Code documentation

Antoine Girardin

December 6, 2025

In this document, I will detail the features contained in the code and how to use them. The code contains three main parts:

1. a method to compute the probability distributions given the sources and measurements of each party.
2. the code to find a local model with a neural network that minimizes a loss function to a target distribution.
3. the code to generate exact strategies with finite cardinality with another neural network based method.

1 Documentation to compute probability distributions

The distribution is computed by the function called "quantum_network_fct" that takes three inputs.

1. source_dims_list : A list that contains all the sources in the network. Each source is given as a Tuple with an array containing the density matrix and a list of the dimensions of the systems. For example, a two-qubit state will have dimensions [2,2]. A three-partite state with systems of dimension 3, 2, 2 will have the list [3,2,2].
2. povm_matrices : The measurements are a list of tuples. Each tuple contains the name of the party as a string and a list containing the POVMs for each input given to the party. If the party has no input, it is then a list containing a single list of POVM.
3. permutation : A list containing the two lists of permutation for the measurements and the states. For the triangle, if the parties perform measurements taking the source on their left as the first input and the source on their right as the second one, we label each subsystem as in Fig. 1. We choose to not permute the systems for the measurements, the permutation list is then $[0, 1, 2, 3, 4, 5]$. The permutation for the sources is given by the label of the subsystems of the sources as we gave them in the second inputs. If the sources are given in the order α, β, γ, then since α is connected to the subsystems 3 and 4, β to the subsystems 5 and 0, and γ to the subsystems 1 and 2, the permutation list will be $[3, 4, 5, 0, 1, 2]$.

This function gives us the probability distribution $p_{abc...}$ obtained as a flat list in numerical order $p_{0...00}, p_{0...01}, ..., p_{0...0i}, p_{0...10}...$ for $[0, 1, ...i]$ the possible outcomes of the parties $a, b, c, ...$.

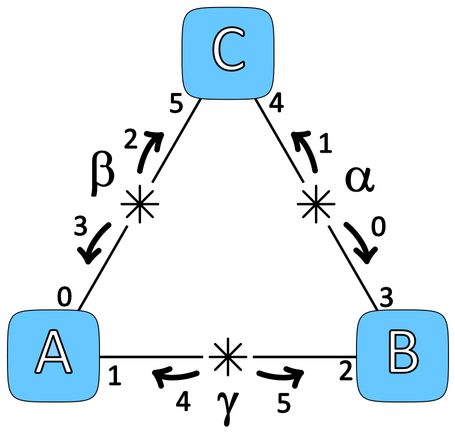

Figure 1: Scheme of the triangle network with the according labelling of the systems

1.1 Permutation of systems for quantum networks

To compute probability distributions in networks, we can still use the Born rule. However, the order of the systems must be chosen carefully. In a network of n parties with m sources, the total measurement is given by $\bigotimes_{i=1}^{n} M_{A_i}$. The sources can be written in a single density matrix as $\bigotimes_{j=1}^{m} \rho_j$, but the basis in which the measurement and the state are written are not the same. One needs to use a permutation matrix P to make the basis coincide. The probability of a given total outcome will finally be given by

$$p(a_1...a_n) = tr\left(\bigotimes_{i=1}^{n} M_{A_i} P \bigotimes_{j=1}^{m} \rho_j P^T\right) \tag{1}$$

Most programming languages have a *permute* function to perform the permutation. Matlab comes with the function by default. In Python, the QuTiP package provides it. With these functions, one needs to give a vector with the requested order. To find this vector, one can write the order of the system in the measurement basis as in Fig. 1, and list the number of the sources in the order of the systems in the measurement basis. For the triangle defined in this figure, the permutation needed to bring the sources to the same basis as the measurement is $[3, 4, 5, 0, 1, 2]$.

2 Documentation of the neural network

To build the neural network that fit the network to study, we create an instance of the class NeuralNetwork. This class can be initialized with the following parameters (all inputs are optional, we write the default values next to the name of the variables. The first two inputs are mandatory for the model to be trainable):

1. config={} : the configuration of the network is given as a dictionary containing the name of the parties with a tuple containing the sources they are connected to and the number of outcomes of each party. For the triangle with four outcomes, the configuration of the network would be given by

   ```
   config = {
           'a': (['beta', 'gamma'], 4),
           'b': (['alpha', 'gamma'], 4),
           'c': (['alpha', 'beta'], 4)
   }
   ```

2. target_distribution=[] : the distribution we want to approach. The order is the same as the probability distribution generated in the section 1
3. width=60 : the width of the neural network for each party
4. depth=4 : the depth of the neural network
5. lr=0.001 : the learning rate
6. dist='kl' : the metric used as loss. 'kl' for the Kullback–Leibler divergence, 'eucl' for the Euclidean distance
7. n_samples=5000 : a default number of samples that will be overwritten when training
8. opt=1 : an option for sampling the local variable (see the function generate_data for the available distributions)
9. scale_blocks_with_n_sources=False : an option to have the width of the neural network of each party to be multiplied with the number of source received

The main function to use the method is called "train_model_and_save". Here are all the inputs the function can take (when the input is optional, we write its default value next to the name of the variable):

1. model : an instance of NeuralNetwork
2. n_samples_in=None : the number of samples for training. If None, the number of samples will adapt as a function of the loss
3. n_training_steps=10000 : the maximal number of training steps
4. model_path='model/model.pth' : the path to save the model
5. threshold=0 : distance when stopping the training
6. bias=4 : if n_samples_in=None, control the number of samples given the loss
7. n_steps_reevaluate=10000 : reevaluate the model if no improvement is found after a number of training steps, increase the bias when reevaluating

8. bias_max=10 : fix the maximal bias increased when reevaluating when n_steps_reevaluate passed without update

9. min_sampling=1000 : if n_samples_in=None, bound the minimum number of samples

10. max_sampling=float('inf') : if n_samples_in=None, fix the maximum number of samples. Useful to avoid running out of memory. Depending on the size of the neural network, 24 GB can get a maximum between 500'000 and 2'000'000 samples.

The use of the other method to build exact strategies with finite cardinalities is similar. All inputs concerning the number of samples and the generation of the local variable are removed. An additional argument can be given when creating the NeuralNetwork instance to fix the cardinality.

To read the results found by the neural network, the file main_read_results.py gives an overview of the different methods to recover the results from the saved models.