Coursework 2 - Modern Computational Techniques

Description Q1

Neural Network Structure

For the memorisation of the 3-input XOR gate, a perceptron with a single hidden layer is trained in `NN_metropolis.py`. Since no generalisation of the training data is intended, the hidden layer could have as many nodes as computationally sensible. As overfitting is favourable in our case, the hidden layer of the used neural network (NN) has 5 nodes, set by the variable `nhidden`. According to the given data, NN has 3 input nodes and one node in the output layer. A bias node that has always value 1 is attached to input and hidden layer, to improve generalisation performance. Therefore the bias node is not connected to the input layer. The resulting layout can is shown in figure 1.

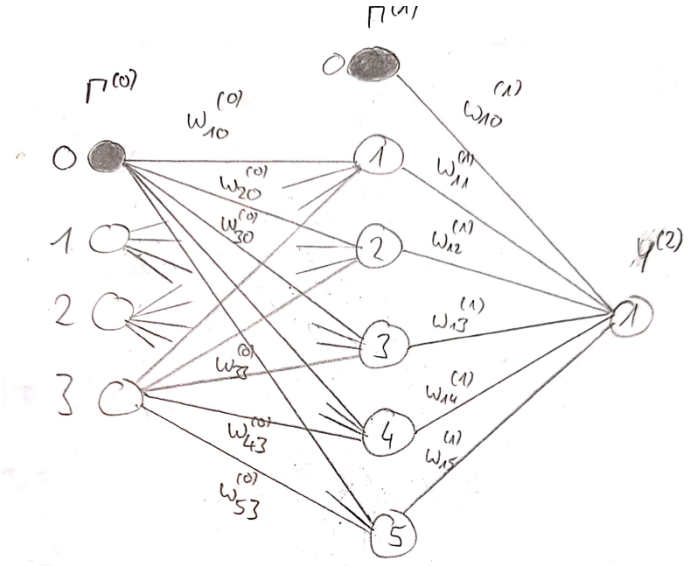

Figure 1: Chosen Neural Network Structure

In the submitted program the shape of the NN is determined by the shape of the provided weight arrays. Each weight $w_{ij}^{(n)}$ can be described by the three indices

$$\begin{aligned} n \in \{0,1\} &: \quad \text{layer index} \\ i &: \quad \text{output node index} \\ j &: \quad \text{input node index} \end{aligned}$$

As a result the corresponding weight matrices for each layer $(\mathbf{W}^{(n)})_{ij}$ are of the vector spaces

$$\mathbf{W}^{(0)} \in \mathbb{R}^{5\times 4} \quad \text{and} \quad \mathbf{W}^{(1)} \in \mathbb{R}^{1\times 6} \tag{1}$$

which also includes the bias weight into the matrix. This definition requires that the 3 dimensional input vector $Y^{(0)}$ from the truth table is extended by the bias to

$$\Gamma^0 = \left(1, Y^{(0)T}\right)^T$$

so that the output of the first layer can be written as

$$Y^{(1)} = \sigma(\mathbf{W^{(0)}}\Gamma^{(0)}). \tag{2}$$

The activation function $\sigma(x) = \frac{1}{1+\exp(-x)}$ is defined element wise on vectors. Generalising this gives

$$Y^{(n)} = \sigma(\mathbf{W^{(n)}}\Gamma^{(n)}). \tag{3}$$

In the code, the training samples are all contained in a single matrix with each row corresponding to the inputs of the truth table. That means that each input vector Y_k of row k is implemented as row of the new matrix. With numpy this makes the code more efficient.

Metropolis Algorithm Implementation

The implemented Metropolis Algorithm minimises the sum of squared error of all inputs. The inverse temperature β is set to an initial value `beta` and is increased linearly in steps of `betastep` after `eqiter` iterations. After `maxiter` updates of β, the algorithm is stopped without result.

The algorithm starts with random weights in the range $[-1, 1]$. In each iteration one weight is selected randomly and changed by a random amount $\Delta w \in [-\alpha, \alpha]$. Each random choice is drawn from a uniform distribution. The metropolis algorithm decides then in the given randomised form if the changed weight is accepted or not.

The used values for the parameters in the code proved most effective in obtaining a trained NN as fast and reliable as possible. However, there are cases in which the algorithm does not find a minimum that corresponds to a trained NN.