ADVANCED LEARNING FOR TEXT AND GRAPH DATA

Lab session 7: Learning on Sets and Graph Generative Models

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

Tuesday, November 28, 2023

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 December 5, 2023 11:59 PM**. No extension will be granted. Late policy is as follows: $]0, 24]$ hours late $\rightarrow$ -5 pts; $]24, 48]$ hours late $\rightarrow$ -10 pts; > 48 hours late $\rightarrow$ 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 graph generative models. 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 graph generation problem, and you will implement a variational graph autoencoder to generate stochastic block model graphs. We will use Python, 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 $f : 2^{\mathfrak{X}} \rightarrow Y$ acting on sets must be permutation invariant to the order of objects in the set, i.e.,

for any permutation $\pi : f(\{x_1, \ldots, x_M\}) = f(\{x_{\pi(1)}, \ldots, x_{\pi(M)}\})$. 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 [3]:

- 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.

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)

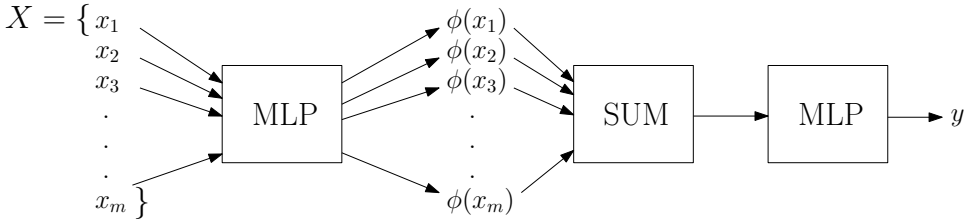

Figure 1: The DeepSets model.

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

Question 1 (4 points)
Please describe briefly whether the LSTM is a permutation invariant model and conclude whether you recommend the use of LSTMs on sets.

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 (GNN) 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 GNN?

3 Graph Generation with Variational Graph Autoencoders

In the second part of the lab, we will implement a variational graph autoencoder, and use it to generate synthetic graphs that are similar to the graphs the model was trained on. Variational autoencoders have attracted a lot of attention in the past years. Instead of embedding the input object to a vector, a variational autoencoder learns parameters of a distribution from which embeddings are sampled, i.e., a random sample $\mathbf{z}$ is drawn from the learned distribution rather than being generated directly from encoder. The decoder is a variational approximation which takes a representation $\mathbf{z}$ and produces the output.

3.1 Dataset

We will train the model that we will implement on a dataset that contains $1,000$ stochastic block model graphs. The stochastic block model is specified in terms of the following three parameters: (1) the number of nodes n; (2) a partition of the node set $\{1, 2, \ldots, n\}$ into r disjoint subsets, known as blocks; (3) a symmetric $r \times r$ matrix $\mathbf{P}$ of edge probabilities. The edge set is then sampled at random as follows: any two vertices that belong to blocks i and j are connected by an edge with probability $\mathbf{P}_{ij}$.

To generate each of the $1,000$ graphs, we sample the number of nodes n from $\{20, 21, \ldots, 40\}$. We then, sample the number of blocks r from $\{2, 3, 4, 5\}$. We group nodes into blocks of equal size (except the last block if n is not exactly divisible by r). The probability of connecting two nodes that belong to the same block by an edge was set equal to 0.8, while the probability of adding an edge between nodes that belong to different blocks was set equal to 0.05. Then, matrix $\mathbf{P}$ is defined as follows:

$$\mathbf{P}_{ij} = \begin{cases} 0.8 & \text{if } i = j \\ 0.05 & \text{otherwise} \end{cases}$$

In other words, $\mathbf{P}$ is an $r \times r$ matrix where the elements on the main diagonal are equal to 0.8, while the off-diagonal elements are set equal to 0.05.

Question 3 (8 points)
In the analysis of graphs we distinguish between homophilic and heterophilic cluster structure. Homophilic cluster structure describes a scenario in which the node set of a graph can be partitioned into subsets, called communities, such that many edges arise between nodes in equal communities and far fewer edges arise between nodes in different communities. Contrariwise, heterophilic cluster structure describes the scenario in which most edges arise between nodes in different communities and few edges are present between nodes in the same community.
1) For a stochastic block model with $r = 2$ please specify two edge probability matrices $\mathbf{P}$ from which homophilic and heterophilic graphs can be sampled.
2) Please calculate, showing your workings, the expected number of edges between nodes in different blocks of a stochastic block model with $n = 20$, containing 4 blocks of 5 nodes each. The diagonal elements of matrix $\mathbf{P}$ are set equal to 0.8, while its off-diagonal elements equal to 0.05).

