ||y_i - y_j||^2}{\sum_{k \neq i} \left( 1 + ||y_k - y_i||^2 \right)^{-1}} \right)^{-1}
\]

Our goal, now, is to make $Q$ as similar to $P$ as possible. A well-known measure of similarity between two probability distributions is the Kullback–Leibler divergence and is given by

$$C = KL(P || Q) = \sum_{ij} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

We have now fully specified our model. We have two probability distributions describing the input and embedding spaces and we have a cost function that tells us how good our embedding is. The only thing remaining is to optimize the cost function. One simple way to optimize differentiable cost functions is gradient descent. To perform gradient descent, we need to work out the gradient, which will be used to update $Y$. The full derivation can be found in $1$. The gradient of the cost KL divergence is given by

$$\frac{\partial C}{\partial y_i} = 4 \sum_{j \neq i} (p_{ij} - q_{ij}) (y_i - y_j) \left(1 + \|y_i - y_j\|^2\right)^{-1}$$

One last thing we have not yet mentioned how to set the bandwidths $\sigma_i$ for the Gaussian kernels centered over each data point in the input space. It is unlikely that one single value of $\sigma_i$ is optimal for all data points because the density of the data is likely to vary. In dense regions, a smaller value of $\sigma_i$ is usually more appropriate than in sparser regions. Perplexity is defined as

$$\text{Perplexity}(p_i) = 2^H(p_i)$$

where $H$ is the Shannon entropy of a discrete distribution

$$H(p_i) = -\sum_i p_{j|i} \log_2 (p_{j|i})$$

Perplexity can be thought of as a continuous analogue to the $k$ nearest neighbours, to which t-SNE will attempt to preserve distances. More concretely, the bandwidths $\sigma_i$ are set such that each Gaussian kernel fits $k$ nearest neighbors within one standard deviation of the probability density.

And that’s it! You now know what t-SNE is and what it does.

### 3.2 Accelerations

Unfortunately, a direct implementation of t-SNE is rather slow. It’s easy to see that that computing all the $p_{ij}$ and $q_{ij}$ requires computing all pair-wise interactions between points and has time complexity $O(N^2)$. This quickly becomes far too slow for any reasonably sized data set. Moreover, the normalization constant for $q_{ij}$ must be computed in every single iteration of the optimization, while $p_{ij}$ can be computed only once, since the points in the input space stay fixed.

Most of the subsequent research on t-SNE has focused on how to accelerate the computation of $p_{ij}$ and $q_{ij}$.

We will begin by rewriting the gradient update rule in the following form

$$\frac{\partial C}{\partial y_i} = 4 \left( \sum_{j \neq i} p_{ij} q_{ij} Z (y_i - y_j) - \sum_{j \neq i} q_{ij}^2 Z (y_i - y_j) \right)$$

where $Z$ is defined as the normalization constant in $q_{ij}$

$$Z = \sum_{k \neq i} (1 + \|y_k - y_i\|^2)^{-1}$$

This equation splits the gradient into two parts which can be interpreted as the attractive and repulsive forces between points in the embedding. The first term includes $p_{ij}$, encouraging nearby points to remain close to each other. The repulsive forces have the natural interpretation of the N-body problem, where all data points exert forces on each other. The optimization process finds an equilibrium between these two forces.
3.2.1 Attractive forces

First, we will address how to speed up the computation of the attractive forces. In practice, this means speeding up the computation of the input similarities \( p_{ij} \). These can be precomputed once before the optimization as the points in the input space remain fixed.

Only nearest neighbors

The first improvement for the computation of the input similarities \( p_{ij} \) in the input space comes from observing that points further than \( 3\sigma \) have nearly infinitesimally small probabilities. These \( p_{ij} \) have practically zero contribution to the KL divergence and can be ignored. Because of the way these bandwidths are computed, it is reasonable to compute and consider only the \( \lfloor 3 \ast \text{Perplexity} \rfloor \) nearest neighbors of each data point and ignore points further away. This means that the affinity matrix \( P \) becomes sparse, and computing the \( p_{ij} \) values includes summing up only the non-zero entries\(^2\).

Approximate neighbors

The second, more recent improvement comes from a theoretical advance which claims that using approximate nearest neighbors works just as well as using exact nearest neighbors. We do not attempt to justify this approach here but the interested reader can find proof in\(^3\).

Previously, the \( k \) nearest neighbors were computed using Vantage Point trees, which have time complexity \( O(N \log N) \), which becomes too expensive with large data sets. Replacing this with approximate methods can lower this time complexity, allowing us to compute \( p_{ij} \) for millions of data points in a reasonable amount of time.

3.2.2 Repulsive forces

We next show how to accelerate the computation of the second term i.e. the repulsive forces. As previously mentioned, these have a natural interpretation of an N-body problem.

Barnes-Hut t-SNE

The first major acceleration draws from particle simulations, which use space partitioning trees to approximate repulsive forces. These are made possible by the observation that given two well-separated clusters of points \( A \) and \( B \), choose \( x \in A \) and \( y, z \in B \) and notice that the repulsive forces from \( y \) onto \( x \) will be roughly the same as \( z \) onto \( x \).

This is true for any point in \( A \) and \( B \), therefore we can compute the interaction for all points from \( B \) onto any point in \( A \) by simply computing the center of mass in \( B \) and using that as a summary for all the points in \( A \).

The Barnes-Hut tree algorithm\(^2\) exploits this fact by constructing a quad-tree and at every node in the tree, deciding whether the center of mass can be used as a summary for all the points in that cell.

Let’s now make precise when a cell can be used as a summary for some point. The condition compares the distance between the cell and the target point and the size of the cell with the following criterion:

\[
\frac{r_{\text{cell}}}{\| y_t - y_{\text{cell}} \|^2} < \theta
\]

where \( r_{\text{cell}} \) is the length of the diagonal in the cell and \( y_{\text{cell}} \) is the center of mass inside the cell. If the condition holds, then the cell is used as a summary. \( \theta \) is a parameter of choice which trades off speed with accuracy. Higher values of


\( \theta \) allow more cells to be summarized leading to worse approximations but faster runtime. Note that when \( \theta = 0 \), all pairwise interactions are computed. Typically, \( \theta \) is set somewhere between 0.2 to 0.8.

Lastly, let’s look at the time complexity of the Barnes-Hut approximation. Constructing the tree is fairly simple with complexity \( \mathcal{O}(N) \). Lookup time is dependent on \( \theta \), but on average takes about \( \mathcal{O}(N \log N) \) time.

**Interpolation-based t-SNE**

A more recent approximation for computing the repulsive forces takes a different route. This method is quite mathematically involved, so we won’t go into it too much, but the key idea is to shift the computation from \( N \) data points to a grid of points that cover the embedding space. We compute the repulsive forces directly between our new points, then use these as interpolation points for our actual data points. The idea is demonstrated in the figure below.

The example also demonstrates one of the possible problems with this method. There are far less blue points (60) representing data samples than there are red interpolation points (225). In this case, directly computing the repulsive forces between the data points would, in fact, be more efficient than this side step using interpolation points. This highlights the fact that while this method can be extremely efficient when \( N \) is large, it can also be much slower when \( N \) is small.

