# Neural Network Potentials

Mark DelloStritto

## 1 Introduction

Neural networks are a powerful computational tool which have gained widespread interest in the last few years in a wide range of disciplines. While often used for categorization or pattern recognition, it should be recognized that neural networks are fundamentally tools designed for fitting arbitrary functions, and therefore have a much wider range of applicability. Recently, it has been recognized that neural networks can be used to fit the energy of atomic systems as a function of the positions. Since the energies depend on the atomic positions, it is possible to take derivatives of the energy and model atomic forces as well. This approach has lead to a growing new field in atomic simulation where neural networks are used to generate highly accurate atomic potentials from ab-initio reference data. In this review we will discuss the properties of neural networks, how they can be applied to simulate atomic systems, and the difficulties and challenges associated with neural network potentials (NNPs).

## 2 Neural Networks

Although neural networks can exhibit complex behavior, they are relatively simple mathematical constructs. A graphical representation of a neural network is shown in Figure 1, where we can see that the network consists of a series of layers of nodes, each of which are fully connected to each adjacent layer through a series of edges. Technically, figure 1 shows a "feed-forward" network, where the inputs to each layer come only from the layer immediately adjacent to it. While this is only one possible construction of a neural network, it is by far the dominant construction in nearly all applications of neural networks and therefore is the only type of network we will consider.

The first layer of the network is known as the "input" layer, the final layer is known as the "output" layer, and all layers in-between are known as "hidden layers." However, it is mathematically simpler to consider the input layer separately from all other layers in the networks, and to consider the output layer together with the hidden layers. That way, we can say that the network posses $N$ sets of edges and $N$ sets of layers, and that the $(l)^{th}$ set of edges connects the $(l-1)$ node layer to the $(l)$ node layer. The $0^{th}$ set of edges then connect the input nodes to the $0^{th}$ set of nodes. Not only does this notation simplify the mathematics of the network, but it is consistent with a network with no hidden layers, i.e. only an input and an output layer.

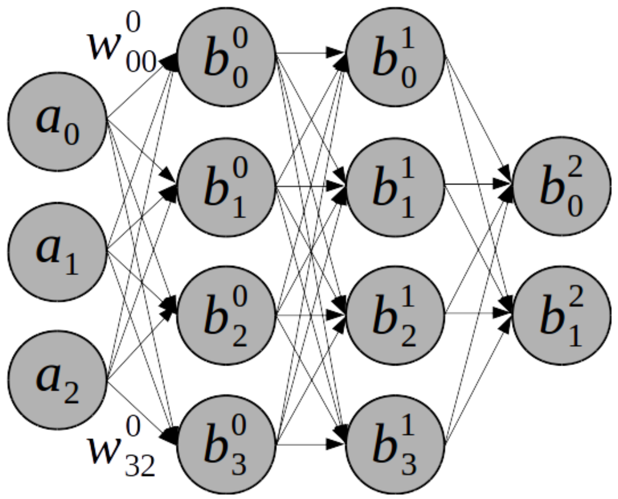

Figure 1: Graphical representation of a feed-forward neural network. The input nodes are labeled by their initial values $a_i$ while all other nodes are labeled by the value of their respective bias $b^l_i$.

We label the value of the $i^{th}$ node in the $l^{th}$ layer as $a^l_i$, while the initial value of the $i^{th}$ input node is labeled simply as $a_i$. Associated with each node is a bias $b^l_i$ and a transfer function $f^l_i(z)$. Each edge is labeled by the layer and the pair of nodes it connects, such that the edge $w^l_{ij}$ points from the $j^{th}$ node in the $(l-1)$ layer to the $i^{th}$ node in the $(l)$ layer. The input to the transfer function we label as $z^l_i$, which is by definition a sum of the bias $b^l_i$ and a sum of the value of all nodes in the previous layer multiplied by the respective edges. We can therefore write the node input and value as:

$$\begin{aligned} z^l_i &= w^l_{ij} a^l_j + b^l_i \\ a^l_i &= f^l_i(z^l_i) \end{aligned} \tag{1}$$

where we use Einstein summation notation. It should now be clear that we can write equation 1 in terms of matrix operations, greatly simplifying their functional form and creating opportunities to use well studied and highly optimized matrix-vector calculations:

$$\begin{aligned} z^l &= w^l a^l + b^l \\ a^l &= f^l(z^l) \end{aligned} \tag{2}$$

where $w^l$ is a matrix, $a^l$ and $b^l$ are vectors, and the function $f^l(z)$ now operates on each element of a vector.

From equation 2 we can see that the output of a neural network is an arbitrary combination of compositions of the transfer function. Thus, the behavior of a neural network is almost entirely defined by the properties of the transfer function. While a wide range of of functional forms are possible, the transfer function is typically either linear, a hyperbolic tangent function ($\tanh(x)$), or a sigmoid function ($\text{sig}(x)$). The use of a sigmoid function is inspired from cognitive science, where biological neurons have both an activation potential and a saturation point which are well modeled by sigmoid function. The $\tanh(x)$ function is designed to emulate the behavior of a sigmoid function while also allowing for negative values. While uncommon in the interior of a network, a linear transfer function is often used for the final edge layer of a network, as it allows the output to be scaled and shifted by an arbitrary amount. In addition, in almost all applications all nodes in a given layer have the same transfer function.

