t-SNE Pseudocode

1. Overview

t-SNE algorithm optimizes the low-dimensional embedding (typically 2D), represented by Y here, by minimizing the KL-divergence between probability distributions p_{ij} derived from pairwise distance in the embedding space and the probability distributions q_{ij} derived from pairwise distances of the input data X. The probability distribution used in the embedding space is t-distribution while the probability distribution used in the input space is Gaussian distribution. Because t-distribution has heavier tails, it encourages the embedding to focus on preserving local distances. The optimization is performed through gradient descent, with momentum. The pseudocode and equation sections describe the complete algorithm, and you can follow the step-by-step guide in implementation. For test purposes, first download and unzip the data file `https://github.com/UTSW-Software-Engineering-Course-2022/module_1_materials/blob/main/day1/tsne_practice/mnist2500.zip`. Also download the tester script `https://github.com/UTSW-Software-Engineering-Course-2022/module_1_materials/blob/main/day1/tsne_practice/tsne.py` and include your implementation of t-SNE to see the visualization of performing the algorithm over MNIST.

1.1. Notations

- $X^{(n \times d)}$: n by d input matrix X (implemented as a numpy array)
- $Y^{(t)}$: embedding matrix Y at step t
- dY: the gradient of KL-divergence between P and Q w.r.t. Y, i.e. $dY = \frac{\partial c}{\partial Y}$ and c is KL-divergence
- $||y_i - y_j||^2$ indicate the L2 norm of vector $y_i - y_j$ (Euclidean distance squared between y_i and y_j)
- We indicate matrices with capital letters (e.g. X, Y, P, Q), and individual rows or entries of the matrix with lower case notations such as x_i, x_j, p_{ij}, q_{ij}

2. Algorithms

Algorithm 1 t-SNE

Input: Data Array $X^{(n \times d)}$
Parameters: no_dims=2, perplexity=30.0, initial_momentum=0.5, final_momentum=0.8, η=500, min_gain=0.01, T=1000
Output: low-dimensional data representation Array $Y^{(n \times no\_dims)}$

```
Precision(β) adjustment based on perplexity (algorithm 2 and code)
Compute pairwise affinities p_ij (equation 1 and note 5.1)
Early exaggerate (multiply) p_ij^(n×n) by 4 and clip the value to be at least 10^-12
Initialize low-dimensional data representation Array Y^(0) using first no_dims of PCs from PCA (code)
Initialize ΔY^(n×no_dims) = 0, gains^(n×no_dims) = 1
for t = 1 to T do
  Compute low-dimensional affinities q_ij (equation 2 and note 5.2) and clip the value to be at least 10^-12
  Compute gradient dY (equation 3 and note 5.3)
  if t < 20 then
    momentum = initial_momentum
  else
    momentum = final_momentum
  end if
  Determine gains based on the sign of dY and ΔY :
  gains = (gains + 0.2) * ((dY > 0) ≠ (ΔY > 0)) + (gains * 0.8) * ((dY > 0.) == (ΔY > 0.))
  Clip gains to be at least min_gain
  ΔY = momentum * ΔY − η * (gains * dY)
  Y = Y + ΔY
  if t == 100 then
    Remove early exaggeration via dividing p_ij^(n×n) by 4
  end if
end for
```

Algorithm 2 Precision(β) adjustment based on perplexity

β represents the inverse of the variance of the Gaussian distribution used for representing input data, i.e. $\beta = \frac{1}{2\sigma^2}$

This algorithm chooses β for each row of X so that the entropy of $p_{j|i}$ (equation 4) is close to log(perplexity)

Input: Data Array $X^{(n \times d)}$
Parameters: tol=1e-5, perplexity=30.0
Output: $p_{j|i}^{(n \times n)}, \beta^{(n \times 1)}$

