Advanced Learning for Text and Graph Data

Lab session 7: Learning on Sets / Graph-based Recommendations

Lecture: Prof. Michalis Vazirgiannis
Lab: Giannis Nikolentzos, Johannes Lutzeyer

Monday, January 24, 2022

This handout includes theoretical introductions, coding tasks and questions. Before the deadline, you should submit here a **.zip** file (max 10MB in size) named `Lab<x>_lastname_firstname.pdf` containing a `/code/` folder (itself containing your scripts with the gaps filled) and an answer sheet, following the template available here, and containing your answers to the questions. Your answers should be well constructed and well justified. They should not repeat the question or generalities in the handout. When relevant, you are welcome to include figures, equations and tables derived from your own computations, theoretical proofs or qualitative explanations. **One submission is required for each student. The deadline for this lab is February 9, 2022 11:59 PM**. No extension will be granted. Late policy is as follows: $]0, 24]$ hours late → -5 pts; $]24, 48]$ hours late → -10 pts; > 48 hours late → not graded (zero).

1 Learning objective

The goal of this lab is to introduce you to machine learning models for data represented as sets. Furthermore, you will be introduced to recommender systems. Specifically, in the first part of the lab, we will implement the DeepSets model. We will evaluate the model in the task of computing the sum of sets of digits and compare it against traditional models such as LSTMs. In the second part of the lab, you will learn about the problem of session-based recommendation, and you will implement a recently-proposed approach which transforms sessions into graphs and uses graph neural networks (GNNs) to deal with the problem. We will use Python 3.6, and the following two libraries: (1) PyTorch (`https://pytorch.org/`), and (2) NetworkX (`http://networkx.github.io/`).

2 DeepSets

Typical machine learning algorithms, such as the Logistic Regression classifier or Multi-layer Perceptrons, are designed for fixed dimensional data samples. Thus, these models cannot handle input data that takes the form of sets. The cardinalities of the sets are not fixed, but they are allowed to vary. Therefore, some sets are potentially larger in terms of the number of elements than others. Furthermore, a model designed for data represented as sets needs to be invariant to the permutations of the elements of the input sets. Formally, it is well-known that a function f transforms its domain $\mathcal{X}$ into its range $\mathcal{Y}$. If the input is a set $X = \{x_1, \ldots, x_M\}$, $x_m \in \mathfrak{X}$, i.e., the input domain is the power set $\mathcal{X} = 2^{\mathfrak{X}}$, and a function $f : 2^{\mathfrak{X}} \to Y$ acting on sets must be permutation invariant to the order of objects in the set, i.e., for any permutation π : $f(\{x_1, \ldots, x_M\}) = f(\{x_{\pi(1)}, \ldots, x_{\pi(M)}\})$. Learning on sets emerges in several real-world applications, and has attracted considerable attention in the past years.

2.1 Dataset Generation

For the purposes of this lab, we consider the task of finding the sum of a given set of digits, and we will create a synthetic dataset as follows: Each sample is a set of digits and its target is the sum of its elements. For instance, the target of the sample $X_i = \{8, 3, 5, 1\}$ is $y_i = 17$. We will generate $100,000$ training samples by randomly sampling between 1 and 10 digits ($1 \leq M \leq 10$) from $\{1, 2, \ldots, 10\}$. With regards to the test set, we will generate $200,000$ test samples of cardinalities from 5 to 100 containing again digits from $\{1, 2, \ldots, 10\}$. Specifically, we will create $10,000$ samples with cardinalities exactly 5, $10,000$ samples with cardinalities exactly 10, and so on.

Task 1
Fill in the body of the `create_train_dataset()` function in the `utils.py` file to generate the training set (consisting of $100,000$ samples) as discussed above. Each set contains between 1 and 10 digits where each digit is drawn from $\{1, 2, \ldots, 10\}$. To train the models, it is necessary that all training samples have identical cardinalities. Therefore, we pad sets with cardinalities smaller than 10 with zeros. For instance, the set $\{4, 5, 1, 7\}$ is represented as $\{0, 0, 0, 0, 0, 0, 4, 5, 1, 7\}$ (Hint: use the `randint()` function of NumPy to generate random integers from $\{1, 2, \ldots, 10\}$).

