Particle Networks: A Variation on Multilayer Perceptrons with Spatial Pairwise Kernels

Adrian W. Lange

November 17, 2015

Abstract

We present a formulation of an Artificial Neural Network akin to the classic Multilayer Perceptron model but with weight matrices replaced by implicit pairwise kernels between units. We call our neural network model a Particle Network. Each unit in a Particle Network is conceptualized as a particle with a variable charge and position in $\mathbb{R}^n$. Units interact between layers via a chosen kernel function, and input data is passed through the network in the usual feed-forward pattern of Multilayer Perceptrons. We exhibit our model with an experiment on the MNIST digit dataset and discuss possible advantages of Particle Networks, such as reduced computer memory cost and parallelism strategies.

1 Introduction

The Multilayer Perceptron (MLP) model is an Artificial Neural Network (ANN) that, in an extremely abstract sense, resembles how information might be received and transformed in an intelligent biological system like a human brain. For instance, the firing of a neuron is modeled mathematically as an activation function associated with each unit of the MLP. Additionally, interneuron connections of a brain are modeled in MLPs as cross-layer unit pair connections controlled by a weight variable.

We are interested in alternative representations of inter-unit connections. The reduction of an interneuron connection into a single independent weight variable seems to us to lack certain features of biological brains, namely spatial relationships of neurons. An so we have been motivated to consider if incorporating the notion of spatial relationships between units yields any benefits.[1] We have thus formulated an alternative version of the MLP in which each node is conceptualized as a particle with a variable charge and position in $\mathbb{R}^n$. We refer to our model as the Particle Network (PN).

Our intent with PNs is not necessarily to make a model more closely resemble a biological system but more of an exploratory study. We are just plain curious to see if the inter-unit connections in PNs might self-organize into interesting clusters or structures, which may be useful for visualization and/or model interpretation—contrasting to the near non-interpretability of weights in MLPs. If such structures appear regularly as part of optimized PNs, it may be possible that one could pre-construct PNs with common structural motifs when fitting a new PN to potentially enhance deep learning similar to starting with trained autoencoders when fine tuning a deep MLP. Moreover, there may exist certain reusable or optimized sets of pairwise kernels and

[1] It is pretty unlikely that we are the first to envision using positions as part of an ANN. However, we (the author, me, Adrian) are not particularly well-versed in the vast amount of ANN literature. And so, we make no claim of novelty here but do claim to have arrived at this idea independently (and naïvely). Others may have come up with similar models, but we still felt like writing a up study about it anyway.

charges, much like the force fields used widely in the field of molecular mechanics simulations. Also along such lines, PNs could maybe benefit from certain algorithmic approaches frequently used in N-body simulations, such as distance cutoffs for pairwise kernels, to reduce computational cost.

We present the formulation of the PN model in the following section, including its analytic gradient and an overview of the algorithm we have used to implement it with reasonable efficiency. We then in Section 3 perform an experiment with a PN on the commonly studied MNIST digit dataset. Comparisons are drawn between PNs and MLPs and discussed throughout this work.

2 Model Formulation and Implementation

We briefly review the formulation of MLPs before introducing the formulation of Particle Networks. We follow the notation of Graves [] as much as possible.

2.1 Multilayer Perceptron Formulation

Consider a hidden unit h in a MLP with activation function θ_h that is connected to I input units, each with input data x_i. The output b_h is given by the activation function applied to the weighted sum

$$a_h = \sum_i^I w_{ih} x_i \tag{1}$$

$$b_h = \theta_h(a_h) \tag{2}$$

where w_{ij} is the weight between unit i and unit j. It is common practice to include an additional fixed input $x_0 = 1$ such that the weight w_{0h} constitutes a constant offset, or bias term. The output of each hidden unit is propagated forward recursively to the l-th layer hidden layer H_l

$$a_h = \sum_{h' \in H_{l-1}} w_{h'h} b_{h'} \tag{3}$$

A bias term may also be included in Eq. 3. The output layer of a MLP, with units a_k for K output units, follows similar to Eq. 3. The output layer generally may employ any activation function, though it is typical for classification problems to chose the so-called softmax function to yield output class probabilities, y_k,