Initialize $\beta^{(n \times 1)} = \mathbf{1}$, u=log(perplexity)
for $i = 1$ **to** n **do**
  betamin=$-\infty$,betamax=∞, tries = 0
  Compute h, $p_{j|i}^{(n \times n)}$ given x_i, β_i(equation 1 and 4)
  hdiff = h - u
  while |hdiff| $>$tol and tries < 50 **do**
    if hdiff > 0 **then**
      betamin = β_i
      if betamax=∞ or betamax=-∞ **then**
        $\beta_i = \beta_i * 2$
      else
        $\beta_i = (\beta_i$+betamax$)/2$
      end if
    else
      betamax = β_i
      if betamin=∞ or betamin=-∞ **then**
        $\beta_i = \beta_i / 2$
      else
        $\beta_i = (\beta_i$+betamin$)/2$
      end if
    end if
    Recompute h, $p_{j|i}^{(n \times n)}$ given x_i, β_i
    hdiff = h - u, tries += 1
  end while
  Set $p_{j|i}^{(n \times n)}$ given the finalized β_i
end for

3. Equations

$p_{j|i}$ is computed from input data X based on the Gaussian distribution (normalized to sum up to 1 across $p_{k|i}$ with all $k \neq i$ for the same i). Let p_{ij} be a symmetrized version of $p_{j|i}$ and $p_{i|j}$ (e.g. average of the two, then normalized to sum up to 1 over all i and j pairs where $i \neq j$)

$$p_{j|i} = \frac{exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{k\neq i} exp(-\|x_i - x_k\|^2/2\sigma_i^2)} \;,\; p_{ij} = \frac{p_{j|i} + p_{i|j}}{\sum\limits_{\substack{k,l \\ l\neq k}} (p_{k|l} + p_{l|k})} \tag{1}$$

q_{ij} computes the probability distribution based on t-distribution derived from the low-dimensional embeeding Y (normalized to sum up to 1 across q_{ij} withall i and j pairs where $i \neq j$)

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum\limits_{\substack{k,l \\ l\neq k}} (1 + \|y_k - y_l\|^2)^{-1}} \tag{2}$$

dY is the gradient of the loss function w.r.t. the embedding Y. This can be computed given Y, and q_{ij} computed from Y, and p_{ij} computed from input data X. The ith row of dY is

$$dY_i = \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1} \tag{3}$$

The variances of Gaussian distribution used to compute p are selected by making sure that entropies h are close to the specified perplexity. For each i,