We now make the extraordinary and somewhat ambiguous statement that a suitably complex neural network can fit any well-behaved function. There are obviously limitations to the statement, for instance a suitably complex network usually must have at least one hidden layer with at least set of edges being non-linear. In addition, the "well-behaved" function we are fitting should not have any divergences or severe discontinuities in the derivative. Even with these restrictions, this implies that neural networks can fit a very wide range of functions, including the functional forms of nearly all physically motivated potential energy surfaces. We do not provide a rigorous proof of the universal fitting behavior of neural networks, as it is beyond the scope of this review, but we do provide a sketch of an argument to illustrate the potential of neural networks.

We begin with the simplest possible neural network, one with a single input node and a single output node with a linear transfer function. It should be obvious that this neural network can fit any linear function. Adding more layers will allow us to fit polynomial functions, with each added layer in increasing the degree of the polynomial which can be fit by the network. Using polynomials to fit functions is generally not recommended though, as doing so often results in erroneous oscillations associated with polynomials of a large degree.

To fit more complex functions, we consider a neural network with at least one hidden layer with a $\tanh(x)$ activation function for the hidden layer and a linear transfer function for the output layer. We model a one-dimensional function by restricting the input and output layers to one node, such that the neural network appears as that in figure 2. We fit a simple Gaussian function by first setting all edges in the first layer to a large value (e.g. 1000), such that the $\tanh(x)$ functions all resemble step functions, and adjust the biases in the first and second layer such that the step functions add up to a crude approximation of the Gaussian. We then relax all weights to a smaller value (e.g. 1-10), such that all $\tanh(x)$ functions become quasi-linear. Despite the simple procedure we have followed, the resulting approximation shown in figure 2 is quite reasonable, and it is should be clear that a more rigorous optimization of the parameters will result in a very close optimization of the function. The power of neural networks with $\tanh(x)$ or $\text{sig}(x)$ activation functions therefore arises from the fact that we are essentially adding up compositions of step functions to fit the objective function. Just as linear approximations to the derivative become exact with vanishing step size, one can think of a sum of step functions becoming an exact approximation when summing over an infinite series of step functions. By allowing the step functions to vary between quasi-linear and a perfect step, we reduce the number of functions required, allowing highly accurate function fitting with a finite number of nodes.

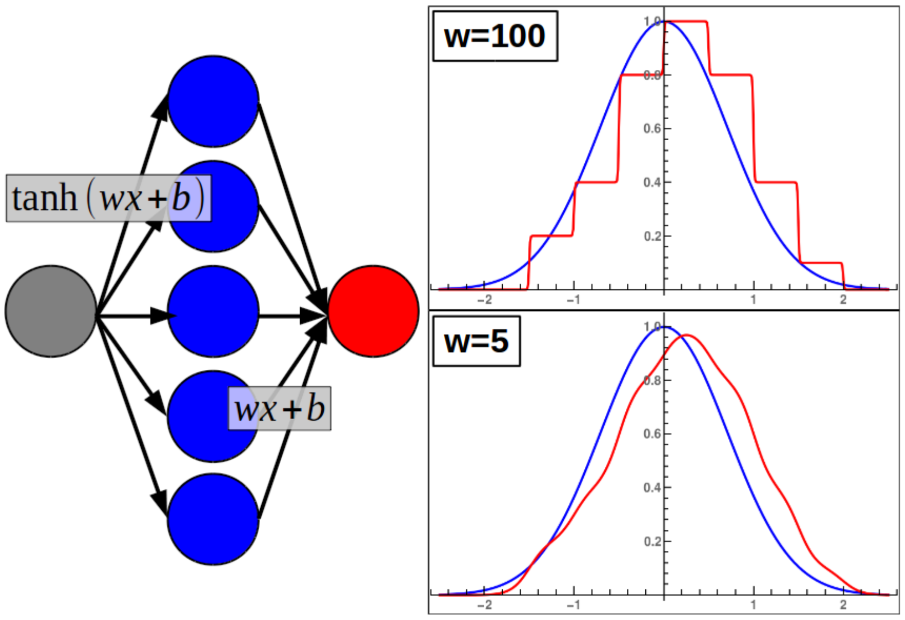

Figure 2: Neural network designed to fit a one-dimensional function, featuring one input node, one output node, a $\tanh(x)$ hidden layer and a linear output layer. The figure on the right shows the approximation of a Gaussian function using tanh transfer functions first with weight $w = 100$ (top) and then with weight $w = 5$ (bottom).

## 3 Symmetry Functions

Although neural networks are powerful tools for function fitting, it is not obvious how they might applied for atomic simulations. Naively, one might assign each input of the neural network to each atomic coordinate. However, such a strategy would result in neural networks with a very large number of inputs, and the network could be applied only to simulations with the exact same number of atoms. Instead, the best approach involves assigning to each node a measure of the local symmetries of the atoms of the simulation. The simulation procedure is to assign a neural network to each atom, assign measures of the symmetry function to each of the inputs, and finally executing the network. The output of the network is interpreted as the energy of the atom, and the total energy of the system is defined as the sum over all atomic energies.

