

Permutation invariance of neural networks

YKY

October 7, 2019

Permutation invariance of a neural network means that components of the input vector can be interchanged without affecting the output of the network. Suppose the input vector is the concatenation of n components, each of length m , *ie*, the input dimension is $n \cdot m$. \mathfrak{S}_n denotes the symmetric group of n elements, where $\sigma \in \mathfrak{S}_n$ is a permutation. All permutations are generated by transpositions of the form $(i \rightleftharpoons j)$.

Invariance means that:

$$\boxed{\text{invariance}} \quad \forall \sigma \in \mathfrak{S}_n. \quad F(\sigma \cdot x) = F(x) \quad (1)$$

The related concept of **equivariance** is defined by:

$$\boxed{\text{equivariance}} \quad \forall \sigma \in \mathfrak{S}_n. \quad F(\sigma \cdot x) = \sigma \cdot F(x) \quad (2)$$

(assuming the output also consists of n components).

There appears to be several ways to achieve permutation invariance in neural networks, but they all have various difficulties or drawbacks. We explore them in this paper:

- (A). By constraining the weights. If the activation function is an **analytic function** (whose Taylor series expansion has an infinite number of terms), there is in general no hope of making the 2 sides of (2) equal, because there is only a finite number of weights on both sides. The only hope of making (2) equal is by using **polynomial** activation functions. Since the composition of polynomials are polynomials, both sides of (2) are polynomials, so we can compare the coefficients of like terms. This leads to a set of (equality) constraints for the weights. The drawback of this method is that the number of constraints grows exponentially as the number of layers increases.
- (B). By mapping the input vector into a **free commutative** group, and embedding its Cayley graph into vector space. As is well known, the Cayley graph of the **free group** F_n is a tree, and can be embedded into the hyperbolic disc. However, the Abelianization of F_n becomes $F_n^{\text{Ab}} \cong \mathbb{Z}^n$, which is not a tree but is like a “grid” of dimension n . It seems impossible to embed \mathbb{Z}^n into lower dimensions unless **fractal** structures are involved. However, fractals are precisely a domain where neural networks may perform badly.
- (C). By transforming the input vector space into the frequency domain.
- (D). By constructing a symmetric function of the form $g(h(x_1), h(x_2), \dots, h(x_n))$ where g is an arbitrary symmetric function. This idea is from PointNet, but it seems to be very restrictive due to the choice of the function g . It seems that no choice of g exists such that the resulting family of functions are **dense** in a function space of interest. The denseness requirement comes from the Stone-Weierstrass theorem (1885, 1937).

1 Prior art

Symmetric NNs have been proposed in:

- Gens2014
- Bie2019
- Ravanbakhsh2016
- Ravanbakhsh2017
- Qi2016
- Qi2017
- Zaheer2017

1.1 Simple symmetrization is too inefficient

A simple fact: If $\mathbf{F}(\mathbf{p}, \mathbf{q})$ is any function, then

$$\mathbf{F}(\mathbf{p}, \mathbf{q}) + \mathbf{F}(\mathbf{q}, \mathbf{p}) \quad \text{or} \quad \mathbf{F}(\mathbf{p}, \mathbf{q})\mathbf{F}(\mathbf{q}, \mathbf{p}) \quad (3)$$

would be symmetric functions in (\mathbf{p}, \mathbf{q}) . This can be easily extended to \mathbb{P}^k .

Thus if \mathbf{F} is a “free” neural network, we can create a symmetric NN (via addition):

$$\mathbf{F}_{\text{sym}}(\mathbf{x}) = \frac{1}{k!} \sum_{\sigma \in \mathfrak{S}_k} \mathbf{F}(\sigma \cdot \mathbf{x}) \quad (4)$$

where \mathfrak{S}_k is the symmetric group of k elements, and $\mathbf{x} = \mathbf{p}_1 \wedge \mathbf{p}_2 \wedge \dots \mathbf{p}_k$. Back-propagation can be easily adapted to such symmetric NNs. However, if K is the size of working memory, it would require to perform forward propagation $k! \cdot {}_K C_k = {}_K P_k$ times for each step, which is computationally inefficient (the same reason why *Rete* was needed in classical AI).

So we would not use this idea, but it is illustrative of the problem.

2 Permutation-invariant polynomial neural networks

Traditional neural network:

$$\begin{array}{ll} \boxed{\text{neuron}} & y = \odot \mathbf{w} \cdot \mathbf{x} \\ \boxed{\text{layer}} & y = \odot W \mathbf{x} \\ \boxed{\text{network}} & y = \odot W \circ \odot W \dots \mathbf{x} \end{array} \quad (5)$$

Quadratic neural network:

$$\begin{array}{ll} \boxed{\text{neuron}} & y = W \mathbf{x} \cdot \mathbf{x} \\ \boxed{\text{layer}} & y = W \mathbf{x} \cdot \mathbf{x} \\ \boxed{\text{network}} & y = W \mathbf{x} \circ W \mathbf{x} \dots \mathbf{x} \end{array} \quad (6)$$

Traditionally, each neuron k with output o_k is defined as:

$$o_k = \mathcal{O}(\text{net}_k) = \mathcal{O} \left(\sum_{j=1}^n w_{jk} o_j \right). \quad (7)$$

This is replaced by our new neuron:

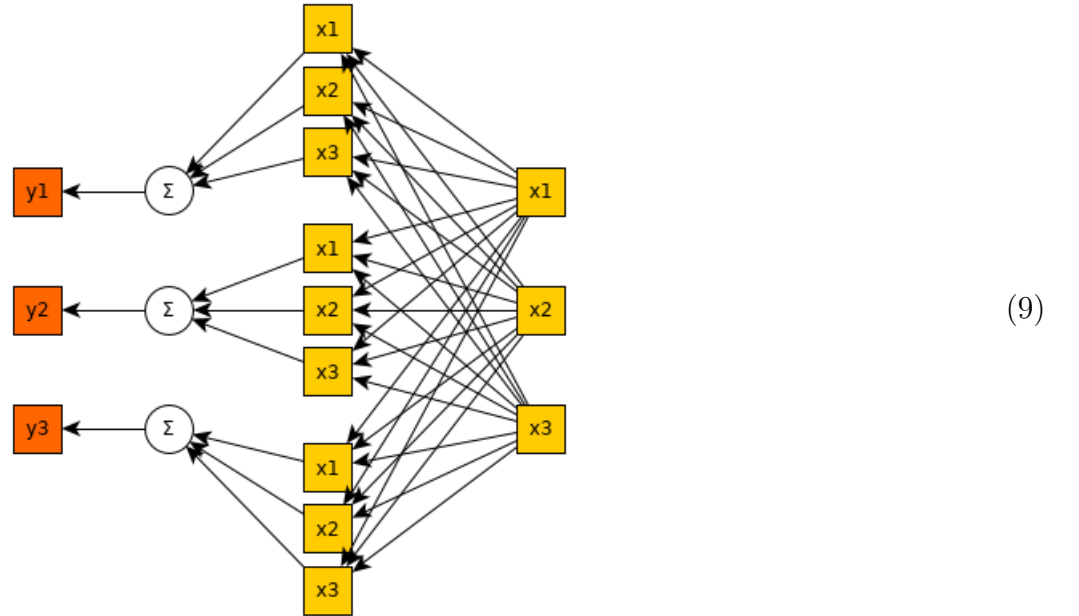
$$\boxed{\text{next layer}} \quad o_k = \text{net}_k = \sum_j \sum_i W_{ij}^k o_i o_j \quad \boxed{\text{current layer}} \quad (8)$$

where the 2 summations can be **interchanged**.

Let $y = F(x)$ denote the (overall) neural network function. In our case $F(x)$ is a polynomial $\in \mathbb{R}[x]^{\mathfrak{S}_n}$ where \mathfrak{S}_n is the symmetric group of n elements.

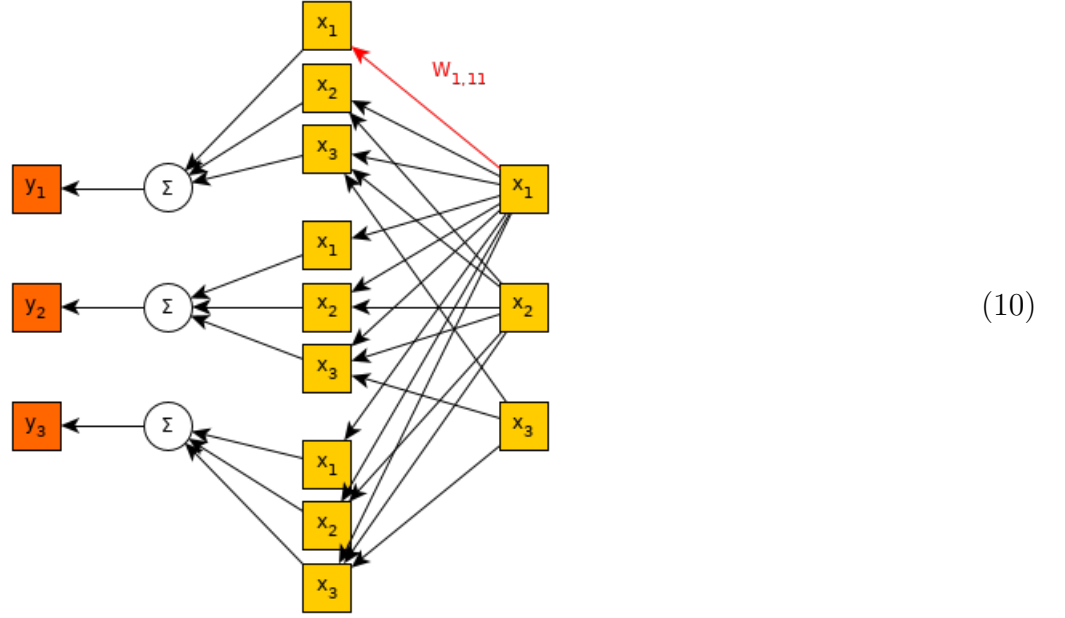
2.1 Redundant terms or coefficients

This is a quadratic layer, fully connected with all W_{ij} links:



but some coefficients are **redundant** because $x_i x_j = x_j x_i$.

One can eliminate redundant edges by imposing $i > j$:

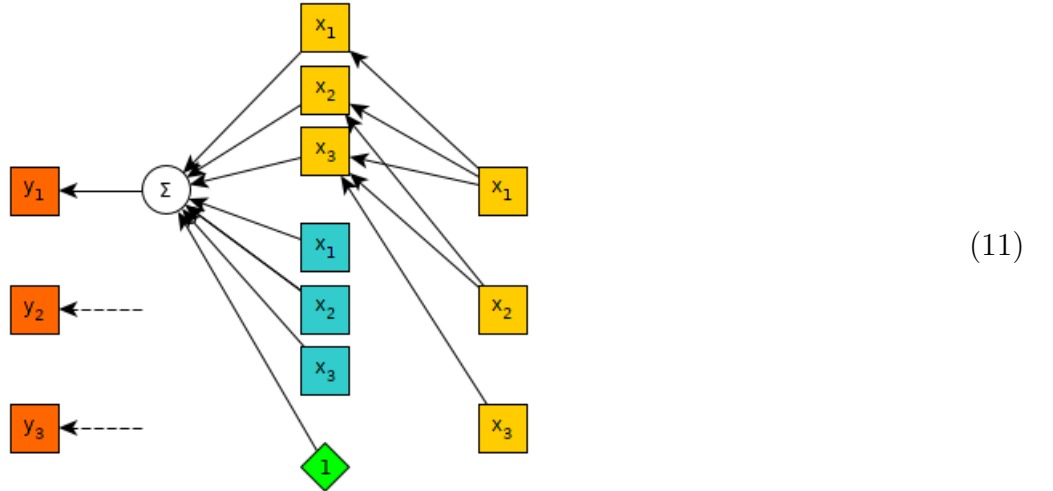


For example, the link in red is W_{11}^1 . However, as we shall see below, this method creates constraints with rather complicated indices. It appears that the first method with redundant weights is algebraically more elegant.