$$h_i = -\sum_{j\neq i} p_{j|i} log(p_{j|i}) \tag{4}$$

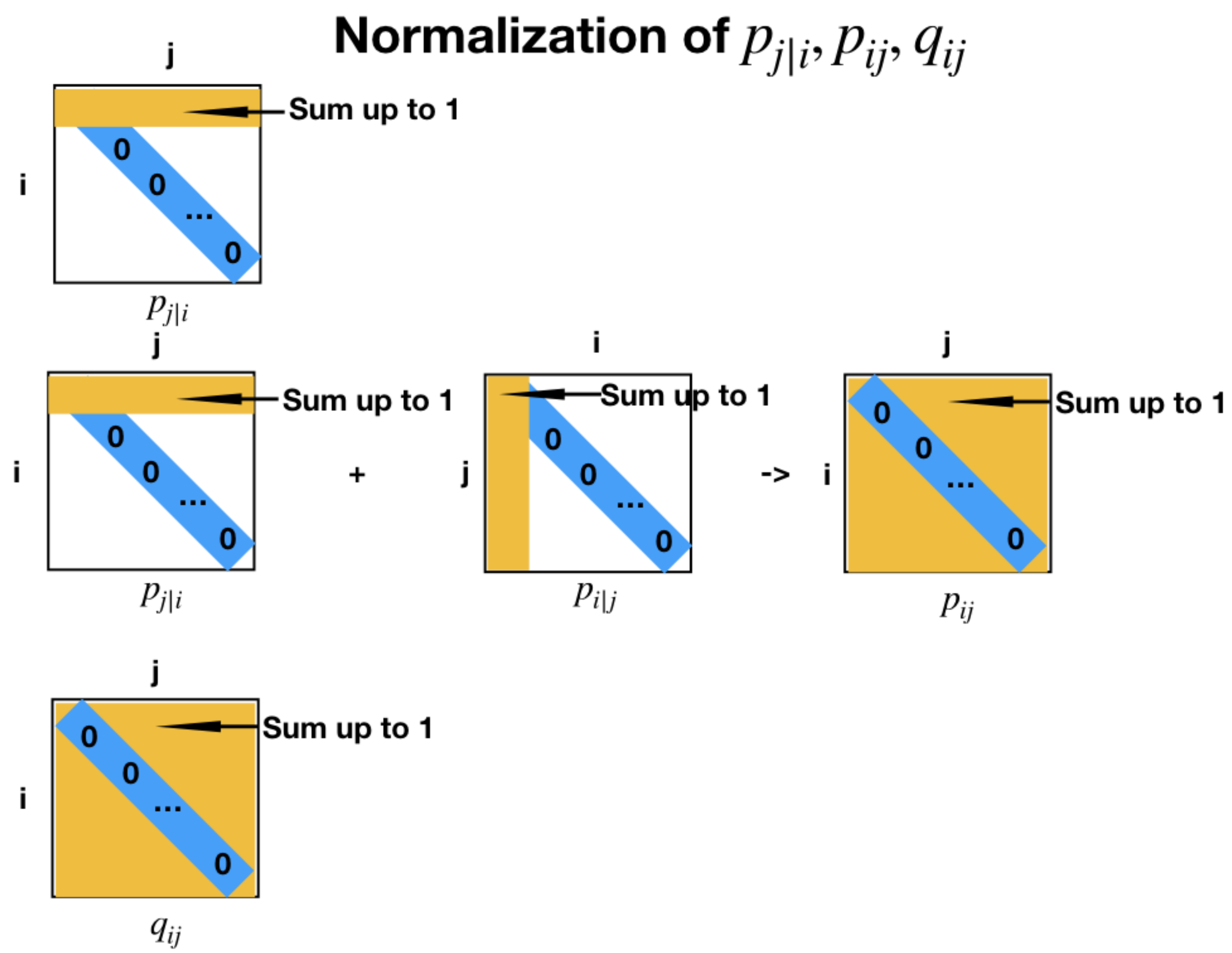
4. Step-by-step Guide

1. Implement computing distance matrix $D^{(n\times n)}$ from Data Array $X^{(n\times d)}$ and $p_{j|i}$ from $D^{(n\times n)}$ and β using equation 1 (Note $\beta = \frac{1}{2\sigma^2}$). Try to use operations on entire arrays (matrices) and avoid for loop in python wherever possible to improve efficiency of your code
2. Use the provided code or write your own according to algorithm 2 to adjust $\beta^{(n\times 1)}$ and calculate $p_{ij}^{(n\times n)}$ from finalized $p_{j|i}^{(n\times n)}$ and normalize it
3. Initialize Y using first no_dims of PCs from PCA (code)
4. Implement T iterations of Y update
    - Compute $q_{ij}^{(n\times n)}$ from Y. Thus we have $(p_{ij} - q_{ij})$ and gradient $dY^{(n\times no\_dims)}$. Follow algorithm 1 for default parameters and Y update

5. Implementation Tips

5.1. equation 1

- $\|x_i - x_j\|^2 = \|x_i\|^2 - 2x_i^T x_j + \|x_j\|^2$. This decomposition is useful for computing $\|x_i - x_j\|^2$ for all i and j simutaneously without for loop. A fast way to compute all $x_i^T x_j$ is to take dot product (**np.dot**) of matrix $X^{(n\times d)}$ and Transpose($X^{(n\times d)}$)
- Fill in probability($p_{j|i}^{(n\times n)}$) matrix row by row. Set the diagonal to be zeros by `P[np.arange(P.shape[0]), np.arange(P.shape[0])] = 0`
- Remember to normalize $p_{ij}^{(n\times n)}$ to sum up to 1 over the entire matrix, early exaggerate (multiply) it by factor of 4 and clip the value to be at least 10^{-12} before the 100-th iteration



