EECS 182 Deep Neural Networks
Fall 2022 Anant Sahai

Homework 4

This homework is due on Saturday, October 1, 2022, at 10:59PM.

1. Implementing RNNs and LSTMs

This problem involves filling out the **RNN_Implementation_and_Gradients.ipynb** notebook.

(A) **Implement Section 1A in the notebook**, which constructs a vanilla RNN layer. This layer implements the function

$$h_t = \sigma(W^h h_{t-1} + W^x x_t + b)$$

where W^h, W^x, and b are learned parameter matrices, x is the input sequence, and σ is a nonlinearity such as tanh. The RNN layer "unrolls" across a sequence, passing a hidden state between timesteps and returning an array of hidden states at all timesteps.

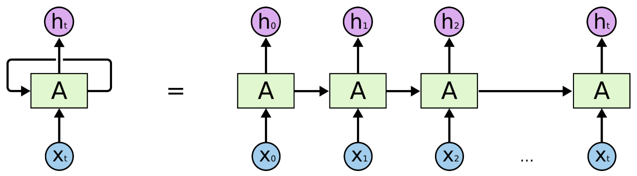


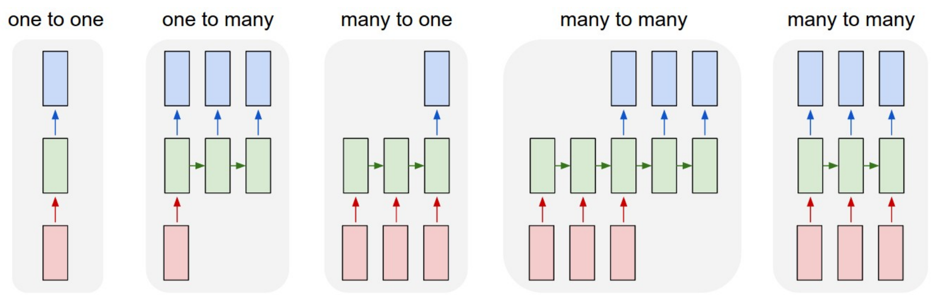
Figure 1: Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

(B) **Implement Section 1.B of the notebook**, in which you'll use this RNN layer in a regression model by adding a final linear layer on top of the RNN outputs.

$$\hat{y}_t = W^f h_t + b^f$$

We'll compute one prediction for each timestep.

(C) RNNs can be used for many kinds of prediction problems, as shown below. In this notebook we will look at many-to-one prediction and aligned many-to-many prediction.



We will use a simple averaging task. The input X consists of a sequence of numbers, and the label y is a running average of all numbers seen so far.

We will consider two tasks with this dataset:

- Task 1: predict the running average at all timesteps
- Task 2: predict the average at the last timestep only

Implement Section 1.C in the notebook, in which you'll look at the synthetic dataset shown and implement a loss function for the two problem variants.

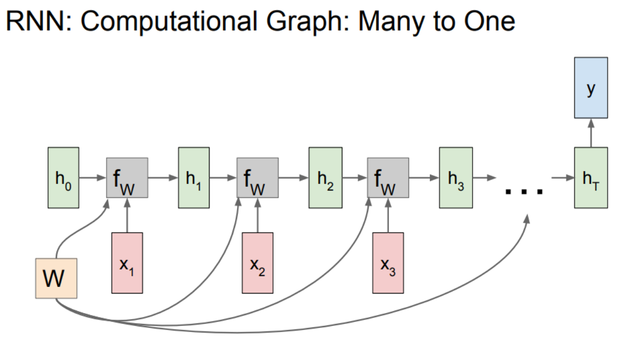


Figure 2: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

(D) Consider an RNN which outputs a single prediction at timestep T. As shown in Figure 2, each weight matrix W influences the loss by multiple paths. As a result, the gradient is also summed over multiple paths:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial h_T}\frac{\partial h_T}{\partial W} + \frac{\partial \mathcal{L}}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial W} + \ldots + \frac{\partial \mathcal{L}}{\partial h_1}\frac{\partial h_1}{\partial W} \tag{1}$$

When you backpropagate a loss through many timesteps, the later terms in this sum often end up with either very small or very large magnitude - called vanishing or exploding gradients respectively. Either problem can make learning with long sequences difficult.

Implement Notebook Section 1.D, which plots the magnitude at each timestep of $\frac{\partial \mathcal{L}}{\partial h_t}$. Play around with this visualization tool and try to generate exploding and vanishing gradients.