The method splits the embedding space into equally sized boxes. Interpolation is performed within each box separately i.e. to compute the repulsive forces for point \( x_i \), we first identify which box it belongs to, then perform interpolation using the 9 interpolation points (in the example above). Clearly, the accuracy of the optimization depends on the number of boxes or the number of interpolation points we use.

We can improve accuracy by using more interpolation points within each box, however, this is generally a bad idea. In the case of equispaced points, interpolation suffers from the Runge phenomenon. When this happens, the interpolation error is very large at the edges of the box.

The Runge phenomenon can be mitigated by instead using Chebyshev nodes for interpolation, which equally distribute the interpolation error along the domain. However, we want to keep our equispaced points because when the interactions between all the interpolation points are put together in a matrix, they form a structured Toeplitz matrix. Toeplitz matrices are computationally convenient for matrix-vector multiplications, which can be accelerated with the Fast Fourier Transform, reducing the computational complexity from \( \mathcal{O}(N^2) \) to \( \mathcal{O}(N \log N) \). Please refer to the original publication for more information³.
Fig. 1: A quad tree evenly splits the space until there is a single point in every cell.
Fig. 2: We demonstrate the Runge phenomenon on the Cauchy kernel using equispaced points. The errors oscillate wildly at the edges of the space when using 5 interpolation points.
So clearly, increasing the number of interpolation points can be problematic, so why not increase the number of boxes instead? By increasing the number of boxes, we also increase the number of interpolation points, but each box will still have only 3 points, eliminating the danger for large errors at boundaries.

By shifting most of the computation onto the interpolation points, we have effectively made the computational complexity dependent on the number of interpolation points $p$ rather than $N$. The computational complexity, therefore, reduces to $O(N)$ with respect to $N$.

### 3.3 Optimization

The t-SNE optimization phase typically runs in two phases. The early exaggeration phase and the normal regime.

The early exaggeration phase is first run for typically 250 iterations with a large value of exaggeration. This increases the attractive forces between points and allows points to move through the embedding more freely to find their true neighbors. Skipping this phase may result in larger clusters being split into several smaller clusters which can be scattered in the embedding.

The normal regime follows the early exaggeration phase and is typically run for 750 iterations. The attractive forces are usually restored to their true values and we allow the embedding to converge to a stable state.

### 3.4 Embedding data into lower dimensions

This section is dedicated to the problems of embedding high dimensional data into lower dimensional embeddings. Methods that attempt to preserve distances between data points e.g. MDS, t-SNE, UMAP face a very tough challenge. High dimensional data sets typically have lower intrinsic dimensionality $d \ll D$ however $d$ may still be larger than 2 and preserving these distances faithfully might not always be possible.

To make this clearer, let’s look at a very simple example of a regular tetrahedron aka an equilateral pyramid.
How might we create a 2-dimensional embedding of this data such that we keep all the distances intact? Perhaps a direct projection?

That doesn’t work. It seems that the points form a star-like topology, which isn’t what we’re after. It’s easy to see there is no way to properly preserve the distances while trying to project this simple tetrahedron into two dimensions. That’s because the tetrahedron is intrinsically 3 dimensional. If the data have even higher intrinsic dimensionality, this problem is further exacerbated.

Let’s see how well t-SNE does with our tetrahedron.

This is likely the best we can do. The distances are somewhat preserved quite well - not perfectly - but probably the best we can hope to achieve.
This is one of the reasons why the interpretation of these kinds of plots is difficult or impossible. In all embeddings, distances between clusters of points can be completely meaningless. It is often impossible to represent complex topologies in 2 dimensions, and embeddings should be approached with the utmost care when attempting to interpret their layout.

t-SNE's objective is very clear - to preserve local neighborhoods. If a set of points cluster together on a t-SNE plot, we can be fairly certain that these points are close to each other. Nothing else can be said with certainty.

UMAP, a recent and popular embedding technique for visualizing high dimensional data sets, promises to better preserve global structure in addition to local neighborhoods. As we have demonstrated, this is simply not possible if the intrinsic dimensionality of the data is much higher. When using both UMAP or t-SNE, one must take care not to overinterpret the embedding structure or distances.

3.5 References
4.1 Perplexity

Perplexity is perhaps the most important parameter in t-SNE and can reveal different aspects of the data. Considered loosely, it can be thought of as the balance between preserving the global and the local structure of the data. A more direct way to think about perplexity is that it is the continuous analogy to the \( k \) number of nearest neighbors for which we will preserve distances.

In most implementations, perplexity defaults to 30. This focuses the attention of t-SNE on preserving the distances to its 30 nearest neighbors and puts virtually no weight on preserving distances to the remaining points. For data sets with a small number of points e.g. 100, this will uncover the global structure quite well since each point will preserve distances to a third of the data set.

For larger data sets, e.g. 10,000 points, considering 30 nearest neighbors will likely do a poor job of preserving global structure. Using a higher perplexity value e.g. 500, will do a much better job for of uncovering the global structure. For larger data sets still e.g. 500k or 1 million samples, this is typically not enough and can take quite a long time to run. Luckily, various tricks can be used to improve global structure\(^1\).

Note that perplexity linearly impacts runtime i.e. higher values of perplexity will incur longer execution time. For example, the embedding in Figure 1a took around 1 minute 30 seconds to compute, while Figure 1b took around 6 minutes.

4.2 Exaggeration

The exaggeration factor is typically used during the early exaggeration phase. This factor increases the attractive forces between points and allows points to move around more freely, finding their nearest neighbors more easily. The most typical value of exaggeration during the early exaggeration phase is 12, but higher values have also been shown to work in combination with different learning rates\(^2\).

Fig. 1: Higher values of perplexity do a better job of preserving global structure, but can obscure local structure. In both a) and b) we run standard t-SNE with perplexities 30 and 500, respectively.

Exaggeration can also be used during the normal optimization regime to form more densely packed clusters, making the separation between clusters more visible.

4.3 Optimization parameters

t-SNE uses a variation of gradient descent optimization procedure that incorporates momentum to speed up convergence of the embedding.

learning_rate: float The learning rate controls the step size of the gradient updates. This typically ranges from 100 to 1000, but usually the default (200) works well enough.

- When dealing with large data sets e.g. 500k samples or more, it may be necessary to increase the learning rate or to increase the number of iterations.

momentum: float Gradient descent with momentum keeps a sum exponentially decaying weights from previous iterations, speeding up convergence. In early stages of the optimization, this is typically set to a lower value (0.5 in most implementations) since points generally move around quite a bit in this phase and increased after the initial early exaggeration phase (typically to 0.8) to speed up convergence.

max_grad_norm: float By default, openTSNE does not apply gradient clipping. However, when embedding new data into an existing embedding, care must be taken that the data points do not “shoot off”. Gradient clipping alleviates this issue.

4.4 Barnes-Hut parameters

Please refer to Barnes-Hut t-SNE for a description of the Barnes-Hut algorithm.

