GCN Guide

Overview : This repository implements a **Graph Convolutional Network (GCN)** pipeline for classifying (and extracting features from) the Indian Pine hyperspectral dataset. The workflow consists of :

1. MATLAB data preparation:
   - Load raw hyperspectral TIFF images (Indian Pine site + train/test masks).
   - Normalize and reshape spectral data into a "bands × pixels" matrix.
   - Extract labeled pixels for training and testing.
   - Build a k-nearest-neighbors graph (heat-kernel weights) and compute the normalized graph Laplacian.
   - Save everything (ALL_X.mat, ALL_Y.mat, ALL_L.mat)for Python.

2. Python GCN training :
   - Load the precomputed MATLAB.mat files.
   - Convert integer labels to one-hot and build train/test masks.
   - Define a two-layer GCN (200→128→16) with masked cross-entropy + ℓ_2 regularization.
   - Train for 400 epochs, printing train/test accuracy every 50 epochs.
   - Extract final 16-dim embeddings for each pixel and save to features.mat.

This is the repository structure :

```
├── 19920612_AVIRIS_IndianPine_Site3.tif    # Raw hyperspectral data (Indian Pine,
AVIRIS)
├── IndianTR123_temp123.tif              # Training mask (single-band TIFF with class
labels)
├── IndianTE123_temp123.tif              # Testing mask (single-band TIFF with class
labels)
├── EuDist2.m                        # MATLAB: fast pairwise Euclidean distance
├── hyperConvert2d.m                     # MATLAB: reshape hyperspectral cube → 2D
├── constructW.m                      # MATLAB: build sparse adjacency matrix W via kNN
├── createLap.m                       # MATLAB: wrapper to compute [W, D, L]
├── GCN_DataPreparation.m                # MATLAB: end-to-end data prep & Laplacian
save
├── ALL_X.mat                         # Saved by MATLAB: features (nTotal×200)
├── ALL_Y.mat                         # Saved by MATLAB: labels (nTotal×1)
├── ALL_L.mat                         # Saved by MATLAB: normalized Laplacian
(nTotal×nTotal)
│
├── tf_utils.py                     # Python: mini-batch & helper functions
└── GCN.py                            # Python: main GCN training & feature extraction
```

TIFF files (`.tif`) are the raw inputs.

MATLAB scripts (`.m`) prepare data and compute graph‑Laplacian.

Saved **`.mat`** files are intermediate outputs used by Python.

Python scripts implement and train the GCN.

Functions :

1. *Hyperconvert2D.m* : This function converts a hyperspectral image cube or single-band image into a 2D matrix where each column is one pixel's spectral signature.

   Inputs : M3, an image of size `(m × n × p)` (hyperspectral cube) or `(m × n)` (single band).

   Output : M2, a 2D matrix. If `M3` was 3D, `M2` is `(p × (m·n))`. If M3 was 2D, M2 is `(1 × (m·n))`.

2. *EuDist2.m* : It Efficiently computes the (squared) Euclidean distance matrix between two sets of feature vectors using vectorized operations.

   Inputs : *fea_a*, an (n_axn) feature matrix.

   fea_b, an (n_bxn) feature matrix. If omitted, computes distances within fea_a.

   bsqrt , if `1` (default), returns the square root of the squared distances. If `0`, returns squared distances.

   Output : D , an (n_axn_b) matrix of distances (or n_axn_a if fea_b is omitted).

3. *constructW.m* : Build a sparse weighted adjacency matrix W for a graph using k-nearest-neighbors and a chosen weighting scheme (heat-kernel, cosine similarity, or binary).

   Inputs : *fea*, a nSamples×nFeatures data matrix (each row is one sample's feature vector).

   Options : a struct containing settings. Common fields :

   NeighbourMode = 'KNN'
   K = 10
   WeightMode = 'HeatKernel'
   t = 1 (heat-kernel σ)
   bSelfConnected = 0 (no self loops)

   Output : W , a sparse `nSamples×nSamples` adjacency matrix.

   W(i, j) = weight between sample i and j.

4. *createLap.m* : Compute adjacency matrix W, degree matrix D, and (unnormalized) Laplacian L = D-W from a feature matrix, by wrapping constructW.

Inputs : X: a bands×nPixels matrix (each column = one pixel). This function transposes it to (nPixels × bands).
K: integer, number of nearest neighbors (e.g. 10).
sigma: double, heat-kernel parameter (e.g. 1.0).

Output : W: sparse adjacency (n×n).
D: diagonal degree matrix (n×n), where D(i,i) = sum_j W(i,j).
L: unnormalized Laplacian (D – W).