2.2 Making all terms homogeneous

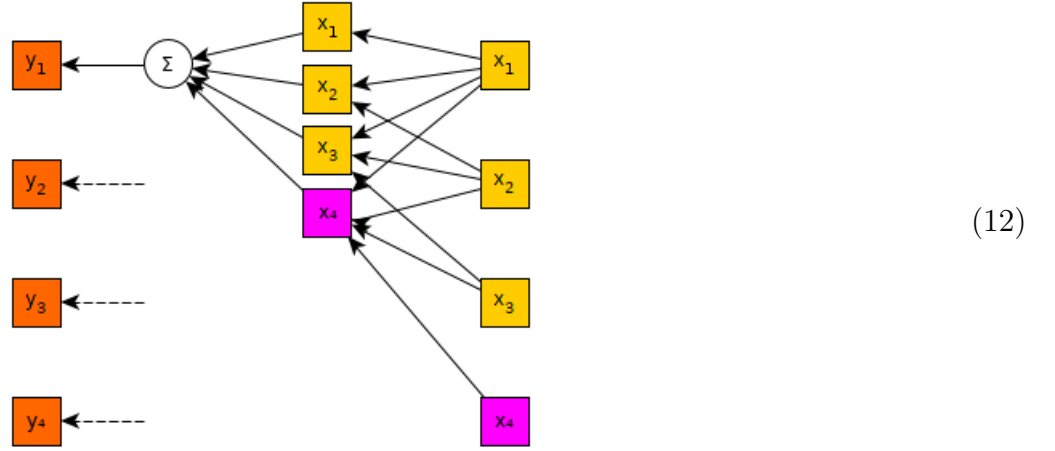
In a “quadratic” layer there would still be linear and constant terms. But we can absorb them all into a single matrix W , making all terms homogeneous of the form: $W_{ij}x_ix_j$.

For example, the following shows part of the network with quadratic, linear, and constant terms. The linear terms are in blue and the constant term is in green. Notice that this block should be repeated for each output y_1, \dots, y_3 :



Homogenization means adding a new input $x_4 \equiv 1$, similar to the “bias” term in traditional neural networks.

All the terms (quadratic, linear, constant) would be absorbed into a single matrix W :



2.3 Linear case: $y = Wx$

$$\boxed{\text{original}} \quad y_j = \sum_i W_{ij} x_i. \quad (13)$$

Equivariance implies:

$$\begin{aligned} \boxed{\text{LHS}} \quad y_j(\sigma(x_j \ x_k)x) &= \sigma \cdot y_j = y_k \quad \boxed{\text{RHS}} \\ \sum_{i \neq j, k} W_{ij} x_i + W_{kj} x_j + W_{jj} x_k &= \sum_{i \neq j, k} W_{ik} x_i + W_{jk} x_j + W_{kk} x_k. \end{aligned} \quad (14)$$

Comparing coefficients on the LHS and RHS yields:

$$\begin{aligned} W_{ij} &= W_{ik} & \forall j, k, (i \neq j, k) \\ W_{kj} &= W_{jk} & \forall j, k \\ W_{jj} &= W_{kk} & \forall j, k. \end{aligned} \quad (15)$$

In other words, the matrix W is of the form:

$$W = \alpha I + \beta 11^T. \quad (16)$$

2.4 Quadratic case: $y_k = W_k x \cdot x$ (with redundant indices)

The general form of a “quadratic” vector function is:

$$y = (Ax) \cdot x + Bx + C. \quad (17)$$

As described in §2.2, we absorb all weights into a single matrix, giving $(Ax) \cdot x$:

$$\boxed{\text{original}} \quad y_k = \sum_j \left[\sum_i a_{ij}^k x_i \right] x_j. \quad (18)$$

Note that the matrix A is “3D” and has $N \times N \times N$ entries.

Equivariance implies:

$$\boxed{\text{LHS}} \quad y_k(\sigma(x_k \ x_h) \cdot x) = \sigma \cdot y_k = y_h \quad \boxed{\text{RHS}} \quad (19)$$

$$\begin{aligned}
LHS &= \sum_j \left[\sum_{i \neq h, k} a_{ij}^k x_i + a_{hj}^k x_k + a_{kj}^k x_h \right] \sigma \cdot x_j \\
&= \sum_{j \neq h, k} \left[\sum_{i \neq h, k} a_{ij}^k x_i + a_{hj}^k x_k + a_{kj}^k x_h \right] x_j + \left[\sum_{i \neq h, k} a_{ih}^k x_i + a_{hh}^k x_k + a_{kh}^k x_h \right] x_k + \left[\sum_{i \neq h, k} a_{ik}^k x_i + a_{hk}^k x_k + a_{kk}^k x_h \right] x_h \\
&= \sum_{j \neq h, k} \sum_{i \neq h, k} a_{ij}^k x_i x_j + \sum_{j \neq h, k} a_{hj}^k x_k x_j + \sum_{j \neq h, k} a_{kj}^k x_h x_j \\
&\quad + \sum_{i \neq h, k} a_{ih}^k x_i x_k + a_{hh}^k x_k^2 + a_{kh}^k x_h x_k \\
&\quad + \sum_{i \neq h, k} a_{ik}^k x_i x_h + a_{hk}^k x_k x_h + a_{kk}^k x_h^2 \\
RHS &= \sum_j \left[\sum_i a_{ij}^h x_i \right] x_j \\
&= \sum_{j \neq h, k} \sum_{i \neq h, k} a_{ij}^h x_i x_j + \sum_{j \neq h, k} a_{kj}^h x_k x_j + \sum_{j \neq h, k} a_{hj}^h x_h x_j \\
&\quad + \sum_{i \neq h, k} a_{ik}^h x_i x_k + a_{kk}^h x_k^2 + a_{hk}^h x_h x_k \\
&\quad + \sum_{i \neq h, k} a_{ih}^h x_i x_h + a_{kh}^h x_k x_h + a_{hh}^h x_h^2
\end{aligned} \tag{20}$$