---

Fig. 2: **Figure 2**: We run t-SNE twice on the 10x genomics mouse brain data set, containing 1,306,127 samples. a) t-SNE was run with the regular early exaggeration phase 12 for 500 iterations, then in the regular regime with no exaggeration for 750 iterations. b) t-SNE was run with the regular early exaggeration phase 12 for 500 iterations, then for another 750 iterations with exaggeration 4.

**theta**: float  The trade-off parameter between accuracy and speed.

### 4.5 Interpolation parameters

Please refer to *Interpolation-based t-SNE* for a description of the interpolation-based algorithm.

- **n_interpolation_points**: int  The number of interpolation points to use within each grid cell. It is highly recommended leaving this at the default value due to the Runge phenomenon described above.

- **min_num_intervals**: int  This value indicates what the minimum number of intervals/cells should be in any dimension.

- **ints_in_interval**: float  Our implementation dynamically determines the number of cells such that the accuracy for any given interval remains fixed. This value indicates the size of the interval/cell in any dimension e.g. setting this value to 3 indicates that all the cells should have side length of 3.

### 4.6 References
We performed benchmarks for several popular t-SNE implementations. Benchmarks were run on a Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz processor.

We repeated each run three times. The data were subsampled from the 10X 1.3 Million Brain Cells from E18 Mice (available here).
t-SNE implementation benchmarks
6.1 Initialization

**openTSNE.initialization.pca** *(X, n_components=2, svd_solver='auto', random_state=None)*

Initialize an embedding using the top principal components.

**Parameters**

- **n_components** (*int*) – The dimension of the embedding space.
- **random_state** (*Union[int, RandomState]*) – If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by *np.random*.

**Returns** initialization

**Return type** *np.ndarray*

**openTSNE.initialization.random** *(X, n_components=2, random_state=None)*

Initialize an embedding using samples from an isotropic Gaussian.

**Parameters**

- **n_components** (*int*) – The dimension of the embedding space.
- **random_state** (*Union[int, RandomState]*) – If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by *np.random*.

**Returns** initialization

**Return type** *np.ndarray*
6.2 Affinity

**class openTSNE.affinity.Affinities**
Compute the affinities between samples.

T-SNE takes as input an affinity matrix $P$, and does not really care about anything else from the data. This means we can use t-SNE for any data where we are able to express interactions between samples with an affinity matrix.

$P$
The $N \times N$ affinity matrix expressing interactions between $N$ initial data samples.

**to_new**(data, return_distances=False)
Compute the affinities of new samples to the initial samples.

This is necessary for embedding new data points into an existing embedding.

**Parameters**
- **data** *(np.ndarray)* – The data points to be added to the existing embedding.
- **return_distances** *(bool)* – If needed, the function can return the indices of the nearest neighbors and their corresponding distances.

**Returns**
- **P** *(array_like)* – An $N \times M$ affinity matrix expressing interactions between $N$ new data points the initial $M$ data samples.
- **indices** *(np.ndarray)* – Returned if return_distances=True. The indices of the $k$ nearest neighbors in the existing embedding for every new data point.
- **distances** *(np.ndarray)* – Returned if return_distances=True. The distances to the $k$ nearest neighbors in the existing embedding for every new data point.

**class openTSNE.affinity.PerplexityBasedNN**(data, perplexity=30, method='approx', metric='euclidean', metric_params=None, symmetrize=True, n_jobs=1, random_state=None)
Compute affinities using nearest neighbors.

Please see the *Parameter guide* for more information.

**Parameters**
- **data** *(np.ndarray)* – The data matrix.
- **perplexity** *(float)* – Perplexity can be thought of as the continuous $k$ number of nearest neighbors, for which t-SNE will attempt to preserve distances.
- **method** *(str)* – Specifies the nearest neighbor method to use. Can be either exact or approx.
- **metric** *(str)* – The metric to be used to compute affinities between points in the original space.
- **metric_params** *(dict)* – Additional keyword arguments for the metric function.
- **symmetrize** *(bool)* – Symmetrize affinity matrix. Standard t-SNE symmetrizes the interactions but when embedding new data, symmetrization is not performed.
- **n_jobs** *(int)* – The number of threads to use while running t-SNE. This follows the scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.
• **random_state** (*Union*[int, RandomState]) – If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by `np.random`.

**check_perplexity** (*perplexity*)

**set_perplexity** (*new_perplexity*)

Change the perplexity of the affinity matrix.

Note that we only allow lowering the perplexity or restoring it to its original value. This restriction exists because setting a higher perplexity value requires recomputing all the nearest neighbors, which can take a long time. To avoid potential confusion as to why execution time is slow, this is not allowed. If you would like to increase the perplexity above the initial value, simply create a new instance.

**Parameters**

- **new_perplexity** (*float*) – The new perplexity.

**to_new** (*data, perplexity=None, return_distances=False*)

Compute the affinities of new samples to the initial samples.

This is necessary for embedding new data points into an existing embedding.

Please see the Parameter guide for more information.

**Parameters**

- **data** (*np.ndarray*) – The data points to be added to the existing embedding.
- **perplexity** (*float*) – Perplexity can be thought of as the continuous k number of nearest neighbors, for which t-SNE will attempt to preserve distances.
- **return_distances** (*bool*) – If needed, the function can return the indices of the nearest neighbors and their corresponding distances.

**Returns**

- **P** (*array_like*) – An $N \times M$ affinity matrix expressing interactions between $N$ new data points the initial $M$ data samples.
- **indices** (*np.ndarray*) – Returned if `return_distances=True`. The indices of the $k$ nearest neighbors in the existing embedding for every new data point.
- **distances** (*np.ndarray*) – Returned if `return_distances=True`. The distances to the $k$ nearest neighbors in the existing embedding for every new data point.

**class** `openTSNE.affinity.MultiscaleMixture` (*data, perplexities, method='approx', metric='euclidean', metric_params=None, symmetrize=True, n_jobs=1, random_state=None*)

Calculate affinities using a Gaussian mixture kernel.

Instead of using a single perplexity to compute the affinities between data points, we can use a multiscale Gaussian kernel instead. This allows us to incorporate long range interactions.

Please see the Parameter guide for more information.

**Parameters**

- **perplexities** (*List[float]*) – A list of perplexity values, which will be used in the multiscale Gaussian kernel. Perplexity can be thought of as the continuous $k$ number of nearest neighbors, for which t-SNE will attempt to preserve distances.
- **method** (*str*) – Specifies the nearest neighbor method to use. Can be either exact or approx.
• **metric** (*str*) – The metric to be used to compute affinities between points in the original space.

• **metric_params** (*dict*) – Additional keyword arguments for the metric function.

• **symmetrize** (*bool*) – Symmetrize affinity matrix. Standard t-SNE symmetrizes the interactions but when embedding new data, symmetrization is not performed.

• **n_jobs** (*int*) – The number of threads to use while running t-SNE. This follows the scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.

• **random_state** (*Union*[int, RandomState]*) – If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by np.random.