One of the most popular set of symmetry functions are those first introduced by Behler and Parinello, with two different categories of radial and angular functions. Both sets of symmetry functions are defined as sums over Gaussian functions multiplied by a cutoff function that enforces a smooth decrease to zero at a given cutoff radius.

The cutoff functions are defined as:

$$\begin{aligned} f_{cos}(r, r_c) &= 0.5(\cos(\pi r/r_c) + 1)\theta(r - r_c) \\ f_{tanh}(r, r_c) &= \tanh^3(1 - r/r_c)\theta(r - r_c) \end{aligned} \tag{3}$$

where $r_c$ is the cutoff radius and $\theta(r)$ is the Heaviside step function.

Two different types of radial functions are introduced as follows:

$$\begin{aligned} G_{1i} &= \sum_{j\neq i} f_c(r_{ij}) \\ G_{2i} &= \sum_{j\neq i} e^{-\eta(r_{ij}-r_s)^2} f_c(r_{ij}) \end{aligned} \tag{4}$$

Two different types of angular functions are introduced as follows:

$$\begin{aligned} G_{3i} &= 2\sum_{j\neq i}\sum_{k\neq i,k>j}(0.5(1+\lambda\cos(\theta_{ijk})))^{\zeta} e^{-\eta(r_{ij}^2+r_{ik}^2+r_{jk}^2)} f_c(r_{ij})f_c(r_{ik})f_c(r_{jk}) \\ G_{4i} &= 2\sum_{j\neq i}\sum_{k\neq i,k>j}(0.5(1+\lambda\cos(\theta_{ijk})))^{\zeta} e^{-\eta(r_{ij}^2+r_{ik}^2)} f_c(r_{ij})f_c(r_{ik}) \end{aligned} \tag{5}$$

where the sum is over all non-unique pairs of nearest-neighbors. The sum is over all non-unique pairs of nearest-neighbors as the symmetry functions are symmetric with respect to the exchange of the positions of the nearest-neighbor pair, and so counting all pairs would double count the symmetry functions.

An example of a set of radial functions is shown in Figure 3, which consists of a sum of shifted Gaussians in the radial direction. While not necessarily optimal, these set of functions will results in different inputs for all interatomic distances. An example of angular symmetry functions is shown inf Figure 4, which consists of a sum of different powers of the cosine functions in $G_3$ and $G_4$. Note that, while using cosines with shifted centers might result in a better fit, such functions will significantly complicate force calculations, reducing the applicability of NNPs to simulation.

Finally, note that we must include a different radial symmetry function for each species in the simulation, and we must include different angular symmetry functions for each pair of species in the simulation. The reason for this is that each radial angular symmetry function must account for the interactions between all possible pairs and triples in the simulation, respectively. Since the atomic neural networks are separate and independent, the different types of possible central atom are not relevant. Thus, for pair interactions, we require a copy of each symmetry function for each species, stored as separate but equivalent symmetry functions for each species. In a simulation of water for instance, we have a radial symmetry function associated with (X-O) interactions and one associated with (X-H) interactions, where X can either be O or H, thereby covering (O-O), (O-H), (H-O), and (H-H) interactions. Similarly, for triple interactions, we need a copy of each symmetry function for each combination of pairs of species. Once again, in a water simulation we need angular symmetry functions for (X-O-O), (X-O-H), (X-H-O), and (X-H-H) interactions, which covers all triple interactions. Thus, the required number of symmetry functions can increase significantly with an increasing number of species, and so the creation of an appropriate set of symmetry functions is an important and difficult task in the process of creating NNPs.

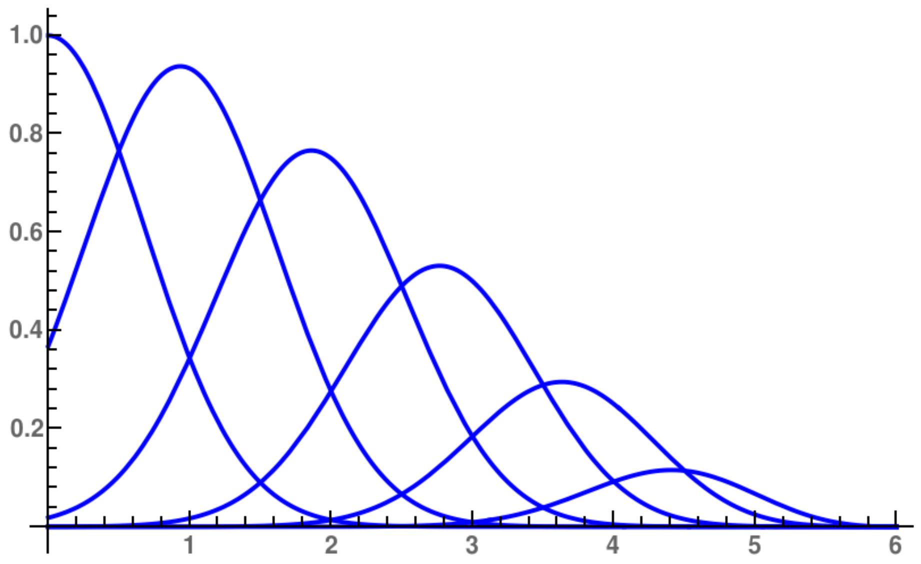

Figure 3: Example of a set of radial symmetry functions, in this case a set of shifted Gaussian functions multiplied by a cutoff function.