5. *GCN_DataPreparation.m* :
   a. Read & normalize the Indian Pine HSI (200 bands).
   b. Read training/test masks and extract only labeled pixels.
   c. Build a kNN graph (k=10, σ=1) and compute the symmetrically normalized adjacency with self-loops: $A=I+D^{-1/2}WD^{-1/2}$
   d. Save three .mat files for Python to use:

      i. ALL_X.mat: variable ALL_X (nTotal × 200)
      ii. ALL_Y.mat: variable ALL_Y (nTotal × 1)
      iii. ALL_L.mat: variable ALL_L (nTotal × nTotal), sparse

Inputs : 19920612_AVIRIS_IndianPine_Site3.tif

IndianTR123_temp123.tif

IndianTE123_temp123.tif

Output : ALL_X.mat
ALL_Y.mat
ALL_L.mat

Python Function Reference :

tf_utils.py :

1. random_mini_batches_GCN(X, Y, L, mini_batch_size, seed) :

Purpose**:**
Create a list of random minibatches for training a GCN when the entire graph can be split into subgraphs of size mini_batch_size.

Inputs:

- X: (nSamples × nFeatures) feature matrix.
- Y: (nSamples × nClasses) label matrix (one-hot).
- L: (nSamples × nSamples) adjacency/Laplacian matrix.
- mini_batch_size: integer, size of each subgraph.
- seed: integer, seed for shuffling.

Outputs:

- mini_batches: list of tuples (mini_batch_X, mini_batch_Y, mini_batch_L), where each has shapes:
  - mini_batch_X: (mini_batch_size × nFeatures)
  - mini_batch_Y: (mini_batch_size × nClasses)
  - mini_batch_L: (mini_batch_size × mini_batch_size) sub-Laplacian.

2. random_mini_batches_GCN1(X, X1, Y, L, mini_batch_size, seed) :

Purpose:
Similar to random_mini_batches_GCN but for two feature matrices X and X1 (e.g., two modalities). Returns (mini_X, mini_X1, mini_Y, mini_L).

3. random_mini_batches(X1, X2, Y, mini_batch_size, seed) :

Purpose:
Create minibatches when you have two disjoint node sets with separate feature matrices X1 and X2. Batches are formed by shuffling X2 in blocks, while X1 and Y remain intact for each batch.

4. random_mini_batches_single(X1, Y, mini_batch_size, seed) :

Purpose:
Create minibatches for a single feature matrix X1 with labels Y. Each batch is a tuple (mini_batch_X1, mini_batch_Y).

5. convert_to_one_hot(Y, C) :

Purpose:
Convert an integer label vector Y (shape (nSamples, 1)) with values in [0..C-1] to a one-hot matrix of shape (C × nSamples). Often transposed to (nSamples × C) for GCN training.

Inputs:

- Y: integer array (nSamples, 1).
- C: integer, number of classes.

Output:

- A (C × nSamples) one-hot matrix.

6. sample_mask(idx, l) :

Purpose**:**

Create a length-`l` boolean mask with True at positions specified by idx. Used to indicate which nodes belong to the training or test set.

Inputs**:**

- idx: list or array of integer indices.
- l: integer, total length of the mask.

Output**:**

- Boolean array (l,) with True at each index in idx, False elsewhere.

GCN.py :

1. create_placeholders(n_x, n_y) :

Purpose**:**

Create a list of random minibatches for training a GCN when the entire graph can be split into subgraphs of size mini_batch_size.

Inputs:

- n_x: integer, feature dimension (200 in this project).
- n_y: integer, number of classes (16 in this project).

Outputs:

- x_in: tf.placeholder(tf.float32, [None, n_x], name="x_in")
- y_in: tf.placeholder(tf.float32, [None, n_y], name="y_in")
- lap: tf.placeholder(tf.float32, [None, None], name="lap")
- mask_tr: tf.placeholder(tf.float32, name="mask_train")
- mask_te: tf.placeholder(tf.float32, name="mask_test")

2. initialize_parameters() :

Purpose**:**

Initialize the two-layer GCN's weight matrices and biases using Xavier (Glorot) initializer and zeros for biases.

Outputs (dictionary):

- "x_w1": TensorFlow variable [200×128]
- "x_b1": TensorFlow variable [128]
- "x_w2": TensorFlow variable [128×16]
- "x_b2": TensorFlow variable [16]

3. GCN_layer(x_in, L_, weights):

Purpose**:** Single graph‑convolution operation.

Inputs:

- x_in: [nTotal, inFeatures] node features.
- L_: [nTotal, nTotal] adjacency or normalized Laplacian.
- weights: [inFeatures, outFeatures] weight matrix.

Outputs:

- x_out: [nTotal, outFeatures] aggregated result.