Task 2
Fill in the body of the `create_test_dataset()` function in the `utils.py` file to generate the test set (consisting of $200,000$ samples) as discussed above. Each set contains from 5 to 100 digits again drawn from $\{1, 2, \ldots, 10\}$. Specifically, the first $10,000$ samples will consist of exactly 5 digits, the next $10,000$ samples will consist of exactly 10 digits, and so on.

2.2 Implementation of DeepSets

It can be shown that if $\mathfrak{X}$ is a countable set and $\mathcal{Y} = \mathbb{R}$, then a function $f(X)$ operating on a set X having elements from $\mathfrak{X}$ is a valid set function, i.e., invariant to the permutation of instances in X, if and only if it can be decomposed in the form $\rho(\sum_{x \in X} \phi(x))$, for suitable transformations ϕ and ρ.

DeepSets achieves permutation invariance by replacing ϕ and ρ with multi-layer perceptrons (universal approximators). Specifically, DeepSets consists of the following two steps:

- Each element x_i of each set is transformed (possibly by several layers) into some representation $\phi(x_i)$.
- The representations $\phi(x_i)$ are added up and the output is processed using the ρ network in the same manner as in any deep network (e.g., fully connected layers, nonlinearities, etc.).

An illustration of the DeepSets model is given in Figure 1.

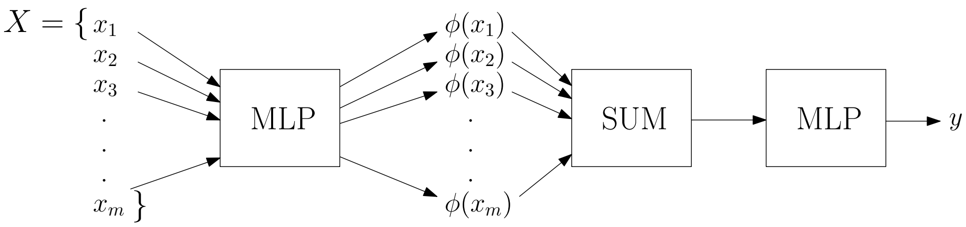

Figure 1: The DeepSets model.

Task 3
Implement the DeepSets architecture in the `models.py` file. More specifically, add the following layers:
– an embedding layer which projects each digit to a latent space of dimensionality h_1
– a fully-connected layer with h_2 hidden units followed by a tanh activation function
– a sum aggregator which computes the sum of the elements of each set
– a fully-connected layer with 1 unit since the output of the model needs to be a scalar (i.e., the prediction of the sum of the digits contained in the set)

Question 1 (5 points)
For the task that we perform here, i.e., determining the sum of a set of integers, what are the optimal MLP parameters (weights and bias) that we expect the DeepSets architecture learn?

We have now defined the DeepSets model. We will compare the DeepSets model against an LSTM, an instance of the family of recurrent neural networks. The next step is thus to define the LSTM model. Given an input set, we will use the LSTM hidden state output for the last time step as the representation of entire set.

Task 4
Implement the LSTM architecture in the `models.py` file. More specifically, add the following layers:
– an embedding layer which projects each digit to a latent space of dimensionality h_1
– an LSTM layer with h_2 hidden units
– a fully-connected layer which takes the LSTM hidden state output for the last time step as input and outputs a scalar

2.3 Model Training

Next, we will train the two models (i.e., DeepSets and LSTM) on the dataset that we have constructed. We will store the parameters of the trained models in the disk and retrieve them later on to make predictions.

Task 5
Fill in the missing code in the `train.py` file, and then execute the script to train the two models. Specifically, you need to generate all the necessary tensors for each batch.

2.4 Predicting the Sum of a Set of Digits

We will now evaluate the two models on the test set that we have generated. We will compute the accuracy and mean absolute error of the two models on each subset of the test set separately. We will store the obtained accuracies and mean absolute errors in a dictionary.

Task 6
In the `eval.py` file, for each batch of the test set, generate the necessary tensors for making predictions. Compute the output of two models. Compute the accuracy and mean absolute error achieved by the two models and append the emerging values to the corresponding lists of the `results` dictionary (Hint: use the `accuracy_score()` and `mean_absolute_error()` functions of scikit-learn).

We will next compare the performance of DeepSets against that of the LSTM. Specifically, we will visualize the accuracies of the two models with respect to the maximum cardinality of the input sets.