**check_perplexities** (*perplexities*)

Check and correct/truncate perplexities.

If a perplexity is too large, it is corrected to the largest allowed value. It is then inserted into the list of perplexities only if that value doesn’t already exist in the list.

**set_perplexities** (*new_perplexities*)

Change the perplexities of the affinity matrix.

Note that we only allow lowering the perplexities or restoring them to their original maximum value. This restriction exists because setting a higher perplexity value requires recomputing all the nearest neighbors, which can take a long time. To avoid potential confusion as to why execution time is slow, this is not allowed. If you would like to increase the perplexity above the initial value, simply create a new instance.

**to_new** (*data*, *perplexities=None*, *return_distances=False*)

Compute the affinities of new samples to the initial samples.

This is necessary for embedding new data points into an existing embedding.

Please see the Parameter guide for more information.

**Parameters**

• **data** (*np.ndarray*) – The data points to be added to the existing embedding.

• **perplexities** (*List*[float]*) – A list of perplexity values, which will be used in the multiscale Gaussian kernel. Perplexity can be thought of as the continuous $k$ number of nearest neighbors, for which t-SNE will attempt to preserve distances.

• **return_distances** (*bool*) – If needed, the function can return the indices of the nearest neighbors and their corresponding distances.

**Returns**

• **P** (*array_like*) – An $N \times M$ affinity matrix expressing interactions between $N$ new data points the initial $M$ data samples.

• **indices** (*np.ndarray*) – Returned if return_distances=True. The indices of the $k$ nearest neighbors in the existing embedding for every new data point.

• **distances** (*np.ndarray*) – Returned if return_distances=True. The distances to the $k$ nearest neighbors in the existing embedding for every new data point.

**class** openTSNE.affinity.Multiscale (*data*, *perplexities*, *method='approx'*, *metric='euclidean'*, *metric_params=None*, *symmetrize=True*, *n_jobs=1*, *random_state=None*)

Calculate affinities using averaged Gaussian perplexities.
In contrast to MultiscaleMixture, which uses a Gaussian mixture kernel, here, we first compute single scale Gaussian kernels, convert them to probability distributions, then average them out between scales.

Please see the Parameter guide for more information.

Parameters

• `data (np.ndarray)` – The data matrix.
• `perplexities (List[float])` – A list of perplexity values, which will be used in the multiscale Gaussian kernel. Perplexity can be thought of as the continuous $k$ number of nearest neighbors, for which t-SNE will attempt to preserve distances.
• `method (str)` – Specifies the nearest neighbor method to use. Can be either `exact` or `approx`.
• `metric (str)` – The metric to be used to compute affinities between points in the original space.
• `metric_params (dict)` – Additional keyword arguments for the metric function.
• `symmetrize (bool)` – Symmetrize affinity matrix. Standard t-SNE symmetrizes the interactions but when embedding new data, symmetrization is not performed.
• `n_jobs (int)` – The number of threads to use while running t-SNE. This follows the scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.
• `random_state (Union[int, RandomState])` – If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by np.random.

```python
class openTSNE.affinity.FixedSigmaNN(data, sigma, k=30, method='approx', metric='euclidean', metric_params=None, symmetrize=True, n_jobs=1, random_state=None)
```

Compute affinities using using nearest neighbors and a fixed bandwidth for the Gaussians in the ambient space. Using a fixed Gaussian bandwidth can enable us to find smaller clusters of data points than we might be able to using the automatically determined bandwidths using perplexity. Note however that this requires mostly trial and error.

Parameters

• `data (np.ndarray)` – The data matrix.
• `sigma (float)` – The bandwidth to use for the Gaussian kernels in the ambient space.
• `k (int)` – The number of nearest neighbors to consider for each kernel.
• `method (str)` – Specifies the nearest neighbor method to use. Can be either `exact` or `approx`.
• `metric (str)` – The metric to be used to compute affinities between points in the original space.
• `metric_params (dict)` – Additional keyword arguments for the metric function.
• `symmetrize (bool)` – Symmetrize affinity matrix. Standard t-SNE symmetrizes the interactions but when embedding new data, symmetrization is not performed.
• `n_jobs (int)` – The number of threads to use while running t-SNE. This follows the scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.
• `random_state (Union[int, RandomState])` – If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance,
then it will be used as the random number generator. If the value is None, the random
number generator is the RandomState instance used by np.random.

to_new(data, k=None, sigma=None, return_distances=False)
Compute the affinities of new samples to the initial samples.
This is necessary for embedding new data points into an existing embedding.

Parameters
- `data` (*np.ndarray*) – The data points to be added to the existing embedding.
- `k` (*int*) – The number of nearest neighbors to consider for each kernel.
- `sigma` (*float*) – The bandwidth to use for the Gaussian kernels in the ambient space.
- `return_distances` (*bool*) – If needed, the function can return the indices of the
  nearest neighbors and their corresponding distances.

Returns
- `P` (*array_like*) – An $N \times M$ affinity matrix expressing interactions between $N$
  new data points the initial $M$ data samples.
- `indices` (*np.ndarray*) – Returned if `return_distances=True`. The indices of the $k$
  nearest neighbors in the existing embedding for every new data point.
- `distances` (*np.ndarray*) – Returned if `return_distances=True`. The distances to the
  $k$ nearest neighbors in the existing embedding for every new data point.

6.3 Callbacks

class openTSNE.callbacks.Callback

__call__(iteration, error, embedding)
This is the main method called from the optimization.

Parameters
- `iteration` (*int*) – The current iteration number.
- `error` (*float*) – The current KL divergence of the given embedding.
- `embedding` (*TSNEEmbedding*) – The current t-SNE embedding.

Returns `stop_optimization` – If this value is set to `True`, the optimization will be interrupted.

Return type `bool`

optimization_about_to_start()
This is called at the beginning of the optimization procedure.

class openTSNE.callbacks.ErrorLogger
Basic error logger.

This logger prints out basic information about the optimization. These include the iteration number, error and
how much time has elapsed from the previous callback invocation.
class openTSNE.sklearn.TSNE(n_components=2, perplexity=30, learning_rate=200,
early_exaggeration_iter=250, early_exaggeration=12, n_iter=750,
exaggeration=None, theta=0.5, n_interpolation_points=3,
min_num_intervals=50, ints_in_interval=1, initialization='pca', metric='euclidean', metric_params=None,
initial_momentum=0.5, final_momentum=0.8, min_grad_norm=1e-08, max_grad_norm=None, n_jobs=1, neighbors='approx',
negative_gradient_method='fft', callbacks=None, callbacks_every_iters=50, random_state=None)

t-Distributed Stochastic Neighbor Embedding.

Please see the Parameter guide for more information.

Parameters

- **n_components** *(int)* – The dimension of the embedding space. This defaults to 2 for easy visualization, but sometimes 1 is used for t-SNE heatmaps. t-SNE is not designed to embed into higher dimension and please note that acceleration schemes break down and are not fully implemented.