$$y_k = \frac{e^{a_k}}{\sum_{k'}^K e^{a_{k'}}} \tag{4}$$

Finally, a loss function $\mathcal{L}(x, z)$ is chosen to score the accuracy of the MLP mapping of input x to predicted output y against the true value z to which the MLP is fit. Several choices for loss function exist, but when fitting a MLP to predict multiple class output, the preferred choice is the categorical cross-entropy function

$$\mathcal{L}(x, z) = -\sum_k^K z_k \ln(y_k) \tag{5}$$

To fit a MLP for a supervised learning problem, the loss function is minimized, which can be accomplished in a number of ways but most often through a gradient-based approach, like stochastic gradient descent. The gradient of the loss function with respect to each unit's weights is computed via the backpropagation technique involving a backward pass of information through the network. For a MLP employing the categorical cross entropy loss function [Eq. 5] in conjunction with an output layer softmax activation function [Eq. 4], one has

$$\frac{\partial \mathcal{L}}{\partial a_k} = y_k - z_k \tag{6}$$

For the output layer, Then, one can recursively compute the following quantity at each layer (and similarly for the output layer)

$$\delta_h = \frac{\partial \mathcal{L}(x, z)}{\partial a_h} = \theta'_h(a_h) \sum_{h' \in H_{l+1}} \delta_{h'} w_{hh'} \tag{7}$$

where $\theta'(a_h)$ is the function $d\theta(a)/da$. Eq. 7 then enters the gradient for each unit through the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}(x, z)}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} = \delta_j b_i \tag{8}$$

2.2 Particle Network Formulation

The central concept of a Particle Network (PN) is to consider letting each h-th unit of a MLP be given a single charge (or weight), q_h, and a position, $\vec{r}_h$, in $\mathbb{R}^n$. We choose in this work to simply use three dimensional Euclidean space, $\mathbb{R}^3$.[2] We then allow the particles (or units) of layer l to interact in a pairwise fashion with the particles of layer $l+1$ as follows,

$$a_h = q_h \sum_{h' \in H_{l-1}} k(\vec{r}_h, \vec{r}_{h'}) b_{h'} \tag{9}$$

The pairwise kernel function, $k(\vec{r}_h, \vec{r}_{h'})$, can generally be any function to couple the coordinates of particles h and h', and we believe a distance based kernel to be the most obvious flavor. For now, we suggest a function that monotonically decreases as distance increases and also does not diverge at zero distance. In this work, we select the basic Gaussian kernel:

$$k(\vec{r}_h, \vec{r}_{h'}) = e^{-|\vec{r}_h - \vec{r}_{h'}|^2} \tag{10}$$

In addition to Eq. 9, we treat the input layer specially such that it too carries a charge and position. The output of the PN input layer is merely set as the scaled input[3]

$$a_i = b_i = q_i x_i \quad \text{for } i \in I \tag{11}$$

Eqs. 9 and 11 along with the choice of a kernel, like Eq. 10, constitute the major variation of a PN from the usual MLP. All other features of the MLP are assumed to be the same in a PN, such as the use of activation functions θ_h.

The gradient of a PN can then be derived and adapted into backpropagation with the following:

$$\frac{\partial a_h}{\partial q_h} = \sum_{h' \in H_{l-1}} k(\vec{r}_h, \vec{r}_{h'}) b_{h'} \tag{12}$$

$$\frac{\partial a_i}{\partial q_i} = x_i \quad \text{for } i \in I \tag{13}$$

$$\frac{\partial a_h}{\partial \vec{r}_h} = \sum_{h' \in H_{l-1}} b_{h'} \frac{\partial k(\vec{r}_h, \vec{r}_{h'})}{\partial \vec{r}_h} \tag{14}$$

$$\frac{\partial k(\vec{r}_h, \vec{r}_{h'})}{\partial \vec{r}_h} = -2e^{-|\vec{r}_h - \vec{r}_{h'}|^2} (\vec{r}_h - \vec{r}_{h'}) \tag{15}$$

