Particle Networks

Adrian W. Lange

April 22, 2017

Abstract

We introduce a variation on the Multilayer Perceptron where weight matrices are replaced by implicit interactions between nodes. We call our model a Particle Network. Each node in a Particle Network is conceptualized as a particle carrying a dipole vector and position in space. Nodes interact between layers through a field generated by a selected pairwise function, and input data is passed through the network in the usual feed-forward pattern. This approach yields a model whose number of free parameters scales proportional to the number nodes in the network, as opposed to the number of connections. We demonstrate Particle Networks in a few experiments and show as much as 10x memory savings/parameter reduction in comparison to an equivalent Multilayer Perceptron with little accuracy loss.

1 Introduction

We are interested in exploring alternative representations of inter-node connections in the Multilayer Perceptron (MLP) model. MLPs place a freely adjustable weight on each connection between nodes, but many of these connections may provide little to no benefit in overall model accuracy, which has led to the development of pruning and parameter reduction approaches (citations). In a similar vein, we consider the potential benefits of representing connection weights with implicit pairwise interactions that capture spatial relationships between nodes. As a result, we have formulated a variant of the MLP in which each node (or particle) is given a dipole vector and position in $\mathbb{R}^n$ space. We refer to this model as the Particle Network (PN).

We present the formulation of the PN model in the following section. The analytic gradient (*i.e.*, back-propagation gradient) of PNs is presented in the Appendix. We then briefly describe certain benefits and caveats of PNs, and in Section 4 we perform a series of experiments on well known datasets. Comparisons are drawn between PNs and MLPs and discussed throughout this work.

2 Model Formulation

2.1 Multilayer Perceptron Formulation

We briefly review the formulation of MLPs before introducing the formulation of Particle Networks in order to establish notation and make clear comparisons.

Consider a single node j in layer l of a MLP with activation function ϕ_j that is connected to N input nodes. The output b_j is given by the activation function applied to the weighted sum

$$a_j^{l+1} = \sum_i^N w_{ij}^l b_i^l \tag{1}$$

$$b_j^l = \phi_j(a_j^l) \tag{2}$$

where w_{ij}^l is the weight of the connection between input node i and output node j for layer l. It is common practice to include an additional fixed input $b_0 = 1$ such that the weight w_{0j}^l constitutes a constant offset, or bias term for each j-th node. Additionally, for the input to the network at layer $l = 0$, we can simply write that $b_i^0 = x_i$, where x_i is the value of the i-th feature of the input data.

MLPs use a feedforward mechanism to map input data to predicted output, wherein the output of the nodes in layer l is propagated forward to layer $l+1$ recursively until the final output layer is reached to yield a predicted output value or vector $\mathbf{y}$.

MLPs are typically fit via a supervised learning approach where a loss function, $\mathcal{L}(\mathbf{x}, \hat{\mathbf{y}})$, is chosen to score the accuracy of the MLP mapping input $\mathbf{x}$ to a predicted output $\mathbf{y}$ against the true value $\hat{\mathbf{y}}$. Learning is achieved by minimizing the loss function for a given data set. This is often accomplished by computing the analytic gradient (see Appendix) of the loss function with respect to weights and using that as input for some standard optimization algorithm, like gradient descent.

2.2 Particle Network Formulation

The central concept behind PNs is to reimagine the abstract network graph of a MLP instead as a concrete spatial arrangement of particles and then to ask how these particles might interact in a way that would implicitly represent the MLP network. To accomplish such, consider letting each node of the network be given a Cartesian coordinate $\vec{r}_j$ in $\mathbb{R}^n$ space along with a a dipole vector $\vec{v}_j$. Then, much like an electric field produced by a set of electric dipoles, the nodes of a PN produce a vector field for inter-layer interaction according to some chosen pairwise function or radial basis function (RBF).

