MVA - Graphs in Machine Learning
TP3: Graph Neural Networks

Ariane ALIX

December 20th, 2019

---

1 Neural Relational Inference

This practical session is based on the paper Neural Relational Inference for Interacting Systems by Kipf et al., 2018. We will use the following material provided by Marc Lelarge and Timothée Lacroix: `https://github.com/timlacroix/nri_practical_session`.

1.1 Motivation and problem formulation

A wide range of dynamical systems can be seen as a group of interacting components. For example, we can think of a set of 2-dimensional particles coupled by springs. Assume that we are given only a set of trajectories of such interacting dynamical system. How can we learn its dynamical model in an unsupervised way?

Formally, we are given as input a set of trajectories of N objects, and each trajectory has length T. Each object i, for $i = 1, ..., N$, is represented by a vertex v_i. Let x_i^t be the feature vector of object i at time t (e.g., position and velocity) with dimension D. Let $x^t = \{x_1^t, ..., x_N^t\}$ be the set of features of all N objects at time t and let $xi = (x_i^1, ..., x_i^T)$ be the trajectory of object i. The input data can be stored in a 3-dimensional array x of shape $NxTxD$, denoted by $x = (x^1, ..., x^T)$, such that $x_{i,t,d}$ is the d-th component of the feature vector of object i at time t.

In addition, we assume that the dynamics can be modeled by a graph neural network (GNN) given an unknown graph z where $z_{i,j}$ represents the discrete 1 edge type between objects v_i and v_j.

In this context, we want to learn, simultaneously:

- The edge types $z_{i,j}$ (edge type estimation);
- A model that, for any time t, takes x^t as input and predicts x^{t+1} as output (future state prediction).

1.2 Model

The Neural Relational Inference (NRI) model consists of:

- An encoder that uses trajectories $x = (x^1, ..., x^T)$ to infer pairwise interaction vectors $z_{i,j} \in \mathbb{R}^K$ for i, j in $\{1, ..., N\}$, where K is the number of edge types.
- A decoder that takes x^t and $z = \{z_{i,j}\}_{i,j}$ as input to infer x^{t+1}.

Both the encoder and the decoder are implemented using graph neural networks. For more details, read Section 3 of the paper here.

2 Questions

2.1 Explain what are the edge types $z_{i,j}$

In our case (evolution of a set of 5 2-dimensional atoms), $z_{i,j}$ can take the value 0 or 1. 1 means that the atoms i and j interact (equivalently, that they are linled by a spring), and 0 means that they do not.

The following figure represents these 5 atoms moving in time (50 time steps, one color per atom), and the lines between them represent their interactions. The image on the right is their corresponding interactions matrix.

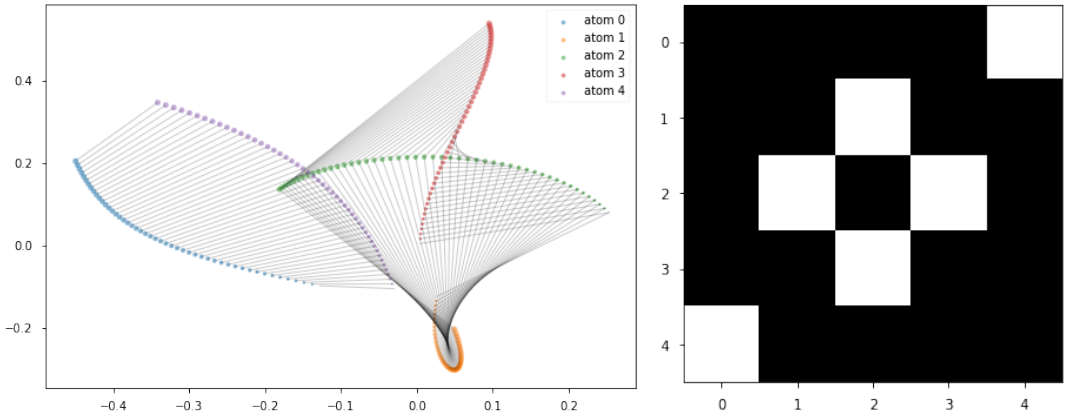

Figure 1: Evolution of the atoms in time and corresponding interactions matrix

More generally, you could have more interaction types than 1 or 0 for linked/not linked. Science consider four interactions between objects: gravity, electromagnetism, weak interaction and strong interaction. By observing the trajectories and distances of particles, we could deduce which interaction is at play. For example, repulsive forces between atoms are often caused by electromagnetic interactions.