3.2 Variational Graph Autoencoder

In the case of autoencoders for graph-structured data, the input to the model is a graph and the objective is to learn a low-dimensional representation for the graph. Let $\mathbf{A}$ be the adjacency matrix of a graph $G = (V, E)$ and $\mathbf{X}$ its feature matrix. For attributed graphs, these features may be set equal to the attribute vectors of the nodes. For instance, in biology, proteins are represented as graphs where nodes correspond to secondary structure elements and the feature vector of each secondary structure element contains its physical properties. For graphs without node labels and node attributes, these vectors can be initialized with a collection of local vertex features that are invariant to vertex renumbering (e.g., degree).

Variational graph autoencoders apply the idea of variational autoencoders on graph-structured data [1]. In our setting, the encoder produces a distribution for each graph. The distribution is usually parameterized as a multivariate Gaussian. Therefore, the encoder predicts the mean and standard deviation of the Gaussian distribution. The low-dimensional representation $\mathbf{z} \in \mathbb{R}^d$ of a graph G is then sampled from this distribution. A high-level illustration of such an autoencoder is given in Figure 2.

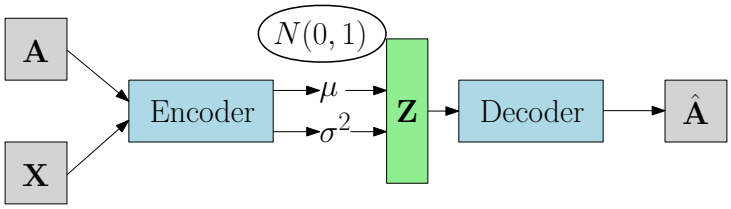

Figure 2: High-level illustration of a variational graph autoencoder.

Formally, the encoder is defined by a variational posterior $q_\phi(\mathbf{z}|G)$ and the decoder by a generative distribution $p_\theta(G|\mathbf{z})$, where ϕ and θ are learned parameters. In our setting, $q_\phi(\mathbf{z}|G) = q_\phi(\mathbf{z}|\mathbf{X}, \mathbf{A}) = N(\mathbf{z}|\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2))$. Both $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are typically produced by graph neural networks (GNNs). Therefore, we have that $\boldsymbol{\mu} = \text{GNN}_{\boldsymbol{\mu}}(\mathbf{X}, \mathbf{A})$ and $\log \boldsymbol{\sigma} = \text{GNN}_{\boldsymbol{\sigma}}(\mathbf{X}, \mathbf{A})$. Then, the latent variable $\mathbf{z}$ can be calculated as $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ where each element of $\boldsymbol{\epsilon}$ is drawn from $N(0, 1)$, and $\odot$ denotes the element-wise multiplication between two vectors.

The decoders of most graph generative models can be categorized into two groups: (1) sequential decoders which generate graphs in a set of consecutive steps [2]; and (2) one-shot decoders which generate the adjacency matrices in one single step [1]. One-shot decoders typically use a multi-layer

perceptron (MLP) to reconstruct the adjacency matrix:

$$\hat{\mathbf{A}} = \text{MLP}(\mathbf{z})$$

where $\hat{\mathbf{A}}$ is the reconstructed adjacency matrix.

3.3 Implementation of Variational Graph Autoencoder

Given the adjacency matrix $\mathbf{A}$ of a graph, we will first normalize it as follows:

$$\bar{\mathbf{A}} = \tilde{\mathbf{D}}^{-1}\,\tilde{\mathbf{A}}$$

where $\tilde{\mathbf{A}} = \mathbf{A}+\mathbf{I}$, and $\tilde{\mathbf{D}}$ is a diagonal matrix such that $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$. The above formula adds self-loops to the graph, and normalizes each row of the emerging matrix such that the sum of its elements is equal to 1. This normalization trick addresses numerical instabilities which may lead to exploding/vanishing gradients when used in a deep neural network model.

We will next implement the Variational Graph Autoencoder. As discussed above, the encoder of the model computes $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ where both $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ correspond to a shared GNN followed by different projection layers. In our implementation, the GNN will consist of two message passing layers. Thus, we have:

$$\begin{aligned}
\mathbf{H}^{(1)} &= \text{MLP}_1\big(\bar{\mathbf{A}}\,\mathbf{X}\big)\\
\mathbf{H}^{(2)} &= \text{MLP}_2\big(\bar{\mathbf{A}}\,\mathbf{H}^{(1)}\big)\\
\mathbf{h} &= \text{READOUT}(\mathbf{H}^{(2)})\\
\mathbf{h}_G &= \mathbf{h}\,\mathbf{W} + \mathbf{b}
\end{aligned}$$

Task 8
Implement the encoder architecture presented above in the `model.py` file. More specifically, add the following layers:
– a message passing layer where the message passing operation is followed by an MLP (use the `torch.mm()` function to perform a matrix-matrix multiplication)
– a second message passing layer where the message passing operation is again followed by another MLP (use the `torch.mm()` function to perform a matrix-matrix multiplication). Let h_1 denote the output size of this second MLP
– a readout function that computes the sum of the node representations
– a fully-connected layer that produces $\mathbf{h}_G$

Once $\mathbf{h}_G$ is computed, we employ two fully connected layers to compute $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ as follows:

$$\begin{aligned}
\boldsymbol{\mu} &= \mathbf{h}_G\,\mathbf{W}_{\boldsymbol{\mu}} + \mathbf{b}_{\boldsymbol{\mu}}\\
\log\boldsymbol{\sigma} &= \mathbf{h}_G\,\mathbf{W}_{\boldsymbol{\sigma}} + \mathbf{b}_{\boldsymbol{\sigma}}
\end{aligned}$$

Then, $\mathbf{z}$ is calculated as $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ where each element of $\boldsymbol{\epsilon}$ is drawn from $N(0,1)$.

