# Neural network based order parameters for classification of binary hard-sphere crystal structures: Supplemental Information

Emanuele Boattini[1], Michel Ram[1], Frank Smallenburg[2], and Laura Filion[1]

[1]Soft Condensed Matter, Debye Institute of Nanomaterials Science, Utrecht University, Princetonplein 5, 3584 CC Utrecht, The Netherlands [2]Laboratoire de Physique des Solides, CNRS, Univ. Paris-Sud, Univ. Paris-Saclay, 91405 Orsay, France

## 1. Python Code and Neural Network Parameters

To ensure that the neural networks we developed here are accessible to a wide range of researchers, we have also provided python code for the SANN trained networks. The code is available in the SI, and contains the following:

- a file "README.txt" which explains how to use the code
- a sample configuration file "conf045.xyz" for testing the code
- the main code consist of "SANN_NN.py" and "particles.py"

Additionally, for those who might want to code their own neural network, we have included a text file which lists the parameters (weights and biases) associated with the SANN networks in the file "parameters.txt".

## 2. Single-layer Neural Networks

In the following, we describe in more detail the implementation of the single-layer neural networks used in this work.

In the most general terms, an artificial neural network can be described as an algorithm that takes an input vector of data and generates a corresponding output. For the networks used here, the input is a vector of averaged bond order parameters associated with a specific particle, and the output tells us in which crystalline (or liquid) environment the particle is most likely to be.

Let us indicate the input vector of particle $n$ as $\mathbf{x^n} = (x_1^n, \dots, x_d^n)$, where $x_i^n$ are $d$ bond order parameters, and suppose we want the network to be able to distinguish between $c$ different local environments. Such a network will produce $c$ outputs (as many as the classes we wish to distinguish) by doing some mathematical operations on the input. The procedure to go from the input to the output is explained in the following.

### *2.1. Output of a single layer neural network*

The first step is to take $c$ different linear combinations of the input

$$a_k^n = \sum_{i=1}^{N} w_{ki} x_i^n + b_k, \qquad \text{with } k = 1, \ldots, c, \tag{1}$$

where the coefficients $w_{ki}$ and $b_k$ are usually called weights and biases, respectively. Then, the final outputs are obtained by applying a so-called activation function to the linear combinations $a_k^n$

$$y_k^n = \phi\left(\sum_{i=1}^{N} w_{ki} x_i^n + b_k\right) = \phi\left(a_k^n\right), \tag{2}$$

where $\phi$ is the activation function and the $y_k^n$ are the outputs.

Many choices for the activation function are available. In order to interpret the final outputs $y_k^n$ as the probabilities that the input vector $\mathbf{x^n}$ belongs to the specific class $k$, we use the Softmax activation function [1]

$$y_k^n = \frac{e^{a_k^n}}{\sum_{k'} e^{a_{k'}^n}}. \tag{3}$$

The action of this function is to map the outputs $y_k^n$ onto numbers between 0 and 1 and guarantees that they sum up to 1 ($\sum_k y_k^n = 1$), and hence they can be interpreted as probabilities. Every input vector $\mathbf{x^n}$ is eventually assigned to the class $k$ with the highest probability $y_k^n$.

### *2.2. Training the network*

To train the network, one typically starts with an initial guess for weights and biases, which are then optimized iteratively during the training process. In this study, we use the normalized initialization proposed by Xavier in Ref. [2], which consists in setting the initial values of the biases to zero, while the weights are initialized randomly from a normal distribution with mean zero and variance $Var[w] = 2/(d+c)$, where $d$ is the dimension of the input vector and $c$ is the number of outputs.

The network is trained using a training data set, which is a set of input vectors (particle environments) corresponding to a known output (crystal structure). During the training, the network is fed these input vectors while weights and biases are adjusted iteratively in order to minimize the difference between the desired output and the actual output that the algorithm produces. To this end, we need to define an error function (or cost function), which quantifies the difference between the desired output and the output produced by the network, and which is minimized when these outputs are equal.

In our case, we use the cross-entropy error function [1], which is defined as follows

$$E(\{t_k^n\}, \{y_k^n\}) = -\sum_{k=1}^{c} t_k^n \ \ln y_k^n, \tag{4}$$

where $\{t_k^n\}$ and $\{y_k^n\}$ are the desired outputs and the outputs produced by the network, respectively, for the $n$th input vector. This function is minimized iteratively during the training process with a gradient descent algorithm [1]. With this algorithm, weights and biases at every iteration are adjusted by moving a small distance in the direction in which $E$ decreases most rapidly, i.e. in the direction of $-\nabla_{\mathbf{w}} E$. By iterating this process, we generate a sequence of weights, which at iteration $t+1$ are calculated as follows

$$w_{ki}^{t+1} = w_{ki}^{t} - \epsilon \left. \frac{\partial E}{\partial w_{ki}} \right|_t , \tag{5}$$

where $\epsilon$ is a positive number called learning rate, which controls the step size at which the algorithm moves towards the minimum of the error function. Under suitable conditions, the sequence of weights generated with Eq. 5 will eventually converge to a point at which $E$ is minimized. The choice of the value of $\epsilon$ is critical, since if it is too small the reduction in error will be very slow, while, if it is too large, divergent oscillations can result. Moreover, since the learning rate is constant, when the gradients get smaller, the weight updates will also be smaller. This may cause the algorithm to become trapped in local minima of the error function. In order to speed up the minimization process and avoid being trapped in these local minima, a momentum term can be added to the gradient descent formula [1]:

$$w_{ki}^{t+1} = w_{ki}^{t} - \epsilon \left. \frac{\partial E}{\partial w_{ki}} \right|_t + \underbrace{\mu \Delta w_{ki}^{t}}_{\text{momentum term}} , \tag{6}$$

where $\mu$ ($\in [0,1]$) is called the momentum parameter and $\Delta w_{ki}^t$ is the variation of the weight at iteration $t$. Inclusion of such a term effectively adds inertia to the motion through weight space and smooths out eventual divergent oscillations, typically leading to a significant improvement in the performance of gradient descent. The optimal choice of the learning rate, $\epsilon$, and the momentum parameter, $\mu$, strongly depends on the specific problem and is usually found by trial and error. In this work, we used $\mu = 0.9$ and $\epsilon$ in the rage $[0.05, 0.5]$.

Another way to assist the network in learning is by doing weight updates with multiple examples at the same time. This method is called batch-gradient descent and is the one we use for minimizing the error function.

## References


[1] C. M. Bishop, *Neural networks for pattern recognition* (Oxford University Press, Oxford, UK, 1995).

[2] X. Glorot and Y. Bengio, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (2010), pp. 249–256.