CS69000-DPL Spring 2021 Homework	— Homework 3—	Last update: April 6, 2021
https://www.cs.purdue.edu/homes/ribeirob/courses/Spring2021		Due: **6:00am**, Saturday, April 3rd

Instructions and Policy: Each student should write up their own solutions independently, no copying of any form is allowed. You MUST to indicate the names of the people you discussed a problem with; ideally you should discuss with no more than two other people.
Chowdhury Mohammad Rakin Haider.
You need to submit your answer in PDF. LaTeX is typesetting is encouraged but not required. Please write clearly and concisely - clarity and brevity will be rewarded. Refer to known facts as necessary.

Q0 (0pts correct answer, -1,000pts incorrect answer: (0,-1,000) pts): A correct answer to the following questions is worth 0pts. An incorrect answer is worth -1,000pts, which carries over to other homeworks and exams, and can result in an F grade in the course.

(1) Student interaction with other students / individuals:

(a) I have copied part of my homework from another student or another person (plagiarism).

(b) **Yes, I discussed the homework with another person but came up with my own answers. Their name(s) is (are) Muhammad Abubakar________________________________.**

(c) No, I did not discuss the homework with anyone.

(2) On using online resources:

(a) I have copied one of my answers directly from a website (plagiarism).

(b) I have used online resources to help me answer this question, but I came up with my own answers (you are allowed to use online resources as long as the answer is your own). Here is a list of the websites I have used in this homework:
Wikipedia, StackOverflow, MLWiki________________________________.

(c) I have not used any online resources except the ones provided in the course website.

Homework 3: Inductive Biases for Graph and Sequence Data

Learning Objectives: Let students understand basic concepts that are used to add inductive biases in deep learning models: for images, sequences, and graphs.

Learning Outcomes: After finishing this homework, students will be able to create new inductive biases to solve new tasks.

3.1 Concepts (3.0 pts)

Please answer the following questions **concisely**. All the answers, along with your name and email, should be clearly typed in some editing software, such as Latex or MS Word.

1. **(0.5 pt)** In class, we saw the **Janossy Pooling** neural network, which uses the Reynolds operator:

$$\Gamma(x, \theta) = \frac{1}{|x|!} \sum_{\pi \in \Pi} f\big(x_\pi; \theta\big),$$

where f is a permutation-sensitive neural network with parameter θ. Show that mean-pooling is a special case of the Jannossy Pooling operator.

Solution: The mean-pooling is defined as follows,

$$\Gamma_{meanpooling}(h) = \sum_i h_i$$

Here, h_i is supposed to be the input from previous layer which can be a neural network (MLP, CNN, RNN etc.).

Now, since mean is a permutation insensitive function, for all $\pi \in \Pi$, $\sum_i h_{\pi_i} = \sum_i h_i$. So we can rewrite the equation as,

$$\Gamma_{meanpooling}(h) = \frac{1}{|h_i|} * |h_i| \sum_i h_i = \frac{1}{|h_i|} * \sum_{\pi \in \Pi} \sum_i h_{\pi_i} = \frac{1}{|h_i|} * \sum_{\pi \in \Pi} f(h_\pi)$$

$$\therefore \Gamma_{meanpooling}(h) = \frac{1}{|h_i|} * \sum_{\pi \in \Pi} f(h_\pi)$$

This representation of mean-pooling is equivalent to the equation of Janossy Pooling.

2. **(0.5 pt)** Consider n-element input sequences $x_1, \cdots, x_n$. Consider the permutation group over these sequence. Prove that the left 1-eigenspace associated with the Reynolds operator of the permutation group is equivalent to mean-pooling times a constant.

   Solution: The permutation operation of an n element sequence of $\boldsymbol{x} = x_1, x_2, \ldots, x_n$ can be defined as $\boldsymbol{Px}$ where $\boldsymbol{P}$ is a $n \times n$ permutation matrix. A permutation matrix is matrix which can be obtained by permutation of the rows or columns of the identity matrix. Therefore, the permutation matrix has one 1 in each row and all the other elements are 0.

   The Reynold's operator of the permutation group is

$$m\bar{T} = \frac{1}{n!} \sum_{\pi \in \Pi} P_\pi$$

   We can obtain the summation using the following idea. For each position π_1 of x_1 there are $(n-1)!$ permutations of the other items. Therefore, we will get $(n-1)!$ permutation matrices with the $\boldsymbol{P}[1, \pi_i] = 1$. Summing them up will make $\bar{\boldsymbol{T}}[1, \pi_i] = (n-1)!$. Using the above idea for each cell of the Reynold's operator we get,

$$\bar{\boldsymbol{T}} = \frac{1}{n!} \begin{bmatrix} (n-1)! & (n-1)! & \ldots & (n-1)! \\ \vdots & \vdots & \vdots & \vdots \\ (n-1)! & (n-1)! & \ldots & (n-1)! \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & \ldots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & \ldots & 1 \end{bmatrix} = \frac{1}{n} \boldsymbol{A}$$

   If $\boldsymbol{v}$ and λ are left eigenvector and eigenvalue of $\boldsymbol{A}$ then, $\boldsymbol{v}\frac{1}{n}\boldsymbol{A} = \frac{\lambda}{n}\boldsymbol{v} = \lambda'\boldsymbol{v}$. So the eigenvectors of Reynold's operator of permutation group are the eigenvectors of $\boldsymbol{A}$. We know that, the only eigenvector of $\boldsymbol{A}$ with non-zero value is the vector $\boldsymbol{v} = \{\left[v_1 v_2 \ldots v_n\right]^T : v_i = v_2 = \cdots = v_n\}$ with eigenvalue $\lambda = n$ and $\lambda' = 1$. That means, the 1-eignespace of $\bar{\boldsymbol{T}}$ includes only one vector where all the elements are same. Let $\boldsymbol{v} = \left[vv \ldots v\right]$, Then $w = \alpha \boldsymbol{v}$.

$$\begin{aligned} \boldsymbol{w}^T \boldsymbol{x} &= \alpha \boldsymbol{v}^T \boldsymbol{x} \\ &= \alpha \sum_{i}^{n} v * x_i \\ &= \frac{v\alpha}{n} \Gamma_{MeanPooling}(\boldsymbol{x}) \end{aligned}$$

3. **(0.5 pt)** Consider the following model families:

- H_0: 1-hidden layer MLP (with softmax outputs)
- H_1: 2-hidden layer MLP (with softmax outputs)
- H_2: Logistic Regression (with softmax outputs)

Describe the relationship between these hypothesis classes. Draw a Venn diagram of H_0, H_1, and H_2 in the model space (where each point in the space represents a different input-output function (different model)).
Hint: If $f'(x) = g(f(x))$, where $g(x) = x$, then f' and f are the same model (same measurable map) since their input-output relations are the same.

Solutions: The general definition of hypothesis space is as follow,

$$H_{MLP} = \{MLP(W_1, \ldots, W_L) : L \in \mathrm{Z}^+, h_1 \in \mathrm{Z}^+, W_1 \in R^{d\times h_1}, \ldots, h_{L-1} \in \mathrm{Z}^+, W_L \in R^{h_{(L-1)}\times C}\} \quad (1)$$

From Eqn. 1, we see the size of hypothesis space is $|H_{MLP}| = R^{d\times h_1} R^{h_1 \times h_2} \ldots R^{h_{L-1}\times C}$. If all the layers have similar dimensions then, $|H_{MLP}| = R^{d\times h} R^{h\times h} \ldots R^{h\times C} = R^{d\times h^{2(L-1)}\times C}$

So, $|H_0| = R^{d\times C}$, $|H_1| = R^{d\times h^2 \times C}$, Therefore, $|H_0| < |H_1|$.