(E) **If the network has no nonlinearities, under what conditions would you expect the exploding or vanishing gradients with for long sequences? Why?** (Hint: it might be helpful to write out the formula for $\frac{\partial \mathcal{L}}{\partial h_t}$ and analyze how this changes with different t). **Do you see this pattern empirically using the visualization tool in Section 1.D in the notebook** with last_step_only=True?

Solution: If we use the MSE loss on a single example (x, y), the gradient $\frac{\partial \mathcal{L}}{\partial h_t} = 2(\hat{y}-y)W^f(W^h)^{T-t}$. (To clarify, the exponents f and h are matrix indicators, but $t-i$ is an exponent.) If the magnitude of the largest eigenvalue of W^h is much greater than 1, the gradient will explode, and if it's much less than 1, the gradient will start to vanish. Gradients are only stable when the largest eigenvalue magnitude is close to 1.

We see the expected pattern empirically.

(F) Compare the magnitude of hidden states and gradients when using ReLU and tanh nonlinearities in Section 1.D in the notebook. **Which activation results in more vanishing and exploding gradients? Why?** (This does not have to be a rigorous mathematical explanation.)

Solution: Hidden states: tanh restricts hidden state values to (-1, 1), so hidden state magnitudes remain small. With ReLU, hidden state values can easily explode.

Gradients: When tanh inputs are large, gradients are close to zero. This results in fewer exploding gradients but more vanishing gradients. Exploding gradients are still possible, however, when the largest eigenvalue of W_h has magnitude > 1, but the hidden states remain close to zero. ReLU activations, in contrast, result in frequent exploding hidden state sizes and gradients, similar to in the no-activation RNN.

(G) **What happens if you set last_target_only = False in Section 1.D in the notebook? Explain why this change affects vanishing gradients. Does it help the network's ability to learn dependencies across long sequences?** (The explanation can be intuitive, not mathematically rigorous.)

Solution: For every timestep k, the model's prediction produces loss $\mathcal{L}_k$. The gradient $\partial L_k/\partial h_k$ is high-magnitude, resulting in high-magnitude logged gradients in the visualization tool, but for any timestep $t << k$, $\partial L_k/\partial h_t$ will still be small. This means the network will still struggle to pass gradients over long sequences, making it hard to learn long-range dependencies.

(H) **Implement Section 1.8 of the notebook** in which you implement a LSTM layer. LSTMs pass a cell state between timesteps as well as a hidden state. **Explore gradient magnitudes using the visualization tool you implemented earlier and report on the results. Solution:** Hidden state outputs remain between +/- 1. Gradients don't explode, but they still vanish. Typically, they don't vanish as fast as vanilla RNNs.

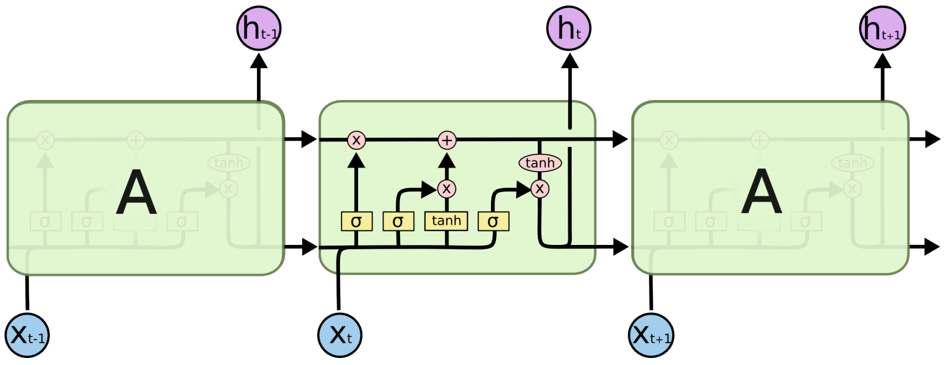


Figure 3: Image source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

The LSTM forward pass is shown below:

$$
\begin{aligned}
f_t &= \sigma(x_t U^f + h_{t-1} W^f + b^f) \\
i_t &= \sigma(x_t U^i + h_{t-1} W^i + b^i) \\
o_t &= \sigma(x_t U^o + h_{t-1} W^o + b^o) \\
\tilde{C}_t &= \tanh(x_t U^g + h_{t-1} W^g + b^g) \\
C_t &= f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \\
h_t &= \tanh(C_t) \circ o_t
\end{aligned}
$$

where $\circ$ represents the Hadamard Product (elementwise multiplication) and σ is the sigmoid function.