There are many ways one can construct the pairwise interactions within the above framework for PNs, but in this work we choose to focus only on one that has proven to yield reasonable success. For particles $i \in l$ interacting with particles $j \in l+1$, we choose the following pairwise function to represent a weight w_{ij}^l:

$$w_{ij}^l = v_{ij} e^{-r_{ij}^2} \tag{3}$$

where $r_{ij} = |\vec{r}_i - \vec{r}_j|$ and $v_{ij} = \vec{v}_i \cdot \vec{v}_j$. We then have the following pre-activation quantity for each j-th node in layer l:

$$a_j^{l+1} = \sum_{i \in l} v_{ij} e^{-r_{ij}^2} b_i^l \tag{4}$$

In addition to the above, we enforce a constraint that the position $\vec{r}_j$ and dipole $\vec{v}_j$ is shared between adjacent layers. That is, the output particles of layer l have the same position and phase as the input particles of layer $l+1$. While this is not necessarily required, we have found it to help in interpretation and parameter reduction. The analytic gradient for this PN model formulation is provided in the Appendix.

3 Model Implementation and Parameter Scaling

We have implemented the above formulation of a PN for arbitrary numbers of nodes and layers in our experimental neural network code called Calrissian (written in Python and available on GitHub [2]). The MLP model is also implemented as well as a handful of common optimization algorithms.

In our implementation, we take a memory efficient approach such that each individual w_{ij} is computed on-the-fly and subsequently discarded when no longer needed. This is done to take advantage of the property that the number of free parameters in a PN, $N_{\text{param}}^{\text{PN},n}$, is proportional to the number of nodes in each layer:

$$N_{\text{param}}^{\text{PN},n} = 2nN_{\text{in}}^0 + (2n+1)\sum_l N_{\text{out}}^l \tag{5}$$

where n is the chosen dimensionality $\mathbb{R}^n$ of the PN, and N_{in}^l and N_{out}^l are the number of input and output nodes for layer l, respectively. This is in comparison to the MLP, whose number of parameters and corresponding memory cost scale proportional to the product of input nodes (plus a bias term) and output nodes for each layer:

$$N_{\text{param}}^{\text{MLP}} = \sum_l (N_{\text{in}}^l + 1) N_{\text{out}}^l \tag{6}$$

For small networks, $N_{\text{param}}^{\text{PN},n}$ may be greater than $N_{\text{param}}^{\text{MLP}}$, such as when $N_{\text{in}}^l < 2n+1$. However, this quickly switches over to $N_{\text{param}}^{\text{PN},n} < N_{\text{param}}^{\text{MLP}}$ as network size increases (see Section 4).

The parameter/memory scaling of PNs is an attractive beneficial feature, possibly enabling learning on hardware with small available memory and/or disk that otherwise may not be able to accommodate a similar MLP. Moreover, the reduced memory could ameliorate the latency of communicating PN models across internet network connections or between host and device on systems with GPGPUs.

We note two important caveats, though: 1) PNs require more computation than the architecture equivalent MLP, assuming that the w_{ij} is computed on-the-fly. 2) The accuracy of a PN is bounded such that it cannot exceed a MLP with an isomorphic topology (*i.e.* an MLP with the same arrangement of nodes and layers). This is readily apparent upon realization that all PN implicit weight matrices are contained as a subset of all possible isomorphic MLP configurations. In other words, a PN maps directly to an MLP, but it is not guaranteed that an arbitrary MLP maps to a PN. A PN can therefore be considered an approximation to an isomorphic MLP achieving equal or greater accuracy.

As a counterpoint to the second caveat, of course, one generally can increase the number of hidden nodes and/or layers in a PN to improve its accuracy, perhaps to a practically negligible difference. We explore this trend in Section 4.

4 Experiments

We demonstrate PNs in a series of experiments on the well studied MNIST data set [4] for classifying handwritten digits.