If logistic regression is f(x) then applying a softmax g(x) on the output of logistic regression f(x) then it will result into the same output g(f(x)) = f(x). Logistic Regression model is simply a 1-hidden layer MLP with sigmoid activation function applied to the output of the 1-hidden layer MLP. Then the size of $|H_2| = |H_0|$.

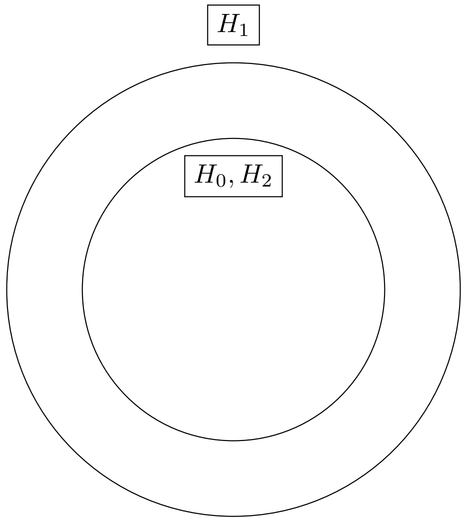


Figure 1: Venn Diagram of H_0, H_1, H_2

4. **(0.5 pt)** Much is talked about the Transformer's ability to learn language structure. Some say large Transformer models trained on massive NLP datasets are "The AI That's Too Dangerous to Release" because it gives the "appearance of intelligence". Show that a Transformer model may be unable to extrapolate beyond the training data distribution.
Hint: Show that Transformers —like any sequence model we have seen in class— would be incapable of logical reasoning. Use the **structural equation models** learned in class to explain your answer. You can use the following example to aid your answer:
(Train) All men are mortal. Socrates is a man. Therefore, Socrates is mortal.
(Train) All programmers are happy. Alice is a programmer. Therefore, Alice is happy.
(Test) All elephants are heavy. Pinocchio is a boy. Therefore, Pinocchio is ⟨predict⟩.
Fun-time: You can play with a large transformer model at `https://talktotransformer.com/`. Playing with it probably will **not** help you answer the above question. In most runs, the (large) GPT-2 model answers "*heavy*". For instance, "All elephants are heavy. Pinocchio is a boy. Therefore, Pinocchio is *heavy. Hard to be a fairy, but you do have to be special. Be special, be brave, be me.* [...]".

Solution: A sequence model tries to predict the probability of the next item in the sequence given previous elements in the sequences. For example, in a sequence $w_1, w_2, \ldots, w_{t-1}$ sequence model will try to predict,

$$P(w_t | w_1, w_2, \ldots, w_{t-1})$$

Let us consider the examples provided for this question. For the sake of simplicity let us simplify the texts.

- Man, Mortal, Socrates, Man, Socrates, Mortal
- Programmer, Happy, Alice, Programmer, Alice, Happy
- Elephant, Heavy, Pinocchio, Boy, Pinocchio, ⟨predict⟩.

The sequences are 6 characters long. Let, each element of the sequence is represented by w_i. We need to predict w_6 given w_1, w_2, w_3, w_4, w_5. According to the training samples, the structural causal equations should be something as shown below.

$$\begin{aligned}
w_1, w_4 &:= f_1(E_1) \\
w_2 &:= f_2(E_2) \\
w_3, w_5 &:= f_3(E_3) \\
w_6 &:= f_4(w_1, w_2, w_3, w_4, w_5, E) \\
E, E_1, E_2, E_3 &\sim Uniform(0, 1)
\end{aligned}$$

The prediction of w_6 from the sequence model will be generated from,

$$P(w_6 | w_1, \ldots, w_5) = g(\Gamma(w_1, \ldots, w_5), E) = g(\Gamma_\epsilon(E_1, E_2, E_3), E)$$

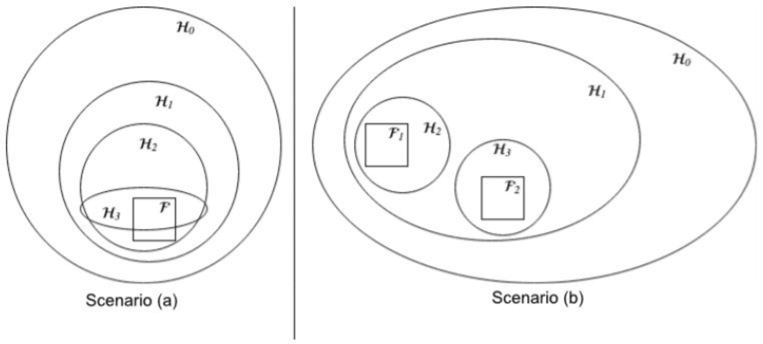
Where, Γ, Γ_ϵ are the representations of the previous sequence in terms of the elements of the sequence and random sampled noise.

However, the test sample doesn't follow the structural causual model of training samples. For the test dataset the structural causal equations might be something similar to the one shown below.

$$\begin{aligned}
w_1 &:= f_1(E_1) \\
w_2 &:= f_2(E_2) \\
w_3, w_5 &:= f_3(E_3) \\
w_4 &:= f_4(E_4) \\
w_6 &:= f_4(w_1, w_2, w_3, w_4, w_5, E) \\
E, E_1, E_2, E_3, E_4 &\sim Uniform(0, 1)
\end{aligned}$$

Trying to predict for the test sample will fail since the sequence model will fail to incorporate E_4 in its prediction and simple predict using E_1, E_2, E_3.

5. **(0.5 pt)** Consider two different multi-task learning scenarios (a) and (b) in figure below. There are two tasks T_1 and T_2. In scenario (a) two tasks we want to learn both come from the hypothesis space $\mathcal{F}$. In scenario (b) task T_1 come from $\mathcal{F}_1$ and T_2 come from $\mathcal{F}_2$. In each scenario ((a) and (b)), describe which hypothesis space is the best hypothesis space to learn these two tasks and why.



Using Scenario (b) in the figure above, explain why **transfer learning** between tasks T_1 and T_2 can fail.
Hint: Consider first training on task T_1 and then transfering to task T_2. Draw the Venn diagram of the hypothesis space of the transfer model for task T_2. Explain why the transfer is probably not useful if we compare against just using H_1 or H_3 as the hypothesis space for task T_2.

Solution: For scenario (a), the best hypothesis space to learn the two tasks is $\mathcal{H}_2$. This is because $\mathcal{H}_2$ is smallest hypothesis space that covers the entire hypothesis space $\mathcal{F}$. If we use $\mathcal{H}_3$, it may not include the true function.

For scenario (b), the best hypothesis space to learn task T_1 and T_2 individually are $\mathcal{H}_2$ and $\mathcal{H}_3$ respectively. But if we try to learn both of them jointly then it will be better to use $\mathcal{H}_1$. This is because using either $\mathcal{H}_2$ or $\mathcal{H}_3$ will result into models that are better for one task but shows poor performance on the other. But using $\mathcal{H}_1$ we can find a representation which is equally better for both tasks and then optimize function g^1 and g^2 using that representation.