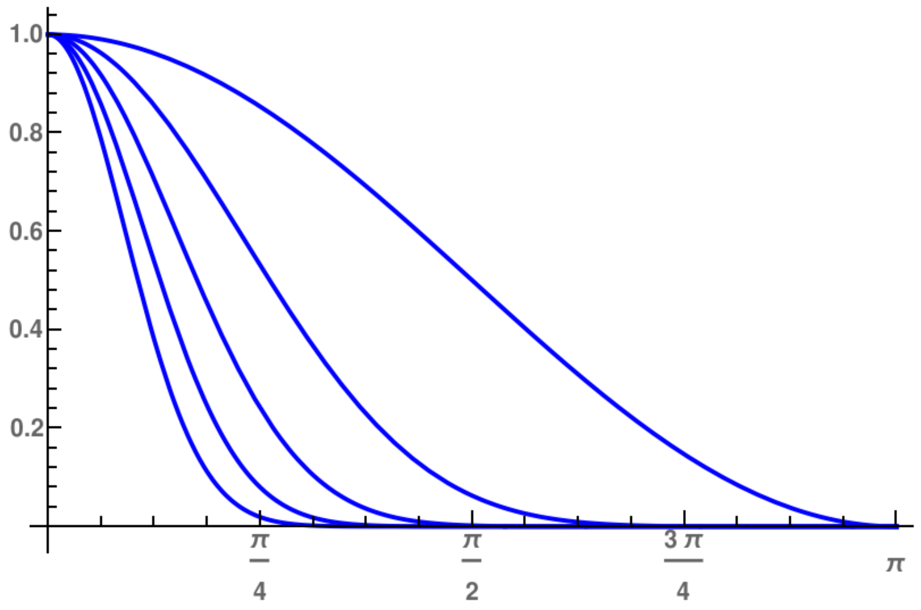

Figure 4: Example of a set of angular symmetry functions, in this case a set of powers of cosine functions.

## 4 Energy and Forces

As noted above, we write the total energy of the system as a sum over each atomic energy:

$$E = \sum_i E_i \tag{6}$$

where the atomic energy $E_i$ is the output of network associated with the $i^{th}$ atom. Since we are interpreting the energy of the total system as a sum over atomic energies, it is useful to consider the energy of an isolated atom. Strictly using a NNP, the energy of an isolated atom would be zero. This is neither a useful nor a physically reasonable result. Note that, since we fit NNPs to ab-initio electronic structure data, the total energy obtained from a NNP can best be thought of as the total *electronic* energy of the system, as the kinetic energy of the atoms is not included in the fitting procedure. Thus, we can instead write the total energy of the system as:

$$E = \sum_i (E_i + E_i^{(0)}) \tag{7}$$

where $E_i^{(0)}$ is the energy of an isolated atom of the same species as atom $(i)$. The exact value of $E_i^{(0)}$ is not very important, as the $E_i^{(0)}$ simply sets the ground state energy for the system, though it is imperative that all $E_i^{(0)}$ are calculated using the same method. Rather, one should simply take care that the cohesive energy of any given system calculated using NNPs, defined as the total energy minus the sum over all $E_i^{(0)}$, is a reasonable, positive number. The addition of $E_i^{(0)}$ to the atomic energy can also significantly help the training procedure by reducing the values of the weights and biases of the network, since the sum over $E_i^{(0)}$ should be close to the total energy of any given system.

Finally, note that the method used to compute $E_i^{(0)}$ need not be the same as the method used to compute the energies of the training data. Again, as the $E_i^{(0)}$ set the ground state energy, they need to be consistent but their exact values will not change the relative energies or forces. Indeed, it is actually recommended that one uses a different method to compute $E_i^{(0)}$ than that used to obtain the training data. The reason being that it is common for methods and codes developed for periodic systems or large supercells to perform poorly for an isolated atom, which can involve complex spin states, require large basis sets, or require relativistic corrections.

In most applications, one desires not only the energy of a configuration, but also the forces on the atoms. Fortunately, one can easily compute forces by taking the gradient of the energy with respect to atomic positions. Using the chain rule, one can write the gradient of the energy as a product of the gradient of the network (gradient of the output with respect to each input) and the gradient of the symmetry function associated with the input. The force on the $i^{th}$ atom can thus be written as:

$$
\begin{aligned}
F_n &= -\frac{\partial}{\partial r_n} E \\
F_n &= -\sum_i \frac{\partial E_i}{\partial r_n} \\
F_n &= -\sum_{i,\alpha} \frac{\partial E_i}{\partial G_{i\alpha}} \frac{\partial G_{i\alpha}}{\partial r_n} \\
F_n &= -\sum_{i,\alpha} \frac{\partial E_i}{\partial G^R_{i\alpha}} \frac{\partial G^R_{i\alpha}}{\partial r_n} - \sum_{i,\alpha} \frac{\partial E_i}{\partial G^A_{i\alpha}} \frac{\partial G^A_{i\alpha}}{\partial r_n}
\end{aligned}
\tag{8}
$$