(I) When using an LSTM, you should still see vanishing gradients, but the gradients should vanish less quickly. **Interpret why this might happen by considering gradients of the loss with respect to the cell state.** (Hint: consider computing $\frac{\partial \mathcal{L}}{\partial C_{T-1}}$ using the terms $\partial \mathcal{L}, \partial C_T, \partial C_{T-1}, \partial h_T, \partial h_{T-1}$).

Solution: In an LSTM, information can be stored in the cell state without needing to persist through a chain of matrix multiplications.

$$\frac{\partial \mathcal{L}}{\partial C_{T-1}} = \frac{\partial \mathcal{L}}{\partial h_T}\left(\frac{\partial h_T}{\partial C_T}\frac{\partial C_T}{\partial C_{T-1}} + \frac{\partial h_T}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial C_{T-1}}\right)$$

Unlike $\frac{\partial h_T}{\partial h_{T-1}}$, which results in decaying gradients due to matrix multiplication, $\frac{\partial C_T}{\partial C_{T-1}}$ is a diagonal matrix with f_t on the diagonal. If the network sets f_t close to 1, then $\frac{\partial C_T}{\partial C_{T-1}}$ will be close to the identity, allowing information stored in the cell state to "pass through" without decay. In practice, however, many elements of f_t are not close to 1, which results in some gradient decay.

In addition, LSTMs also contain a hidden state in which to store information, and this "pathway" through the network will decay like in a vanilla RNN, potentially resulting in some overall gradient decay.

(J) Consider a ResNet with simple resblocks defined by $h_{t+1} = \sigma(W_t h_t + b_t) + h_t$. **Draw a connection between the role of a ResNet's skip connections and the LSTM's cell state in facilitating gradient propagation through the network.**

Solution: ResNet skip connections and LSTM cell states both allow information to pass through the network without requiring a matrix multiplication at each layer. As a result, they both mitigate vanishing gradients.

(K) We can create multi-layer recurrent networks by stacking layers as shown in Figure 4. The hidden state outputs from one layer become the inputs to the layer above.

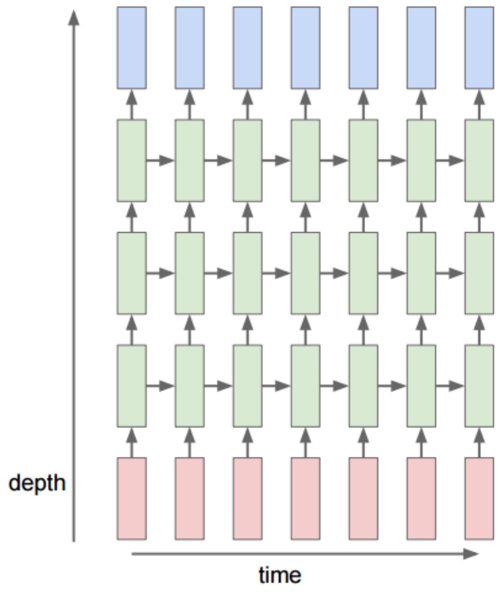


Figure 4: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

Implement notebook Section 1.K and run the last cell to train your network. You should be able to reach training loss < 0.001 for the 2-layer networks, and <.01 for the 1-layer networks.

2. Running RNNs and LSTMs

Implement all the TODOs in Real_RNNs_LSTMs.ipynb.

A. Implementing the network

B. Preprocessing the dataset

C. Training the model
D. Using the LSTM model

3. Directed and Undirected Graphs

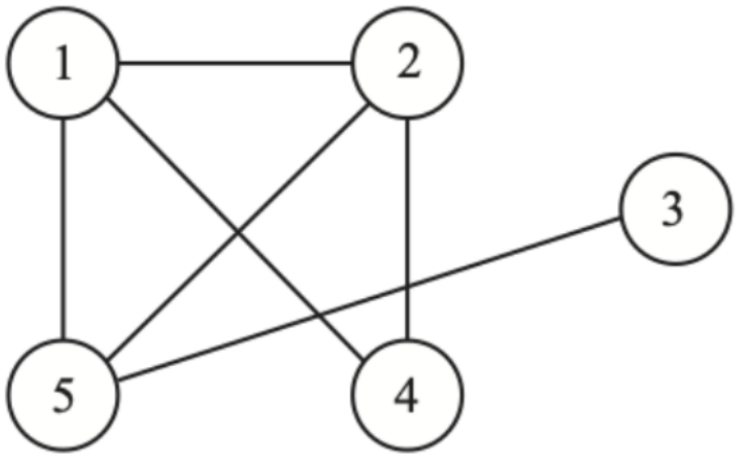


Figure 5: Simple Undirected Graph

Figure 5 shows a simple undirected graph whose adjacency matrices we want to make sure you can write down. Generally, an unnormalized adjacency matrix between the nodes of a directed or undirected graph is given by:

$$A_{i,j} = \begin{cases} 1 : \text{if there is an edge between node i and node j,} \\ 0 : \text{otherwise.} \end{cases} \tag{2}$$

