Deep Neural Networks **(UC Berkeley CS182/282A, Spring 2023)**

Lecture 12: GNNs and RNNs

3 Mar 2023

Lecturer: Anant Sahai *Scribes: Yishu Yan and Will Yang*

1 GNN: Node-level Tasks

Last lecture: Graph-level tasks to Node-level tasks

Graph-level tasks involve predicting a single label/attribute for the entire input graph while node-level tasks involve predicting a label/attribute for each individual node in the input graph.

(1) In CNNs (e.g., U-net), some image tasks are performed at the pixel-level. Similarly, in GNNs, some tasks are performed at the node-level.
(2) Many graph algorithms are already working at the node-level. These interesting questions exist at the node-levels in graphs that can be dealt with GNNs.
(3) Weight sharing is a key feature of GNNs that allows them to operate across the entire graph, regardless of the number of connections between nodes. This makes it possible to support node-level tasks.

1.1 Holding out nodes

1st scenario: the entire graph has actual labels for every node; **2nd scenario**: the graph contains a lot of nodes, only a subset of them are labeled, and some of the labeled nodes are held back as a validation set.

Two general approaches:
(1) Simply deleting the held-out nodes and all associated edges during training. However, it can potentially damage the topology of the graph because the deleted nodes and edges might be important for the overall structure of the graph and removing them could change the way the remaining nodes are connected.

(2) Keeping the nodes and associated edges, but removing the "information" in the node. The removed information is not in the training loss (simply deleting the label information). It typically involves "Removal" of node label (feature vector). It can be done either by replacing with $\vec{0}$ or replacing with a special label "MASK" (if 0 is a meaningful label). It allows messages to pass over that node using message-passing language.

An example of these two approaches is sketched in Fig. **??**.

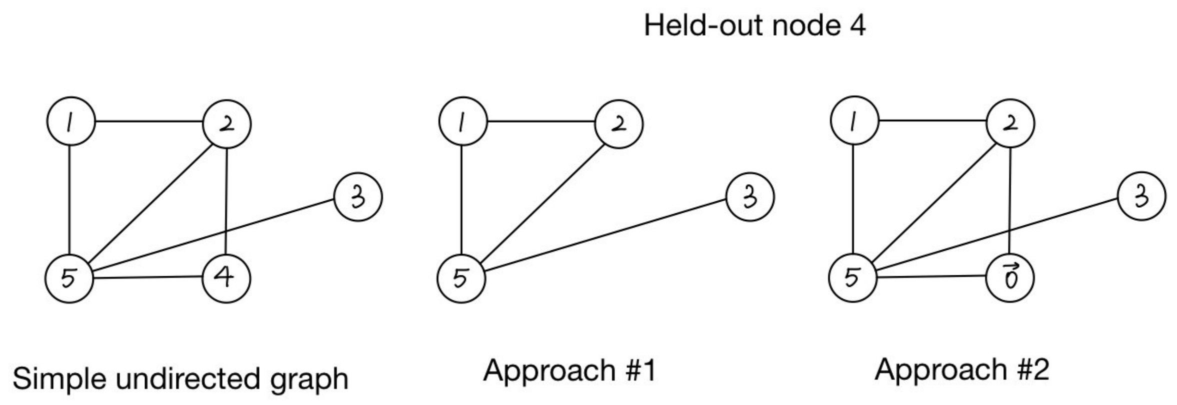


Figure 1: An example of held-out node with approach 1 (simply deleting the held-out nodes and all associated edges) or approach 2 (replacing node 4 with $\vec{0}$ when $\vec{0}$ is a meaningless label.)

Example of "MASK":

The processing of graph neural nets is all based on real vectors. You can't necessarily have this special symbol but you want it.

Example solution: Put something that is not going to show up naturally:

Original data is a vector of length 3, $\begin{bmatrix} a & b & c \end{bmatrix}^T$; embed into a vector of length 4, $\begin{bmatrix} a & b & c & 1 \end{bmatrix}^T$, if this is valid information; if it is not valid information: $\begin{bmatrix} a & b & c & 0 \end{bmatrix}^T$.

During training, you want the neural network to learn to ignore this additional element and focus on the meaningful information in the original vector.

Another approach is embedding the vectors in a larger space where invalid information has a totally different direction.