Finally, in scenario (b) transfer learning between task T_1 and T_2 can fail. We first assume a hypothesis $f^1(x) = g^1(\Gamma(x))$ is picked for task T_1 and the learning process used hypothesis space $\mathcal{H}_1$. Since $\Gamma(x)$ is part of the hypothesis f^1, it is a subset of $\mathcal{H}_2$. Now, if we want to optimize a function g^2 such that $f^2(x) = g^2(\Gamma(x))$ approximates the function f^2 for task T_2 then the appropriate hypothesis space for learning is $\mathcal{H}_4$ as shown in Fig. 2 which is far bigger than $\mathcal{H}_3$.

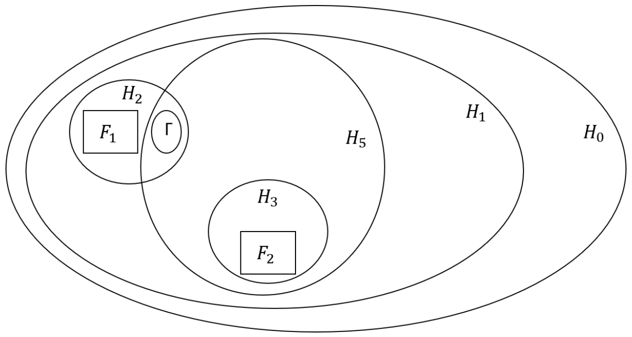


Figure 2: Scenario (b) Transfer Learning

On the other hand, if $\mathcal{H}_1$ was used then it might be possible to find a better representation $\Gamma^2(x)$ which results in better models for T_2. Therefore, it is better to use $\mathcal{H}_3$ or $\mathcal{H}_1$ than using transfer learning of task T_1 and T_2 in this scenario.

6. **(0.5 pt)** We saw in class that gradient matching is a type of meta-learning procedure, where we have a model $f^1 = g^1\big(\Gamma^1(x)\big)$ which we use to learn one task and a model $f^2 = g^2\big(\Gamma^2(x)\big)$ which we use for the second tasks. With gradient matching, we seek to match the gradients $\frac{\partial}{\partial x}\Gamma^1(x) = \frac{\partial}{\partial x}\Gamma^2(x)$. Explain why matching gradients can be better than making $\Gamma^1 = \Gamma^2$. Draw a Venn diagram of the model space specified by Γ^1 and Γ^2 in each case.

   Solution: Matching gradient can be better than making $\Gamma^1 = \Gamma^2$ since there may not be a good representation Γ^2 in the vicinity of Γ^1. Using gradient matching allows us to learn a representation Γ^2 that is simply a shifted version of Γ^1. The venn diagram of the model space specified by Γ^1 and Γ^2 are show below.

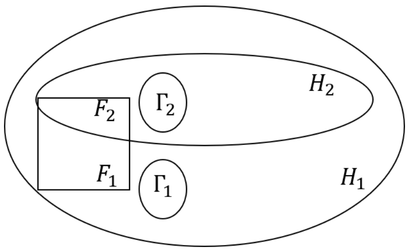


Figure 3: Venn Diagram of hypothesis space.

   Gradient matching is also helpful when the tasks are given in sequence. That means we cannot train a joint representation since the training samples for each task are not available at the same time. Let's assume, first we were given task 1 and we trained the function $f^1 = g^1(\Gamma^1(x))$ using hypothesis space $\mathcal{H}_1$. Now, we are given task to and we need to find out f^2. At this point we do not know if Γ^1 is a proper representation for Task 2. Therefore, we need to learn a new representation by adapting Γ^1. In the venn diagram we see that the representations for task 1 and task 2 are different from each other. If we use gradient matching, we will be able to introduce inductive bias in learning through matching the gradients of the two representations and we will be able to use $\mathcal{H}_2$ (a smaller hypothesis space) to learn $f^2 = g^2(\Gamma^2(x))$. This means we are able to learn efficiently, by using previous knowledge from previous tasks and ensuring a proper representation is learned for each task.

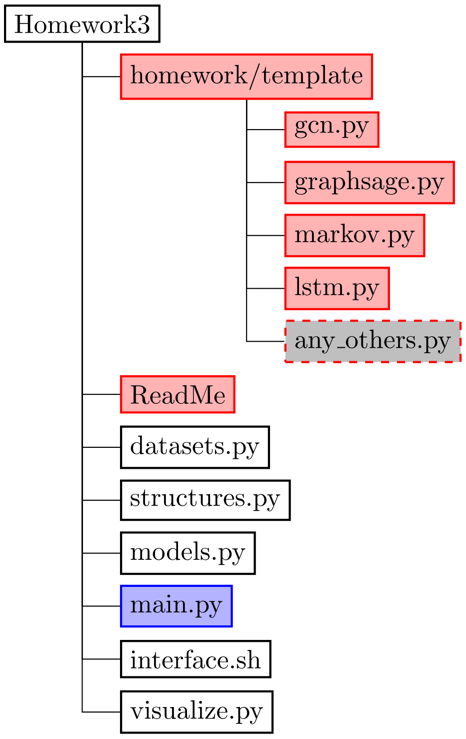
3.2 Programming (7.0 pts)

In this part, you are going to implement (1) Graph Neural Network with mean aggregator, (2) Graph Neural Network with LSTM set aggregator, (3) Markov Chain language model and (4) LSTM language model. The rule of thumbs is that you can do any changes in the `homework/template` files (not changing their names), but need to **keep the main executable outside `homework/template`, such as `main.py`, `models.py`, untouched**. That is, you can change any function you want in the code provided inside the folder `homework/template` and add extra files. But all other files outside `homework/template` should remain intact on submission and the code should execute in the scholar cluster via `interface.sh`.

Skeleton Package: A skeleton package is available on Piazza with the execution scripts. You should be able to download it and use the folder structure provided. **A GPU is essential for some of the tasks, thus make sure you follow the instruction to set up the GPU environment on scholar.rcac.purdue.edu.** A tutorial is available at `https://www.cs.purdue.edu/homes/ribeirob/courses/Spring2021/lectures/cluster-how-to/cluster-how-to.html`.

HW Overview

This HW is a combination of two distinct tasks. You are going to fill in a few new components to the HW3 package given on Piazza.
Here is the folder structure that you should use:



- **Homework3:** the top-level folder that contains all the files required in this homework.
- **ReadMe:** Your ReadMe should begin with a couple of **example commands**, e.g., "python hw3.py data", used to generate the outputs you report. TA would replicate your results with the commands provided here. More detailed options, usages and designs of your program can be followed. You can also list any concerns that you think TA should know while running your program. Note that put the information that you think it's more important at the top. Moreover, the file should be written in pure text format that can be displayed with Linux "less" command. You can refer to interface.sh for an example.
- **main.py:** The main executable to run training with graph neural networks and languange models.
- **visualize.py:** The executable script to render plots for your results. It can give you a better understanding of your implementations.
- **interface.sh:** The executable bash script to give you examples of main.py usage. It also works as an example for writing ReadMe.
- **datasets.py:** The module defines the Cora and PTB dataset used for this homework.
- **structures.py:** The module defines different dataset structures for this homework. It will automatically split raw dataset into training, validation and test chunks.
- **models.py:** The module defines the learning framework for this homework. It also evaluates learned model performance on Cora and PTB dataset separately in this homework.
- **homework/template:** Your Python neural network package. The package name in this homework is **homework/template**, which should NOT be changed for submission. But you can change any function you want in the code provided inside the folder `homework/template`. All other files outside `homework/template` should remain intact. The four modules that you are going to develop and should be at least included:
  - **gcn.py**
  - **graphsage.py**
  - **markov.py**
  - **lstm.py**

  Details will be provided in the description of each task. All other modules are just there for your convenience. It is not required to use them, but exploring that will be a good practice of re-using code. Again, you are welcome to architect the code inside `homework/template` in your own way, as long as the code works with interface.sh and does what it is supposed to do. For instance, you can add another module, called `utils.py`, to facilitate your implementation.

