ARTICLE ANALYSIS

By : Thibault Lahire (thibault.lahire@student.isae-supaero.fr)

Abstract
This document is a report done for the course "Graphs in Machine Learning". It is based on the understanding of the paper *Neural Relational Inference for Interacting Systems* by Kipf et al., 2018, and the github repository `https://github.com/timlacroix/nri_practical_session`.

1. Explain what are the edge types $\mathbf{z}_{i,j}$.

The NRI model is an encoder-decoder structure. The input of the NRI can be seen as a 3-dimensional array $\mathbf{x}$ of shape $N \times T \times D$, as explained in the TP handout. This 3D-tensor is a set of N objects (or nodes) that we observe during T time steps. At each time step, a node has D features.

The goal of the encoder is to learn from this tensor relations between objects. Since we want to represent these relations with a graph structure, we use a GNN (Graph Neural Network) as an encoder, and we expect it to output K graphs. K is also called the number of edge types.

If we note $e_{i,j}$ an edge from node v_i to node v_j, then the encoder is supposed to output K edges $e_{i,j}$. Let's note the k-th edge $e^k_{i,j}$. These K edges can be concatenated in $\mathbf{z}_{i,j}$, which is of dimension K.

K is a hyperparameter that is chosen before running the algorithm. We say that there are K edge types in our network. In the paper studied, K is generally equal to 2. Sometimes it has a higher value (for example 4) but experiments with 20 edge types have lead to overfitting.

It is important to note that the values behind the edges $e_{i,j}$ are learnt by the network, so they do not have necessarily a clear meaning or a physical interpretation.

The vector $\mathbf{z}_{i,j}$ obeys certain rules. In theory, we want the vector $\mathbf{z}_{i,j}$ to be a one-hot vector. In this case, if the 1 is in position k, then the edge from v_i to v_j exists only in the k-th graph. This way, we can represent K types of relations between nodes.

In practice, using a one-hot vector is a bad idea because we cannot back-propagate information with such a vector. Hence, we use a relaxation of the one-hot case, where $\mathbf{z}_{i,j}$ is a vector for which its components sum to one.

The case of a one-hot vector is shown in Fig. [11]. There are 5 nodes. A node does not interact with itself, so the diagonal is white. For each tile outside the diagonal, if the tile is white (which means 0) for an edge type, it is black (which means 1) for the other one.

The case of a "relaxed" one-hot vector can be seen in Fig. [8]. Instead of having black and white tiles, we have dark blue and light blue tiles. Once again, there is a complementarity. If a tile (outside the diagonal) is dark blue for a given edge type, then it is light blue for the other one.

In the "relaxed" case, we can no more say what I wrote some lines above : "*if the* 1 *is in position* k*, then the edge from* v_i *to* v_j *exists only in the* k*-th graph*". In this case, an edge from v_i to v_j may exist in multiple graphs, but with different coupling constants.

To conclude my answer to this question, I want to add that "edge type" is a flexible concept that can support some priors. Indeed, the first edge type can be "hard-coded" to

"indicate the presence of non-edge in the structure". In more simple words, this means we can assign to the first edge type the role of learning the existence or the absence of an interaction between two objects. This is useful in the example of the N particles where only some particles are connected by a spring.

2. Explain how the encoder and the decoder work.

As explained in question 1, the encoder is a GNN that outputs K graphs from the D features of the N objects seen during T time steps.

More formally, the aim of the encoder is to learn the latent space $\mathbf{z}$ from the observation $\mathbf{x}$. If we note ϕ the parameters of the encoder, the aim of the encoder is to output the factorized distribution over egde types $q_\phi(\mathbf{z}|\mathbf{x})$ through *message passing operations*. A message passing operation is an aggregation of information from nodes to edges ($v \to e$) or from edges to nodes ($e \to v$). We have in order :

$$\mathbf{h}_j^1 = f_{emb}(\mathbf{x}_j) \tag{1}$$

f_{emb}, such as f_e^1, f_v^1, f_e^2, etc... are neural networks that map between the different representations.

$$v \to e : \mathbf{h}_{(i,j)}^1 = f_e^1([\mathbf{h}_i^1, \mathbf{h}_j^1]) \tag{2}$$

$$e \to v : \mathbf{h}_j^2 = f_v^1\left(\sum_{i \neq j} \mathbf{h}_{(i,j)}^1\right) \tag{3}$$

$$v \to e : \mathbf{h}_{(i,j)}^2 = f_e^2([\mathbf{h}_i^2, \mathbf{h}_j^2]) \tag{4}$$

Finally, we model the edge type posterior as $q_\phi(\mathbf{z}_{i,j}|\mathbf{x}) = \text{softmax}(\mathbf{h}_{(i,j)}^2)$.

Once we have the latent space composed of K graphs, these K graphs feed K decoders. The aim of the decoders is to predict the future trajectories of the N objects given the latent graphs and the first T time steps of the N objects.