To make the above derivatives more explicit without losing generality, we write the coordinate dependence of the symmetry functions explicitly. That is: $G^R_{i\alpha}$ we write as $\sum_{j\neq i} \rho^{(\alpha)}_{ij}(r_{ij})$ and $G^A_{i\alpha}$ we write as $\sum_{j\neq i}\sum_{k\neq j,i} \gamma^{(\alpha)}_{ijk}(r_{ij}, r_{ik}, r_{jk})$. Note that each "component" of the symmetry functions $\rho^{(\alpha)}_{ij}$ and $\gamma^{(\alpha)}_{ijk}$ are labeled by the symmetry function index $\alpha$ and the indices $(i, j, k)$, which label each atom in the simulation. This is necessary because the specific form and number of symmetry functions can change depending on the atomic species of both the center and neighbor atoms. Typically, the symmetry functions will not change with the species of the central atom, but we emphasize that identity of the central and neighbor atoms need to be tracked when computing the forces from the symmetry functions

We can write the gradient of the radial component of the energy as:

$$
F^R_n = -\sum_{i,j\neq i}\sum_\alpha \dot{E}^R_{i\alpha} \partial_{r_n} \rho^{(\alpha)}_{ij}(r_{ij}) \tag{9}
$$

where $\dot{E}^R_{i\alpha} \equiv \partial E_i / \partial G^R_{i\alpha}$ and $\partial_{r_n}$ is the gradient with respect to $r_n$. Equation 9 then becomes:

$$
\begin{aligned}
F^R_n &= -\sum_{i,j\neq i}\sum_\alpha [\dot{E}^R_{i\alpha}\dot{\rho}^{(\alpha)}_{ij}(r_{ij})\hat{r}_i\delta_{ni} - \dot{E}^R_{i\alpha}\dot{\rho}^{(\alpha)}_{ij}(r_{ij})\hat{r}_j\delta_{nj}] \\
F^R_n &= -\sum_{i,j\neq i}\sum_\alpha \dot{E}^R_{i\alpha}\dot{\rho}^{(\alpha)}_{ij}(r_{ij})\hat{r}_{ij}\delta_{ni} - \sum_{i,j\neq i}\sum_\alpha \dot{E}^R_{i\alpha}\dot{\rho}^{(\alpha)}_{ij}(r_{ij})\hat{r}_{ij}\delta_{nj} \\
F^R_n &= -\sum_{j\neq n}\sum_\alpha \dot{E}^R_{n\alpha}\dot{\rho}^{(\alpha)}_{nj}(r_{nj})\hat{r}_{nj} - \sum_{i\neq n}\sum_\alpha \dot{E}^R_{i\alpha}\dot{\rho}^{(\alpha)}_{in}(r_{in})\hat{r}_{in} \\
F^R_n &= -\sum_{i\neq n}\sum_\alpha \left[\dot{E}^R_{n\alpha}\dot{\rho}^{(\alpha)}_{ni}(r_{ni}) + \dot{E}^R_{i\alpha}\dot{\rho}^{(\alpha)}_{in}(r_{in})\right]\hat{r}_{ni}
\end{aligned}
\tag{10}
$$

Thus, the final result for the radial component of the force is:

$$
F^R_i = -\sum_{j\neq i}\sum_\alpha \left[\dot{E}^R_{i\alpha}\dot{\rho}^{(\alpha)}_{ij}(r_{ij}) + \dot{E}^R_{j\alpha}\dot{\rho}^{(\alpha)}_{ji}(r_{ji})\right]\hat{r}_{ij} \tag{11}
$$

We can simplify equation 11 even further by recognizing that the expression within the brackets obeys Newtons' third law. That, is the term within the square brackets is identical with respect to permuting the indices $(i, j)$. Thus, when computing the force on atom $(i)$, the second term does not need to be computed, as long as one remembers to add the negative of the first term to the force on atom $(j)$. For a given symmetry function $(\alpha)$ and pair of atoms $(i, j)$, the contribution to the forces can be written as:

$$\begin{aligned} f_{ij}^{R\alpha} &\equiv \dot{E}_{i\alpha}^{R} \dot{\rho}_{ij}^{(\alpha)}(r_{ij}) \\ F_i^R &\rightarrow F_i^R + f_{ij}^{R\alpha} \\ F_j^R &\rightarrow F_j^R - f_{ij}^{R\alpha} \end{aligned} \tag{12}$$

Thus, one can use equation 12 to compute the radial forces on all atoms by computing the network gradients and symmetry function gradients for each atom in a linear sequence.

We can write the gradient of the angular component of the energy as:

$$F_n^A = -\frac{1}{2} \sum_{i,j \neq i} \sum_{k \neq i,j} \sum_{\alpha} \dot{E}_{i\alpha}^{A} \partial_{r_n} \gamma_{ijk}^{(\alpha)}(r_{ij}, r_{ik}, r_{jk}) \tag{13}$$

where we sum over all pairs of nearest-neighbor atoms, including a factor of 1/2 in order to account for double counting of forces due to equivalent pairs of nearest-neighbor atoms.