3.2.1 GNN with Mean Aggregator (3.0 pts)

This is a warm-up task. You are going implement mean aggregator based on provided formula and supporting files under GNN folder.

In the following GNN related coding assignment, you will work with the Cora dataset. The Cora dataset consists of 2708 scientific publications classified into one of seven classes. The citation network consists of 5429 links (including self loops and replcations). Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from the dictionary. The dictionary consists of 1433 unique words.

You can regard it as a labeled graph $G = (V, E, X, Y)$ where $V = \{1, 2, \cdots, 2708\}$ is the set of $n = 2708$ nodes, E is the set of 5429 undirected edges, X and Y are the feature and label matrices assigned to the graph. Although the raw dataset is a directed graph, we will regard it as an undirected graph for convenience. For each node $v \in V$, we have a feature vector $\boldsymbol{x}_v$ and a label y_v. Thus, you will have

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix} \quad \boldsymbol{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \tag{2}$$

where each $\boldsymbol{x}_u$ is a 1433 dimensional vector and $\boldsymbol{X}$ is a 2708×1433 matrix.

In GNN of this project, features of a node u and its neighbors N_u will be aggregated together from layer k to next layer $k + 1$:

$$\boldsymbol{h}_u^{(k+1)} = \begin{cases} \sigma\left(\boldsymbol{W}^{(k+1)}\left[\boldsymbol{h}_u^{(k)}, \overline{\overline{f}}\left(\{\boldsymbol{h}_v^{(k)}\}_{v \in N_u}\right)\right] + \boldsymbol{b}^{(k+1)}\right) & k+1 > 0 \\ \boldsymbol{x}_u & k+1 = 0 \end{cases} \tag{3}$$

where $\boldsymbol{h}_u^{(k)}$ is the embedding features of node u at layer k, $\boldsymbol{W}^{(k)}$ and $\boldsymbol{b}^{(k)}$ are weight and bias matrices of layer k, σ is a specific activation and $\overline{\overline{f}}$ is a specific permutation invariant function. In this project, $\overline{\overline{f}}$ will be the *mean* function. Thus, you will have a graph embedding matrix $\boldsymbol{H}^{(k)}$ for each layer k. The final embedding matrix will then be used to classify labels Y for graph G.

$$\boldsymbol{H}^{(k)} = \begin{bmatrix} \boldsymbol{h}_1^{(k)} \\ \boldsymbol{h}_2^{(k)} \\ \vdots \\ \boldsymbol{h}_n^{(k)} \end{bmatrix}. \tag{4}$$

Related Modules:

- homework/template/gcn.py

Action Items:

1. The formula above is designed to work on each node separately which is slow for deep GNNs in practice. For the mean aggregator of this project, there will be a matrix version of the formula which allows us

to exploit the efficiency of matrix computations. To be specific, you are given an embedding matrix $\boldsymbol{H}^{(k)}$ of layer k, adjacency matrix $\boldsymbol{A}$ where

$$A_{uv} = \begin{cases} 0 & u \text{ and } v \text{ are disconnected} \\ 1 & u \text{ and } v \text{ are connected} \end{cases}. \tag{5}$$

Rewrite the above formula from Equation 3, for mean as $\overline{\overline{f}}$, so that you can get $\boldsymbol{H}^{(k+1)}$ from $\boldsymbol{H}^{(k)}$ and A. You can only define constant matrices in this question.
Hint: You can compute the mean operation through a matrix multiplication.

2. Go through all the related modules. Specifically, you should understand **datasets.CoraDataset**, **structures.CoraDataStructure** and **models.MulticlassNodeClassification** well to use provided data properly. Implement the GCN formula you derived above in in `template/gcn.py`. **In the report, provide the min, max and average degrees of training, validation and test nodes of the Cora dataset.** (To be more specific, you shoud get those statistics from forward function of your GCN implementation.)

3. After understanding the matrix version of the formula and all the related modules, you should implement the formula into the **data.py** and **model.py** to make GNN run correctly. Run the **main.py** with default arguments: `python main.py`. Save the logging outputs from your console. **In the report, you should include:**
   (a) Training curves (validation nodes) of loss function of for each epoch.
   (b) Training curves of (validation nodes) accuracy of for each epoch.
   (c) Loss functions and accuracies of test nodes for the saved models. (You should achieve this from log file with pseudo early stopping.)

4. Run with `python main.py ... --num-graph-layers 2`, and
   `python main.py ... --num-graph-layers 20`.
   In your report, include:
   (a) Describe what happens during the training process
   (b) For the model from `python main.py ... --num-graph-layers 2`, get $\boldsymbol{H}^{(1)}$; Report the maximum MSE of all pairs of embedding vectors in $\boldsymbol{H}^{(1)}$.
   (c) For the model from `python main.py ... --num-graph-layers 20`, get $\boldsymbol{H}^{(19)}$; Report the maximum MSE of all pairs of embedding vectors in $\boldsymbol{H}^{(19)}$.

	Min	Max	Average
Training			
Validation			
Test			

Solution: The following action items are performed.

1. The matrix version of the formula is as follows,

$$\boldsymbol{h}_u^{(k+1)} = \begin{cases} \sigma\left(\left[\boldsymbol{H}^k, diag(\boldsymbol{e}\boldsymbol{A})^{-1} * \boldsymbol{A} * \boldsymbol{H}^k\right] * \boldsymbol{W}^{(k+1)} + \boldsymbol{b}^{(k+1)}\right) & k+1 > 0 \\ \boldsymbol{x}_u & k+1 = 0 \end{cases} \tag{6}$$

Here, $\boldsymbol{e}_i = 1 \forall i$. That means $\boldsymbol{e}$ is a row vector with all ones. Then $\boldsymbol{eA}$ gives the degree vector of the graph. The $diag(\boldsymbol{eA})^{-1}$ can be computed efficiently because the inverse of a diagonal matrix only requires inverse of the diagonal elements.

2. The min, max and average number of degrees of training, validation and test nodes of the Cora dataset are as shown in the table.