2.2 In the NRI model, explain how the encoder and the decoder work.

We consider a fully-connected graph where the nodes are the particles with associated features (2D location and velocity in the example), and the edges are the interactions between them.

The encoder

Its role is to predict the interactions $z_{i,j}$ between the particles by looking at their trajectories $(x^1, ..., x^T)$. The encoder of the NRI model described in the paper `https://arxiv.org/pdf/1802.04687.pdf` is based on a GNN (Graph Neural Network) used on the graph that represents the particles system.

The GNN consists 3 operations, each using a neural network: passing information from vertices to edges, then from edges to vertices using the previous results, then back from vertices to edges using the results from the previous step. The goal of these exchanges of information is to use binary terms (look at pair of nodes one by one), but still take into account possible multiple interactions: thanks to that the information on an edge (i, j) will take into account the information coming from other edges connected to i and j.

The last step consists in applying a *softmax* to the ouput of the GNN on the graph to get a probability distribution of the interactions given the trajectories.

This could be summarized as :

$$q_\phi(z_{i,j}|x) = \text{softmax}(f_{enc,\phi}(x)_{i,j})$$

where $f_{enc,\phi}$ is the GNN and ϕ is its set of parameters.

The decoder

Its role is to learn the dynamical model of the particles; and from there predict the next state of the system. It takes the results from the encoder (the probability distribution of the interactions) as an input. It is also based on a GNN.

It has two steps: passing information from vertices to edges then from edges to vertices to estimate the movement between t and $t + 1$. Each step uses a different Neural Network, that also depends on the type of edge. The type of edge used is either a sample from the outputed probability distribution from previous step, either a weighted sum using those probabilities.

The GNN finally returns a predicted position μ_i^{t+1} for each particle at time $t + 1$. From this we can deduce a probability distribution given previous step, interaction graph z and a fixed variance:

$$p(x_i^{t+1}|x^t, z) = \mathcal{N}(\mu_i^{t+1}, \sigma\mathbf{I})$$

.

2.3 Explain the LSTM baseline used for joint trajectory prediction. Why is it important to have a “burn-in” phase?

LSTM stands for Long Short-Term Memory, and is a Recurrent Neural Network (RNN). One of the main difference with the NRI model is that it is supervised.

For one system of particles

It runs over the series of states of the system (meaning for each time step t), and each step used the previous prediction as input except in the "burn-in" phase taht uses the ground truth. Each step is composed of 3 sub-steps:

- The data of state t is passed through an encoder which is a Multi-Layers Perceptron (MLP), that encodes the configurations of the system,
- An RNN is applied on the encoded input,
- The output configuration is decoded by another MLP, which gives us the differnce (in position and velocity for each particle) between time step t and $t + 1$.

The interesting thing about using an RNN here is that it remembers things learns from the previous time step of a system, instead of only during training like classical networks. This is what the hidden states are here for: as a consequence a same input could produce a different output depending on previous inputs in the series.

On all the systems of the training dataset

The algorithm described above is applied succesively to all the systems in the training dataset. At the end of the application on each system, a loss is computed and the optimizer takes a step in the gradient direction to optimize the parameters of the LSTM.

In the Notebook implementation, the loss used is the Negative Log-Likelihood between prediction and ground truth, which can be written as $loss = \frac{(\text{pred} - \text{ground\_truth})^2}{2\sigma^2}$ where σ is the chosen variance.

The "burn-in" phase

At the beginning of each application of the LSTM on a system, there is a "burn-in" phase that consists in using the ground truth as input instead of the prediction from the previous step like in the rest of the LSTM. The definition of "burn-in" for hardware production is "the process by which components of a system are exercised prior to being placed in service (and often, prior to the system being completely assembled from those components). This testing process will force certain failures to occur under supervised conditions so an understanding of load capacity of the product can be established". In the case of the RNNs, it means setting preceding observations for the RNN to learn good initial states and predicts the next steps from there.

2.4 Consider the training of the LSTM baseline. Notice that the negative loglikelihood is lower after the burn-in than before. Why is this surprising? Why is this happening?