- **perplexity** *(float)* – Perplexity can be thought of as the continuous k number of nearest neighbors, for which t-SNE will attempt to preserve distances.

- **learning_rate** *(float)* – The learning rate for t-SNE optimization. Typical values range between 100 to 1000. Setting the learning rate too low or too high may result in the points forming a “ball”. This is also known as the crowding problem.

- **early_exaggeration_iter** *(int)* – The number of iterations to run in the early exaggeration phase.

- **early_exaggeration** *(float)* – The exaggeration factor to use during the early exaggeration phase. Typical values range from 12 to 32.

- **n_iter** *(int)* – The number of iterations to run in the normal optimization regime.

- **exaggeration** *(float)* – The exaggeration factor to use during the normal optimization phase. This can be used to form more densely packed clusters and is useful for large data sets.

- **theta** *(float)* – Only used when negative_gradient_method="bh" or its other aliases. This is the trade-off parameter between speed and accuracy of the tree approximation method. Typical values range from 0.2 to 0.8. The value 0 indicates that no approximation is to be made and produces exact results also producing longer runtime.

- **n_interpolation_points** *(int)* – Only used when negative_gradient_method="fft" or its other aliases. The number of interpolation points to use within each grid cell for interpolation based t-SNE. It is highly recommended leaving this value at the default 3.

- **min_num_intervals** *(int)* – Only used when negative_gradient_method="fft" or its other aliases. The minimum number of grid cells to use, regardless of the ints_in_interval parameter. Higher values provide more accurate gradient estimations.

- **ints_in_interval** *(float)* – Only used when negative_gradient_method="fft" or its other aliases. Indicates how large a grid cell should be e.g. a value of 3 indicates a grid side length of 3. Lower values provide more accurate gradient estimations.
• **initialization** (*Union[np.ndarray, str]*) – The initial point positions to be used in the embedding space. Can be a precomputed numpy array, *pca* or *random*. Please note that when passing in a precomputed positions, it is highly recommended that the point positions have small variance (var(Y) < 0.0001), otherwise you may get poor embeddings.

• **metric** (*str*) – The metric to be used to compute affinities between points in the original space.

• **metric_params** (*dict*) – Additional keyword arguments for the metric function.

• **initial_momentum** (*float*) – The momentum to use during the *early exaggeration* phase.

• **final_momentum** (*float*) – The momentum to use during the normal optimization phase.

• **min_grad_norm** (*float*) – If the gradient norm is below this threshold, the optimization will be stopped.

• **max_grad_norm** (*float*) – Maximum gradient norm. If the norm exceeds this value, it will be clipped. This is most beneficial when adding points into an existing embedding and the new points overlap with the reference points, leading to large gradients. This can make points “shoot off” from the embedding, causing the interpolation method to compute a very large grid, and leads to worse results.

• **n_jobs** (*int*) – The number of threads to use while running t-SNE. This follows the scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.

• **neighbors** (*str*) – Specifies the nearest neighbor method to use. Can be either *exact* or *approx*.

• **negative_gradient_method** (*str*) – Specifies the negative gradient approximation method to use. For smaller data sets, the Barnes-Hut approximation is appropriate and can be set using one of the following aliases: *bh*, *BH* or *barnes-hut*. For larger data sets, the FFT accelerated interpolation method is more appropriate and can be set using one of the following aliases: *fft*, *FFT* or *interpolation*.

• **callbacks** (*Union[Callable, List[Callable]]*) – Callbacks, which will be run every *callbacks_every_iters* iterations.

• **callbacks_every_iters** (*int*) – How many iterations should pass between each time the callbacks are invoked.

• **random_state** (*Union[int, RandomState]*) – If the value is an int, *random_state* is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by *np.random*.

**fit** (*X*, *y=None*)

Fit X into an embedded space.

**Parameters**

• **X** (*np.ndarray*) – The data matrix to be embedded.

• **y** (*ignored*) –

**fit_transform** (*X*, *y=None*)

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

**Parameters**

• **X** (*np.ndarray*) – The data matrix to be embedded.
• \textbf{y (ignored)} –

\textbf{Returns} Embedding of the training data in low-dimensional space.

\textbf{Return type} np.ndarray

\textbf{transform} \((X, *\text{args}, **\text{kwargs})\)

Apply dimensionality reduction to \(X\).

See openTSNE.TSNEEmbedding.transform() for additional parameters.

\textbf{Parameters} \(X (\text{np.ndarray})\) – The data matrix to be embedded.

\textbf{Returns} Embedding of the training data in low-dimensional space.

\textbf{Return type} np.ndarray

\textbf{class} openTSNE.TSNE \((n\_components=2, \ \text{perplexity}=30, \ \text{learning\_rate}=200, \ \text{early\_exaggeration\_iter}=250, \ \text{early\_exaggeration}=12, \ \text{n\_iter}=750, \ \text{exaggeration}=\text{None}, \ \text{theta}=0.5, \ \text{n\_interpolation\_points}=3, \ \text{min\_num\_intervals}=50, \ \text{ints\_in\_interval}=1, \ \text{initialization}='\text{pca}', \ \text{metric}='\text{euclidean}', \ \text{metric\_params}=\text{None}, \ \text{initial\_momentum}=0.5, \ \text{final\_momentum}=0.8, \ \text{min\_grad\_norm}=1e-08, \ \text{max\_grad\_norm}=\text{None}, \ \text{n\_jobs}=1, \ \text{neighbors}='\text{approx}', \ \text{negative\_gradient\_method}='\text{fft}', \ \text{callbacks}=\text{None}, \ \text{callbacks\_every\_iter}=50, \ \text{random\_state}=\text{None})\)

t-Distributed Stochastic Neighbor Embedding.

Please see the Parameter guide for more information.

\textbf{Parameters}

• \textbf{n\_components} \((\text{int})\) – The dimension of the embedding space. This defauls to 2 for easy visualization, but sometimes 1 is used for t-SNE heatmaps. t-SNE is not designed to embed into higher dimension and please note that acceleration schemes break down and are not fully implemented.

• \textbf{perplexity} \((\text{float})\) – Perplexity can be thought of as the continuous \(k\) number of nearest neighbors, for which t-SNE will attempt to preserve distances.

• \textbf{learning\_rate} \((\text{float})\) – The learning rate for t-SNE optimization. Typical values range between 100 to 1000. Setting the learning rate too low or too high may result in the points forming a “ball”. This is also known as the crowding problem.

• \textbf{early\_exaggeration\_iter} \((\text{int})\) – The number of iterations to run in the \textit{early exaggeration} phase.

• \textbf{early\_exaggeration} \((\text{float})\) – The exaggeration factor to use during the \textit{early exaggeration} phase. Typical values range from 12 to 32.

• \textbf{n\_iter} \((\text{int})\) – The number of iterations to run in the normal optimization regime.

• \textbf{exaggeration} \((\text{float})\) – The exaggeration factor to use during the normal optimization phase. This can be used to form more densely packed clusters and is useful for large data sets.