Taking the derivative of each position in turn, we expand equation 13 into:

$$\begin{aligned} F_n^A = &-\frac{1}{2} \sum_{j \neq n} \sum_{k \neq j,n} \sum_{\alpha} \dot{E}_{n\alpha}^{A} \partial_{r_n} \gamma_{njk}^{(\alpha)}(r_{nj}, r_{nk}, r_{jk}) \\ &-\frac{1}{2} \sum_{j \neq n} \sum_{k \neq j,n} \sum_{\alpha} \dot{E}_{j\alpha}^{A} \partial_{r_n} \gamma_{jnk}^{(\alpha)}(r_{nj}, r_{jk}, r_{nk}) \\ &-\frac{1}{2} \sum_{j \neq n} \sum_{k \neq j,n} \sum_{\alpha} \dot{E}_{j\alpha}^{A} \partial_{r_n} \gamma_{jkn}^{(\alpha)}(r_{jk}, r_{nj}, r_{nk}) \end{aligned} \tag{14}$$

The derivative of the angular functions is more complex than the derivative of the radial functions because of its dependence on the cosine of the angle between the $r_{ij}$ and the $r_{ik}$ vectors. The best approach to calculating the derivative is to split the symmetry function into a term that depends only the cosine and a term that depends only upon interatomic distances:

$$\gamma_{ijk}^{(\alpha)}(r_{ij}, r_{ik}, r_{jk}) = f_{ijk}^{(\alpha)}(\cos(\theta_{ijk})) h_{ijk}^{(\alpha)}(r_{ij}, r_{ik}, r_{jk}) \tag{15}$$

We can then write equation 14 as:

$$
\begin{aligned}
F_n^A = &-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{n\alpha}\left[\partial_{r_n} f^{(\alpha)}_{njk}(\cos(\theta_{njk}))h^{(\alpha)}_{njk}(r_{nj}, r_{nk}, r_{jk})\right] \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{j\alpha}\left[\partial_{r_n} f^{(\alpha)}_{jnk}(\cos(\theta_{jnk}))h^{(\alpha)}_{jnk}(r_{nj}, r_{jk}, r_{nk})\right] \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{j\alpha}\left[\partial_{r_n} f^{(\alpha)}_{jkn}(\cos(\theta_{jkn}))h^{(\alpha)}_{jkn}(r_{jk}, r_{nj}, r_{nk})\right] \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{n\alpha}\left[f^{(\alpha)}_{njk}(\cos(\theta_{njk}))\partial_{r_n} h^{(\alpha)}_{njk}(r_{nj}, r_{nk}, r_{jk})\right] \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{j\alpha}\left[f^{(\alpha)}_{jnk}(\cos(\theta_{jnk}))\partial_{r_n} h^{(\alpha)}_{jnk}(r_{nj}, r_{jk}, r_{nk})\right] \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{j\alpha}\left[f^{(\alpha)}_{jkn}(\cos(\theta_{jkn}))\partial_{r_n} h^{(\alpha)}_{jkn}(r_{jk}, r_{nj}, r_{nk})\right]
\end{aligned}
\tag{16}
$$

Using the chain rule, we can write the above equation as:

$$
\begin{aligned}
F_n^A = &-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{n\alpha}\dot{f}^{(\alpha)}_{njk}(\cos(\theta_{njk}))h^{(\alpha)}_{njk}(r_{nj}, r_{nk}, r_{jk})\partial_{r_n}\cos(\theta_{njk}) \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{j\alpha}\dot{f}^{(\alpha)}_{jnk}(\cos(\theta_{jnk}))h^{(\alpha)}_{jnk}(r_{nj}, r_{jk}, r_{nk})\partial_{r_n}\cos(\theta_{jnk}) \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{j\alpha}\dot{f}^{(\alpha)}_{jkn}(\cos(\theta_{jkn}))h^{(\alpha)}_{jkn}(r_{jk}, r_{nj}, r_{nk})\partial_{r_n}\cos(\theta_{jkn}) \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{n\alpha}f^{(\alpha)}_{njk}(\cos(\theta_{njk}))\partial_{r_n} h^{(\alpha)}_{njk}(r_{nj}, r_{nk}, r_{jk}) \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{j\alpha}f^{(\alpha)}_{jnk}(\cos(\theta_{jnk}))\partial_{r_n} h^{(\alpha)}_{jnk}(r_{nj}, r_{jk}, r_{nk}) \\
&-\frac{1}{2}\sum_{j\neq n}\sum_{k\neq j,n}\sum_{\alpha}\dot{E}^A_{j\alpha}f^{(\alpha)}_{jkn}(\cos(\theta_{jkn}))\partial_{r_n} h^{(\alpha)}_{jkn}(r_{jk}, r_{nj}, r_{nk})
\end{aligned}
\tag{17}
$$

where $\dot{f}_{ijk}(x) \equiv \partial f_{ijk}(x)/\partial x$. Finally, we compute the gradients of the cosine function and the distance-dependent part of the symmetry function. We write the gradient of the cosine function below