Task 7
Visualize the accuracies of the two models with respect to the maximum cardinality of the input sets (Hint: you can use the `plot()` function of Matplotlib).

Question 2 (5 points)
What are the architectural differences between the Graph Neural Network used for Graph-Level Tasks that we implemented in Lab 6 and the DeepSets architecture? What is the difference between a set, such as the ones taken as input by the DeepSets architecture and a graph without edges, which could be processed using for example a Graph Neural Network?

3 Graph-based Recommendations

Recommender systems have attracted a lot of attention in the past years since they allow users to locate potentially interesting items, while they also serve as filters helping users alleviate the problem of information overload. Most recommender systems are developed with the assumption that every user interacting with the system can be identified and a user profile can be built for each and every user. However, this is not always true, especially in the case of small retailers in e-commerce, news websites, etc. In these cases, instead of a matrix of ratings, we have access to a sequentially ordered log of user interactions such as item views, purchases, listening or viewing events. The goal of session-based recommendation is to assess the preferences or interests of users from such a small set of interactions.

In this part of the lab, we will implement a session-based recommender system which is known as SR-GNN. This system represents the input sessions as graphs and then capitalizes on GNNs to make predictions about the preferences of the users. Each session s can be modeled as a directed graph $G_s = (V_s, E_s)$. In this session graph, each node $v_i \in V_s$ represents an item. Each edge $(v_i, v_j) \in E_s$ represents the click behavior of the user, i.e., that the user first clicked item v_i and then item v_j in the session. Since several items may appear in the sequence repeatedly, the weights of the edges are normalized such that the outdegree (i.e., sum of weights of outgoing edges) of each node is equal to 1. Suppose for example that we have a dataset consisiting of sessions that correspond to sequences of songs the different users have listened to. An example of such a session is the following: $s =$ [”Gimme Shelter”, ”Layla”, ”Comfortably Numb”, ”Paranoid”, ”Layla”, ”Imagine”, ”Hotel California”]. The graph representation of that session is shown in Figure 2. The objective of session-based recommendation is to predict the next song that the user would listen to and to recommend that song to the user.

3.1 Dataset Generation and Preprocessing

We will evaluate the SR-GNN model on Diginetica, a dataset provided by DIGINETICA and its partners containing anonymized search and browsing logs, product data, anonymized transactions, and a large dataset of product images (the dataset was released in the context of the CIKM Cup 2016). We will only

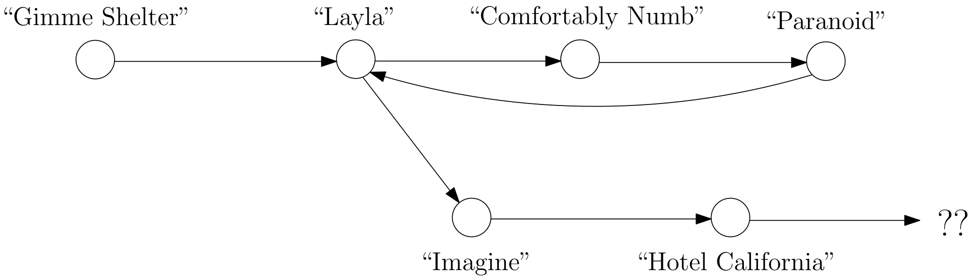

Figure 2: Example of a session.

use data related to transactions, and all sessions of length 1 and items appearing less than 5 times in the training and test sets have been filtered out.

Task 8
Fill in the body of the `load_dataset()` function in the `utils.py` file. The function loads and returns the training and test sets of the Diginetica dataset. Compute and print the number of training sessions and the number of test sessions. Furthermore, find and print the number of unique items and the largest item id in the dataset. The function also needs to return the largest item id in the dataset.

You are provided with a function (`generate_batches()`) which, given a list of sessions transforms them into graphs, splits the dataset into batches and generates all the necessary tensors to be used by the SR-GNN model. Specifically, for each batch of sessions, the function generates: (1) a sparse block diagonal matrix that contains the normalized adjacency matrices of all graphs; (2) a matrix that contains the ids of the nodes (i.e., items) of all the graphs of the batch; (3) a vector that indicates the graph to which each node belongs; (4) a vector that indicates the position in the matrix of features associated with the last item of each session of the batch; and (5) a vector that contains the ids of the target items. Then, the above NumPy/Scipy arrays are converted into PyTorch tensors.