• \textbf{theta} \((\text{float})\) – Only used when negative\_gradient\_method=\"bh\" or its other aliases. This is the trade-off parameter between speed and accuracy of the tree approximation method. Typical values range from 0.2 to 0.8. The value 0 indicates that no approximation is to be made and produces exact results also producing longer runtime.

• \textbf{n\_interpolation\_points} \((\text{int})\) – Only used when negative\_gradient\_method=\"fft\" or its other aliases. The number of interpolation points to use within each grid cell for interpolation based t-SNE. It is highly recommended leaving this value at the default 3.
• **min_num_intervals** (*int*) – Only used when `negative_gradient_method="fft"` or its other aliases. The minimum number of grid cells to use, regardless of the `ints_in_interval` parameter. Higher values provide more accurate gradient estimations.

• **ints_in_interval** (*float*) – Only used when `negative_gradient_method="fft"` or its other aliases. Indicates how large a grid cell should be e.g. a value of 3 indicates a grid side length of 3. Lower values provide more accurate gradient estimations.

• **initialization** (*Union[np.ndarray, str]*) – The initial point positions to be used in the embedding space. Can be a precomputed numpy array, `pca` or `random`. Please note that when passing in a precomputed positions, it is highly recommended that the point positions have small variance (`\text{var}(Y) < 0.0001`), otherwise you may get poor embeddings.

• **metric** (*str*) – The metric to be used to compute affinities between points in the original space.

• **metric_params** (*dict*) – Additional keyword arguments for the metric function.

• **initial_momentum** (*float*) – The momentum to use during the *early exaggeration* phase.

• **final_momentum** (*float*) – The momentum to use during the normal optimization phase.

• **min_grad_norm** (*float*) – If the gradient norm is below this threshold, the optimization will be stopped.

• **max_grad_norm** (*float*) – Maximum gradient norm. If the norm exceeds this value, it will be clipped. This is most beneficial when adding points into an existing embedding and the new points overlap with the reference points, leading to large gradients. This can make points “shoot off” from the embedding, causing the interpolation method to compute a very large grid, and leads to worse results.

• **n_jobs** (*int*) – The number of threads to use while running t-SNE. This follows the scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.

• **neighbors** (*str*) – Specifies the nearest neighbor method to use. Can be either `exact` or `approx`.

• **negative_gradient_method** (*str*) – Specifies the negative gradient approximation method to use. For smaller data sets, the Barnes-Hut approximation is appropriate and can be set using one of the following aliases: `bh`, `BH` or `barnes-hut`. For larger data sets, the FFT accelerated interpolation method is more appropriate and can be set using one of the following aliases: `fft`, `FFT` or `interpolation`.

• **callbacks** (*Union[Callable, List[Callable]]*) – Callbacks, which will be run every `callbacks_every_iters` iterations.

• **callbacks_every_iters** (*int*) – How many iterations should pass between each time the callbacks are invoked.

• **random_state** (*Union[int, RandomState]*) – If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by `np.random`.

`fit(X)`

Fit a t-SNE embedding for a given data set.
Runs the standard t-SNE optimization, consisting of the early exaggeration phase and a normal optimization phase.

**Parameters**

- **X** (*np.ndarray*) – The data matrix to be embedded.

**Returns**

- A fully optimized t-SNE embedding.

**Return type** *TSNEEmbedding*

**prepare_initial (X)**

Prepare the initial embedding which can be optimized as needed.

**Parameters**

- **X** (*np.ndarray*) – The data matrix to be embedded.

**Returns**

- An unoptimized *TSNEEmbedding* object, prepared for optimization.

**Return type** *TSNEEmbedding*

**class openTSNE.TSNEEmbedding**

A t-SNE embedding.

Please see the Parameter guide for more information.

**Parameters**

- **embedding** (*np.ndarray*) – Initial positions for each data point.

- **affinities** (*Affinities*) – An affinity index which can be used to compute the affinities of new points to the points in the existing embedding. The affinity index also contains the affinity matrix $P$ used during optimization.

- **learning_rate** (*float*) – The learning rate for t-SNE optimization. Typical values range between 100 to 1000. Setting the learning rate too low or too high may result in the points forming a “ball”. This is also known as the crowding problem.

- **exaggeration** (*float*) – The exaggeration factor is used to increase the attractive forces of nearby points, producing more compact clusters.

- **momentum** (*float*) – Momentum accounts for gradient directions from previous iterations, resulting in faster convergence.

- **negative_gradient_method** (*str*) – Specifies the negative gradient approximation method to use. For smaller data sets, the Barnes-Hut approximation is appropriate and can be set using one of the following aliases: bh, BH or barnes-hut. For larger data sets, the FFT accelerated interpolation method is more appropriate and can be set using one of the following aliases: fft, FFT or interpolation.

- **theta** (*float*) – This is the trade-off parameter between speed and accuracy of the tree approximation method. Typical values range from 0.2 to 0.8. The value 0 indicates that no approximation is to be made and produces exact results also producing longer runtime.

- **n_interpolation_points** (*int*) – Only used when negative_gradient_method="fft" or its other aliases. The number of interpolation points to use within each grid cell for interpolation based t-SNE. It is highly recommended leaving this value at the default 3.

- **min_num_intervals** (*int*) – Only used when negative_gradient_method="fft" or its other aliases. The minimum number of grid cells to use, regardless of the ints_in_interval parameter. Higher values provide more accurate gradient estimations.

- **random_state** (*Union[int, RandomState]*) – The random state parameter follows the convention used in scikit-learn. If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used...
as the random number generator. If the value is None, the random number generator is the
RandomState instance used by np.random.

• **n_jobs** *(int)* – The number of threads to use while running t-SNE. This follows the
scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.

• **callbacks** *(Callable[[int, float, np.ndarray] -> bool]*) – Callbacks, which will be run every callbacks_every_iters iterations.

• **callbacks_every_iters** *(int)* – How many iterations should pass between each
time the callbacks are invoked.

• **optimizer** *(gradient_descent)* – Optionally, an existing optimizer can be used
for optimization. This is useful for keeping momentum gains between different calls to
optimize().

**kl_divergence**

The KL divergence or error of the embedding.

Type float

.. _kl_divergence:

**optimize** *(n_iter, inplace=False, propagate_exception=False, **gradient_descent_params)*)

Run optimization on the embedding for a given number of steps.

Please see the Parameter guide for more information.

Parameters

• **n_iter** *(int)* – The number of optimization iterations.

• **learning_rate** *(float)* – The learning rate for t-SNE optimization. Typical values
range between 100 to 1000. Setting the learning rate too low or too high may result in the
points forming a “ball”. This is also known as the crowding problem.

• **exaggeration** *(float)* – The exaggeration factor is used to increase the attractive
forces of nearby points, producing more compact clusters.

• **momentum** *(float)* – Momentum accounts for gradient directions from previous itera-
tions, resulting in faster convergence.

• **negative_gradient_method** *(str)* – Specifies the negative gradient approxima-
tion method to use. For smaller data sets, the Barnes-Hut approximation is appropriate
and can be set using one of the following aliases: bh, BH or barnes-hut. For larger
data sets, the FFT accelerated interpolation method is more appropriate and can be set
using one of the following aliases: fft, FFT or interpolation.