Comparing coefficients on the LHS and RHS yields:

$$\begin{aligned}
a_{ij}^h &= a_{ij}^k & \forall h, k, (i \neq h, k, j \neq h, k) \\
a_{kj}^h &= a_{hj}^k & \forall h, k, (j \neq h, k) \\
a_{hj}^h &= a_{kj}^k & \forall h, k, (j \neq h, k) \\
a_{ik}^h &= a_{ih}^k & \forall h, k, (i \neq h, k) \\
a_{ih}^h &= a_{ik}^k & \forall h, k, (i \neq h, k) \\
a_{kk}^h &= a_{hh}^k & \forall h, k \\
a_{hh}^h &= a_{kk}^k & \forall h, k \\
a_{hk}^h + a_{kh}^h &= a_{hk}^k + a_{kh}^k & \forall h, k.
\end{aligned} \tag{21}$$

The last constraint is not a simple equality of weights. It arises because $x_h x_k = x_k x_h$, a consequence of the **redundancy** of the weights as explained in §2.1.

How many distinct weights?

$$\begin{aligned}
N = 2 & \dots 6 & / & 8 & = 75\% \\
N = 3 & \dots 9 & / & 27 & = 33.3\% \\
N = 4 & \dots 11 & / & 64 & = 17/2\% \\
N = 5 & \dots 13 & / & 125 & = 10.4\% \\
N = 6 & \dots 15 & / & 216 & = 6.9\%
\end{aligned} \tag{22}$$

There would be N **blocks** of $N \times N$ matrices.

All diagonals consists of 2 colors, regardless of N (from 2nd and 3rd equations). This leaves $N(N - 1)$ non-diagonal entries per block.

Non-diagonal entries of different blocks are equal, if the block indices are different from the row and column indices. Out of N blocks there would be 2 different sets of non-diagonal weights. (This comes from the 1st equation.)

The last equation causes non-diagonal weights to have a certain symmetry about the diagonal.

2.5 Quadratic case: $y_k = W_k x \cdot x$ with ordered indices

As I said in §2.1, ordering the weights eliminates redundancy but also leads to complicated indexing.

Recall the general form $(Wx) \cdot x$:

$$\boxed{\text{original}} \quad y_k = \sum_j \left[\sum_{i \leq j} W_{ij}^k x_i \right] x_j. \quad (23)$$

The matrix W is “3-dimensional” and has $N \times N \times N$ entries.

Let $\sigma := (x_k \rightleftharpoons x_h)$, meaning **transposition** of the two elements. Equivariance implies:

$$\boxed{\text{LHS}} \quad y_k(\sigma \cdot x) = \sigma \cdot y_k = y_h \quad \boxed{\text{RHS}} \quad (24)$$

$$\begin{aligned}
\boxed{\text{LHS}} &= y_k(\sigma \cdot x) = \sum_j \left[\sum_{i \leq j} W_{ij}^k \sigma \cdot x_i \right] \sigma \cdot x_j \\
&= \sum_j \left[\sum_{\substack{i \leq j \\ i \neq h, k}} W_{ij}^k x_i + \underbrace{W_{hj}^k x_k}_{\text{if } h \leq j} + \underbrace{W_{kj}^k x_h}_{\text{if } k \leq j} \right] \sigma \cdot x_j \quad \boxed{\text{applied } \sigma \cdot x_i} \\
&= \sum_{j \neq h, k} \left[\sum_{\substack{i \leq j \\ i \neq h, k}} W_{ij}^k x_i + \underbrace{W_{hj}^k x_k}_{\text{if } h \leq j} + \underbrace{W_{kj}^k x_h}_{\text{if } k \leq j} \right] x_j \quad \boxed{\text{applied } \sigma \cdot x_j} \\
&\quad + \left[\sum_{\substack{i < h \\ i \neq k}} W_{ih}^k x_i + W_{hh}^k x_k + \underbrace{W_{kh}^k x_h}_{\text{if } k \leq h} \right] x_k + \left[\sum_{\substack{i < k \\ i \neq h}} W_{ik}^k x_i + \underbrace{W_{hk}^k x_k}_{\text{if } h \leq k} + W_{kk}^k x_h \right] x_h \\
&= \sum_{j \neq h, k} \sum_{\substack{i \leq j \\ i \neq h, k}} W_{ij}^k x_i x_j + \sum_{j > h} W_{hj}^k x_k x_j + \sum_{j > k} W_{kj}^k x_h x_j \\
&\quad + \sum_{\substack{i < h \\ i \neq k}} W_{ih}^k x_i x_k + W_{hh}^k x_k^2 + \underbrace{W_{kh}^k x_h x_k}_{\text{if } k \leq h} \\
&\quad + \sum_{\substack{i < k \\ i \neq h}} W_{ik}^k x_i x_h + \underbrace{W_{hk}^k x_k x_h}_{\text{if } h \leq k} + W_{kk}^k x_h^2 \\
\boxed{\text{RHS}} &= \sum_j \left[\sum_{i \leq j} W_{ij}^h x_i \right] x_j \\
&= \sum_j \left[\sum_{\substack{i \leq j \\ i \neq h, k}} W_{ij}^h x_i + \underbrace{W_{hj}^h x_h}_{\text{if } h \leq j} + \underbrace{W_{kj}^h x_k}_{\text{if } k \leq j} \right] x_j \\
&= \sum_{j \neq h, k} \left[\sum_{\substack{i \leq j \\ i \neq h, k}} W_{ij}^h x_i + \underbrace{W_{hj}^h x_h}_{\text{if } h \leq j} + \underbrace{W_{kj}^h x_k}_{\text{if } k \leq j} \right] x_j \\
&\quad + \left[\sum_{\substack{i < h \\ i \neq k}} W_{ih}^h x_i + W_{hh}^h x_h + \underbrace{W_{kh}^h x_k}_{\text{if } k \leq h} \right] x_h + \left[\sum_{\substack{i < k \\ i \neq h}} W_{ik}^h x_i + \underbrace{W_{hk}^h x_h}_{\text{if } h \leq k} + W_{kk}^h x_k \right] x_k \\
&= \sum_{j \neq h, k} \sum_{\substack{i \leq j \\ i \neq h, k}} W_{ij}^h x_i x_j + \sum_{j > h} W_{hj}^h x_h x_j + \sum_{j > k} W_{kj}^h x_k x_j \\
&\quad + \sum_{\substack{i < h \\ i \neq k}} W_{ih}^h x_i x_h + W_{hh}^h x_h^2 + \underbrace{W_{kh}^h x_k x_h}_{\text{if } k \leq h} \\
&\quad + \sum_{\substack{i < k \\ i \neq h}} W_{ik}^h x_i x_k + \underbrace{W_{hk}^h x_h x_k}_{\text{if } h \leq k} + W_{kk}^h x_k^2
\end{aligned} \tag{25}$$