	Min	Max	Average
Training	1	23	3.3282
Validation	1	78	4.7560
Test	1	168	4.9941

3. The training curves of loss function and accuracy is shown in Fig. 4 and 5.

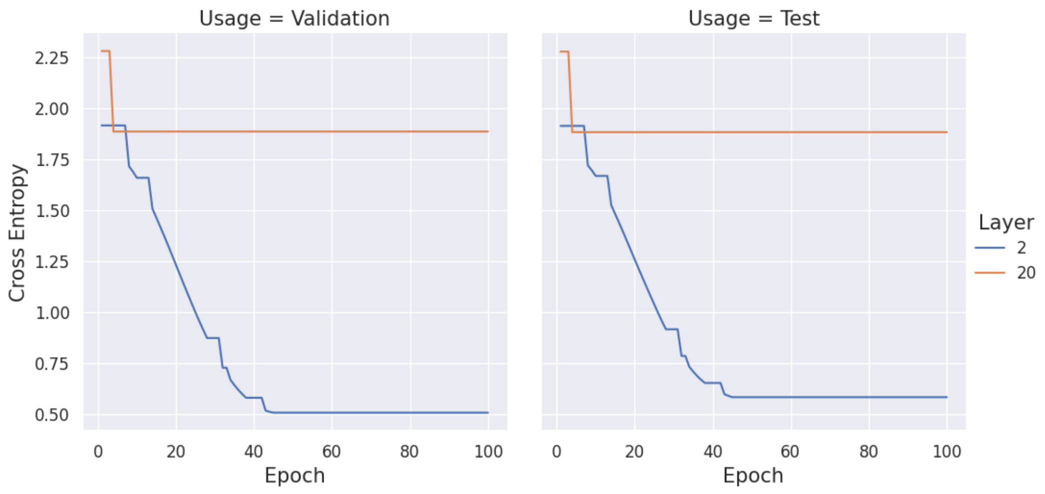


Figure 4: Training curves of loss for each epochs

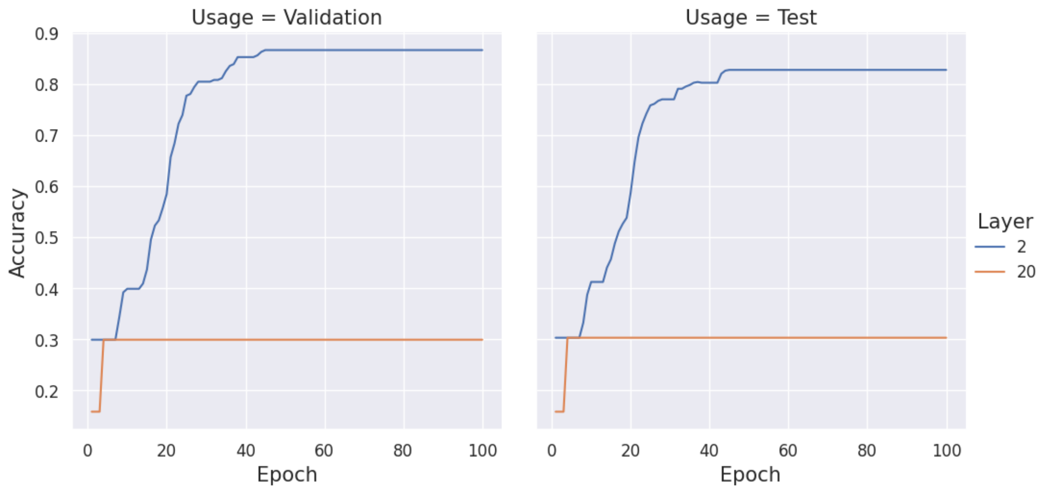


Figure 5: Training curves of accuracy for each epochs

The loss and accuracy of test nodes for the saved model with pseudo early stopping (model with best validation accuracy) are shown in Table 1

Layers	Test Loss	Test Acc
2	0.5822234749794006	82.72%
20	1.881252646446228	30.28%

Table 1: Test loss and accuracy of GCN with number of layers 2 and 20.

4. (a) During the training process when GCN is trained with small number of layers (num_layers = 2) the validation accuracy improves with epohcs. That means as the training goes on the GCN learns better representation of each node. Using two layers means the representation learned for each node is influenced by the representation of all the 2-hop neighbors of the node.

   However, when training GCN with 20 layers the accuracy stops increasing after a few iterations. This is because when we train a GCN with 20 layers, the representation of each node is influenced by all the 20-hop neighbors of the node. In a real network usually all the nodes lies within a small number of hops between each other (also known as small world phenomenon). Therefore, when considering k-hop neighbors if k is high that means considering the entire graph. Therefore, each node considers all the other nodes and consequently all the nodes learn the same representation. This is also evident when we computed the MSE of each node pair using $\boldsymbol{H}^{(19)}$ and found that the MSE is 0. That means all the nodes have the same representation.

   (b) The maximum MSE of all pairs of embedding vectors in $\boldsymbol{H}^{(1)}$ is **0.455431**.

   (c) The maximum MSE of all pairs of embedding vectors in $\boldsymbol{H}^{(19)}$ is **0.000000**.

3.2.2 GNN with Mean and LSTM Aggregator (2.0 pts)

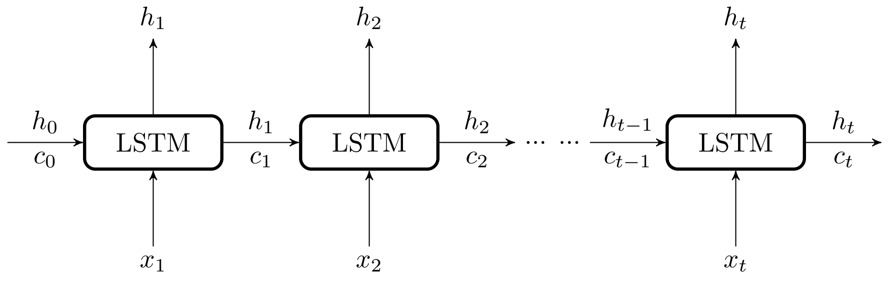
This is an advanced version of previous task. You are going implement a more general GNN supporting both mean and LSTM aggregator. This model is similar to what is known as the GraphSAGE architecture. **Pay attention that the design of this project will have time efficiency issues, so we highly recommend you to do your experiments on GPU resources, e.g. the scholar cluster. Also it is recommended to understand PyTorch sparse algebra, e.g., index_add_. If it is not available, you should start as soon as possible. Running on CPU will take hours to finish.**

In the GNN of this project, features of a node u and its neighbors N_u will be aggregated together from layer k to next layer $k+1$:

$$\boldsymbol{h}_u^{(k+1)} = \begin{cases} \sigma\Big(\boldsymbol{W}^{(k+1)}\Big[\boldsymbol{h}_u^{(k)}, \overline{\overline{f}}\Big(\big\{\boldsymbol{h}_v^{(k)}\big\}_{v\in N_u}\Big)\Big] + \boldsymbol{b}^{(k+1)}\Big) & k+1>0 \\ x_u & k+1=0 \end{cases}$$

where $\boldsymbol{h}_u^{(k)}$ is the embedding features of node u at layer k, $\boldsymbol{W}^{(k)}$ and $\boldsymbol{b}^{(k)}$ are weight and bias matrices of layer k, σ is a specific activation and $\overline{\overline{f}}$ is a specific permutation invariant function. But this time, $\overline{\overline{f}}$ will be either the mean function or LSTM architectures with Janossy pooling.

Consider a general LSTM architecture, as we have seen in class:



To use the LSTM as a neighbor aggregator, we will have a sequence of neighbor embeddings $\boldsymbol{h}_{v_1}, \boldsymbol{h}_{v_2}, \cdots, \boldsymbol{h}_{v_t}$ (where $t = |N_u|$ is the number of neighbors to aggregate), which we'll give as the input of the LSTM. To avoid confusion, we'll call the short term outputs from the LSTM by $\boldsymbol{m}_1, \boldsymbol{m}_2, \cdots, \boldsymbol{m}_t$, so the final output of the LSTM is going to be $\boldsymbol{c}_t, \boldsymbol{m}_t = \vec{f}(\boldsymbol{h}_{v_1}, \boldsymbol{h}_{v_2}, \cdots, \boldsymbol{h}_{v_t})$.

