Extending t-SNE

Megan Morrison and Benjamin Liu (Primary)
Department of Applied Mathematics
University of Washington

Abstract

t-distributed Stochastic Neighbor Embedding (t-SNE) is a data visualization method dimensionality reduction. In this paper we explore several variants of the standard t-SNE algorithm that aim to produce more accurate representations of the original data or improve the quality of clusters. We apply t-SNE to a Black Friday shopping dataset as well as MNIST. We use k-means clustering to determine the cluster quality that t-SNE produces.

1 Introduction

t-distributed Stochastic Neighbor Embedding (t-SNE) is a clustering algorithm for dimension reduction and visualization that maps data from a high-dimensional origin space to a low (two or three dimensional) embedding space. t-SNE is most commonly used to attempt to find patterns in data, particularly clusters. In this paper we investigate t-SNE's ability to cluster data with different characteristics, propose modifications to the algorithm, and test our modified algorithms on a Black Friday shopping dataset and the MNIST dataset.

2 Variants of the t-SNE algorithm

We implemented standard t-SNE based on the original paper by Maaten and Hinton [1] in Python using the PyTorch library. Using PyTorch we were able to implement gradient descent using autodifferentiation, which allowed us to rapidly explore variations on the standard t-SNE algorithm without having to compute analytic gradients.

2.1 Conditional t-SNE

The standard t-SNE algorithm defines the joint distribution P that measures similarity in the origin space in terms of a conditional distribution P_i,

$$p_{j|i} = \frac{\exp\left(-\|x_i - x_j\|_2^2 / 2\sigma_i^2\right)}{\sum_{k=1}^{n} \exp\left(-\|x_i - x_k\|_2^2 / 2\sigma_i^2\right)}. \tag{1}$$

The value of σ_i is determined for each data point x_i to achieve a user-specified *perplexity* value. The perplexity effectively determines 'how many neighbors' each point has, so that the characteristic radius of the point's neighborhood (given by σ_i) is small for tightly packed points and larger for more loosely packed points. A single joint distribution P is obtained from $p_{i,j} = (p_{j|i} + p_{i|j})/(2n)$.

The distribution Q that measures similarity in the embedding space is, in contrast, defined directly as a joint distribution,

$$q_{i,j} = \frac{\left(1 + \|x_i - x_j\|_2^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|x_k - x_l\|_2^2\right)^{-1}}. \tag{2}$$

Preprint. Work in progress.

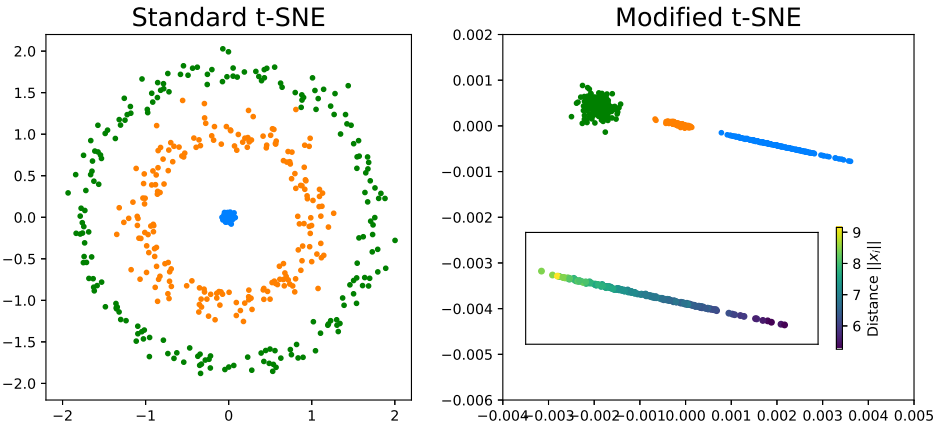


Figure 1: Comparison of standard t-SNE and a modified version of the algorithm. For original t-SNE, we set Perp $= 200$. For the modified version we used Perp $= 80$. The modified version produces an embedding that is qualitatively very different: the inset shows the points in the most tightly packed cluster colored in according to their distance from the center of the cluster.