Comparing coefficients yields the following equations (except for the “colorful” ones):

$$\begin{aligned}
&W_{ij}^h = W_{ij}^k \quad \forall i, j, h, k. \ i, j \neq h, k; \ i \leq j \\
\boxed{\text{“Black” equations}} \quad &W_{hh}^k = W_{kk}^h \quad \forall h, k. \\
&W_{kk}^k = W_{hh}^h \quad \forall h, k.
\end{aligned} \tag{26}$$

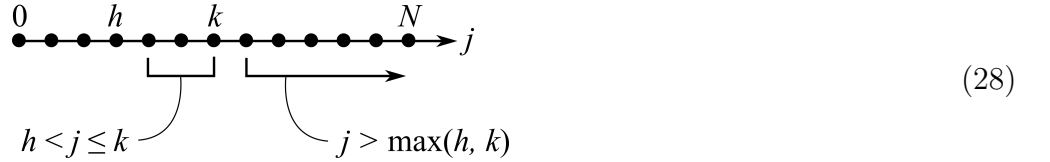
The colored terms lead to these constraints:

| | cases | $h < k :$ |
|-----------------|-----------------------|------------------------------|
| cyan = cyan | $W_{ki}^k = W_{hi}^h$ | $\forall i, h, k. i > k$ |
| olive = olive | $W_{hi}^k = W_{ki}^h$ | $\forall i, h, k. i > k$ |
| red = red | $W_{ik}^k = W_{ih}^h$ | $\forall i, h, k. i < h$ |
| blue = blue | $W_{ih}^k = W_{ik}^h$ | $\forall i, h, k. i < h$ |
| red = cyan | $W_{ik}^k = W_{hi}^h$ | $\forall i, h, k. h < i < k$ |
| olive = blue | $W_{hi}^k = W_{ik}^h$ | $\forall i, h, k. h < i < k$ |
| violet = violet | $W_{hk}^h = W_{hk}^k$ | $\forall h, k. h < k$ |
| | cases | $k < h :$ |
| cyan = cyan | $W_{ki}^k = W_{hi}^h$ | $\forall i, h, k. i > h$ |
| olive = olive | $W_{hi}^k = W_{ki}^h$ | $\forall i, h, k. i > h$ |
| red = red | $W_{ik}^k = W_{ih}^h$ | $\forall i, h, k. i < k$ |
| blue = blue | $W_{ih}^k = W_{ik}^h$ | $\forall i, h, k. i < k$ |
| cyan = red | $W_{ki}^k = W_{ih}^h$ | $\forall i, h, k. k < i < h$ |
| blue = olive | $W_{ih}^k = W_{ki}^h$ | $\forall i, h, k. k < i < h$ |
| green = green | $W_{kh}^h = W_{kh}^k$ | $\forall h, k. k < h$ |

(27)