This will be a symmetric matrix for undirected graphs. For a directed graph, we have:

$$A_{i,j} = \begin{cases} 1 : \text{if there is an edge from node i to node j,} \\ 0 : \text{otherwise.} \end{cases} \tag{3}$$

This need not to be symmetric for a directed graph, and is in fact typically not a symmetric matrix when we are thinking about directed graphs (otherwise, we'd probably be thinking of them as undirected graphs).

Similarly, the degree matrix of an undirected graph is a diagonal matrix that contains information about the degree of each vertex. In other words, it contains the number of edges attached to each vertex and it is given by:

$$D_{i,j} = \begin{cases} deg(v_i) : \text{if i == j,} \\ 0 : \text{otherwise.} \end{cases} \tag{4}$$

where the degree $\deg(v_i)$ of a vertex counts the number of times an edge terminates at that vertex.

For directed graphs, the degree matrix could be *In-Degree* when we count the number of edges coming into a particular node and *Out-Degree* when we count the number of edges going out of the node. We'll use the terms in-degree matrix or out-degree matrix to make it clear which one we are invoking.

Sometimes, imbalanced weights may undesirably affect the matrix spectrum (eigenvalues and eigenvectors). This occurs when a vertex with a large degree results in a large diagonal entry in the Laplacian matrix dominating the matrix properties. To solve that issue, a normalization scheme is applied which aims to make the influence of such vertices more equal to that of other vertices, by dividing the entries of the Adjacency matrix by the vertex degrees.

In that sense, a normalized adjacency matrix is given by:

$$A^{Normalized} = AD^{-1} \tag{5}$$

and a symmetrically normalized adjacency matrix is given by

$$A^{SymNorm} = D^{-1/2}AD^{-1/2} \tag{6}$$

Additionally, the Laplacian matrix relates many useful properties of a graph. In fact, the spectral decomposition of the Laplacian matrix of a graph allows for the construction of low-dimensional embeddings that appear in many machine learning applications. In other words, there is a relation between the properties of a graph and the spectra (eigenvalues and eigenvectors) of matrices associated with the graph, such as its adjacency matrix or Laplacian matrix.

Given a simple graph G with n vertices $v_1, ..., v_n$, its unnormalized Laplacian matrix $L_{n\times n}$ is defined element-wise as:

$$L_{i,j} = \begin{cases} deg(v_i) : \text{if i == j}, \\ -1 : \text{if i != j and } v_i \text{ is adjacent to } v_j, \\ 0 : \text{otherwise.} \end{cases} \tag{7}$$

or equivalently by the matrix:

$$L = D - A \tag{8}$$

where D is the degree matrix and A is the adjacency matrix of the graph.

We could also compute the symmetrically normalized Laplacian which is inherited from the adjacency matrix normalization scheme as shown below:

$$L^{SymNorm} = I - A^{SymNorm} \tag{9}$$

where I is the identity matrix, A is the unnormalized adjacency matrix, and L is the unnormalized Laplacian.

A **Show that $L^{SymNorm}$ could also be written as**:

$$L^{SymNorm} = D^{-1/2}LD^{-1/2} \tag{10}$$

where D is the dregree matrix, and L is the unnormalized Laplacian.

Solution:

$$\begin{aligned} L^{SymNorm} &= I - A^{SymNorm} \\ &= I - D^{-1/2}AD^{-1/2} \\ &= DD^{-1} - D^{-1/2}AD^{-1/2} \\ &= D^{-1/2}DD^{-1/2} - D^{-1/2}AD^{-1/2} \\ &= D^{-1/2}(D-A)D^{-1/2} \\ &= D^{-1/2}LD^{-1/2} \end{aligned} \tag{11}$$

B **Write the unnormalized adjacency A, the degree matrix, D, and the symmetrically normalized adjacency matrix, $A^{SymNorm}$, of the graph in Figure. 5.**

Solution: $A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$ $D = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{bmatrix}$ $A^{SymNorm} = \begin{bmatrix} 0 & 1/3 & 0 & 1/\sqrt{6} & 1/3 \\ 1/3 & 0 & 0 & 1/\sqrt{6} & 1/3 \\ 0 & 0 & 0 & 0 & 1/\sqrt{3} \\ 1/\sqrt{6} & 1/\sqrt{6} & 0 & 0 & 0 \\ 1/3 & 1/3 & 1/\sqrt{3} & 0 & 0 \end{bmatrix}$