This definition differs from that of P in two important ways: there is no intermediate conditional distribution, and there is no concept of the local point density.

We developed modified versions of the t-SNE algorithm where the target distribution Q is defined through an intermediate conditional distribution, Q_i that mirrors the definition of of P_i; this variant we call conditional t-SNE. We also developed a variant where Q re-uses the values of σ_i obtained when calculating the distribution P_i. These variants were also combined to form the "conditional–σ_i" variant. For full details see Appendix A.

Figure 1 shows embeddings produced by standard t-SNE and the conditional–σ_i variant on a synthetic data set consisting of three nested Gaussian clouds with standard deviations $1, 5, 25$. The embeddings have significant qualitative differences. Original t-SNE produces rings or annular arrangements of the data. The modified algorithm separates the three groups into a colinear arrangement. Examining the most tightly packed group, we see that the points are arranged by their proximity to the center of the cluster. The resulting clusters are clearly demarcated and superior to those of the standard t-SNE algorithm in terms of ease of visual separation.

2.2 Fat-tailed t-SNE

In standard t-SNE, the distribution used to measure similarity in the target space is the t-distribution with degrees of freedom $\nu = 1$ (also known as the Cauchy distribution). We modified standard t-SNE to allow for any number of degrees of freedom in the target distribution. Figure 2 shows plots of the clusterings resulting from setting $\nu_{\text{target}} = 30, 2, 1, 0.1, 0.01, 0.001$ on a data set consisting of 15 distinct clusters. The case where $\nu_{\text{target}} = 1$ corresponds to standard t-SNE. We see that values in the range of $0.01 \leq \nu_{\text{target}} \leq 0.1$ produce more easily distinguished clusters. Since the t-distribution increases in tail fatness as ν is decreased, we refer to this variant of t-SNE as "fat-tailed t-SNE".

3 MNIST results

We applied t-SNE and the four variations of the t-SNE algorithm detailed in Appendix A to a subset of the MNIST dataset consisting of $N = 2000$ points. We varied the set perplexity from Perp $= 50$ to Perp $= 200$ and performed k-means clustering with $k = 10$ clusters in order to find ten clusters, ideally corresponding to digits 0-9 (Fig. 3).

We found each cluster's domain and determined the percentage of correctly clustered data points by taking the average number of correctly clustered points over all regions (Fig. 4). The standard t-SNE algorithm allows a large spread for each cluster, blurring the boundaries between clusters (Fig. 3a, 4a), while the fat-tailed t-SNE algorithm forces each cluster to be more tightly packed while also