• **theta** *(float)* – This is the trade-off parameter between speed and accuracy of the tree
approximation method. Typical values range from 0.2 to 0.8. The value 0 indicates that
no approximation is to be made and produces exact results also producing longer runtime.

• **n_interpolation_points** *(int)* – Only used when
negative_gradient_method=”fft" or its other aliases. The number of in-
terpolation points to use within each grid cell for interpolation based t-SNE. It is highly
recommended leaving this value at the default 3.

• **min_num_intervals** *(int)* – Only used when
negative_gradient_method=”fft" or its other aliases. The minimum number
of grid cells to use, regardless of the ints_in_interval parameter. Higher values
provide more accurate gradient estimations.

• **inplace** *(bool)* – Whether or not to create a copy of the embedding or to perform
updates inplace.
• **propagate_exception**(bool) – The optimization process can be interrupted using callbacks. This flag indicates whether we should propagate that exception or to simply stop optimization and return the resulting embedding.

• **min_grad_norm**(float) – If the gradient norm is below this threshold, the optimization will be stopped.

• **max_grad_norm**(float) – Maximum gradient norm. If the norm exceeds this value, it will be clipped. This is most beneficial when adding points into an existing embedding and the new points overlap with the reference points, leading to large gradients. This can make points “shoot off” from the embedding, causing the interpolation method to compute a very large grid, and leads to worse results.

• **random_state**(Union[int, RandomState]) – The random state parameter follows the convention used in scikit-learn. If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by `np.random`.

• **n_jobs**(int) – The number of threads to use while running t-SNE. This follows the scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.

• **callbacks**(Callable[[int, float, np.ndarray] -> bool]) – Callbacks, which will be run every callbacks_every_iters iterations.

• **callbacks_every_iters**(int) – How many iterations should pass between each time the callbacks are invoked.

**Returns** An optimized t-SNE embedding.

**Return type** TSNEEmbedding

**Raises** OptimizationInterrupt – If a callback stops the optimization and the propagate_exception flag is set, then an exception is raised.

**prepare_partial**(X, initialization='median', k=25, **affinity_params)

Prepare a partial embedding which can be optimized.

**Parameters**

• **X**(np.ndarray) – The data matrix to be added to the existing embedding.

• **initialization**(Union[np.ndarray, str]) – The initial point positions to be used in the embedding space. Can be a precomputed numpy array, median, weighted or random. In all cases, median of weighted should be preferred.

• **k**(int) – The number of nearest neighbors to consider when initially placing the point onto the embedding. This is different from perplexity because perplexity affects optimization while this only affects the initial point positions.

• **affinity_params**(dict) – Additional params to be passed to the Affinities.to_new method. Please see individual Affinities implementations as the parameters differ between implementations.

**Returns** An unoptimized PartialTSNEEmbedding object, prepared for optimization.

**Return type** PartialTSNEEmbedding

**transform**(X, perplexity=5, initialization='median', k=25, learning_rate=0.1, early_exaggeration=2, early_exaggeration_iter=0, exaggeration=None, n_iter=250, initial_momentum=0.5, final_momentum=0.8, max_grad_norm=0.25)

Embed new points into the existing embedding.
This procedure optimizes each point only with respect to the existing embedding i.e. it ignores any interactions between the points in \( X \) among themselves.

Please see the Parameter guide for more information.

**Parameters**

- **\( X (\text{np.ndarray}) \)** – The data matrix to be added to the existing embedding.

- **perplexity (float)** – Perplexity can be thought of as the continuous \( k \) number of nearest neighbors, for which t-SNE will attempt to preserve distances. However, when transforming, we only consider neighbors in the existing embedding i.e. each data point is placed into the embedding, independently of other new data points.

- **initialization(\text{Union[\text{np.ndarray, str}]])** – The initial point positions to be used in the embedding space. Can be a precomputed numpy array, median, weighted or random. In all cases, median of weighted should be preferred.

- **\( k (\text{int}) \)** – The number of nearest neighbors to consider when initially placing the point onto the embedding. This is different from perplexity because perplexity affects optimization while this only affects the initial point positions.

- **learning_rate (float)** – The learning rate for t-SNE optimization. Typical values range between 100 to 1000. Setting the learning rate too low or too high may result in the points forming a “ball”. This is also known as the crowding problem.

- **early_exaggeration_iter (int)** – The number of iterations to run in the early exaggeration phase.

- **early_exaggeration (float)** – The exaggeration factor to use during the early exaggeration phase. Typical values range from 12 to 32.

- **\( n_{\text{iter}} (\text{int}) \)** – The number of iterations to run in the normal optimization regime.

- **exaggeration (float)** – The exaggeration factor to use during the normal optimization phase. This can be used to form more densely packed clusters and is useful for large data sets.

- **initial_momentum (float)** – The momentum to use during the early exaggeration phase.

- **final_momentum (float)** – The momentum to use during the normal optimization phase.

- **max_grad_norm (float)** – Maximum gradient norm. If the norm exceeds this value, it will be clipped. This is most beneficial when adding points into an existing embedding and the new points overlap with the reference points, leading to large gradients. This can make points “shoot off” from the embedding, causing the interpolation method to compute a very large grid, and leads to worse results.

**Returns** The positions of the new points in the embedding space.

**Return type** `PartialTSNEEmbedding`

class openTSNE.PartialTSNEEmbedding
A partial t-SNE embedding.

A partial embedding is created when we take an existing `TSNEEmbedding` and embed new samples into the embedding space. It differs from the typical embedding in that it is not possible to add new samples to a partial embedding and would generally be a bad idea.

Please see the Parameter guide for more information.

**Parameters**
• **embedding** (*np.ndarray*) – Initial positions for each data point.

• **reference_embedding** *(TSNEEmbedding)* – The embedding into which the new samples are to be added.

• **P** (*array_like*) – An $N \times M$ affinity matrix containing the affinities from each new data point $n$ to each data point in the existing embedding $m$.

• **learning_rate** (*float*) – The learning rate for t-SNE optimization. Typical values range between 100 to 1000. Setting the learning rate too low or too high may result in the points forming a “ball”. This is also known as the crowding problem.

• **exaggeration** (*float*) – The exaggeration factor is used to increase the attractive forces of nearby points, producing more compact clusters.

• **momentum** (*float*) – Momentum accounts for gradient directions from previous iterations, resulting in faster convergence.

• **negative_gradient_method** (*str*) – Specifies the negative gradient approximation method to use. For smaller data sets, the Barnes-Hut approximation is appropriate and can be set using one of the following aliases: bh, BH or barnes-hut. For larger data sets, the FFT accelerated interpolation method is more appropriate and can be set using one of the following aliases: fft, FFT or interpolation.

• **theta** (*float*) – This is the trade-off parameter between speed and accuracy of the tree approximation method. Typical values range from 0.2 to 0.8. The value 0 indicates that no approximation is to be made and produces exact results also producing longer runtime.