C **Write the symmetrically normalized Laplacian matrix of the graph in Figure. 5.**

Solution: $L^{SymNorm} = \begin{bmatrix} 1 & -1/3 & 0 & -1/\sqrt{6} & -1/3 \\ -1/3 & 1 & 0 & -1/\sqrt{6} & -1/3 \\ 0 & 0 & 1 & 0 & -1/\sqrt{3} \\ -1/\sqrt{6} & -1/\sqrt{6} & 0 & 1 & 0 \\ -1/3 & -1/3 & -1/\sqrt{3} & 0 & 1 \end{bmatrix}$

D **Compute A^2, A^3**

Solution: $A^2 = \begin{bmatrix} 3 & 2 & 1 & 1 & 1 \\ 2 & 3 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 2 & 2 \\ 1 & 1 & 0 & 2 & 3 \end{bmatrix}$ $A^3 = \begin{bmatrix} 4 & 5 & 1 & 5 & 6 \\ 5 & 4 & 1 & 5 & 6 \\ 1 & 1 & 0 & 2 & 3 \\ 5 & 5 & 2 & 2 & 2 \\ 6 & 6 & 3 & 2 & 2 \end{bmatrix}$

We now want to estimate the traffic flow of inner downtown Berkeley and we know the road network shown below. The goal of the estimation is to estimate the traffic flow on each road segment. The flow estimates should satisfy the conservation of vehicles exactly at each intersection as indicated by the arrows.

The intersections are labeled a to h. The road segments are labeled 1 to 22. The arrows indicate the direction of traffic.

Hint: think about how you might represent the road network in terms of matrices, vectors, etc.

E **Write the unnormalized adjacency matrix of the graph in Figure 6.**

Solution:

A = $\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

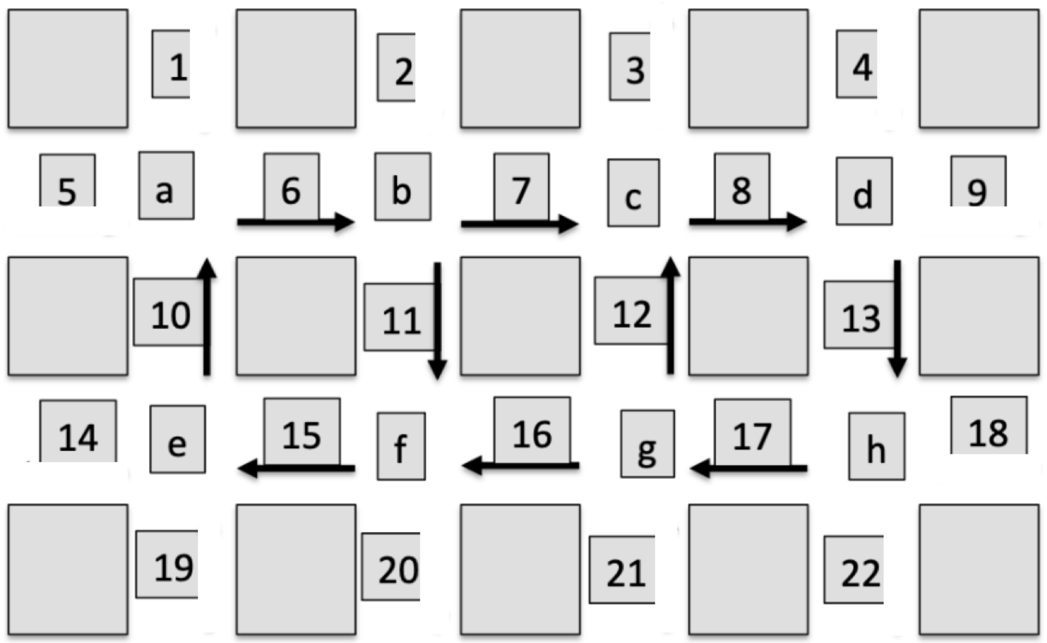


Figure 6: Simple Directed Graph

F **Write the In-degree D_{in} and Out-degree D_{out} matrix of the graph in Figure. 6.**

Solution: $$D_{in} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} D_{out} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