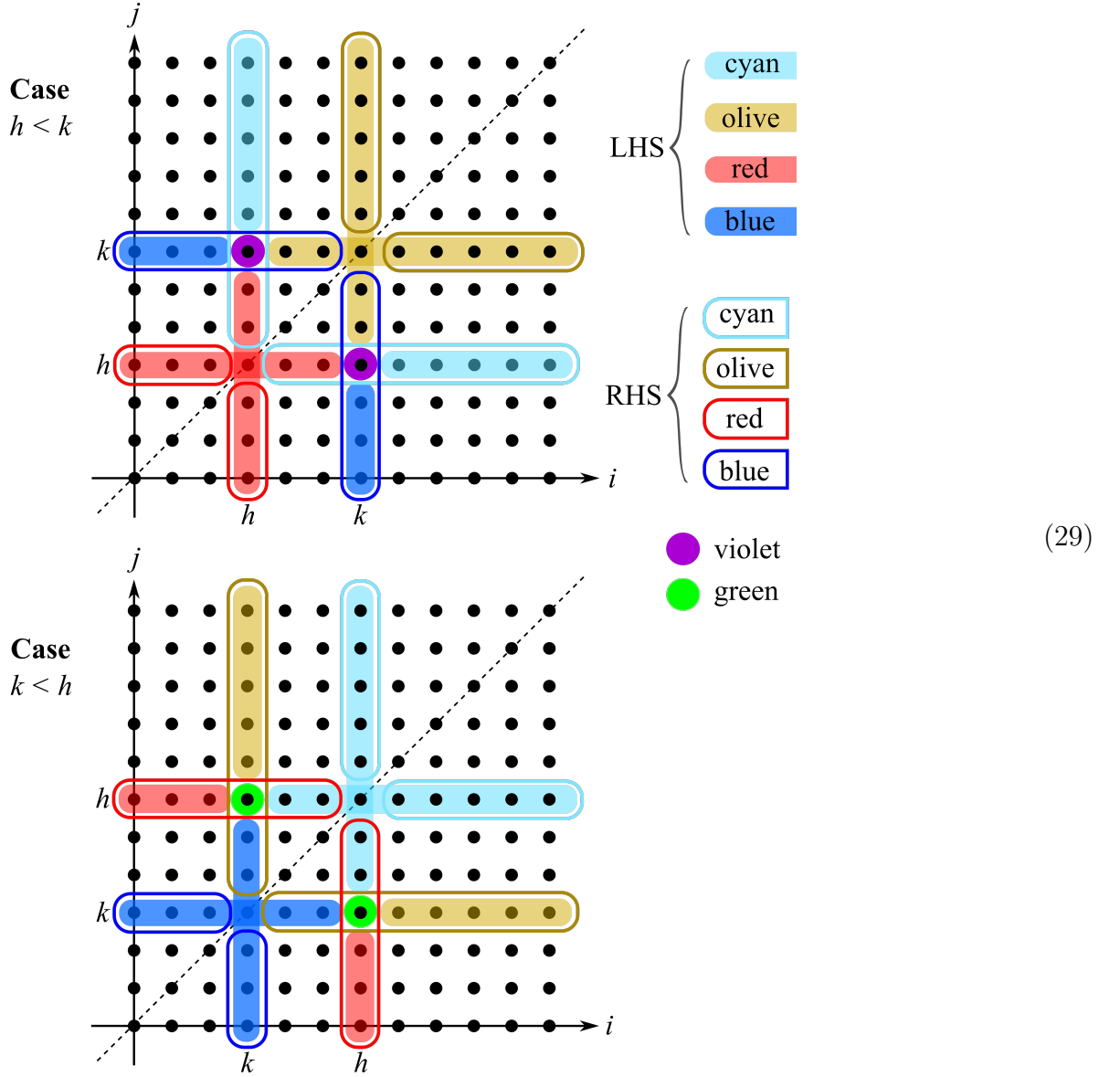
where notations such as “red = red” mean that the term of the first color on the LHS is compared to the term of the second color on the RHS.

The indexing situation can get a bit complicated. For example, the following diagram shows the situation of $k > h$ with the index j :



The following diagram shows the indexing situation for the colored terms in equation (25). The LHS and RHS

of this equation has terms with different indexings conditions:



It is not easy to see how many distinct weights are permitted by the constraints. Surprisingly, Python experiments suggest that the number of distinct weights $\equiv 4$ for all N (except for $N = 1, 2$ which has only 1 and 3 weights).

For example, the number of monomials in 3 variables and of degree ≤ 8 is:

$$\begin{aligned} & \binom{10}{8} + \binom{9}{7} + \binom{8}{6} + \dots + \binom{4}{2} + \binom{3}{1} + \binom{3}{0} \\ & = 45 + 36 + 28 + 21 + 15 + 10 + 6 + 3 + 1 = 165 \end{aligned} \quad (30)$$

2.6 Programmatic way to find invariant constraints on weights

Define a scheme to index all the weights in the multi-layer NN: $W_{ij}^{k\ell}$. Then the entire NN function can be expanded as a polynomial with coefficients from W .

For one quadratic layer, there would be a total of $n \left(n^2 - \frac{n(n-1)}{2} \right) = \frac{n^2(n+1)}{2}$ terms. The coefficients for each term would be composed out of $W_{ij}^{k\ell}$. Permuting the input would require coefficients of **like** terms to be equal. We should try all pairwise permutations of n inputs, of which there are $n(n-1)/2$.

On the second layer, the output would be composed of sums and products of polynomials with second-layer weights. Thus the new coefficients would be **polynomials** in multi-layer weights. Invariance or equivariance requires that we compare coefficients of like terms, thus yielding **equalities with polynomials** on both sides. Such conditions seems much more complex than the single-layer conditions for equivariance.

This means that the weights would be in an **algebraic variety** of reduced dimension. Our objective is to update / learn the weights **within** this variety.

2.7 Quadratic, 2-layer case

In the last section we have **equivariant** layers composed together to form a neural network. Now we relax the constraints so that the multi-layer network is free to have any weights except that the output must be **invariant**.

2.8 Back-propagation algorithm for constrained weights

作者: zighthouse

链接: <https://www.zhihu.com/question/327765164/answer/704606353>

来源: 知乎 / 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

神经网络是对一类内部结构固定的非线性函数的俗称，这类函数是输出关于输入以及隐含内部状态的函数，输出与输入呈现非线性特性。当一份输出只与一份输入有关时，常用卷积神经网络来实现。当一份输出与一个相继表达的输入序列相关时，可以用回归神经网络来实现。一般地，神经网络可以技术性地分解成神经元的复合，这里的神经元是在这个神经网络中的一种最基本的非线性函数的俗称，管理着属于它的内部状态，并基于这些内部状态在神经网络中负责着分配到它的非线性处理。每多一重基本非线性函数的复合，则多一层神经元。如果在某一重复合中出现了两类或者更多类基本非线性函数项的合并，则出现了分支。