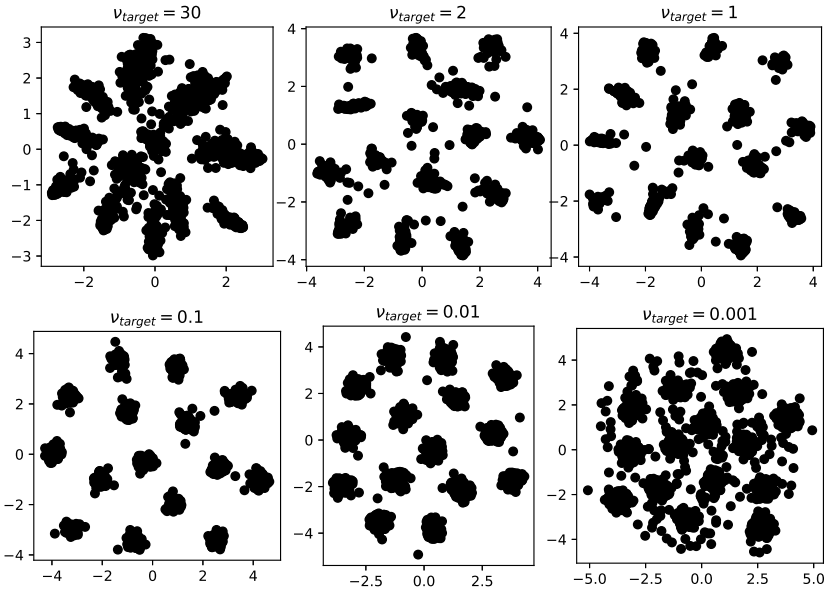
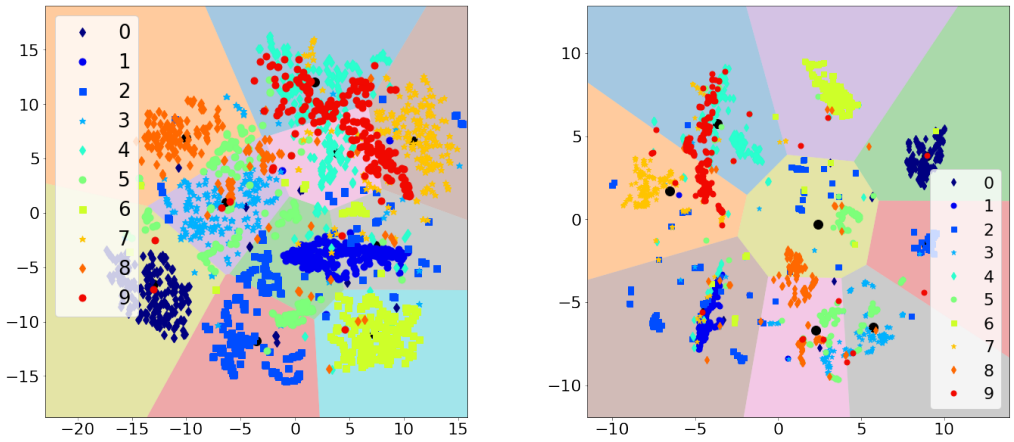


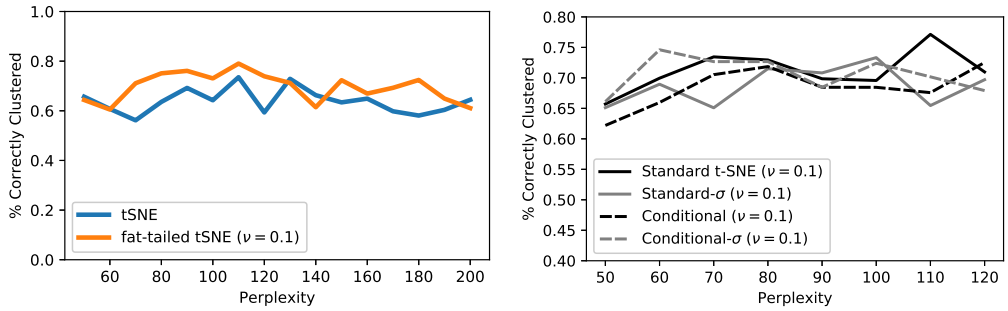
Figure 2: Comparison of t-SNE embeddings using various values of ν (fat-tailed variant). Characteristics of the resulting clusterings vary with ν: for ν very large the clusters pull together and for ν very small the clusters lose cohesion. The standard algorithm used $\nu = 1$, but from experimentation we can see that values of $\nu = 0.1, 0.01$ can yield superior results with respect to visual separation of clusters.

separating distinct clusters (Fig. 3b,4a). We found that fat-tailed t-SNE performed slightly better according to our k-means classification than standard t-SNE at most perplexity values (Fig. 4a). The four variations to the t-SNE algorithm appear to generate similar classification accuracy's across perplexity values (Fig. 4b). Because of the trial-to-trial variation, an average of many trials at each parameter setting would be needed in order to compare average classification quality.