Based on the definition of LSTM design, we know that state memory c_t can be regarded as an embedding of sequence $\boldsymbol{h}_{v_1}, \boldsymbol{h}_{v_2}, \cdots, \boldsymbol{h}_{v_t}$. The problem is that the LSTM function $\vec{f}$ is a permutation sensitive function while we are seeking for permutation invariant function $\overline{\overline{f}}$.

From the class, we know that Janossy pooling can be a method to get a permutation invariant function $\overline{\overline{f}}$ from permutation sensitive function $\vec{f}$ by

$$\overline{\overline{f}}(\boldsymbol{h}_{v_1}, \boldsymbol{h}_{v_2}, \cdots, \boldsymbol{h}_{v_t}) = \frac{1}{t!}\sum_{\pi\in\Pi(t)} \vec{f}(\boldsymbol{h}_{v_{\pi(1)}}, \boldsymbol{h}_{v_{\pi(2)}}, \cdots, \boldsymbol{h}_{v_{\pi(t)}}) \tag{7}$$

where $\Pi(t)$ will be all possible permutations to length t. In practice, enumerate $t!$ permutations is not tractable. In this project, we will use π-SGD by sampling a permutation rather than traversing all permutations.

$$\overline{\overline{f}}(\boldsymbol{h}_{v_1}, \boldsymbol{h}_{v_2}, \cdots, \boldsymbol{h}_{v_t}) \approx \vec{f}(\boldsymbol{h}_{v_{\hat{\pi}(1)}}, \boldsymbol{h}_{v_{\hat{\pi}(2)}}, \cdots, \boldsymbol{h}_{v_{\hat{\pi}(t)}}), \quad \text{where } \hat{\pi} \sim \text{Uniform}(\Pi(t)) \tag{8}$$

The above procedure is expensive if the node have large degrees. In what follows we consider 5-ary dependencies to ease the computational borden.

5-ary Janossy pooling. Another issue of LSTM is gradient vanishing or explosion when backpropagating through too many time steps. In this project, a node may have more than 100 neighbors which will require LSTM to backpropagate through more than 100 time steps. Rather, we will consider 5-ary Janossy pooling, trained with π-SGD. The key idea is only perform a forward and backward pass over the first 5 neighbors of $\hat{\pi}$ (ignoring the remaining neighbors). At inference time, where we will sample 20 permutations $\hat{\pi}$ (described later in the homework) we also need to only consider the first 5 neighbors from $\hat{\pi}$ in the forward pass.

Related Modules:

- homework/template/graphsage.py

Action Items:

1. You should address the difference between this task and previous task. **Report what you expect would happen if we also did the 20-layer experiment in this project.**
2. Fill in missing parts of `template/graphsage.py`. **Hint:** If necessary, consult PyTorch's documentation of the LSTM: `https://pytorch.org/docs/stable/nn.html?highlight=lstm#torch.nn.LSTM`.
3. Run with and without `python main.py ... --janossy`. Save the logging outputs from your console. **In the report, you should include:**
   (a) Training curves (validation node) of loss function of for each epoch.
   (b) Training curves (validation node) of accuracy for each epoch.
   (c) Loss function and accuracy of test nodes for the saved model. (You should achieve this from log file with pseudo early stopping.)
4. In the previous question with LSTM aggregator, you will also randomly sample a permutation in test stage. Run the code again, bug use the average prediction of 20 permutations (average of 20 times of test stage) as final output. Use `python main.py ... --num-perms 20`. **What is the test accuracy over those 20 permutations? Explain why the results improve.**

Solution: The action items performed for this problem are as follows,

1. The `SparseGCN` implementation is same as `DenseGCN` except it uses a sparse adjacency matrix. Since the adjacency matrix of a graph is usually sparse, the process to compute $h_{neigh}^{(k)}$ can be efficiently done using sparse matrix multiplication ideas. (Per epoch time of SparseGCN $\sim$ 0.005s and Per epoch time of DenseGCN $\sim$ 0.012s).

However, the `SparseJanossy` implementation uses a lstm inplace of a permutation invariant function. LSTM is not a permutation invariant function. Therefore, to make the LSTM function permutation invariant the Janossy Pooling technique was used. But since Janossy pooling will intractable π-SGD was used to solve the problem of generating high number of permutations. Finally, LSTM also suffers from long term dependancies. Therefore, if a node has high degree, LSTM can face vanishing or exploding gradient. To solve this issue only first 5 neighbor of the permutation of the neighbors was sampled as the representatives of the neighbors. The main take away from `SparseJanossy` will be we can use any function in place of mean pooling, but we need to ensure that it is permutation-invariant function. If it is not permutation-invariant Janossy Pooling technique can be used to make it permutation-invariant.

Even if we used Janossy Pooling the problem of higher number of layers will still exists since number of layers only determine how many hops we want each node to propagate its representation to other nodes. On the other hand, Janossy Pooling is only concerned with learning a permutation invariant representation from the neighbors of the node.

2. `SparseGCN` and `SparseJanossy` classes are completed.

3. The training curves of loss function and accuracy is shown in Fig. 6 and 7.

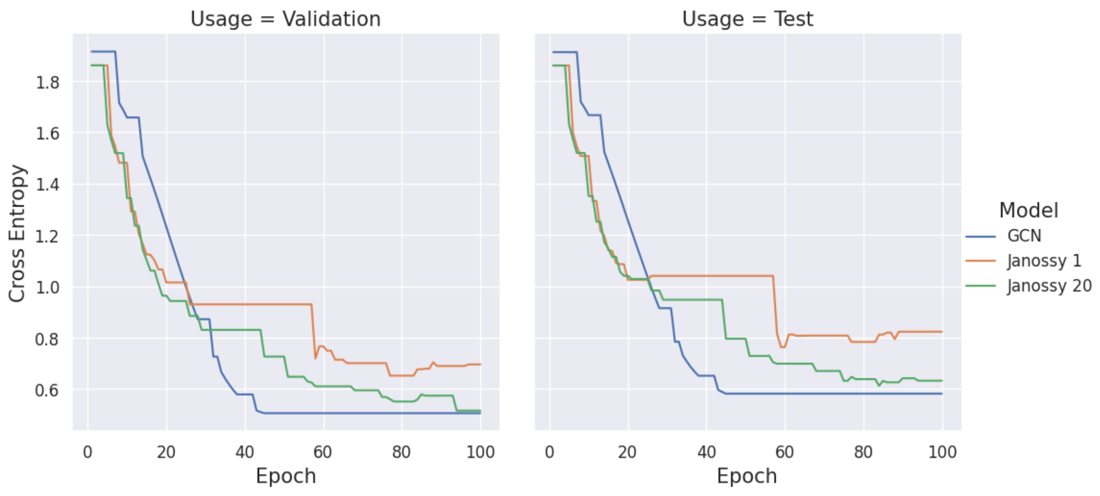


Figure 6: Training curves of loss for each epochs

The loss and accuracy of test nodes for the saved model (model with best validation accuracy) are shown in Table 2.

Configuration	Validation Accuracy	Test Loss	Test Accuracy
Without Janossy	86.60%	0.5822234749794006	82.72%
Janossy (num-perm = 1)	85.91%	0.823553740978241	81.39%
Janossy (num-perm = 20)	87.63%	0.6326681971549988	84.49%

Table 2: Test Loss and Accuracy.