• **n_interpolation_points** (*int*) – Only used when **negative_gradient_method**="fft" or its other aliases. The number of interpolation points to use within each grid cell for interpolation based t-SNE. It is highly recommended leaving this value at the default 3.

• **min_num_intervals** (*int*) – Only used when **negative_gradient_method**="fft" or its other aliases. The minimum number of grid cells to use, regardless of the **ints_in_interval** parameter. Higher values provide more accurate gradient estimations.

• **random_state** *(Union[int, RandomState]*) – The random state parameter follows the convention used in scikit-learn. If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by np.random.

• **n_jobs** (*int*) – The number of threads to use while running t-SNE. This follows the scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.

• **callbacks** *(Callable[[int, float, np.ndarray] -> bool]*) – Callbacks, which will be run every **callbacks_every_iters** iterations.

• **callbacks_every_iters** (*int*) – How many iterations should pass between each time the callbacks are invoked.

• **optimizer** *(gradient_descent)* – Optionally, an existing optimizer can be used for optimization. This is useful for keeping momentum gains between different calls to optimize().

**kl_divergence**
The KL divergence or error of the embedding.

Type float
```python
optimize(n_iter, inplace=False, propagate_exception=False, **gradient_descent_params)
```

Run optimization on the embedding for a given number of steps.

**Parameters**

- **n_iter (int)** – The number of optimization iterations.
- **learning_rate (float)** – The learning rate for t-SNE optimization. Typical values range between 100 to 1000. Setting the learning rate too low or too high may result in the points forming a “ball”. This is also known as the crowding problem.
- **exaggeration (float)** – The exaggeration factor is used to increase the attractive forces of nearby points, producing more compact clusters.
- **momentum (float)** – Momentum accounts for gradient directions from previous iterations, resulting in faster convergence.
- **negative_gradient_method (str)** – Specifies the negative gradient approximation method to use. For smaller data sets, the Barnes-Hut approximation is appropriate and can be set using one of the following aliases: bh, BH or barnes-hut. For larger data sets, the FFT accelerated interpolation method is more appropriate and can be set using one of the following aliases: fft, FFT or interpolation.
- **theta (float)** – This is the trade-off parameter between speed and accuracy of the tree approximation method. Typical values range from 0.2 to 0.8. The value 0 indicates that no approximation is to be made and produces exact results also producing longer runtime.
- **n_interpolation_points (int)** – Only used when negative_gradient_method="fft" or its other aliases. The number of interpolation points to use within each grid cell for interpolation based t-SNE. It is highly recommended leaving this value at the default 3.
- **min_num_intervals (int)** – Only used when negative_gradient_method="fft" or its other aliases. The minimum number of grid cells to use, regardless of the ints_in_interval parameter. Higher values provide more accurate gradient estimations.
- **inplace (bool)** – Whether or not to create a copy of the embedding or to perform updates inplace.
- **propagate_exception (bool)** – The optimization process can be interrupted using callbacks. This flag indicates whether we should propagate that exception or to simply stop optimization and return the resulting embedding.
- **random_state (Union[int, RandomState])** – The random state parameter follows the convention used in scikit-learn. If the value is an int, random_state is the seed used by the random number generator. If the value is a RandomState instance, then it will be used as the random number generator. If the value is None, the random number generator is the RandomState instance used by np.random.
- **n_jobs (int)** – The number of threads to use while running t-SNE. This follows the scikit-learn convention, -1 meaning all processors, -2 meaning all but one, etc.
- **callbacks (Callable[[int, float, np.ndarray] -> bool])** – Callbacks, which will be run every callbacks_every_iters iterations.
- **callbacks_every_iters (int)** – How many iterations should pass between each time the callbacks are invoked.

**Returns**

An optimized partial t-SNE embedding.

**Return type** *PartialTSNEEmbedding*
Raises `OptimizationInterrupt` – If a callback stops the optimization and the `propagate_exception` flag is set, then an exception is raised.

```
exception openTSNE.OptimizationInterrupt (error, final_embedding)
```

Optimization was interrupted by a callback.

**Parameters**

- `error (float)` – The KL divergence of the embedding.
- `final_embedding (Union[TSNEEmbedding, PartialTSNEEmbedding])` – Is either a partial or full embedding, depending on where the error was raised.
Python Module Index

O

openTSNE, 65
openTSNE.affinity, 58
openTSNE.callbacks, 62
openTSNE.initialization, 57
openTSNE.sklearn, 63
Symbols

__call__() (openTSNE.callbacks.Callback method), 62

A
Affinities (class in openTSNE.affinity), 58

C
Callback (class in openTSNE.callbacks), 62
callbacks (openTSNE.callbacks method), 62
callbacks (module), 62

check_perplexities()
(checkTSNE.affinity.MultiscaleMixture method), 60

check_perplexity()
(checkTSNE.affinity.PerplexityBasedNN method), 59

E
ErrorLogger (class in openTSNE.callbacks), 62

F
fit () (openTSNE.sklearn.TSNE method), 64
fit() (openTSNE.TSNE method), 66
fit_transform() (openTSNE.sklearn.TSNE method), 64

FixedSigmaNN (class in openTSNE.affinity), 61

K
kl_divergence (openTSNE.affinity.MultiscaleMixture attribute), 71

kl_divergence (openTSNE.affinity. PerplexityBasedNN attribute), 68

M
Multiscale (class in openTSNE.affinity), 60
MultiscaleMixture (class in openTSNE.affinity), 59

O
openTSNE (module), 65
openTSNE.affinity (module), 58

openTSNE.callbacks (module), 62
openTSNE.initialization (module), 57
openTSNE.sklearn (module), 63

optimization_about_to_start()
(openTSNE.callbacks.Callback method), 62

OptimizationInterrupt, 73

optimize() (openTSNE. PartialTSNEEmbedding method), 71
optimize() (openTSNE. TSNEEmbedding method), 68

P
P (openTSNE.affinity. Affinities attribute), 58
PartialTSNEEmbedding (class in openTSNE), 70

pca() (in module openTSNE.initialize), 57

PerplexityBasedNN (class in openTSNE.affinity), 58

prepare_initial() (openTSNE.TSNE method), 67
prepare_partial() (openTSNE.TSNEEmbedding method), 69

R
random() (in module openTSNE.initialize), 57

S
set_perplexities()
(openTSNE.affinity.MultiscaleMixture method), 60

set_perplexity() (openTSNE.affinity. PerplexityBasedNN method), 59

T
to_new() (openTSNE.affinity. Affinities method), 58
to_new() (openTSNE.affinity. FixedSigmaNN method), 62
to_new() (openTSNE.affinity. MultiscaleMixture method), 60
to_new() (openTSNE.affinity. PerplexityBasedNN method), 59

transform() (openTSNE.sklearn.TSNE method), 65
transform() (openTSNE.TSNEEmbedding method), 69
TSNE (class in openTSNE), 65
TSNE (class in openTSNE.sklearn), 63
TSNEEmbedding (class in openTSNE), 67