(a) (b)

Figure 3: MNIST clusters using the standard t-SNE algorithm (a), and the fat-tailed t-SNE algorithm (b) at perplexity value $p = 110$. Fat-tailed t-SNE generates tighter clusters than standard t-SNE, leading to a higher classification accuracy when using k-means clustering to classify the low-dimensional data.



(a) (b)

Figure 4: K-means classification accuracy as a function of perplexity p, using standard t-SNE and fat-tailed t-SNE low-dimensional data (a), and using four variations of the fat-tailed t-SNE algorithm (b)

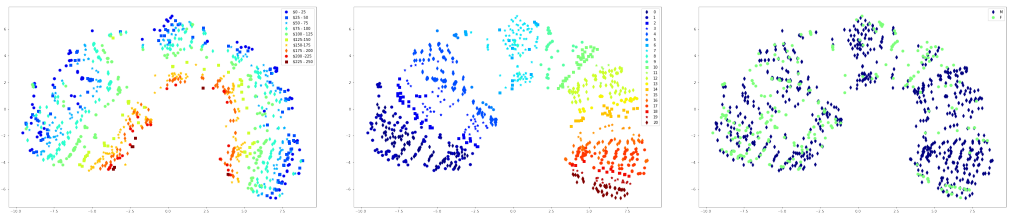
4 Black Friday results

We tested t-SNE, fat-tailed t-SNE, and the four variations of the t-SNE algorithm on the Black Friday shopping dataset in order to identify possible clusters of different types of shoppers. We were unable to find clusters. However, we found that two of the seven features (amount spend, and occupation level) were the primary determinants of distant between points in the high dimensional space and therefore drove the resulting distribution of points in the low-dimensional space (Fig. 5). This datset is not well suited for t-SNE analysis since some of the features are non-ordered, categorical data (e.g. city category, product category) and t-SNE uses the relative distances between points in the feature space to construct the resulting projection.

5 Conclusion

We developed several variants of the t-SNE algorithm in order to generate cleaner clusters for data that exists in high-dimensional spaces. Using these variants, we produced better clustering results on synthetic data sets where the clusters appeared more clearly visually separated. In the case of the conditional–σ_i variant, the resulting clusers were qualitatively very different from those produced by standard t-SNE, and enhanced different aspects of the underlying data.

We found that we could generate tighter clusters when embedding the MNIST dataset using our t-SNE modifications. We could not find meaningful clusters in the black Friday data, and believe this is because the data set contains many nominal or non-numeric fields.



(a) (b) (c)

Figure 5: Black Friday clusters using the standard t-SNE algorithm and labeled by amount spent (a), occupation level (b), and gender (c). Data points differentiate themselves according to spending and occupation level but not gender.

References

[1] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

A Variants of t-SNE

A.1 Conditional t-SNE

The "conditional" variant of t-SNE alters the definition of the distribution Q in the embedding space so that it is defined by an intermediate conditional distribution,

$$q_{j|i} = \frac{\left(1 + \|x_i - x_j\|_2^2\right)^{-1}}{\sum_{k \neq i}^{n} \left(1 + \|x_k - x_i\|_2^2\right)^{-1}} \tag{3}$$

and

$$q_{i,j} = \frac{q_{j|i} + q_{i|j}}{2n}.$$

This has the effect of mirroring the manner in which the distribution P is constructed in the origin space.

A.2 Conditional t-SNE with local point density σ_i

The "conditional–σ_i" variant of t-SNE extends the "conditional" variant of t-SNE by also incorporating the parameters σ_i that measure local point density in the origin space,

$$q_{i,j} = \frac{q_{j|i} + q_{i|j}}{2n} \qquad\qquad q_{j|i} = \frac{\left(1 + \|x_i - x_j\|_2^2 / \sigma_i^2\right)^{-1}}{\sum_{k \neq i}^{n} \left(1 + \|x_k - x_i\|_2^2 / \sigma_i^2\right)^{-1}}. \tag{4}$$

Each σ_i is tuned during the computation of P so that a fixed perplexity is achieved. The effect is that more tightly packed points receive a smaller value of σ_i, whereas more loosely packed points receive a larger value of σ_i. The "conditional–σ_i" variant re-uses these values when computing distance in the embedding space.