We construct an experiment to compare PNs and MLPs for varying network topologies, denoted by

Topology	Particle Network, $n = 2$ A_{train}	A_{test}	N_{param}	Particle Network, $n = 3$ A_{train}	A_{test}	N_{param}	Multilayer Perceptron A_{train}	A_{test}	N_{param}
786-10	92.4	92.2	3186	93.1	92.6	4774	95.3	96.0	1000
786-16-10	94.5	94.2	3266	96.0	94.9	4886	95.3	96.0	1000
786-32-10	95.4	94.8	3346	96.5	95.7	4998	95.3	96.0	1000
786-64-10	95.8	95.3	3506	97.3	96.2	5222	95.3	96.0	1000
786-128-10	96.3	95.4	3826	97.7	96.4	5670	95.3	96.0	1000
786-256-10	96.1	95.5	4466	97.7	96.5	6566	95.3	96.0	1000
786-512-10	96.3	95.3	5746	97.6	96.7	8358	95.3	96.0	1000
786-64-32-10	96.5	95.7	3666	97.3	96.1	5446	95.3	96.0	1000
786-128-64-10	96.3	95.6	4146	97.9	96.6	6118	95.3	96.0	1000
786-256-128-10	96.4	95.4	5106	98.0	96.6	7462	95.3	96.0	1000

Table 1: foo

hyphen-separated number of nodes in each layers in the feedforward pattern from left to right (Table 1). Hidden layers are given a rectified linear unit (ReLU) activation function. The output layer is a softmax function, and the loss function used is categorical cross-entropy. The MLPs are initialized with weights according to Glorot uniform random sampling [**?**]. PN biases also use Glorot uniform random sampling. The PNs are initialized with position and dipole components randomly sample from a zero-centered normal distribution with standard deviation 0.5. The dipoles of PNs are subsequently normalized to unit length after sampling. PNs and MLPs are optimized according to the RMSProp [] algorithm with learning rate 0.005 and $\gamma = 0.99$. Models are run for 50 epochs using mini-batches of 100 data points.

Results of this experiment are shown in Table 1.

5 Discussion

This is a discussion.

Fewer parameters, less likely to overfit. Overfitting a concern with so many parameters. Cite AlexNet paper. Mention Circulant matrix paper. Limited to be square matrix; use cutoffs and/or zero filling to fake out rectangular matrix. (This is not an issue in PNs)

Connections in PNs a bit weaker for fitting than connections in MLPs. May require more connections to be comparable to MLPs. Tradeoff.

6 Conclusions

This is a conclusion.

7 Appendix: Analytic Gradient

In a MLP, the analytic gradient of the loss function $\mathcal{L}(\mathbf{x}, \hat{\mathbf{y}})$ with respect to weights w_{ij}^l can be obtained by recursively computing (*i.e.*, backpropagating) the following quantity for each layer

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial a_j^l} = \phi_j'(a_j^l) \sum_{i \in l+1} \delta_i^{l+1} w_{ij}^{l+1} \tag{7}$$

where $\phi_j'(a_j)$ is the corresponding derivative function $d\phi(x)/dx$. Eq. 7 then enters the loss function gradient via the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^l} = \frac{\partial \mathcal{L}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{ij}^l} = \delta_j^l b_i^l \tag{8}$$

The gradient of a PN can be derived similarly and adapted into backpropagation making use of the chain rule:

$$\frac{\partial \mathcal{L}}{\partial q_j^l} = \frac{\partial \mathcal{L}}{\partial a_j^l} \frac{\partial a_j^l}{\partial q_j^l} \tag{9}$$

Analogous results are straightforwardly obtained for $\partial \mathcal{L}/\partial \vec{r}_j$ and $\partial \mathcal{L}/\partial \theta_j$. The modified version of Eq. 7 for a PN is

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial a_j^l} = \phi_j'(a_j^l) \sum_{i \in l+1} \delta_i^{l+1} q_j^{l+1} k_{ij}^{l+1} \tag{10}$$