$$
\begin{aligned}
\partial_{r_i}\cos(\theta_{ijk}) &= (r_{ik}^{-1} - \cos(\theta_{ijk})r_{ij}^{-1})\hat{r}_{ij} + (r_{ij}^{-1} - \cos(\theta_{ijk})r_{ik}^{-1})\hat{r}_{ik} \\
\partial_{r_j}\cos(\theta_{ijk}) &= -r_{ij}^{-1}\hat{r}_{ik} + \cos(\theta_{ijk})r_{ij}^{-1}\hat{r}_{ij} \\
\partial_{r_k}\cos(\theta_{ijk}) &= -r_{ik}^{-1}\hat{r}_{ij} + \cos(\theta_{ijk})r_{ik}^{-1}\hat{r}_{ik}
\end{aligned}
\tag{18}
$$

We then substitute equation 18 into equation 17 and write out the derivatives to find the final equation for the angular force:

$$
\begin{aligned}
F_i^A = &-\frac{1}{2}\sum_{j\neq i}\sum_{k\neq j,i}\sum_{\alpha} \\
&+\dot{E}^A_{i\alpha} \dot{f}^{(\alpha)}_{ijk}(\cos(\theta_{ijk}))h^{(\alpha)}_{ijk}(r_{ij},r_{ik},r_{jk})\big[(r_{ik}^{-1}-\cos(\theta_{ijk})r_{ij}^{-1})\hat{r}_{ij}+(r_{ij}^{-1}-\cos(\theta_{ijk})r_{ik}^{-1})\hat{r}_{ik}\big] \\
&+\dot{E}^A_{j\alpha} \dot{f}^{(\alpha)}_{jik}(\cos(\theta_{jik}))h^{(\alpha)}_{jik}(r_{ij},r_{jk},r_{ik})\big[-r_{ji}^{-1}\hat{r}_{jk}+\cos(\theta_{jik})r_{ji}^{-1}\hat{r}_{ji}\big] \\
&+\dot{E}^A_{j\alpha} \dot{f}^{(\alpha)}_{jki}(\cos(\theta_{jki}))h^{(\alpha)}_{jki}(r_{jk},r_{ij},r_{ik})\big[-r_{ji}^{-1}\hat{r}_{jk}+\cos(\theta_{jki})r_{ji}^{-1}\hat{r}_{ji}\big] \\
&+\dot{E}^A_{i\alpha} f^{(\alpha)}_{ijk}(\cos(\theta_{ijk}))\big[h^{(\alpha 1)}_{ijk}(r_{ij},r_{ik},r_{jk})\hat{r}_{ij}+h^{(\alpha 2)}_{ijk}(r_{ij},r_{ik},r_{jk})\hat{r}_{ik}\big] \\
&+\dot{E}^A_{j\alpha} f^{(\alpha)}_{jik}(\cos(\theta_{jik}))\big[h^{(\alpha 1)}_{jik}(r_{ij},r_{jk},r_{ik})\hat{r}_{ij}+h^{(\alpha 3)}_{jik}(r_{ij},r_{jk},r_{ik})\hat{r}_{ik}\big] \\
&+\dot{E}^A_{j\alpha} f^{(\alpha)}_{jki}(\cos(\theta_{jki}))\big[h^{(\alpha 2)}_{jki}(r_{jk},r_{ij},r_{ik})\hat{r}_{ij}+h^{(\alpha 3)}_{jki}(r_{jk},r_{ij},r_{ik})\hat{r}_{ik}\big]
\end{aligned}
\tag{19}
$$

where we define $h^{(\alpha\beta)}_{ijk}(x_1,x_2,x_3)\equiv \partial h^{(\alpha)}_{ijk}(x_1,x_2,x_3)/\partial x_\beta$.

Once again, we can simplify equation 19 greatly by using Newton's second law and recognizing that we can compute all forces on all atoms by computing the gradients for each atom in sequence. We thus group all terms in the force on atom $(i)$ which include the gradient of the energy $E_i$ of atom $(i)$ and drop all other terms. For a given symmetry function $(\alpha)$ and triple of atoms $(i,j,k)$, we can write the force as:

$$
\begin{aligned}
ga^{(\alpha)}_{ijk} =& -\frac{1}{2}\dot{E}^A_{i\alpha}\dot{f}^{(\alpha)}_{ijk}(\cos(\theta_{ijk}))h^{(\alpha)}_{ijk}(r_{ij},r_{ik},r_{jk}) \\
gd1^{(\alpha)}_{ijk} =& -\frac{1}{2}\dot{E}^A_{i\alpha}f^{(\alpha)}_{ijk}(\cos(\theta_{ijk}))h^{(\alpha 1)}_{ijk}(r_{ij},r_{ik},r_{jk}) \\
gd2^{(\alpha)}_{ijk} =& -\frac{1}{2}\dot{E}^A_{i\alpha}f^{(\alpha)}_{ijk}(\cos(\theta_{ijk}))h^{(\alpha 2)}_{ijk}(r_{ij},r_{ik},r_{jk}) \\
F_i^A \rightarrow& F_i^A + ga^{(\alpha)}_{ijk}\big[(r_{ik}^{-1}-\cos(\theta_{ijk})r_{ij}^{-1})\hat{r}_{ij}+(r_{ij}^{-1}-\cos(\theta_{ijk})r_{ik}^{-1})\hat{r}_{ik}\big] \\
&+\big[gd1^{(\alpha)}_{ijk}\hat{r}_{ij}+gd2^{(\alpha)}_{ijk}\hat{r}_{ik}\big] \\
F_j^A \rightarrow& F_j^A - ga^{(\alpha)}_{ijk}(r_{ik}^{-1}-\cos(\theta_{ijk})r_{ij}^{-1})\hat{r}_{ij} - gd1^{(\alpha)}_{ijk}\hat{r}_{ij} \\
F_k^A \rightarrow& F_k^A - ga^{(\alpha)}_{ijk}(r_{ij}^{-1}-\cos(\theta_{ijk})r_{ik}^{-1})\hat{r}_{ik} - gd2^{(\alpha)}_{ijk}\hat{r}_{ik}
\end{aligned}
\tag{20}
$$