A.3 Fat-tailed t-SNE

Fat-tailed t-SNE adds an additional parameter $\nu > 0$ that controls the degree of fatness of the tails of the t-distribution used to calculate the distribution Q in the embedding space.

$$q_{i,j} = \frac{q_{j|i} + q_{i|j}}{2n} \qquad\qquad q_{j|i} = \frac{\left(1 + \|x_i - x_j\|_2^2 / (\nu\sigma_i^2)\right)^{-\left(\frac{\nu+1}{2}\right)}}{\sum_{k \neq i}^{n} \left(1 + \|x_k - x_i\|_2^2 / (\nu\sigma_i^2)\right)^{-\left(\frac{\nu+1}{2}\right)}}. \tag{5}$$

The parameter ν corresponds to the same parameter that is typically seen in Student's t-distribution. In classical statistics ν is often confined to positive integer values that correspond to sample sizes. For analytic reasons $\nu > 1/2$ is required for integrabiltiy of the probability density. In the present use case, this restriction can be relaxed to allow for $\nu > 0$ because the number of data points is finite.

B Python implementation of t-SNE (PyTorch)

```
import torch

def calc_pairwisesquareddists(X):
    """Calculate the pairwise squared-distances among all points in X
    Calculates ||x_i - x_j||_2^2 for each pair of rows x_i and x_j

    Args:
        X : Data in (Npts x d) PyTorch tensor.
    """

    tmp = torch.mm(X,X.transpose(0,1));
    selfdots = torch.diag(tmp);

    res = (selfdots.reshape(-1,1)+selfdots) - tmp - tmp.transpose(0,1);
```