Task 9
Add the following layers in the of the `loss_function()` function of the variational autoencoder (in the `model.py` file):
– a fully-connected layer with h_2 hidden units (i.e., $\mathbf{W}_{\boldsymbol{\mu}} \in \mathbb{R}^{h_1 \times h_2}$) that produces $\boldsymbol{\mu}$
– a fully-connected layer with h_2 hidden units (i.e., $\mathbf{W}_{\boldsymbol{\sigma}} \in \mathbb{R}^{h_1 \times h_2}$) that produces $\log \boldsymbol{\sigma}$

The decoder is implemented as follows:

$$\begin{aligned}\mathbf{T}_1 &= \text{MLP}(\mathbf{z}) \\ \mathbf{T}_2 &= \mathbf{T}_1\mathbf{W} + \mathbf{b} \\ \hat{\mathbf{A}} &= \sigma\left(\frac{\mathbf{T}_2 + \mathbf{T}_2^\top}{2}\right)\end{aligned}$$

where $\sigma(\cdot)$ is the sigmoid function. Note that the output of the projection layer that follows the MLP is a n^2-dimensional vector and needs to be reshaped such that $\mathbf{T}_2 \in \mathbb{R}^{n \times n}$. Matrix $\mathbf{T}_2$ is not necessarily symmetric, thus we compute the sum of $\mathbf{T}_2$ and its transpose to produce a symmetric matrix.

Task 10
Implement the decoder architecture presented above in the `model.py` file. More specifically, add the following layers:
– a MLP that consists of a series of fully connected layers. Between one fully connected layer and the next, apply the ReLU activation function, layer normalization and dropout
– a fully connected layer that transforms the h-dimensional output of the MLP (for each graph) into a n^2-dimensional vector (i.e., $\mathbf{W} \in \mathbb{R}^{h \times n^2}$)
– reshape the output of the fully connected layer such that for each graph the output is an $n \times n$ matrix (use the `torch.reshape()` function)
– compute the sum of the $n \times n$ matrix and its transpose and divide by 2 (use the `torch.transpose()` function)

l'objectif est que \bar{A} soit le plus proche de \tilde{A} possible

Note that the elements of matrix $\hat{\mathbf{A}}$ take values between 0 and 1, and ideally, we would like these values to be as close as possible to the correponding values contained in $\tilde{\mathbf{A}}$. To measure how well the model reconstructs the input data, i.e., the adjacency matrix, we will employ the binary cross entropy as follows:

$$\mathcal{L} = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbf{A}_{ij}\log(\hat{\mathbf{A}}_{ij}) + (1 - \mathbf{A}_{ij})\log(1 - \hat{\mathbf{A}}_{ij}) \tag{1}$$

For very sparse adjacency matrices $\mathbf{A}$, it can be beneficial to re-weight the nonzero terms or alternatively sub-sample pairs of nodes for which $\mathbf{A}_{ij} = 0$. We choose the former strategy for our implementation.

Question 4 (3 points)
The reconstruction loss in (1) is appropriate in the context of unweighted graphs. Could you briefly propose a loss function that would be more suitable for the reconstruction of weighted graphs, i.e., graphs for which the entries of the adjacency matrix are not constrained to only take values equal to 0 or 1?

3.4 Graph Generation

Once the model is trained, we can generate graphs that look like the ones the model was trained on by sampling vectors from the mutivariate standard normal distribution and feeding them into the decoder. We can thus use the model to generate stochastic block model graphs.

Task 11
– Create a tensor $\mathbf{Z} \in \mathbb{R}^{5 \times d}$ consisting of 5 rows where each row is a vector sampled from the multivariate normal distribution (use the `torch.randn()` function)
– Execute the `main.py` file to train the model on the dataset that contains stochastic block model graphs and to generate and visualize 5 synthetic graphs

References

[1] Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. In *Proceedings of the 27th International Conference on Artificial Neural Networks*, pages 412–422, 2018.

[2] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. In *Proceedings of the 35th International Conference on Machine Learning*, pages 5708–5717, 2018.

[3] Manzil Zaheer, Satwik Kottur, Siamak Ravanbhakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J Smola. Deep Sets. In *Advances on Neural Information Processing Systems*, pages 3394–3404, 2017.