By symmetry of the distance based pairwise kernel (i.e., Newton's 3^{rd} Law), we have

$$\frac{\partial a_h}{\partial \vec{r}_{h'}} = -\sum_{h' \in H_{l-1}} b_{h'} \frac{\partial k(\vec{r}_h, \vec{r}_{h'})}{\partial \vec{r}_h} \tag{16}$$

which can also be applied to the input layer by letting $h' \to i$ and $H_{l-1} \to I$. Thus, the modified version of Eq. 7 for a PN is

$$\delta_h = \frac{\partial \mathcal{L}(x,z)}{\partial a_h} = \theta'_h(a_h) \sum_{h' \in H_{l+1}} \delta_{h'} q_{h'} k(\vec{r}_h, r_{h'}^{\vec{}}) \tag{17}$$

The loss function gradient with respect to charges is then

$$\frac{\partial \mathcal{L}}{\partial q_h} = \frac{\partial \mathcal{L}(x,z)}{\partial a_h} \frac{\partial a_h}{\partial q_h} = \delta_h a_h / q_h \tag{18}$$

$$\frac{\partial \mathcal{L}}{\partial q_i} = \frac{\partial \mathcal{L}(x,z)}{\partial a_i} \frac{\partial a_i}{\partial q_i} = \delta_i x_i \tag{19}$$

For the loss function gradient with respect to position, one must be careful to sum Eq. 14 and Eq. 16 across each layer in which the h-th particle's position appears. For example, for unit h with $h \in H_l$ and $h' \in H_{l+1}$,

$$\frac{\partial \mathcal{L}}{\partial \vec{r}_h} = \frac{\partial \mathcal{L}(x,z)}{\partial a_h} \frac{\partial a_h}{\partial \vec{r}_h} + \sum_{h' \in H_{l+1}} \frac{\partial \mathcal{L}(x,z)}{\partial a_{h'}} \frac{\partial a_{h'}}{\partial \vec{r}_h} \tag{20}$$

which, for completeness, can be expanded by substitution of the above equations into each term

$$\frac{\partial \mathcal{L}(x,z)}{\partial a_h} \frac{\partial a_h}{\partial \vec{r}_h} = \delta_h \sum_{h'' \in H_{l-1}} b_{h''} \frac{\partial k(\vec{r}_h, \vec{r}_{h''})}{\partial \vec{r}_h} \tag{21}$$

$$\frac{\partial \mathcal{L}(x,z)}{\partial a_{h'}} \frac{\partial a_{h'}}{\partial \vec{r}_h} = -\delta_{h'} \sum_{h'' \in H_{l+1}} b_{h''} \frac{\partial k(\vec{r}_h, \vec{r}_{h''})}{\partial \vec{r}_h} \tag{22}$$

[2] It is entirely conceivable to make other choices, such as $\mathbb{R}^2$ or, say, hyperbolic space, if that's your thing.

[3] Scaling the input may or may not be useful, and we compare models with and without input scaling later in this work.

For the input layer, having no layer preceding it, Eq. 23 is reduced to

$$\frac{\partial \mathcal{L}}{\partial \vec{r}_i} = \sum_{h \in H_{l+1}} \frac{\partial \mathcal{L}(x,z)}{\partial a_h} \frac{\partial a_h}{\partial \vec{r}_i} \tag{23}$$

which can be computed similar to Eq. 22.

2.3 Particle Network Implementation

The equations for the PN model appear somewhat daunting at first glance in comparison to the MLP model, but their implementation in code is more straightforward than might appear. We have implemented the PN model (for arbitrary number of units and layers) in our experimental neural network code, called Calrissian (available on GitHub []), which is written in Python and makes heavy use of the NumPy library for efficient vectorized computation.