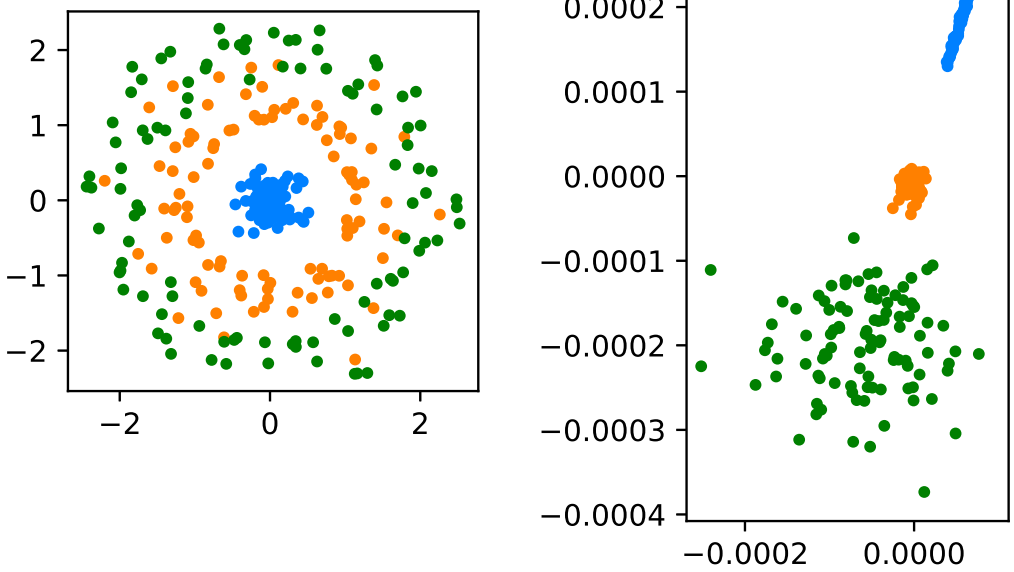
```
    return res;

def calc_Pjgiveni_fixedsigma2s(d2,s2,nu=float("Inf")):
    """Calculate p_{j|i} for specified values of sigma_i^2

    Args:
        X : Data in (Npts x d) PyTorch tensor.
        nu=Inf : Tail fatness of distribution used (Inf = Gaussian)
    """
    Npts = len(s2);

    if nu < float("Inf"):
        numerators=1./(1.+ (d2/s2.reshape(-1,1).repeat(1,N)/nu) )**((nu+1.)/2.)
        lognumerators=-(nu+1.)/2.*torch.log(1.+(d2/s2.reshape(-1,1).repeat(1,N)/nu));
    else:
        lognumerators = -d2 / (2.*s2.reshape(-1,1).repeat(1,Npts));
        numerators = torch.exp( lognumerators );

    # Zero out diagonal entries
    numerators = torch.triu(numerators,1) + torch.tril(numerators,-1);

    tmp = torch.sum(numerators,dim=1,keepdim=True); # Calculate row sums

    # Turn each row into a conditional distribution
    Pjgiveni = numerators / tmp.repeat(1,Npts)

    logPjgiveni2 = lognumerators - torch.log(tmp).repeat(1,Npts);

    prods = Pjgiveni*logPjgiveni2;
    perp = 2.**(-1.*torch.sum(prods,dim=1)/ torch.log(torch.tensor(2.)));
    return Pjgiveni, perp;

def calc_Pjgiveni(X,setPerp=200,s2_low=1e-4,s2_high=5e5,Niter=40,nu=float("Inf")):
    """Calculate p_{j|i} for a fixed perplexity value setPerp.
    Performs bisection method to find value of sigma_i for each data point
    so that p_{j|i} has specified perplexity value.

    Args:
        X : Data in (Npts x d) PyTorch tensor.
        setPerp=200 : User-specified perplexity
        nu=Inf : Tail fatness of distribution used (Inf = Gaussian)

        Niter : number of iterations of bisection method to perform
        s2_low : low value of sigma^2 to start bisection
        s2_high : high value of sigma^2 to start bisection
    """
    Npts = X.size()[0];

    d2 = calc_pairwisesquareddists(X);

    s2_low = torch.ones(Npts)*s2_low;
    s2_high = torch.ones(Npts)*s2_high;

    _,perps_low = calc_Pjgiveni_fixedsigma2s(d2,s2_low,nu=nu);
    _,perps_high = calc_Pjgiveni_fixedsigma2s(d2,s2_high,nu=nu);

    notlowenough = perps_low > setPerp;
    nothighenough = perps_high < setPerp;

    if torch.any(notlowenough):
        print('WARNING: sigma_low not low enough')
    if torch.any(nothighenough):
        print('WARNING: sigma_high not high enough')

    for nn in range(Niter):
```

```
        s2_mid = (s2_low + s2_high)*.5;
        Pjgiveni,perps_mid = calc_Pjgiveni_fixedsigma2s(d2,s2_mid,nu);


        gobig = perps_mid > setPerp;
        gobig = torch.tensor(gobig);
        s2_low[~gobig] = s2_mid[~gobig];
        s2_high[gobig] = s2_mid[gobig];

    return Pjgiveni, s2_mid, perps_mid;


def tsne(p,target_dim=2,eta=200.,Niter=100,s2=None,Y0=None,nu=1.,variant='original'):
    """Perform t-SNE to obtain data embeddings representing the origin distribution p

    Args:
        p : similarity distribution of original data
        target_dim=2 : dimension of target space to embed data into
        nu=1 : tail fatness to use in computation of target similarities
        variant='original' : variant of t-SNE to perform
        s2=None : sigma_i^2 values to use when computing a t-SNE variant
        Y0=None : initial condition to use for data embedding

        eta=200 : learning rate or step size for gradient descent
        Niter=100 : number of iterations to perform
    """

    nu = float(nu);
    eta = float(eta);

    N = p.size()[0];

    if Y0 is None:
        Y = torch.randn(N,target_dim)*1e-6;
    else:
        Y = Y0;

    if s2 is None:
        s2 = torch.ones(N);

    Y = torch.tensor(Y, requires_grad=True);

    Cs = torch.zeros(Niter);

    for nn in range(Niter):
        d2 = calc_pairwisesquareddists(Y);


        if variant == 'conditional_s2':
            numers = 1./ (1.+ (d2/s2.reshape(-1,1).repeat(1,N)/nu) )**((nu+1.)/2.)
            numers = torch.tril(numers,-1) + torch.triu(numers,1);
            rowsums = torch.sum(numers,dim=1);
            qjgiveni = numers / rowsums.reshape(-1,1).repeat(1,N);
            q = (qjgiveni + qjgiveni.transpose(0,1))/(2*N);
        elif variant == 'conditional':
            numers = 1./ (1.+ (d2/nu) )**((nu+1.)/2.)
            numers = torch.tril(numers,-1) + torch.triu(numers,1);
            rowsums = torch.sum(numers,dim=1);
            qjgiveni = numers / rowsums.reshape(-1,1).repeat(1,N);
            q = (qjgiveni + qjgiveni.transpose(0,1))/(2*N);
        elif variant == 'original_s2':
            numers = 1./ (1.+ (d2/s2.reshape(-1,1).repeat(1,N)/nu) )**((nu+1.)/2.)
            numers = torch.tril(numers,-1) + torch.triu(numers,1);
            totalmass = torch.sum(numers);
            q = numers / totalmass;
```

```
            q = (q + q.transpose(0,1))/2.;
        elif variant == 'original':
            numers = 1./ (1.+ (d2/nu) )**((nu+1.)/2.)
            numers = torch.tril(numers,-1) + torch.triu(numers,1);
            totalmass = torch.sum(numers);
            q = numers / totalmass;

        q = q + torch.eye(N); # avoid problems with autograd and nan

        tmp1 = p*torch.log(p);
        tmp1[p==0] = 0;

        tmp2 = p*torch.log(q);
        tmp2[q==0] = 0;
        tmp = tmp1 - tmp2;
        tmp = torch.triu(tmp,1) + torch.tril(tmp,-1);
        C = torch.sum(tmp);
        Cs[nn] = C.detach();

        C.backward();
        Ygrad = Y.grad;

        Y = Y - eta*Ygrad;
        Y = torch.tensor(Y.detach(),requires_grad=True)
    return Y,Cs
```