Below is a plot of the losses after each application of the model on a system of particle (prediction on its 50 time steps for each system):

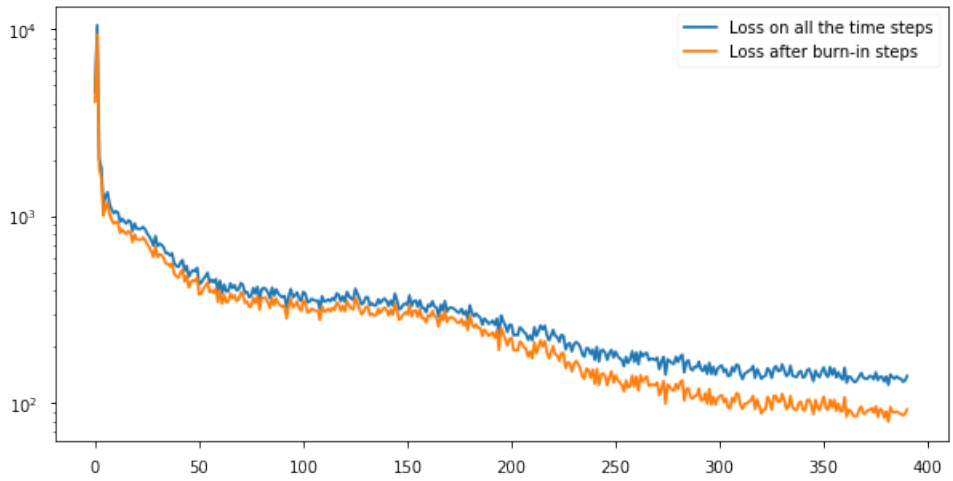

Figure 2: Evolution of Negative Log-Likelihood during training with LSTM (logarithmic scale)

We notice that the loss on the time steps after the "burn-in" ones is always lower. This is surprising because we could intuitively think that the prediction would be better on the "burn-in" steps where the inputs are taken from the ground truth.

It is happening because the RNN is not perfect and because it can not adjust its weights so that the prediction of the first steps matches the reality exactly, and because those steps are a prefix of states on which the network does not receive updates: the information in the hidden nodes does not match the state received as input. However, the steps after the "burn-in" can use all the information accumulated during that phase and are correctly updated.

2.5 Consider the problem of trajectory prediction. What are the advantages of the NRI model with respect to the LSTM baseline?

The advantages I noted are:

- Unlike the LSTM, the NRI model is unsupervised. It means that it can be used to predict the futur states of a system without knowing and using any ground truth to train it. It is therefore more practical when faced with real-worl problems, for which we often do not have enough or even no ground truth.
- In the appendix of the paper, it is shown that the NRI is more accurate for the prediction of trajectories for long sequences. Their explanation is that the LSTM fails to capture the dynamics of the system and will therefore go "out-of-sync" after a certain numbers of steps, while the NRI captures this dynamic with the learned interactions graph.

- The training of the NRI is much faster than the LSTM: in the Colab Notebook, 1 epoch of LSTM took more than 6 min and 30 seconds, while 1 epoch of NRI took around 2 minutes.

2.6 Consider the training the of NRI model. What do you notice about the edge accuracy during training? Why is this surprising?

We plotted the evolution of the edge accuracy at each step of 2 training epochs for the Encoder:

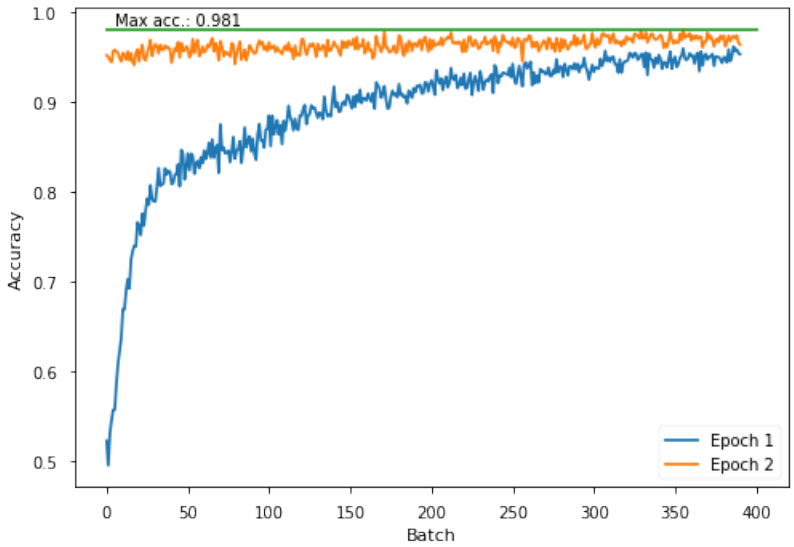

Figure 3: Evolution of the edge accuracy during the Encoder training

And this is the edge accuracy at each step of 10 training epochs for the Neural Relational Inference model:

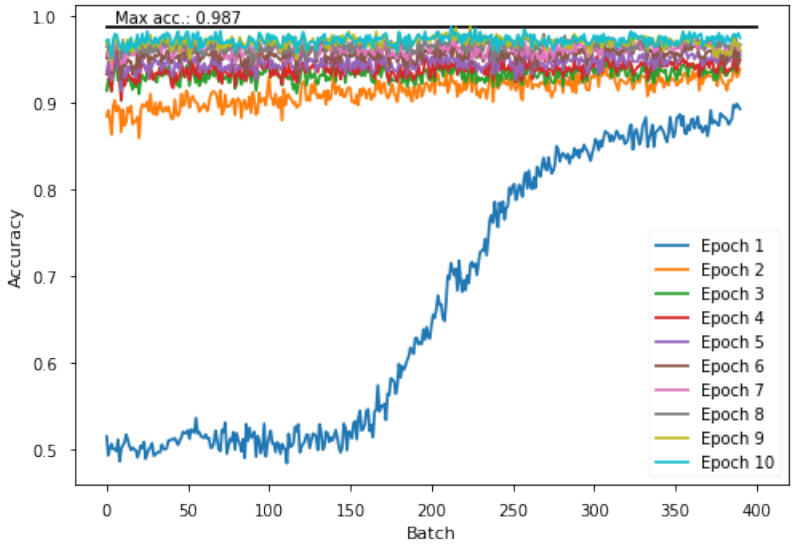

Figure 4: Evolution of the edge accuracy during global NRI training

What is surprising bere is the jump in accuracy after the first 150 batches or so. It is even more surprising since the training of the Encoder by itself reached a high accuracy much faster.

2.7 What do you expect to happen with the NRI model when there is no interaction between the objects?

Since the NRI model is good at identifying the correct edge types, including the type meaning that there is no interaction, it should also perform well when there is no interaction. In the appendix of the NRI paper (page 11), they present the results of a test where they tried the NRI model on "spring" systems without any interaction between the particles. It could correctly identify 98.4% of the edges.

However, this edge accuracy is lower than the one on typical case (99.9%, cf. page 6 of the paper). Since the identification of the interactions between particles was what made the NRI more accurate than the LSTM in the prediction of the future states, we can expect a decrease of performance on that matter and results closer to the ones of the LSTM.

2.8 Bonus

Plotting the ground truth and the prediction with the final trained NRI model for the first 3 systems of particles:

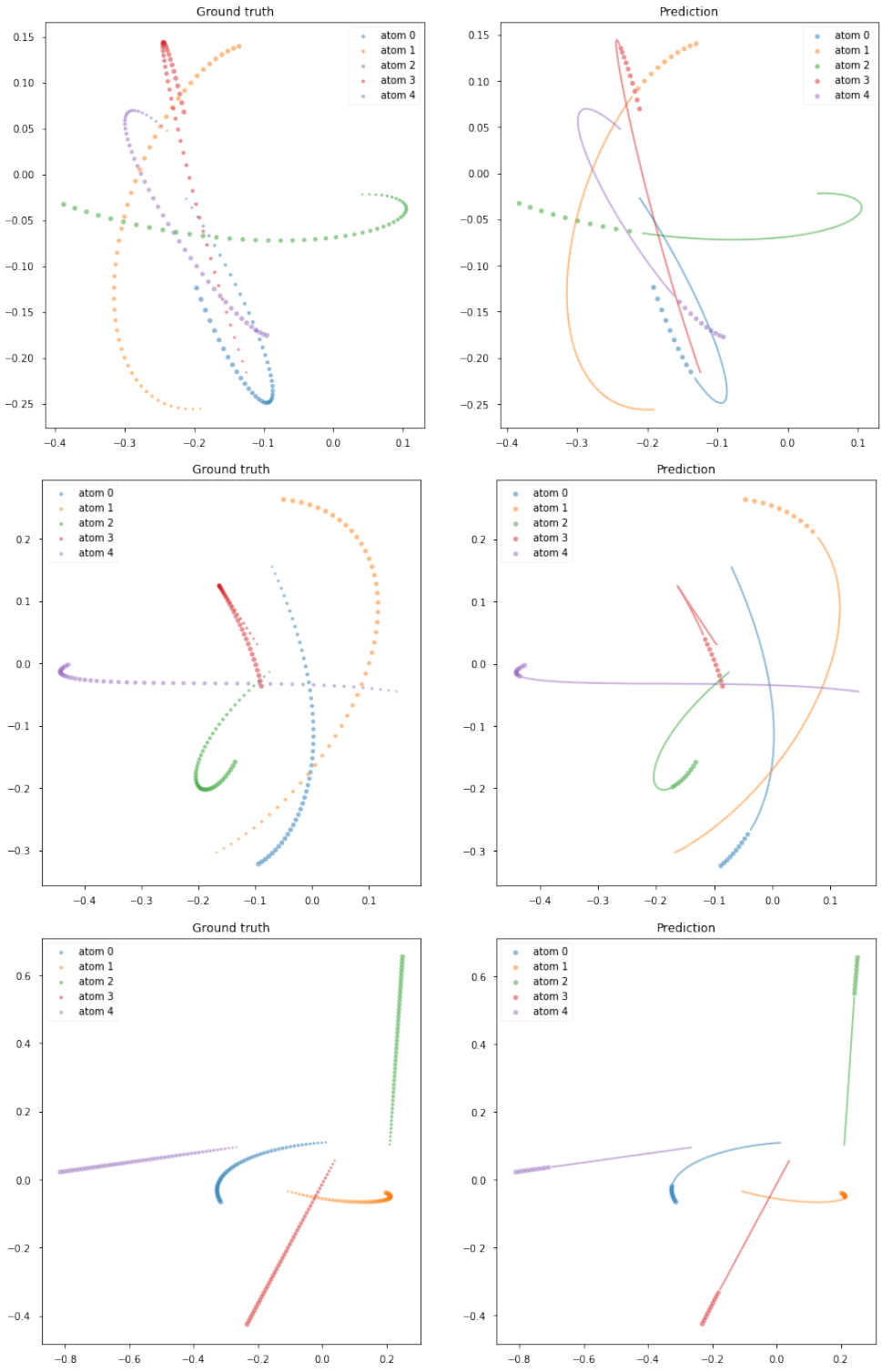

Figure 5: Ground truth vs prediction of the NRI for 3 systems
(line for the "burn-in" phase, dots for after)

And here is a close-up on the atom 0 of the last system:

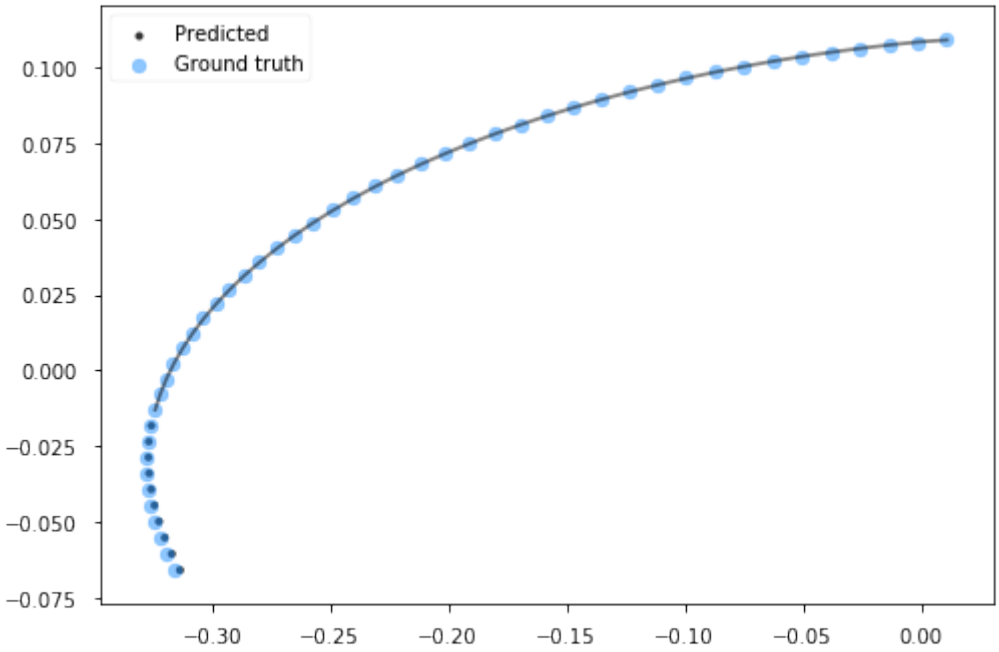

Figure 6: Superposition for the trajectory for one atom