G **Write both of the symmetrically normalized In-degree and Out-degree Laplacian matrix of the graph in Figure. 6.**

$$L_{in}^{SymNorm} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1/\sqrt{2} & 0 & 0 & -1/\sqrt{2} & 0 & 0 \\ 0 & 0 & 1 & -1/\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1/\sqrt{2} & 1 & 0 & 0 \\ 0 & 0 & -1/\sqrt{2} & 0 & 0 & -1/\sqrt{2} & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

Solution:

$$L_{out}^{SymNorm} = \begin{bmatrix} 1 & -1/\sqrt{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1/\sqrt{2} & 0 & 0 & -1/\sqrt{2} & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1/\sqrt{2} & 0 & 0 & -1/\sqrt{2} & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1/\sqrt{2} & 1 \end{bmatrix}$$

H ***[Optional]*** It is good to Read `https://arxiv.org/pdf/1609.02907.pdf` and `https://distill.pub/2021/understanding-gnns/` to observe the importance of the Adjacency and Laplacian matrices in graph representation.

4. Graph Dynamics

Some graph neural network methods operate on the full adjacency matrix. Others, such as those discussed here `https://distill.pub/2021/gnn-intro/`, at each layer apply the same local operation to each node based on inputs from its neighbors.

This problem is designed to:

- show connections between these methods.
- show that for a positive integer k, the matrix A^k has an interesting interpretation. That is, the entry in row i and column j gives the number of walks of length k (i.e., a collection of k edges) leading from vertex i to vertex j.

To do this, let's consider a very simple deep linear network that is built on an underlying graph with n vertices. In the 0-th layer, each node has a single input with weight 1 that is fed a one-hot encoding of its own identity — so node i in the graph has a direct input which is an $n-$dimensional vector that has a 1 in position i and 0s in all other positions. You can view these as n channels if you want.

The weights connecting node i in layer k to node j in layer $k+1$ are simply 1 if vertices i and j are connected in the underlying graph and are 0 if those vertices are not connected in the underlying graph. At each layer, the operation at each node is simply to sum up the weighted sum of its inputs and to output the resulting n-dim vector to the next layer. You can think of these as being depth-wise operations if you'd like.

A Let A be the $n \times n$ size adjacency matrix for the underlying graph where the entry $A_{i,j} = 1$ if vertices i and j are connected in the graph and 0 otherwise. **Write the output of the j-th node at layer k in this network in terms of the matrix A.**

(Hint: This output is an n-dimensional vector since there are n output channels at each layer.)

Solution: A_j^k

Explanation: Think of the initial graph as a matrix of length n, where every node is a row. The initial graph is just the identity.

At each timestep, we can compute the next timestep of the graph by taking $G^{t+1} = AG_t$. This works b/c every row j in G^{t+1} is produced by summing the rows of G_t specified by the 1s in A_j - i.e. the rows corresponding to the neighbors of node j.

If we do this repeatedly, we get that the value at node j is $(A^k G_0)_j = A^k_j$.

B Here is some helpful notation: Let $V(i)$ be the set of vertices that are connected to vertex i in the graph. Let $L_k(i, j)$ be the number of distinct paths that go from vertex i to vertex j in the graph where the number of edges traversed in the path is exactly k. Recall that a path from i to j in a graph is a sequence of vertices that starts with i, ends with j, and for which every successive vertex in the sequence is connected by an edge in the graph. The length of the path is 1 less than the number of vertices in the corresponding sequence. **Show that the i-th output of node j at layer k in the network above is the count of how many paths there are from i to j of length k**, where by convention there is exactly 1 path of length 0 that starts at each node and ends up at itself.

Solution: **Proof**: We can prove the result by induction on k. For k = 1, the result follows from the very definition of A. Let $L_k(i, j)$ denote the number of paths of length k between nodes i and j, and assume that the result we wish to prove is true for some given $h \geq 1$, so that $L_h(i, j) = [A_h]_{i,j}$.

We next prove that it must also hold that $L_{h+1}(i, j) = [A_{k+1}]_{i,j}$, thus proving by inductive argument that $L_k(i, j) = [A_k]_{i,j}$ for all $k \geq 1$.

Indeed, to go from a node i to a node j with a walk of length h + 1, one needs first reach, with a walk of length h, a node l linked to j by an edge. Thus:

$$L_{h+1}(i, j) = \Sigma_{l \in V(j)} L_h(i, l) \tag{12}$$

where V(j) is the neighbor set of j, which is the set of nodes connected to the j-th node, that is, nodes l such that $A_{l,j} \neq 0$. Thus:

$$L_{h+1}(i, j) = \Sigma_{l=1}^{n} L_h(i, l) A_{l,j} \tag{13}$$

But we assumed that $L_h(i, j) = [A_h]_{i,j}$, hence the previous equation can be written as:

$$L_{h+1}(i, j) = \Sigma_{l=1}^{n} [A_h]_{i,l} A_{l,j} \tag{14}$$

In the above we recognize the (i,j)-th element of the product $A^h A = A_{h+1}$, which proves that $L_{h+1}(i, j)$ = $[A_{h+1}]_{i,j}$, and hence concludes the inductive proof.