If we note θ_k the parameters of the k-th decoder, the aim of this decoder is to output $p_{\theta_k}(\mathbf{x}^{t+1}|\mathbf{x}^t, ..., \mathbf{x}^1, \mathbf{z})$. When the dynamics are Markovian, i.e. when we can write $p_{\theta_k}(\mathbf{x}^{t+1}|\mathbf{x}^t, ..., \mathbf{x}^1, \mathbf{z}) = p_{\theta_k}(\mathbf{x}^{t+1}|\mathbf{x}^t, \mathbf{z})$, we can in general use any GNN algorithm as our decoder. However, if the dynamics are not Markovian, then we are obliged to use more complex architectures, i.e recurrent decoders using GRU units. We detail the equations for a Markovian dynamic :

$$v \to e : \overset{\sim}{\mathbf{h}}{}^t{}_{(i,j)} = \sum_k z_{ij,k} \tilde{f}{}_e^k([\mathbf{x}_i^t, \mathbf{x}_j^t]) \tag{5}$$

$$e \to v : \mathbf{u}_j^{t+1} = \mathbf{x}_j^t + \tilde{f}{}_e^k\left(\sum_{i \neq j} \overset{\sim}{\mathbf{h}}{}^t{}_{(i,j)}\right) \tag{6}$$

Then, $p_{\theta_k}(\mathbf{x}^{t+1}|\mathbf{x}^t, \mathbf{z}) = \mathcal{N}(\mathbf{u}_j^{t+1}, \sigma^2\mathbf{I})$. Note that $z_{ij,k}$ denotes the k-th element of vector $\mathbf{z}_{i,j}$ and σ^2 a fixed variance. When $\mathbf{z}_{i,j}$ is a one-hot vector, the messages $\overset{\sim}{\mathbf{h}}{}^t{}_{(i,j)}$ are $\tilde{f}{}_e^k([\mathbf{x}_i^t, \mathbf{x}_j^t])$ for the selected edge type k, and for the continuous relaxation we get a

weighted sum.

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

We introduce a baseline to be able to compare the algorithm proposed (here, NRI) with more common architectures. LSTM is a well-known technique, so it can serve as a baseline for our study.

The NRI learns the interactions (encoder) and then computes the future trajectory (decoder). The LSTM computes directly the future trajectory from the previous time steps.

The burn-in phase plays a role similar to the encoder in the NRI model, in the sense that it allows the LSTM to discover which types of objects it is dealing with. During the B time steps of the burn-in phase, the LSTM is trained by receiving as input the true trajectory so that it can compare its prediction with the ground-truth and make adjustement (i.e. optimize the network parameters to have a loss as small as possible).

The burn-in phase allows a *fairer* comparison between the LSTM baseline and the NRI model. The way the LSTM baseline is used for trajectory prediction is summarized in Fig. [1].

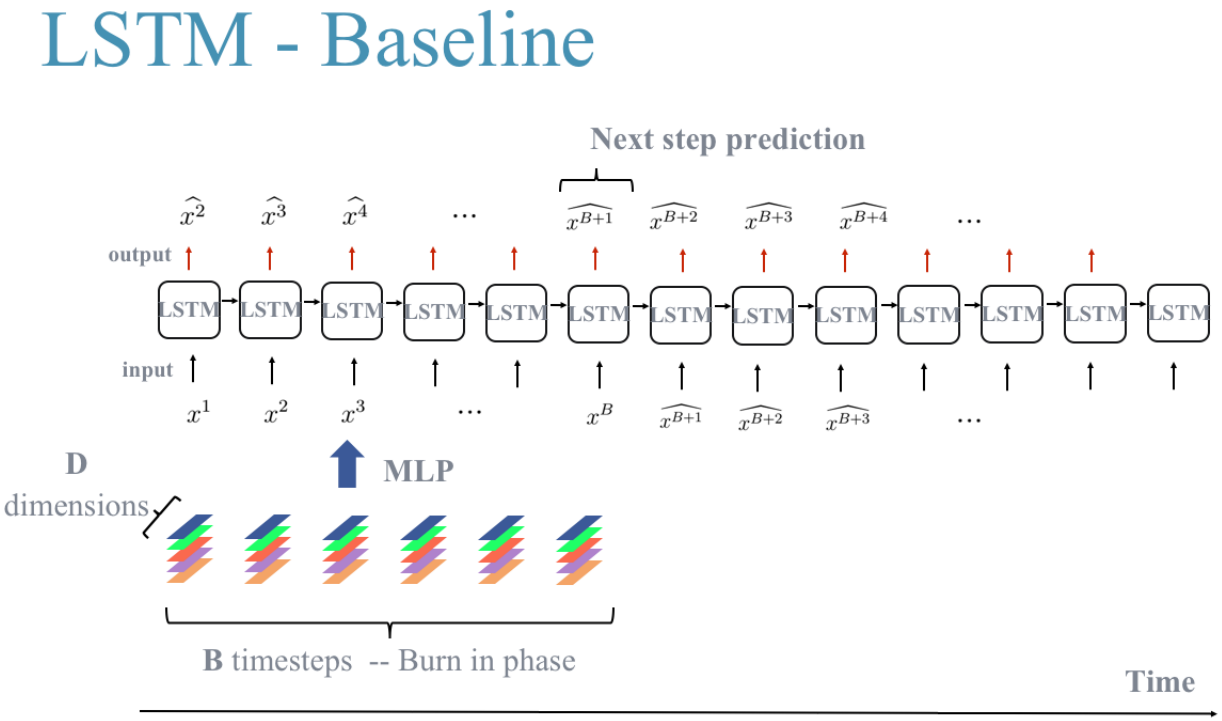

Figure 1: LSTM baseline

4. Consider the training of the LSTM baseline. Show and explain the difference between the NLL before and after the burn-in phase. Is it surprising ?