Is it possible to perform surgery on the operation of the nodes graph? We want to implement weight sharing across different nodes. Thus, the function needs to have a good way of removing dependence on self. It depends on the structure of the update function. If the update function is some function where we can simply take out the self, like an average, then removing dependence on self is easy. However, for more complex update functions where we need our own information, like weighted average by a similarity metric, simply removing self is not possible.

In general, the amount of data needed should be larger than the complexity of the function trying to learn.

2 RNN Introduction

There are two approaches in how to think about RNNs.

From GNN to RNN: In GNNs, you have the basis of graph algorithms on graph. It can be interesting in natural to want to do the same operation over and over again. Different conceptual layers of processing might share weights among them. Thinking about RNNs which are designed to process sequential data, we could think about where else that could be natural.

From Conv-net to RNN:
We will use an analogy from SP(Signal processing)-perspective:
In a conv-net, we had filters defined by finite convolutions. SP analogy: FIR filters
In an RNN, we had filters defined by internal state and weights. SP analogy: IIR filters

FIR: Finite Impulse Response: $\sum_i h[t-i]u[i]$
IIR: Infinite Impulse Response: Get infinite response length through the use of filter state

$$y_{t+1} = \beta_1 y_t + \beta_2 u_t$$

Where u_t is the input signal at time step t, and y_t is the filter output at time step t

RNNs are native to sequential data. FIR filters can be applied anywhere since they only require a local neighborhood of data to be defined. In contrast, IIR filters require sequential processing and internal state that evolves over time.

There are analogies in deep learning to the difference between open-loop control and closed-loop control, although they are still being built out. Certain types of RNN architectures can be conceptualized using the understanding of closed-loop control, although the concept is not yet fully developed. These analogies become particularly important when discussing things like diffusion models and generative models, which rely on feedback from the output to adjust the input and are closely related to the control-type ideas.

3 Kalman Filter-inspired Example

Kalman filter is probably the most famous example of the use of filter in practical problems and the most practical example in real world deployment of the infinite pulse response type and is a filter that has internal states.

3.1 Classical Approach

You have linear evolution of a true system with a description as

$$\vec{h}_{t+1} = A_{\text{true}}\vec{h}_t + B\vec{\omega}_t, \tag{1}$$

where $\{\vec{h}_t\}$ are the underlying (hidden) states of the system, $\{\vec{\omega}_t\}$ are some driving terms (e.g. white noise) and A_{true} and B are some coefficient matrices. A noisy observation is

$$\vec{x}_t = C\vec{h}_t + V_t, \tag{2}$$

where $\{V_t\}$ is some other noise.
You want an estimator that takes the noisy observations and gives you an estimate of the true states. Namely,

$$\hat{h}_{t+1} = A'\hat{h}_t + B'\vec{x}_t, \tag{3}$$

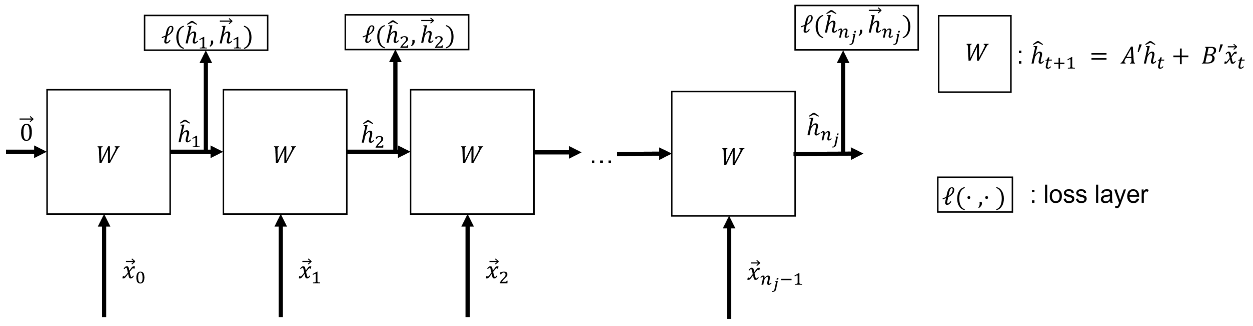
where $\{\hat{h}_t\}$ are the estimations of the true states $\{\hat{h}_t\}$.
Classically, one would make Gaussian assumptions on what's happening and compute the explicit Bayes rule estimator, which will give a MAP or MSE estimate for the state. Then, one would compute the update of that state and get an equation like Eq. **??**. The actual implementation is sophisticated, but the key idea is having the system Eq. **??** in hand and get Eq. **??**.