The PN model presented in Section 2.2 has the following quantities for its gradient:

$$\frac{\partial a_j}{\partial q_j} = \sum_{i \in l-1} b_i k_{ij} = \frac{a_j}{q_j} \tag{11}$$

$$\frac{\partial a_j}{\partial \vec{r}_j} = \sum_{i \in l-1} b_i q_j \frac{\partial k_{ij}}{\partial \vec{r}_j} \tag{12}$$

$$\frac{\partial a_j}{\partial \theta_j} = \sum_{i \in l-1} b_i q_j \frac{\partial k_{ij}}{\partial \theta_j} \tag{13}$$

$$\frac{\partial k_{ij}}{\partial \vec{r}_j} = -2e^{-r_{ij}^2} \cos(\theta_{ij})(\vec{r}_j - \vec{r}_i) \tag{14}$$

$$\frac{\partial k_{ij}}{\partial \theta_j} = e^{-r_{ij}^2} \sin(\theta_{ij}) \tag{15}$$

Since the PN model shares $\vec{r}_j$ and θ_j between adjacent layers, we must also compute the following quantities, which hold by symmetry of the pairwise interaction:

$$\frac{\partial a_j}{\partial \vec{r}_i} = -\sum_{i \in l-1} b_i \frac{\partial k_{ij}}{\partial \vec{r}_j} \tag{16}$$

$$\frac{\partial a_j}{\partial \theta_i} = -\sum_{i \in l-1} b_i \frac{\partial k_{ij}}{\partial \theta_j} \tag{17}$$

We then arrive at the following:

$$\frac{\partial \mathcal{L}}{\partial q_j^l} = \frac{\partial \mathcal{L}}{\partial a_j^l} \frac{\partial a_j^l}{\partial q_j^l} = \delta_j \frac{a_j^l}{q_j^l} \tag{18}$$

$$\frac{\partial \mathcal{L}}{\partial \vec{r}_j^{\,l}} = \delta_j^l \frac{\partial a_j^l}{\partial \vec{r}_j^{\,l}} + \sum_i \delta_i^{l+1} \frac{\partial a_i^{l+1}}{\partial \vec{r}_j^{\,l}} \tag{19}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_j^{\,l}} = \delta_j^l \frac{\partial a_j^l}{\partial \theta_j^{\,l}} + \sum_i \delta_i^{l+1} \frac{\partial a_i^{l+1}}{\partial \theta_j^{\,l}} \tag{20}$$

For the loss function gradient with respect to shared positions and phases, one must be careful to also sum across each adjacent layer.

In addition to the above, we point out that because r_{ij} and θ_{ij} are invariant under uniform coordinate translations, we find it advisable to subtract the first moment of the full analytic gradient (*i.e.* the mean translational gradient) for each spatial and phase dimension. This ensures that the PN does not needlessly include bulk convective drift. Analogously, a PN is invariant to uniform coordinate rotations in the spatial dimension. It may be important to also remove such rotations from the gradient, but in our experiments so far we have not found this to be quite as prominent as translations. In this work, we ignore subtracting mean rotations and only subtract mean translations out of simplicity.

8 Appendix: Relationship to Radial Basis Function Networks

PNs somewhat superficially resemble Radial Basis Function (RBF) networks, but there are a few important differences worth pointing out. In general, the output b_j of a classic RBF network layer's j-th node can be written as:

$$b_j = \alpha_j \psi(|\mathbf{x} - \mathbf{c}_j|) \tag{21}$$

where ψ is some chosen radial basis function (*e.g.* a Gaussian function) α_j is a freely adjustable parameter, and $\mathbf{c}_j$ is the centroid vector of the j-th node. RBFs compute the distance between the input vector $\mathbf{x}$ and $\mathbf{c}_j$, which naturally both must be of the same length. Consequently, RBF networks require storage of $N_{\text{out}}(N_{\text{in}} + 1)$ parameters for each layer, scaling proportional to the number of connections, like MLPs. In addition, RBF networks do not possess the mapping to equivalent MLPs that PNs do (Eq. 3).

References

[1] GitHub, https://github.com/awlange/brainsparks as of November 2016.

[2] UCI Machine Learning Repository, http://archive.ics.uci.edu/ml/ as of November 2016.

[3] MNIST Database of Handwritten Digits, http://yann.lecun.com/exdb/mnist/ as of November 2016.