4. mynetwork(x, parameters, Lap) :

Purpose**:** Build the two‑layer GCN forward pass:

Layer 1: $Z1 = L \times (X\ W1) + b1$, $A1 = \mathrm{ReLU}(Z1)$

Layer 2: $Z2 = L \times (A1\ W2) + b2$ (logits)

Compute $l2\_loss = ||W1||^2 + ||W2||^2$

Inputs:

- x: [nTotal, 200] feature matrix.
- parameters: dict with keys 'x_w1', 'x_b1', 'x_w2', 'x_b2'.
- Lap: [nTotal, nTotal] adjacency/Laplacian matrix.

Outputs:

- z2: [nTotal, 16] logits (pre-softmax).
- l2_loss: scalar tensor representing ℓ_2 penalty.

5. mynetwork_optimaization(y_est, y_re, l2_loss, mask, reg, learning_rate, global_step) :

Purpose: Given logits, true labels, ℓ₂ penalty, and a mask, compute masked cross-entropy cost and create an Adam optimizer.

Inputs:

- y_est: [nTotal, nClasses] logits (Z2).
- y_re: [nTotal, nClasses] one-hot true labels.
- l2_loss: scalar ℓ₂ penalty.
- mask: [nTotal,] boolean or float mask indicating which nodes to include.
- reg: float, regularization coefficient.
- learning_rate: float.
- global_step: TensorFlow variable to track iteration count.

Outputs (tuple) :

- cost: scalar tensor (masked loss + reg term).
- optimizer: optimizer operation (minimize cost).

6. masked_accuracy(preds, labels, mask):

Purpose: Compute accuracy only on nodes indicated by the mask

Inputs:

- preds: [nTotal, nClasses] predicted logits or probabilities.
- labels: [nTotal, nClasses] one-hot true labels.
- mask: [nTotal,] boolean or float mask.

Outputs:

- Scalar mean accuracy over masked nodes.

7. train_mynetwork(x_all, y_all, L_all, mask_in, mask_out, learning_rate, beta_reg, num_epochs, print_cost):

Purpose: Build, train, and evaluate the two-layer GCN on the entire graph.

Inputs:

- x_all: (nTotal × 200) feature matrix (NumPy).
- y_all: (nTotal × 16) one-hot label matrix (NumPy).
- L_all: (nTotal × nTotal) adjacency/Laplacian (NumPy).
- mask_in: boolean array (nTotal,), True for training nodes.
- mask_out: boolean array (nTotal,), True for test nodes.
- learning_rate: (default=0.001)

- beta_reg: (default=0.001) ℓ_2 regularization factor.
- num_epochs: (default=400) number of epochs to train.
- print_cost: (default=True) whether to print metrics every 50 epochs.

Outputs:

- parameters: dict of trained weights & biases (x_w1, x_b1, x_w2, x_b2).
- None: placeholder (unused).
- features_all: (nTotal × 16) NumPy array of final embeddings (logits from second layer).

8. Main function :

a. Load MATLAB data

b. Build Masks :

Training nodes: indices 0..694.

Test nodes: indices 696..10365.

(Note: index 695 is not used.)

c. Convert Labels to One-Hot

d. Call train_mynetwork

e. Save Final Embeddings :

features.mat contains the learned node embeddings as a (nTotal × 16) matrix

Training Results & Evaluation Metric :

Below are the recorded training and validation metrics at regular 50‑epoch intervals. “Train_loss” and “Val_loss” refer to the masked cross‑entropy loss (including ℓ_2 regularization) on the training and test masks, respectively. “Train_acc” and “Val_acc” correspond to the masked node‑classification accuracy on the training and test sets.

This is because in a full‑graph GCN, all nodes share the same adjacency and feature matrix. We only want the gradients from training nodes to drive learning, and only want to measure accuracy on test nodes. A raw “accuracy over all nodes” would be meaningless if some nodes are never used for supervision.

```
epoch   0:  Train_loss=3.9567, Val_loss=3.4077, Train_acc=0.0604, Val_acc=0.2479

epoch  50:  Train_loss=1.7786, Val_loss=1.7717, Train_acc=0.4504, Val_acc=0.4972

epoch 100:  Train_loss=1.5599, Val_loss=1.5569, Train_acc=0.5237, Val_acc=0.5036

epoch 150:  Train_loss=1.4240, Val_loss=1.4215, Train_acc=0.5683, Val_acc=0.5539

epoch 200:  Train_loss=1.3023, Val_loss=1.3001, Train_acc=0.6532, Val_acc=0.6017

epoch 250:  Train_loss=1.1929, Val_loss=1.1907, Train_acc=0.6906, Val_acc=0.6229

epoch 300:  Train_loss=1.0999, Val_loss=1.0982, Train_acc=0.7108, Val_acc=0.6449

epoch 350:  Train_loss=1.0307, Val_loss=1.0297, Train_acc=0.7281, Val_acc=0.6672

epoch 400:  Train_loss=0.9787, Val_loss=0.9777, Train_acc=0.7540, Val_acc=0.6749
```