2.3.1 Linear Scaling Memory and Computational Complexity

An interesting feature of PNs is that it does not explicitly require storing weight matrices in memory for each layer (or weight vectors per unit, depending on implementation) like a MLP does. In fact, one can recover MLP weights from PNs by setting

$$q_j k(\vec{r}_i, \vec{r}_j) \rightarrow w_{ij} \tag{24}$$

where w_{ij} is the corresponding MLP weight from 3. (It is not clear to use at this point that the reverse mapping from MLP weight to PN charge and position is uniquely defined/computed given that q_h of Eq. 9 appears in each term, suggesting that, for example, weights of alternating sign cannot be reverse mapped.) In PNs, one could explicitly construct a distance matrix or kernel matrix for all pairs of units and have the same memory requirement of MLPs, but this is not strictly necessary. A more memory efficient approach for PNs is to implicitly build these matrices by computing $k(\vec{r}_i, \vec{r}_j)$ on-the-fly as needed and subsequently discarding the value from memory.

With such an implementation, for a network having N units in layer h connected to M units in layer $h+1$, the memory required between these layers for a PN scales as $O(mN + mM)$, where $m = n + 1$ with n as the number of dimensions in $\mathbb{R}^n$ and the one from the single charge parameter. For comparison, the MLP weight matrix for the same sort of layer and unit setup requires memory scaling as $O(NM)$. Assuming $N \sim M$, we can argue that MLPs have a roughly quadratic scaling memory cost, whereas PNs have a roughly linear scaling memory cost for the same number of units. Still, the computational scaling of MLPs and NPs remains roughly the same for the same number of units, being $O(NM)$.

Linear scaling memory could be a potentially useful feature of PNs in parallelization strategies requiring communication of parameters across distributed memory and/or between CPU and GPU memory. The latter is sometimes cited as a concerning bottleneck when using GPUs for parallelization, and having to transfer a linear scaling amount of memory as opposed to a quadratic scaling amount of memory could yield a performance boost. Or, one might possibly be able to squeeze larger/deeper PN networks on a GPU. Experiments will be needed to evaluate such a claim, of course, as well as to understand the balance of memory benefit and model predictive power.

PNs, however, might require slightly more floating point operations than MLPs. On the other hand, PNs might be able to exploit spatial sparseness to reduce computational complexity by invoking distance cutoffs. For instance, for distant unit pairs with kernels of approximately zero, one can skip their computation. In N-body simulations with short-range kernels, cutoffs and neighbor lists can be used to achieve nearly linear scaling computational costs in systems with many particles spanning relatively large distances. Additionally, fast multipole methods and/or domain decomposition could possibly be used to achieve $O(N\ln N)$ complexity and/or scalable model parallelization, respectively. Again, the realization of any of these claimed potential benefits will need some experimental evidence as support.

2.3.2 Forward Pass

The forward pass of a PN proceeds quite similar to that of an MLP forward pass. To take advantage of linear scaling memory, though, we need to structure the algorithm in such a way that each element of the kernel matrix computed on-the-fly is available for each input instance. See Algorithm 1, which saves w_{ij} in memory only for the innermost loop over the N input instances of data. Of course, if the feed forward procedure were being called many times for a constant set of PN parameters, it might be useful to explicitly cache w_{ij} as a matrix rather than recompute for every forward pass. But, for PN learning, the parameters change with every update in the optimization algorithm, defeating the purpose of caching. Naturally, it is up to the developer to decide if caching is appropriate or not.

Algorithm 1 PN Forward Pass for input data set with N instances

```
procedure FORWARDPASS
    for i ∈ input, n ∈ N instances do
        b_i^n = q_i x_i^n
    end for
    for l ∈ H_l do
        a = 0
        for i ∈ l − 1, j ∈ l do
            w_ij = q_j k(r_i, r_j)
            for n ∈ N do
                a_j^n = a_j^n + w_ij b_i^n
            end for
        end for
        b = θ_l(a)
    end for
    return b
end procedure
```

2.3.3 Backpropagation

To compute the analytic gradient of a PN, one can apply the backpropagation technique as is done in MLPs. Like the forward pass, PN backpropagation can be written in such a way as to avoid explicit construction of matrices (Algorithm 2). Much like MLP backpropagation, a forward pass is first performed to compute the activations of each layer. Then, a backward pass is performed in which the gradient information of one layer is passed back to previous layers recursively. As with the forward pass, the loops can be structured so as to avoid explicit matrix construction.