In the NRI model, the objectiv is to maximize the ELBO (or minimize the negative

ELBO) :

$$\mathcal{L} = -\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] + \mathrm{KL}\left[q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z})\right] \tag{7}$$

In the case of the LSTM baseline, there is no Kullback-Leibler divergence, so the loss is simply the negative log-likelihood : $\mathcal{L} = -\mathbb{E}\left[\log p_\theta(\mathbf{x})\right]$.

In the code provided, we have the lines "for epoch in tnrange(1): nll, pred = train(epoch)". If we print nll and pred, we obtain something like nll = 350 and pred = 300 (for one epoch). nll corresponds to the negative log-likelihood computed for all the time steps involved in the experiment (i.e. 49 time steps), whereas pred is the loss for the time steps after the burn-in phase only (i.e. 10 time steps).

As we see, the negative log-likelihood is about 50 for the burn-in phase and about 300 after the burn-in phase. This may seem surprising because there are 39 time steps in the burn-in phase, and only 10 after it. So, without *a priori*, we could expect a higher loss for the burn-in phase because there are 4 times more time steps. However, we observe that the loss is lower during the burn-in phase than after. This can be explained with the help of Fig. [1]. During the burn-in phase, the LSTM baseline always predicts $\widehat{x^{t+1}}$ given the true trajectory x^t. The LSTM may make mistakes but these mistakes are corrected at each time step because the LSTM receives as input the true trajectory. On the contrary, after the burn-in phase, the LSTM predicts $\widehat{x^{B+t+1}}$ given its prediction $\widehat{x^{B+t}}$. This is a huge source of errors ! Indeed, the mistakes done by the LSTM are not rectified at each time step and they accumulate all along the future predicted trajectory.

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

Let's motivate our answer with a comparison between fully-connected neural networks (MLP) and convolutional neural networks (CNN) on the task of image classification. We know that CNN performs better than MLP on this task. Indeed, images have a convolutional nature. Therefore, we understand the efficiency of CNNs on this task because they are by definition convolutional, so they learn faster and with better accuracy. To reach the same level of performance, MLPs need much more time and much more energy because they have to learn the convolution.

Now, replace the word CNN by NRI, MLP by LSTM, image classification by relational inference and read the previous paragraph again : you have the first part of the answer to the question.

Indeed, in our study case, the goal is to learn interactions (or relations) between objects. One efficient way to do this is to use a graph structure. As a consequence, an architecture that directly deals with graph structures performs better on the studied tasks (connected springs, Kuramoto, charged particles, basketball players (and their body)), because the inherent nature of these tasks are graphical. Imposing a graph structure in the architecture can be seen as an *inductive bias* on our model.

The other advantage of the NRI model with respect to the LSTM baseline is related to the decoder part. The encoder produces K graphs that feed K decoders. Contrarily to other GNNs that could be used to solve the problem of trajectory prediction (such as the interaction network of Battaglia et al. [1]), the NRI is forced to use the K graphs. If the NRI was not forced, it would have the possibility not to use these latent representations, and it would have to learn that it is a good idea to use them. Forcing the NRI to use the K graphs can also be seen as another *inductive bias*, and it allows better results in a

faster way.

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

We call edge accuracy a quantity in $[0; 1]$ proportional to the number of edges correctly classified. Note that, in the case of the TP, we only use two edge types for the network : *on* or *off*, as written in the notebook. During training, the edge accuracy increases and is above 0.95 after two epochs. This seems coherent : during the training, the model learns and becomes more accurate on the training data.

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

The authors of the paper have tested the case where there is no interaction between objects as a variant of the spring simulation experimental setting. To paraphrase the paper, they create a test set of 1000 simulations with 5 non-interacting particles and test an unsupervised NRI model which was trained on the spring simulation dataset with also 5 particles. The NRI model achieves an accuracy of 98.7% in identifying "no interaction" edges.

To say it in a different way, and given the notebook we have, this means that the encoder of the NRI model has identified the presence of only one edge type. Since two edge types were imposed (during the choice of hyperparameters), this means that the encoder attributes (almost) all the edges to a certain edge type. In other words, the encoder produces two graphs : one which is (almost) complete, and one which is (almost) empty.

Bibliography

[1] Battaglia, P. et al. 'Relational inductive biases, deep learning, and graph networks' `https://arxiv.org/pdf/1806.01261.pdf`