3.2 Implementation of SR-GNN

We will next implement the following very simple message passing layer:

$$\mathbf{Z} = \text{MP}(\hat{\mathbf{A}}, \mathbf{X}) = \hat{\mathbf{A}} \ \mathbf{X} \ \mathbf{W}$$

where $\hat{\mathbf{A}}$ is the normalized adjacency matrix, $\mathbf{X}$ is a matrix that contains the representations of the nodes, and $\mathbf{W}$ is a trainable matrix (we omit bias for clarity).

Task 9
Fill in the body of the `forward()` function of the `MessagePassing` class in the `model.py` file. Implement the message passing layer presented above. More specifically, add the following layers:
- a fully-connected layer with h hidden units (i.e., $\mathbf{W} \in \mathbb{R}^{d \times h}$)
- a neighborhood aggregation layer

We will next implement the SR-GNN model. Note that the model does not make use of any external information, but each session is represented directly based on the nodes involved in that session. The first layer of the model is an embedding layer which projects all items into a vector space. Thus, this layer consists of a matrix $\mathbf{E} \in \mathbb{R}^{m \times d}$ where m is the number of all items and d the dimensionality of the embeddings (the i^{th} row of $\mathbf{E}$ corresponds to the embedding of the i^{th} item). Then, given a session

$s = [v_1, v_2, \dots, v_n]$ consisting of n unique items (some items may appear more than once), let $\hat{\mathbf{A}} \in \mathbb{R}^{n \times n}$ denote the normalized adjacency matrix of its graph representation and $\mathbf{X} \in \mathbb{R}^{n \times d}$ a matrix that contains the embeddings of the items of the session. The model then uses a GNN to update the representations of the session's nodes as follows:

$$\mathbf{Z} = \text{GNN}(\hat{\mathbf{A}}, \mathbf{X})$$

More specifically, the GNN that we will implement consists of two message passing layers. The first message passing layer is followed by a ReLU activation function as follows:

$$\mathbf{H} = \text{ReLU}\big(\text{MP}(\hat{\mathbf{A}}, \mathbf{X})\big)$$

where MP is the message passing layer defined above. The second layer of the model is again a message passing layer:

$$\mathbf{Z} = \text{MP}(\hat{\mathbf{A}}, \mathbf{H})$$

Next, the model computes a vector $\mathbf{s} \in \mathbb{R}^d$ for the entire session by combining the representation of the last-clicked item of the session with the representations of all items of the session (contained in matrix $\mathbf{Z}$). Let $\mathbf{v}_1, \dots, \mathbf{v}_n$ denote the representations of the nodes of session s, and $\mathbf{s}_l$ denote the representation of the last-clicked item, i.e., $\mathbf{s}_l = \mathbf{v}_n$. Let also $\mathbf{s}_g$ denote the global embedding of the session graph G_s which emerges by aggregating all node vectors (see below). Then, the representation of the session is the concatenation of the above two vectors:

$$\mathbf{s} = [\mathbf{s}_l | \mathbf{s}_g]$$

where $[\cdot|\cdot]$ denotes concatenation of two vectors. To compute the global embedding $\mathbf{s}_g$ of the session graph, the model employs a soft-attention mechanism as follows:

$$\begin{aligned} \alpha_i &= \mathbf{w}^{(3)\top} \sigma(\mathbf{W}^{(1)}\, \mathbf{v}_n + \mathbf{W}^{(2)}\, \mathbf{v}_i + \mathbf{b}^{(1)} + \mathbf{b}^{(2)}) \\ \mathbf{s}_g &= \sum_{i=1}^{n} \alpha_i\, \mathbf{v}_i \end{aligned}$$

where $\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \in \mathbb{R}^{d \times d}$ and $\mathbf{w}^{(3)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)} \in \mathbb{R}^d$ are trainable matrices / vectors / biases, and $\sigma(\cdot)$ is the sigmoid activation function. Finally, the model computes the hybrid embedding $\mathbf{s}_h$ by feeding the representation of the session into a fully connected layer:

$$\mathbf{s}_h = \mathbf{W}^{(4)}\, \mathbf{s} + \mathbf{b}^{(4)}$$

where matrix $\mathbf{W}^{(4)} \in \mathbb{R}^{d \times 2d}$ compresses two combined embedding vectors into the latent space $\mathbb{R}^d$, and $\mathbf{b}^{(4)}$ is the bias. To make a prediction, the model computes the score $\hat{\mathbf{z}}_i$ for each candidate item $i \in \{1, \dots, m\}$ by computing the inner product between its embedding $\mathbf{E}_{i:}$ and the representation of the session $\mathbf{s}_h$:

$$\hat{\mathbf{z}}_i = \mathbf{E}_{i:}\, \mathbf{s}_h$$

Finally, the model applies the softmax function to compute the output vector $\hat{\mathbf{y}}$:

$$\hat{\mathbf{y}} = \text{softmax}(\hat{\mathbf{z}})$$

where $\hat{\mathbf{z}} \in \mathbb{R}^m$ denotes the recommendation scores over all candidate items and $\hat{\mathbf{y}} \in \mathbb{R}^m$ denotes the probabilities of items to be directly clicked in sessions. The SR-GNN model is illustrated in Figure 3.

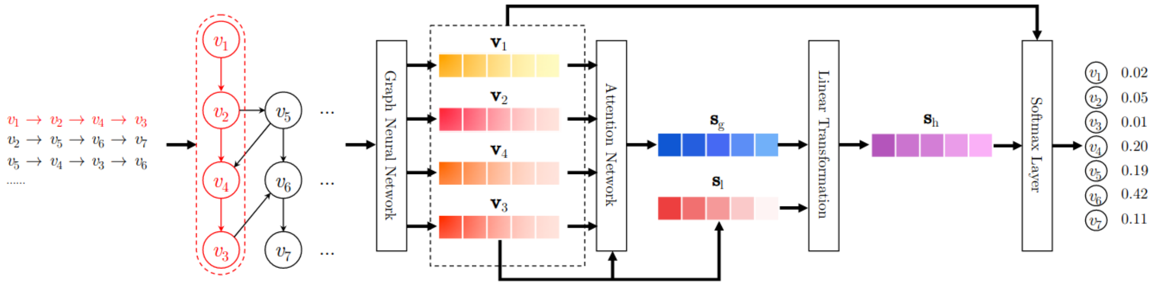

Figure 3: The SR-GNN model.

Task 10
Implement the architecture presented above in the `model.py` file. More specifically, add the following layers:
- an embedding layer which maps the nodes (i.e., items) to their d-dimensional embeddings
- a message passing layer with d hidden units followed by a ReLU activation function
- a dropout layer with p ratio of dropped outputs
- a message passing layer with d hidden units
- a dropout layer with p ratio of dropped outputs
- an indexing operation that extracts the representations of the last items of the sessions $\mathbf{s}_l$
- the attention mechanism presented above to compute $\mathbf{s}_g$ followed by a dropout layer with p ratio of dropped outputs
- an operator that concatenates $\mathbf{s}_g$ with $\mathbf{s}_l$
- a fully-connected layer with d hidden units
- a layer that multiplies the vector representations of the sessions with the embeddings of the items to compute a score for each item

Question 3 (5 points)
Why does the SR-GNN uses an embedding layer to map the nodes of the session graphs into a vector space and not a fully-connected layer instead?

3.3 Model Training and Evaluation

Finally, we will train and evaluate the model on the Diginetica dataset. We will iterate over the different batches to train the model. At each epoch, we will also evaluate the model and print the best performance so far. The loss function is the cross-entropy between the predictions and the ground-truth.

Task 11
Execute the `main.py` script to train and evaluate the model.

Question 4 (5 points)
Would it be a good idea to apply DeepSets to the above problem of predicting the next item of a session and why?

References

[1] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*, pages 3744–3753, 2019.

[2] Konstantinos Skianis, Giannis Nikolentzos, Stratis Limnios, and Michalis Vazirgiannis. Rep the Set: Neural Networks for Learning Set Representations. In *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics*, pages 1410–1420, 2020.

[3] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. Session-based Recommendation with Graph Neural Networks. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, pages 346–353, 2019.

[4] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep Sets. In *Advances in Neural Information Processing Systems*, pages 3391–3401, 2017.