B.1 Example

```
import numpy as np
import matplotlib.pyplot as plt

Nincloud = 100;
origin_dim = 50;

torch.manual_seed(2018)

X = torch.zeros(0,origin_dim);
sds = [1., 50.];
sds = [1., 5., 25.];

for jj,sd in enumerate(sds):
    Xtmp = sd*torch.randn(Nincloud,origin_dim);
    X = torch.cat( (X,Xtmp), dim=0)

N = X.size()[0];

Pjgiveni,s2,perps = calc_Pjgiveni(X,setPerp=80.,Niter=40,s2_high=1e6,nu=float("Inf"));
p = (Pjgiveni + Pjgiveni.transpose(0,1)) / (2*N);

Y_orig,Cs    = tsne(p,target_dim=2,Niter=100,variant='original');
Y_conds2,Cs = tsne(p,target_dim=2,Niter=100,variant='conditional_s2', s2=s2);


Ys = [Y_orig, Y_conds2];

mycs = [[0,.5,1], [1,.5,0], [0,.5,0]];

plt.clf();

for ii,Y in enumerate(Ys):
    plt.subplot(1,2,1+ii);
    Yplot = Y.detach().numpy();
    for jj in range(len(sds)):
        res = np.arange(jj*Nincloud, (jj+1)*Nincloud);
        plt.plot(Yplot[res,0],Yplot[res,1],'.',color=mycs[jj]);
```


2
1
0
−1
−2
−2
0
2
0.0003
0.0002
0.0001
0.0000
−0.0001
−0.0002
−0.0003
−0.0004
−0.0002
0.0000