4. The results improve with higher number of permutation because the resulting representation of the neighbors in more permutation-invariant when more permutations are used. When more permutations are used we obtain a better estimate of the Reynold's operator of Janossy Pooling. Therefore, we obtain better results.

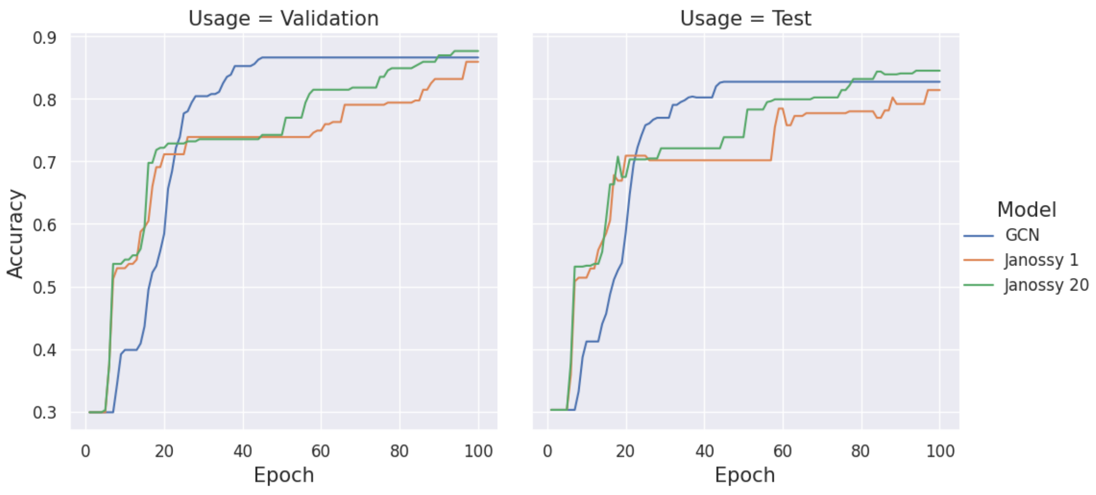


Figure 7: Training curves of accuracy for each epochs

3.2.3 Markov Chain Modeling (1.0 pts)

In the following LM related tasks, you will work with the Penn Treebank (PTB) dataset. The PTB dataset is a collection of words and sentences. In the provided loader, an additional word "eos" will be added to each sentences to force language model to learn "end-of-sentence" embeddings. Given the first several words of a sentence and its context, your task is to predict next word.

In k-order Markov chain language modeling, we are computing the statistics over training data. Suppose for the t-th word w_t, we have its context $c_t = \big[w_{t-k}, w_{t-k+1}, \cdots, w_{t-1}\big]$. If the words of context go out of the context, we will just use token OOV ("out-of-vocabulary") for padding.

$$P(w_t|c_t) = \frac{\#(c_t, w_t)}{\sum\limits_w \#(c_t, w)} \tag{9}$$

where $\#(c_t, w)$ is the number of appearance of sequence $\big[w_{t-k}, w_{t-k+1}, \cdots, w_{t-1}, w\big]$ in the whole training subset of the whole dataset.

Equation (9) could lead to zero probabilities. To solve this problem, we will use a simple smoothing schema.

$$P(w_t|c_t) = \begin{cases} \frac{\#(c_t,w)}{\Big(\sum\limits_w \#\big(c_t,w\big)\Big)+1} & \#(c_t, w) > 0 \\ \frac{1}{\Big(\sum\limits_w \#\big(c_t,w\big)\Big)+1} \frac{1}{\#_{\text{oov}}\big(c_t\big)} & \#(c_t, w) = 0 \end{cases} \tag{10}$$

where $\#_{\text{oov}}\big(c_t\big) = \sum_w \mathbb{1}\Big(w \Big| \#\big(c_t, w\big) = 0\Big)$ is number of unique out-of-vocabulary cases.

Related Modules:

- homework/template/markov.py

Action Items:

1. Finish the missing part of the file. Feel free to modify the files as long as you can use Markov chain to predict cross entropy loss and accuracy on test data correctly.
2. Run `python main.py ... --num-internals 1`, `python main.py ... --num-internals 2`, and `python main.py ... --num-internals 10`. **Report their perplexity on test subsets.** (You should achieve this from log file with pseudo early stopping.) Perplexity is a common metric used to evaluate the performance of language models, and it is indeed the exponent of cross entropy loss in the homework.

$$\exp\Big(\frac{1}{N}\sum_{i=1}^{N} -\log P\big(w_i\big|w_1, \cdots, w_{i-1}\big)\Big) \tag{11}$$

Solution: The following action items are performed.

1. The missing parts of the file `markov.py` are implemented.
2. The perplexity on the test subset for different k-order Markov Chain Language Models are shown in Table 3.

k	Test Perplexity
1	448.56
2	1735.62
10	8955.97

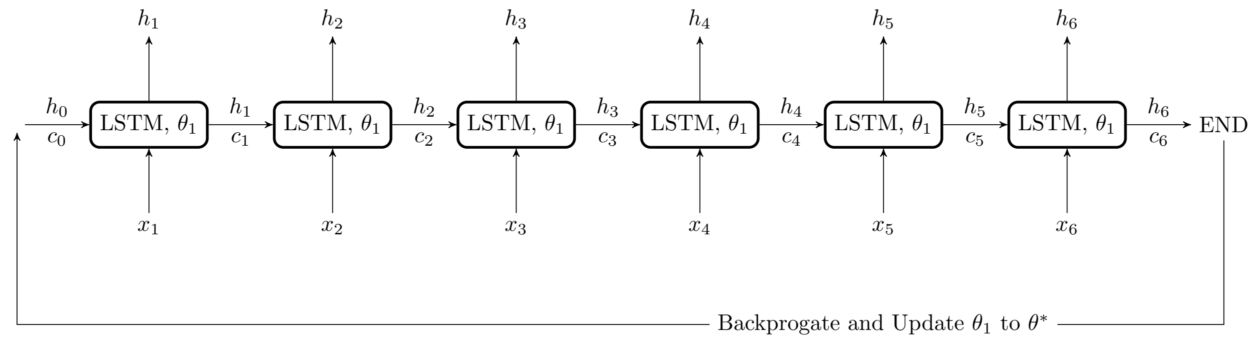
Table 3: Perplexity on Test Subset

3.2.4 LSTM Modeling (1.0 pts)

In LSTM language modeling, we are computing the statistics over training data.

$$P\big(w_t \big| w_1, \cdots, w_{t-1}\big) \tag{12}$$

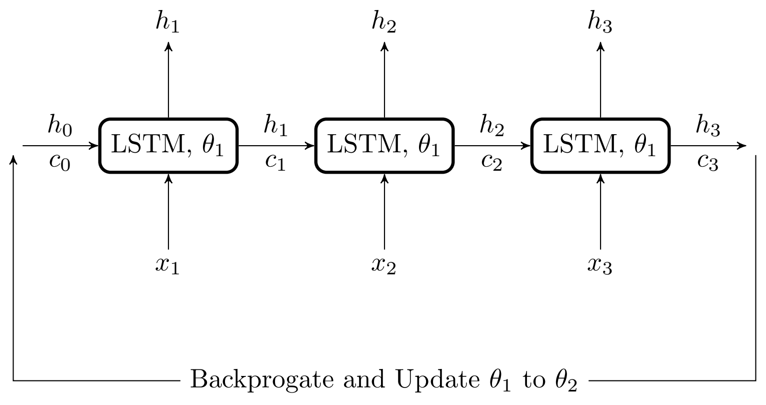
As mentioned in the class, one of the problem of LSTM is gradient vanishing or explosion over long sequences. To solve this problem, we often do truncated backpropagation through time rather than backprogration though the whole sequence. To be specific, for a sequence $x_1, x_2, x_3, x_4, x_5, x_6$ with common backpropagation, a full backprogation will work in the following way:



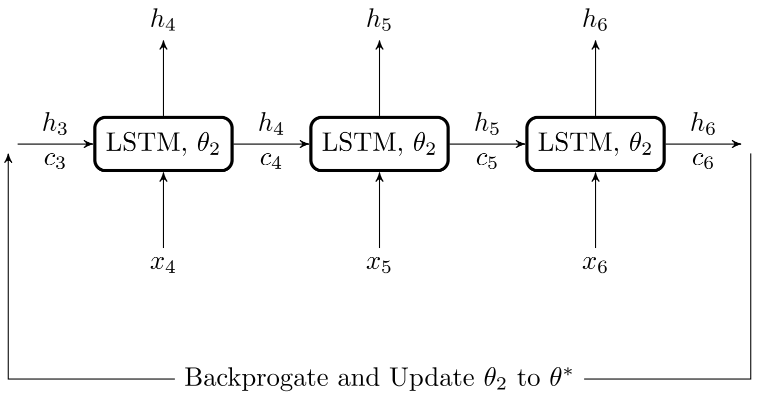
h_1 h_2 h_3 h_4 h_5 h_6
h_0 c_0
LSTM, θ_1
h_1 c_1 h_2 c_2 h_3 c_3 h_4 c_4 h_5 c_5 h_6 c_6
END
x_1 x_2 x_3 x_4 x_5 x_6
Backprogate and Update θ_1 to θ^*

We will feed forward all 6 time steps with parameter θ_1, compute the loss over output all 6 time steps, backpropagate gradients through all 6 time steps and get new parameter θ^*.

For a truncated backprogragation through time with length 3, a full backprogation will work in the following way:



h_1 h_2 h_3
h_0 c_0
LSTM, θ_1
h_1 c_1 h_2 c_2 h_3 c_3
x_1 x_2 x_3
Backprogate and Update θ_1 to θ_2



We will treat the first 3 time steps as a full sequence, and feed forward and backpropagate just like the previous one with parameter θ_1. After the parameter are updated to θ_2, we take the last hidden states h_3, c_3 as the initial hidden states of next process. Take the last 3 time steps as a full sequence (because there is not enough time step to generate a 3-time-step sequence), and feed forward and backpropagate just like the previous one but with parameter θ_2 and initial hidden states h_3, c_3. After parameters are updated, we get new parameter θ^*.

For using batches, we just equally divide the whole dataset sequence into several independent segments. For example, for sequence x_1 to x_6, if we want to use a batch size of 2, we just split it into 2 sequences x_1, x_2, x_3 and x_4, x_5, x_6, and training them together.

Warning: For this task you really need to use GPUs, otherwise the optimization will be very slow.

Related Modules:

- homework/structures.py
- homework/template/lstm.py

Action Items:

1. Understand the truncated backprogatation process, and finish the missing part of the file (t_bptt_preprocess) to support a data preprocessing for truncated backprogation through time process. You should correctly split training sequences to earn full points for this question.
2. Run `python main.py ... --truncate 5`, `python main.py ... --truncate 35`, and `python main.py ... --truncate 80`. **In the report, you should include, for each of the three runs:**
   (a) **Training curves (validation data) of loss function, i.e. plot the value of the loss for each epoch of training.**

(b) **Loss function of test data for the saved models. (You should achieve this from log file with pseudo early stopping.)**

Solution: The action items performed are as follow,

1. `t_bptt_preprocess` function is implemented. The input is fed to the forward function as a (truncate * batch_size) matrix (batch_size is the second dimension since we used LSTM with batch_first set to False).

2. (a) Training curves of loss function for each epoch of training is shown in Fig. 8,

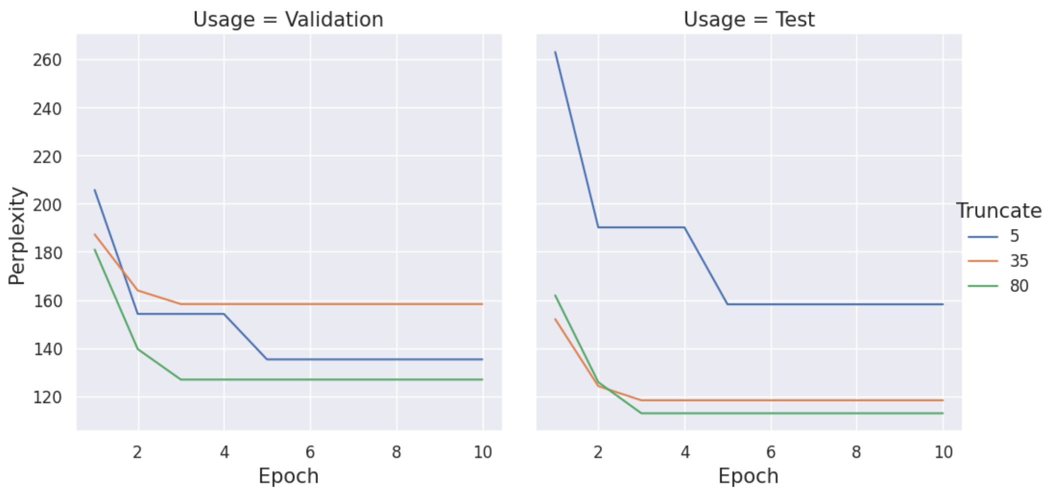


Figure 8: Training curves of loss function for each epoch of training

(b) The loss function of test data for models trained with different truncation size are as follows,

Truncation Size	Perplexity
5	158.2
35	125.62
80	114.71

Table 4: Caption

Submission Instructions

Please read the instructions carefully. Failed to follow any part might incur some score deductions.

PDF upload

The report PDF must be uploaded on Gradescope (see link at Brightspace).

Code upload

Naming convention: [firstname]_[lastname]_hw3
All your submitting code files, a ReadMe, should be included in one folder. The folder should be named with the above naming convention. For example, if my first name is "Bruno" and my last name is "Ribeiro", then for Homework 3, my file name should be "bruno_ribeiro_hw3".

Tar your folder: [firstname]_[lastname]_hw3.tar.gz
Remove any unnecessary files in your folder, such as training datasets. Make sure your folder structured as the tree shown in Overview section. Compress your folder with the the command: **tar -czvf bruno_ribeiro_hw3.tar.gz Homework3**.

Submit: TURNIN INSTRUCTIONS
Please submit your compressed file on **data.cs.purdue.edu** by turnin command line, e.g. **turnin -c cs690dl -p hw3 bruno_ribeiro_hw3.tar.gz** . Please make sure you didn't use any library/source explicitly forbidden to use. If such library/source code is used, you will get 0 pt for the coding part of the assignment. If your code doesn't run on scholar.rcac.purdue.edu, then even if it compiles in another computer, your code will still be considered not-running and the respective part of the assignment will receive 0 pt.