5.2. equation 2

- Initialize low-dimensional data representation(Y) using first no_dims of PCs from PCA (code)
- Compute low-dimensional affinities($Q^{(n \times n)}$) using equation 2 and set the diagonal to be zeros. clip the value to be at least 10^{-12}

5.3. equation 3

- Compute gradient(dY) row by row using equation 3. You may use **np.tile** to repeat $(p_{ij} - q_{ij})$ and y_i or use numpy broadcasting (recommended) which bypasses the need to repeat to be more efficient (e.g. `Y[i,:][None,:]-Y`)
- Monitor KL-divergence of p_{ij} and q_{ij} every 10 iterations as a diagnostic measure

5.4. Debugging Checkpoints

You can verify each step of your implementation by comparing with the provided reference output below. Note that because PCA gives random sign to each PC, it is OK if your Y and dY differ from the provided output by only signs.

```
#beta
[[0.14508629]
[0.1286459 ]
[0.1658268 ]
...
```

```
[0.25783539]
[0.11305857]
[0.12134361]]

#p_ij (Note the diagonal values may be 4e-12 or 0 depending on your order of operations.
#All of them are ok and will not affect the final results)
[[1.00000000e-12 2.96283109e-09 3.91945187e-12 ... 9.51654682e-11
4.91379646e-10 7.00047775e-10]
[2.96283109e-09 1.00000000e-12 1.00000000e-12 ... 8.44553710e-12
4.20011851e-11 9.71642230e-10]
[3.91945187e-12 1.00000000e-12 1.00000000e-12 ... 9.67308857e-11
1.33251332e-08 5.02785520e-09]
...
[9.51654682e-11 8.44553710e-12 9.67308857e-11 ... 1.00000000e-12
1.45140069e-11 1.64668540e-09]
[4.91379646e-10 4.20011851e-11 1.33251332e-08 ... 1.45140069e-11
1.00000000e-12 2.68320022e-09]
[7.00047775e-10 9.71642230e-10 5.02785520e-09 ... 1.64668540e-09
2.68320022e-09 1.00000000e-12]]
```

Note that the p_ij provided here are after early exaggeration so it is 4x the original p_ij

```
#initial value of Y
[[ 0.61344587  1.37452188]
[ 5.00379081  1.94540396]
[ 0.31463237 -2.11658407]
...
[-3.52302175  4.1962009 ]
[ 0.81387035 -2.43970416]
[ 2.25717018  3.67177791]]

#q_ij (first iteration)
[[1.00000000e-12 7.64034225e-08 1.18549108e-07 ... 6.03703315e-08
1.00971265e-07 1.75292562e-07]
[7.64034225e-08 1.00000000e-12 3.98599937e-08 ... 1.99814301e-08
4.16569001e-08 1.36580156e-07]
[1.18549108e-07 3.98599937e-08 1.00000000e-12 ... 2.83199414e-08
1.16277857e-06 4.11193224e-08]
...
[6.03703315e-08 1.99814301e-08 2.83199414e-08 ... 1.00000000e-12
2.46537311e-08 4.53787118e-08]
[1.00971265e-07 4.16569001e-08 1.16277857e-06 ... 2.46537311e-08
1.00000000e-12 3.89280250e-08]
[1.75292562e-07 1.36580156e-07 4.11193224e-08 ... 4.53787118e-08
3.89280250e-08 1.00000000e-12]]

#dY (first iteration)
[[ 1.04351471e-04  1.01968819e-04]
[-4.90092162e-04  8.77456965e-05]
[-1.20622378e-04  2.47448040e-04]
...
[ 2.84159732e-04  1.26005087e-04]
[-1.75165952e-05 -4.43805766e-04]
[ 2.28990676e-05  1.13318128e-04]]
```