神经网络的权值是分解到具体神经元管理的一种内部状态。用反向传播方法来更新神经网络的权值是基于这样一个基本的假设：在一个确定的输入（或者输入序列）并产生当前输出的这个点（权值构成的线性空间中的点）上，输出在这个点上是连续的。即权值点的连续微小变化会导致输出点相应的连续微小变化。这样，当我们希望调节当前权值以使此输出向特定点靠拢时，就得出基于权值空间中错误/误差/惩罚的梯度的反向传播算法。

如果想在某个神经网络中的两个权值间建立一种约束关系，这两个权值自然就不再相互独立，可以通过考查整个权值构成的线性空间，秩会变小。约束条件只要不改变连续假设，仍然可以求出带约束条件下的梯度。如果改变了连续假设，则意味着非线性分解不恰当，需要重新分解神经网络的基本结构。

Quadratic NN setup

Traditional neural network:

| | | |
|--------------------|--|------|
| <div>neuron</div> | $y = \odot \boldsymbol{w} \cdot \boldsymbol{x}$ | |
| <div>layer</div> | $y = \odot W \boldsymbol{x}$ | |
| <div>network</div> | $y = \odot W \circ \odot W \dots \boldsymbol{x}$ | (31) |

Quadratic neural network:

$$\begin{array}{ll}
\boxed{\text{neuron}} & y = \langle W\mathbf{x}, \mathbf{x} \rangle \\
\boxed{\text{layer}} & y = \langle W\mathbf{x}, \mathbf{x} \rangle \\
\boxed{\text{network}} & y = \langle W\mathbf{x}, \mathbf{x} \rangle \circ \langle W\mathbf{x}, \mathbf{x} \rangle \dots \mathbf{x}
\end{array} \tag{32}$$

Traditionally, each neuron k with output o_k is defined as:

$$o_k = \mathcal{O}(\text{net}_k) = \mathcal{O} \left(\sum_{j=1}^n w_{jk} o_j \right). \tag{33}$$

This is replaced by our new neuron:

$$\boxed{\text{next layer}} \quad o_k = \text{net}_k = \sum_j \sum_i W_{ij}^k o_i o_j \quad \boxed{\text{current layer}} \tag{34}$$

where the 2 summations can be **interchanged**.

Classical back-prop algorithm

What follows is just a re-working of traditional back-propagation. For ease of comparison, let's recall the classic derivation of the back-prop algorithm (from Wikipedia):

The weight update rule is:

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} = -\eta o_i \delta_j \tag{35}$$

where δ_j is the so-called “local gradient”. o_i is the i -th output, η is the learning rate.

The gradient is:

$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial W_{ij}} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial W_{ij}}. \tag{36}$$

In the last factor on the RHS, only one term in net_j depends on W_{ij} , so:

$$\frac{\partial \text{net}_j}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} \left(\sum_k W_{kj} o_k \right) = \frac{\partial}{\partial W_{ij}} W_{ij} o_i = o_i. \tag{37}$$

The middle factor on the RHS is the partial derivative of the activation function:

$$\frac{\partial o_j}{\partial \text{net}_j} = \frac{\partial \mathcal{O}(\text{net}_j)}{\partial \text{net}_j}. \tag{38}$$

The first factor is easy to evaluate if the neuron is in the output layer:

$$\frac{\partial E}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} (o^* - o_j)^2 = o_j - o^* = \epsilon \tag{39}$$

where ϵ is the error.

Otherwise, assume \mathcal{C} is the set of all child neurons **receiving** input from neuron j , one gets a **recursive** expression for the derivative:

$$\frac{\partial E}{\partial o_j} = \sum_{c \in \mathcal{C}} \left(\frac{\partial E}{\partial \text{net}_c} \frac{\partial \text{net}_c}{\partial o_j} \right) = \sum_{c \in \mathcal{C}} \left(\frac{\partial E}{\partial o_c} \frac{\partial o_c}{\partial \text{net}_c} \frac{\partial \text{net}_c}{\partial o_j} \right) = \sum_{c \in \mathcal{C}} \left(\frac{\partial E}{\partial o_c} \frac{\partial o_c}{\partial \text{net}_c} W_{jc} \right) \quad (40)$$

So we get the classic back-prop algorithm :

$$\begin{aligned} \frac{\partial E}{\partial W_{ij}} &= \delta_j \cdot o_i \\ \delta_j &= \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial \text{net}_j} = \begin{cases} \frac{\partial E}{\partial o_j} \cdot \frac{\partial \mathcal{O}(\text{net}_j)}{\partial \text{net}_j} & \text{if } j = \text{output neuron} \\ \sum_c W_{jc} \cdot \delta_c \cdot \frac{\partial \mathcal{O}(\text{net}_j)}{\partial \text{net}_j} & \text{if } j = \text{inner neuron} \end{cases} \end{aligned} \quad (41)$$

Quadratic back-prop algorithm

Applying the chain rule as in (36):

$$\frac{\partial E}{\partial W_{ij}^k} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial W_{ij}^k}. \quad (42)$$

Notice that we don't use an activation function, so the olive factor disappears.

For the RHS's second factor, only one term in the sum depends on W_{ij}^k , so:

$$\frac{\partial o_k}{\partial W_{ij}^k} = \frac{\partial}{\partial W_{ij}^k} \left(\sum_{i'} \sum_{j'} W_{i'j'}^k o_{i'} o_{j'} \right) = \frac{\partial}{\partial W_{ij}^k} W_{ij}^k o_i o_j = o_i o_j. \quad (43)$$

If k is an inner neuron, let $\mathcal{C} = \{u, v, \dots, w\}$ be the **next layer** of neurons receiving input from neuron k . Consider E as a function with the inputs being all neurons in \mathcal{C} :

$$\begin{aligned} \frac{\partial E(o_k)}{\partial o_k} &= \frac{\partial E(o_u, o_v, \dots, o_w)}{\partial o_k} \\ \boxed{\text{next layer}} \quad o_c &= \sum_j \sum_i W_{ij}^c o_i o_j \quad \boxed{\text{current layer}}. \end{aligned} \quad (44)$$