C The structure of the neural network in this problem is compatible with a straightforward linear graph neural network since the operations done (just summing) are locally permutation-invariant at the level of each node and can be viewed as essentially doing the exact same thing at each vertex in the graph based on inputs coming from its neighbors. This is called "aggregation" in the language of graph neural nets. In the case of the computations in previous parts, **what is the update function that takes the aggregated inputs from neighbors and results in the output for this node?**

Solution: If we represent the graph as a matrix with a node value in each row, the update function is to multiply by A.

If we represent the graph as a set of nodes, the update function is

$v_j = \sum_{i \in V(j)} \text{v}_i$

D The simple GNN described in the previous parts counts paths in the graph. If we were to replace sum aggregation with max aggregation, **what is the interpretation of the outputs of node j at layer k?**

Solution: It is 1 if there is a path from i to j of length k and 0 otherwise.

5. The power of the graph perspective in clustering

Implement all the TODOs in the the_power_of_graph_perspective_in_clustering.ipynb notebook and answer the written questions below

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

from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

from scipy.linalg import svd
from scipy.spatial import distance
from sklearn.preprocessing import normalize

def get_data(n_samples, seed):
    T_matrix = [[-0.60834549, -0.63667341], [0.40887718, 0.85253229]]
    X_orig, y_orig = make_blobs(n_samples=n_samples, random_state=170)
    X = np.dot(X_orig, T_matrix)
    return X

def show_data_results(X, num_plot=2, y_pred=None, cmap='jet'):
    if num_plot==1:
        plt.scatter(X[:, 0], X[:, 1])
        plt.title ("input␣data")
    elif num_plot==2:
        try:
            assert y_pred is not None
            fig = plt.figure(figsize=(10, 3))
            ax1 = fig.add_subplot(121)
            ax1.scatter(X[:, 0], X[:, 1])
            ax1.set(xticks=[],yticks=[],title ="input␣data")

            ax2 = fig.add_subplot(122)
            ax2.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=cmap)
            ax2.set(xticks=[], yticks=[], title ="clustered␣data")
        except:
            print('y_pred␣is␣required␣for␣2␣plots')

n_samples, seed = 2000, 170
X = get_data(n_samples, seed)
```

A Given.

Solution:

```
y_pred = KMeans(n_clusters=3, random_state=seed).fit_predict(X)
show_data_results(X, num_plot=1)
```

B We used the KMeans algorithm implementation of sklearn, and showed our attempt to cluster this dataset into 3 classes. **Comment on the output the KMeans algorithm? Did it work? If so explain why, if not, explain not.**

Solution: As could be seen above, it did not work. Note that the provided dataset is not linearly separable and the Kmeans clustering algorithm minimizes within-cluster variances (squared Euclidean distances), but not regular Euclidean distances.

This algorithm works by finding centroids and looks at points around each centroid inside a given radius to determine which points correspond to which cluster.

Even if we try giving the kmeans algorithm the correct means of each cluster directly, it still will classify the points in the wrong way because as given, they have the wrong embedding. In that sense, each point votes for its cluster in an isolated way.

Kmeans works using centroids which is the representation of the data center and each point is assigned to the nearest centroid.

C As given, the data points in our dataset are represented simply with their 2D Cartesian coordinates. Let's now interpret every single point as a node in a graph. Our goal is to find a way to relate every node in the graph in such way that the points that are closer together and points that are far apart maintain that relationship explicitly.

That is, we will choose to look at every point in the dataset as a vertex in a graph where the edge connection between two vertexes is determined by the weighted distances between them. Write a function that takes in the input dataset and some coefficient gamma and returns the adjacency matrix A. **Is this a directed or an undirected graph?**

$$A_{i,j} = e^{-\gamma||x_i - x_j||^2} \tag{15}$$

where x_i and x_j represent each point in the provided dataset, γ is positive. You may find the $dictance$ module from $scipy.spatial$ useful.

Solution: please, see notebook for the function; undirected