For completeness, we note that since we have for this work chosen the Gaussian pairwise kernel (Eq. 10), we have

$$\begin{aligned}\partial_{\vec{r}_j} k(\vec{r}_i, \vec{r}_j) &= -2|\vec{r}_j - \vec{r}_i| e^{-|\vec{r}_j - \vec{r}_i|^2} \frac{(\vec{r}_j - \vec{r}_i)}{|\vec{r}_j - \vec{r}_i|} \\ &= -2e^{-|\vec{r}_j - \vec{r}_i|^2} (\vec{r}_j - \vec{r}_i) \qquad (25)\end{aligned}$$

3 Experiment

Experiment

MNIST digits with no augmentation.

4 Conclusions

This is a conclusion.

Algorithm 2 PN Backpropagation for input data set with N instances

```
procedure BACKPROPAGATION
    ForwardPass to compute b, θ'(a) ∀ l ∈ H_l
    for k ∈ K, n ∈ N do
        δ_k^n = y_k^n − z_k^n
    end for
    l = L
    while l >= 0 do
        δ' = 0
        for i ∈ l − 1, j ∈ l do
            w_ij = q_j k(r_i, r_j)
            for n ∈ N do
                δ'_i^n = δ'_i^n + θ'(a_i) w_ij δ_j^n
                ∂_q L_j^l = ∂_q L_j^l + k(r_i, r_j) Σ_n b_i^n δ_j^n
                Δ = Σ_n ∂_{r_j} k(r_i, r_j) b_i^n δ_j^n
                ∂_r L_j^l = ∂_r L_j^l + Δ
                ∂_r L_i^l = ∂_r L_i^l − Δ
            end for
        end for
        δ = δ'
        l = l − 1
    end while
    for j ∈ l_input do
        ∂_q L_j^{l_input} = ∂_q L_j^{l_input} + Σ_n b_j^n δ_j^n
    end for
    return ∂_q L, ∂_r L
end procedure
```

procedure BACKPROPAGATION
- ForwardPass to compute $b, \theta'(a)\ \forall\ l \in H_l$
- **for** $k \in K$, $n \in N$ **do**
  - $\delta_k^n = y_k^n - z_k^n$
- **end for**
- $l = L$
- **while** $l >= 0$ **do**
  - $\delta' = 0$
  - **for** $i \in l-1$, $j \in l$ **do**
    - $w_{ij} = q_j k(\vec{r}_i, \vec{r}_j)$
    - **for** $n \in N$ **do**
      - ${\delta'}_i^n = {\delta'}_i^n + \theta'(a_i) w_{ij} \delta_j^n$
      - $\partial_q \mathcal{L}_j^l = \partial_q \mathcal{L}_j^l + k(\vec{r}_i, \vec{r}_j) \sum_n b_i^n \delta_j^n$
      - $\Delta = \sum_n \partial_{\vec{r}_j} k(\vec{r}_i, \vec{r}_j) b_i^n \delta_j^n$
      - $\partial_{\vec{r}} \mathcal{L}_j^l = \partial_{\vec{r}} \mathcal{L}_j^l + \Delta$
      - $\partial_{\vec{r}} \mathcal{L}_i^l = \partial_{\vec{r}} \mathcal{L}_i^l - \Delta$
    - **end for**
  - **end for**
  - $\delta = \delta'$
  - $l = l - 1$
- **end while**
- **for** $j \in l_{input}$ **do**
  - $\partial_q \mathcal{L}_j^{l_{input}} = \partial_q \mathcal{L}_j^{l_{input}} + \sum_n b_j^n \delta_j^n$
- **end for**
- **return** $\partial_q \mathcal{L}, \partial_{\vec{r}} \mathcal{L}$

end procedure