3.2 Neural Network Approach

Now, we want a learned a filter to do this job without knowing models like Eqs. **??** and **??**. The way we do this is by treating A' and B' as learnable weights to be learned from data.

3.2.1 The Easier Case

Now the question is: what does our data look like? Starting with the easiest case: we have sequence$\underline{\text{s}}$ of measured trajectories with ground truth. (i.e. we have the states $\left(\vec{h}_{t,j}, \vec{x}_{t,j}\right)_{t=0}^{n_j}$ for trajectories $j = 1, \cdots, m$). Then, how to use PyTorch to solve the problem? Namely, we have some data traces $\left(\vec{h}_{t,j}, \vec{x}_{t,j}\right)_{t=0}^{n_j}$ and want to use PyTorch to learn weights A', B' and evaluate the performance on some held-out sets. The standard neural net way to do it is to set up layers and do backpropagation with a loss layer. A example network is sketched in Fig. **??**.



$\ell(\hat{h}_1, \vec{h}_1)$
$\ell(\hat{h}_2, \vec{h}_2)$
$\ell(\hat{h}_{n_j}, \vec{h}_{n_j})$
$\vec{0}$
W
$\hat{h}_1$
$\hat{h}_2$
$\hat{h}_{n_j}$
$\vec{x}_0$
$\vec{x}_1$
$\vec{x}_2$
$\vec{x}_{n_j-1}$
W : $\hat{h}_{t+1} = A'\hat{h}_t + B'\vec{x}_t$
$\ell(\cdot,\cdot)$: loss layer

Figure 2: An example network. Each weight block stands for Eq. **??** with two inputs. The weights are A' and B'. The initial estimate of the state is set to zero vector.

Now, we have a setup for a PyTorch implementation and we can use our standard approach (i.e. run GD, SGD, ADAM, etc. with appropriate learning rate, initialize the weights W). Note that the same weights are shared across all layers. For backprop, the gradient of all the losses are summed up. All the cautions taken previously in this course should also be done like avoiding exploding and vanishing gradients. An open question to those who know Kalman filters well: if you initialize the weights to the Kalman filter as it should be, what would happen to the gradient in backprop?

In response to a student's question: If the weights are initialized to the analytical (symbolic) solutions of the Kalman filter in steady state, one would expect the gradient to be zero. But when you do it on PyTorch, the backprop is actually done on a per sample basis and due to the presence of observation noise, the gradient is not going to be zero. In general, there is no way to get zero loss on a single sample even at optimality. Instead, under the average of the right distribution, all the gradients cancel out.

3.2.2 The Harder Case

What if we only have the observation traces (i.e. $(\vec{x}_{t,j})_{t=0}^{n_j}$ for trajectories $j = 1, \cdots, m$) without knowing the ground truth?
You can modify your estimator such that it predicts the next observed state $\vec{x}_t$. The network in Fig. **??** needs to be modified. Several things to do: use C in Eq. **??** as an additional learnable parameter, calculate the loss on next-observation prediction. This is left as an exercise for the students.

4 RNN

The example in the previous section is of a linear model. An RNN is a generalization to include non-linearity to make more expressive models. Generic recipes to include non-linearity: 0. add activation, 1. replace matrix multiplies with MLPs. For the latter, namely, one can take Eq. **??** or a W block in Fig. **??**, writing it as the block matrix multiplication

$$A'\hat{h}_t + B'\vec{x}_t = \begin{bmatrix} A', B' \end{bmatrix} \begin{bmatrix} \hat{h}_t \\ \vec{x}_t \end{bmatrix} \tag{4}$$

and replace it by MLPs. Now, a box in the network becomes

$$\vec{h}_{t+1} = \mathrm{MLP}_W(\vec{h}_t, \vec{x}_t) \tag{5}$$

Note that here $\vec{h}$ instead of $\hat{h}$ is used as the deep neural net community don't use $\hat{h}$ and the reason is that you are agnostic as to whether there's actually a true h or not. Another modification from Kalman filter is that biases are added.
To make the network more expressive and sophisticated, one can expand the MLPs by adding more weights. This will be talked in detail next time.