Then the **recursive** expression for the derivative is obtained:

$$\frac{\partial E}{\partial o_k} = \sum_{c \in \mathcal{C}} \left(\frac{\partial E}{\partial o_c} \frac{\partial o_c}{\partial o_k} \right) = \sum_{c \in \mathcal{C}} \left(\frac{\partial E}{\partial o_c} \sum_j W_{kj}^c o_j \right) \quad (45)$$

Substituting, we obtain the algorithm:

$$\begin{aligned} \boxed{\text{quadratic back-prop}} \quad \frac{\partial E}{\partial W_{ij}^k} &= \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial W_{ij}^k} = \frac{\partial E}{\partial o_k} o_i o_j := \delta_k \cdot o_i o_j \\ \delta_k &:= \sum_{c \in \mathcal{C}} \left(\delta_c \sum_j W_{kj}^c o_j \right) \end{aligned} \quad (46)$$

Shared weights

For the single-layer (stacked) case, it is good to know that all the weight-sharing occurs **within** each layer, never across layers.

The coefficient of $x_i x_j$ is W_{ij} , for the y_k component. For each weight W_{ij}^k we need to calculate the gradient $\frac{\partial E}{\partial W_{ij}^k}$, but the weights are in equivalence classes.

Say if the following 2 weights are shared:

$$W_0 := W_{ij} \equiv W_{i'j'} \quad (47)$$

Then the network:

$$y = \sum \sum W_{ij} x_i x_j \quad (48)$$

would contain the shared components:

$$W_0(x_i x_j + x_{i'} x_{j'}) + \text{other terms} \dots \quad (49)$$

Equality constraints

For simple **equality** of weights, the weights should be collected together. (43) should simply be $\sum o_i o_j$ for the equivated weights.

Additive constraints

The most tricky part is the “additive” constraint:

$$W_{hk}^h + W_{kh}^h = W_{hk}^k + W_{kh}^k \quad \forall h, k. \quad (50)$$

This is just like having 4 “not quite independent” variables x, y, u, v satisfying:

$$x + y = u + v \quad (51)$$

and asking what is

$$\frac{\partial(x + y)}{\partial x} ? \quad (52)$$

And the solution is to make one of the variables **depend** on the other 3.

For each layer, iterate over every neuron representative, which has a collection of coefficients.

We need to consider equations (43) and (45), but (45) is unaffected by weight-sharing.

In (43), for each equivalence class,

3 Free Abelian group method

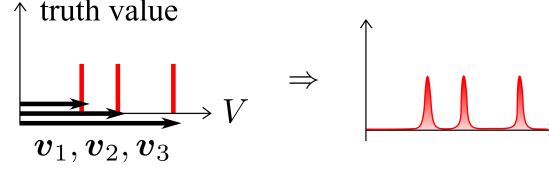
3.1 Representation theory

Representation theory seems to be of no use in this situation. A theorem says that any finite-dimensional irreducible representation (over \mathbb{C}) of an Abelian group G is always 1-dimensional. So G has exactly $|G|$ complex representations [Qiu2011].

A recent book talks about metric embeddings into Banach space [Ostrovskii2013], but the space appears to be infinite-dimensional. Not sure if it is useful for our purpose.

4 Fourier transform method

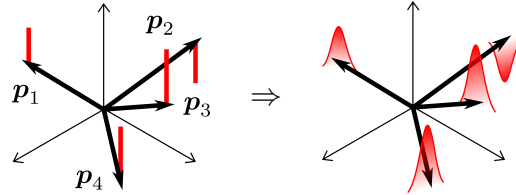
Now consider another simple idea. Suppose we have 3 vectors $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$ in a 1-dimensional vector space V , and we attach a **truth value** $\top = 1$ to each vector. Then we try to approximate the **graph** of truth values $\top(\mathbf{v})$ as a “wave” using Fourier or wavelet transform:


(53)

The resulting representation has some nice properties:

- If we permute $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$, the graph (and thus its spectrum) remains the same, *ie*, the representation is invariant under permutations
- We can add more vectors to the graph without changing the size of the spectral representation, *ie*, it is relatively insensitive to the number of vectors

We can extend this idea to the **multi-dimensional** case where the literal proposition vector $\mathbf{p} \in \mathbb{P} = \mathbb{R}^{3d}$ * and the state \mathbf{x} consists of k vectors $= \mathbf{p}_1 \wedge \mathbf{p}_2 \wedge \dots \mathbf{p}_k$. In other words, we need to apply Fourier transform to a wave over $3d$ dimensions. Moreover, we can have truth values in the range $[-1, 1]$, which can be construed as **fuzzy** truth values, or in the range $[0, 1]$, regarded as the probability of **stochastic actions** (as is common in policy-gradient methods).


(54)

Notice that our NN would *input* from the spectral representation and *output* the normal representation.

With this approach, the geometric shapes of (??) and (??) would be distorted in very complicated ways. It remains to be verified whether NNs can learn this task effectively.

5 PointNet’s $g(h(x_1), \dots, h(x_n))$ method

6 Conclusion

This is unrelated to the paper’s topic, but I would digress a bit into the area of AGI.

It seems that permutation invariance of neural networks is very difficult to achieve no matter which method we adopt. So perhaps an alternative is to abandon the invariance entirely.

*For example, a typical d from Word2Vec or GloVe is 200, so $3d = 600$.

The idea is to use an attention mechanism to select a particular conjunction, to which the rule-applier performs the forward deduction step. In this case, the conjunction can be a **sorted** concatenation of proposition vectors. But this has the problem that the matcher and the rule-applier seems to be learning the same structures.

Assume that we have a conjunction that is ready for rule-application. The possible patterns of deduction may be limited, and may be enumerable. But we want to relax the “rigid quantifications”.