```
def get_adjacency_matrix(gamma, X):
    return np.exp(- gamma * distance.cdist(X, X,
    metric='sqeuclidean'))
```

D The degree matrix of an undirected graph is a diagonal matrix that contains information about the degree of each vertex. In other words, it contains the number of edges attached to each vertex and it is given by Eq (4) in problem 3. Note that in the traditional definition of the adjacency matrix, this boils down to the diagonal matrix in which elements along the diagonals are the column-wise sum of the elements in the adjacency matrix. Using the same idea, **write a function that takes in the adjacency matrix as an argument and returns the degree matrix.**

Solution: please, see notebook for the function

```
def get_degree_matrix(adjacency_matrix):
    degree_matrix = np.sum(adjacency_matrix, axis=1)
    return degree_matrix
```

E Using γ = 7.5, compute the adjacency matrix A, degree matrix D and the symmetrically normalized adjacency matrix matrix M,

$$M = A^{SymNorm} = D^{-1/2}AD^{-1/2} \tag{16}$$

Solution: Please, see for computation

```
gamma, num_clusters = 7.5, 3
adjacency_matrix = get_adjacency_matrix(gamma, X)
degree_matrix = get_degree_matrix(adjacency_matrix)
M = np.multiply(np.sqrt(1/degree_matrix)[np.newaxis, :],
                np.multiply(adjacency_matrix,
                            np.sqrt(1/degree_matrix)[:, np.newaxis]))
```

Note that another interpretation of the matrix M is that it shows the probability of moving/jumping from one node to another.

F Using SVD decomposition, **write a function that selects the top 3 vectors (corresponding to the highest singular values) in the matrix U and performs the same KMeans clustering used above on them; show the plots. What do you observe? Did it work? If so explain why, if not, explain not.**

Intuition: By selecting the top 3 vectors of the U matrix, we are selecting a new representation of the data points which could be seen as a construction of a low dimension embedding of the data points as mentioned in problem 3.

Solution: while it did better than the first try, it is still not separating the upper 2 clusters properly

```
def cluster_data(M):
    U = svd(M, full_matrices=False, lapack_driver='gesvd')[0]
    Usub = U[:, :num_clusters]
    y_pred = KMeans(n_clusters=num_clusters,
    random_state=seed).fit_predict(Usub)
    return Usub, y_pred

y_pred_spectral = cluster_data(M)[1]
show_data_results(X, 2, y_pred_spectral)
```

G Now let's think of the symmetrically normalized adjacency matrix obtained above as the transition matrix in of a Markov Chain. That is, it represents the probability of jumping from one node to another. In order to fully interpret M in such way, it needs to be a proper stochastic matrix which means that the sum of the elements in each column must add up to 1. **Write a function that takes in the matrix M and returns M_{stoch}, the stochastic version of M; compute the stochastic matrix.**

Solution: please, see notebook for the function; yes it worked

```
def stochastic_matrix_converter(M):
    M_stoch = M / np.sum(M, axis=0)[np.newaxis, :]
    return M_stoch
```

or using a *for loop*

```
def stochastic_matrix_converter(M):
    M_stoch = np.zeros(M.shape)
    for i in range(M_stoch.shape[1]):
        M_stoch[:, i] = M[:, i]/sum(M[:, i])
    return M_stoch
```

Using SVD decomposition on the newly obtained stochastic matrix M_{stoch}, **use your function in part F to select the top 3 vectors of the matrix U_{stoch} and perform the same KMeans clustering used above on them and show the plots. What do you observe? Did it work?**

Solution: please, see notebook for the function; yes it worked

```
y_pred_spectral_stoch = cluster_data(M_stoch)[1]
show_data_results(X, 2, y_pred_spectral_stoch)
```

H Now, let's investigate how we could have made the matrix M work directly in our original interpretation. To do this, normalize those 3 vectors first before performing the clustering and **show the plots. What do you observe? Did it work? If so explain why normalizing the vectors gives what is expected.**

Hint:you may use:

```
from sklearn.preprocessing import normalize
```

Solution: please, see the notebook for the function; yes it worked now; this is because was not a proper stochastic matrix. As such the top 3 U vectors are not unit vectors and by normalizing them here, we made up for the discrepancy.

```
Usub_norm = normalize(cluster_data(M)[0][:, :num_clusters])
y_pred_spectral_norm = KMeans(n_clusters=3,
random_state=seed).fit_predict(Usub_norm)
show_data_results(X, 2, y_pred_spectral_norm)
```

6. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!
We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework? Write it down here where you'll need to remember it for the self-grade form.**

Contributors:

- Olivia Watkins.
- Jerome Quenum.
- Anant Sahai.

- Anrui Gu.
- Matthew Lacayo.
- Past EECS 282 and 227 Staff.