## 5 Energy and Forces - Algorithms

Although NNPs are much less expensive compared to the ab-initio methods, the calculation of symmetry functions and neural networks can be computationally intensive. Thus, it is imperative that we design algorithms for evaluation of NNPs that are as efficient as possible. In this section we will discuss the evaluation of neural networks, the logistics of data storage, and efficient evaluation of the symmetry functions and forces.

As stated earlier, neural networks are best thought of as vectors of nodes connected by matrices of edges. This allows us to write nearly all operations involving neural networks in terms of matrix-vector operations. In addition, both the optimization of the neural network and the gradients of the outputs of the neural network can be framed in terms of back-propagation, which involves propagating fluctations in the output backwards through the network by matrix multiplication. When we consider the relatively small sizes of neural networks associated with atomic simulation, the evaluation of neural networks is generally *not* a computational bottleneck. Indeed, evaluation of neural networks is often one of the fastest components of NNPs.

Due to the cost of NNPs, and as the targets for NNPs include simulations with thousands of atoms, we must consider how to parallelize a NNP code. This is a somewhat difficult task due to the size of NNs. Even a small NN will typically require hundreds of floating point parameters, and roughly double that number are needed for computing and storing derivatives. Assigning a NN to each atom in a simulation is clearly impractical, further, since the NNs do not differ between species, this will result a great deal of redundant data.

The best parallelization strategy is to assign one NN per species to each processing unit (PU) in the simulation, and have each PU calculate energies and forces using those NNs. The NNs are constants which are set by a training procedure separate and apart from the MD simulation, and so the NNs need to be broadcast only once at the beginning of a simulation. This then requires each PU to evaluate the energies and forces of each atom in a serial manner (leaving out hyper-threading). This is of course much more expensive than a simple pair potential, such as a Lennard-Jones interactions, but as NN evaluations usually only involve a few modest matrix multiplications (i.e. 15 x 15), NNPs are still much less expensive than ab-initio methods.

The evaluation of the symmetry functions is one of the more computationally intensive and difficult aspects of using NNPs. First, computing symmetry functions requires a large number of evaluations of expensive functions, such as Gaussian, cosine, and tanh functions. Secondly, most applications of NNPs require 6-12 radial symmetry functions per species and an equal number for each pair of species, resulting in a large number of evaluations. Finally, evaluation of the symmetry functions is difficult to optimize as the local environment around an atom is, well, *local.* For each atom, one must sum over the contributions of the symmetry functions for each atomic distance in the local neighborhood. By keeping track of atoms with the same species, one might be able to eliminate some computation using symmetry, but the bookkeeping required to do so correctly would likely erase any potential efficiency gains. Thus, the symmetry functions and their associated cutoffs should be the most optimized aspects of the code, as their evaluation will take up a large portion of both the operation count and the total runtime.

The evaluation of forces, though it involves evaluation of symmetry functions, is much more open to optimization due to the symmetry of the problem and the simpler nature of force calculations. Rather than keep track of different network inputs corresponding to different symmetry functions for different pair and triple interactions, we simply need to sum the force vectors generated by the networks and symmetry functions associated with the local environment around an atom. In addition, because of Newton's third law, we can take advantage of symmetry to greatly reduce the computational effort.

The computational expense of calculating the radial force can be greatly reduced by noting that, although the force depends on the neural network gradients of each atom, the gradients can be computed separately for each atom. Rather than perform each loop in equation 11 for each atom, it is far more efficient to compute the sum over $\alpha$ for the first term only in 11. After the sum over $\alpha$ is completed, the resulting force is added to the total force on atom $(i)$ and subtracted from atom $(j)$. By performing this procedure on the full neighbor list for a given atom, we skip the step of separately computing the second term in 11 for each atom, but still guarantee that term is added to the total force on each atom.

A similar approach can be taken to simplify computation of the angular component of the force as well. Specifically, we only need to calculate the first and fourth term of equation 19. The resulting forces along the $\hat{r}_{ij}$ and $\hat{r}_{ik}$ vectors computed in the loop over $\alpha$ are then added to atom $(i)$ and subtracted from atom $(j)$ and $(k)$, respectively.