Understanding and Collapsing Symmetries in Neural Network Parameter Spaces

Understanding and collapsing symmetries in neural network parameter spaces

With an application to weight averaging

Hikari Sorensen

A thesis submitted in partial fulfillment
of the requirements for the joint degree
of Bachelor of Arts in
Computer Science and Mathematics

Departments of Computer Science and Mathematics
Harvard College
Cambridge, MA
April 10, 2020

Abstract

It has been mentioned numerous times in the deep learning research field that neural network parameter spaces contain many redundancies. However, there seems to be little work that addresses specifically whence redundancy arises, and those papers that do consider redundant parameterizations by and large address the matter from a statistical perspective, in terms of the frequency at which local optima sampled from the loss surface seem to have identical or near-identical loss values.

I here consider the redundancy in neural network parameter spaces from a combinatorial perspective as a matter of symmetries between permutations of nodes in layers of neural networks. Moreover, I present a way to identify networks that are symmetric in this way by establishing a notion of a "universal basis" with respect to which networks can be uniquely expressed. This further becomes of great interest when considering weight averaging.

Contents

1 Introduction

I will in this paper assume a standard feedforward neural network architecture, however the methods and results I present can be generalized to networks of other forms, including convolutional neural networks (CNNs).

1.1 Definitions and notation

Although the term "neural network" is commonly used to refer to both the class of functions specified by a neural network architecture and any particular member of the class specified by a parameterization, in this paper it is important that we distinguish these two notions.

Let us define a *neural network architecture* of depth $d \in \mathbb{N}$, $m \in \mathbb{N}$ inputs and $k \in \mathbb{N}$ outputs to be a tuple

$$A = (n_0, n_1, ..., n_d) \in \mathbb{N}^{d+1}, \quad N_0 = m, \ N_d = k$$

where each $N_i, i \in \{0, ..., d\}$ specifies the number of nodes in layer i. Let

$$\mathcal{A}_d = \{(n_0, n_1, ..., n_d) \in \mathbb{N}^{d+1}\}$$

be the set of all neural network architectures of depth d. We can define the *space of parameterizations*, or *parameter space*, of a neural network with architecture A as

$$\Theta_A = \prod_{\ell=1}^{d+1} (\mathbb{R}^{n_{\ell-1} \times n_\ell}, \mathbb{R}^{n_\ell}) \ \in \ \Theta = \bigcup_{A \in \mathcal{A}_d} \Theta_A$$

where Θ is the collection of all such spaces corresponding to architectures in $\mathcal{A}_d$. A *neural network parameterization* is then a sequence of pairs

$$\theta = ((W_1, b_1,), ..., (W_d, b_d)) \in \Theta_A.$$

Each θ then specifies a function

$$f_\theta : X \to Y = H_d \circ \sigma_d \circ H_{d-1} \circ ... \circ H_1 \circ \sigma_1 \circ H_0,$$

where $X = \mathbb{R}^m$ and $Y = \mathbb{R}^k$ are the input and output spaces, respectively; each $\sigma_i, i \in \{1, ..., d\}$ is a (generally, non-linear) function applied elementwise; and we define the linear map

$$H_\ell : \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_\ell}, \qquad H_\ell(x) = xW_\ell + b_\ell \tag{1}$$

for each layer $\ell \in \{0, ..., d\}$ in terms of the *weights* W_ℓ and *biases* b_ℓ corresponding to layer ℓ specified by the parameterization θ. The functions σ_i, known as *activation functions*, are widely treated as implicit structure independent of the parameterization θ. Since θ then uniquely specifies the function f_θ, there is a *realization* map [Berner et al., 2019]

$$\begin{aligned} \mathcal{R} : \Theta &\to C(\mathbb{R}^m, \mathbb{R}^k) \\ \theta &\mapsto f_\theta. \end{aligned}$$

Note that $\mathcal{R}$ is not injective. [Berner et al., 2019] present as a counterexample the parameterizations

$$\theta_1 = ((W_1, b_1), (W_2, b_2), ..., (0, 0)) \qquad \text{and} \qquad \theta_2 = ((V_1, c_1), (V_2, c_2), ..., (0, 0)),$$

for which $\mathcal{R}(\theta_1) = \mathcal{R}(\theta_2)$ regardless of the values of $W_\ell, b_\ell, V_\ell, c_\ell$. Here, $\mathcal{R}(\theta_1)$ and $\mathcal{R}(\theta_2)$ are both equivalent to the 0 map - whose equivalence class we can denote [0].

2 Permutations and node symmetries

A more subtle form of equivalence class of neural network parameterizations, and that which will be the main focus of this paper, comes as a consequence of the following essentially trivial fact about matrix multiplication and permutation matrices:

Fact 2.0.1. Given matrices A, B such that the matrix product $A \cdot B$ is well-defined, then we can write $A \cdot B$ as

$$A \cdot B = (A\pi^{-1}) \cdot (\pi B)$$

where π is a permutation matrix.

Now, as trivial as this fact is when stated mathematically, it is of non-trivial consequence when realized in the context of neural network parameterizations. The important implication of Fact 2.0.1 is that, in particular, if A is an $m_1 \times m_2$ matrix, and B an $m_2 \times m_3$ matrix, then there are $m_2! = |S_{m_2}|$ (where S_{m_2} is the symmetric group on $\{1, ..., m_2\}$) equivalent ways of writing the product $A \cdot B$ by permuting rows of A and columns of B.

As somewhat of an aside, note that it is precisely because we distinguish between neural networks as, on the one hand, objects with some parameterization, and on the other hand, functions that correspond to those parameterized objects, that the triviality above becomes meaningful. That is, it is because we work with and manipulate neural networks at the level of their *representation*[1] rather than in terms of their realizations as functions more generally that we can interpret the statement

$$A \cdot B = (A\pi^{-1}) \cdot (\pi B)$$

as an equivalence of representations rather than as a mathematical equality.

We can realize this equivalence in graphical form by considering the standard representation of a neural network architecture as a directed acyclic graph (DAG) (see Figure 1).

In this representation, the node $h_{\ell,j}$ (the j-th node of layer ℓ, $\ell \in \{1, ..., d-1\}$) represents the value

$$h_{\ell,j} = \sum_{\alpha=1}^{n_{\ell-1}} h_{\ell-1,\alpha} W_{\ell_{\alpha,j}} + b_{\ell_j} = H_\ell(h_{\ell-1}) \tag{2}$$

where $h_{\ell-1} = (h_{\ell-1,1}, h_{\ell-1,2}, ..., h_{\ell-1,n_{\ell-1}})$ and H_ℓ is as in Equation 1. That is, a node $h_{\ell,j}$ in a hidden layer is the linear combination (plus the constant "bias" term b_{ℓ_j}) of the values of all the nodes $h_{\ell-1,\alpha}$ for $\alpha \in \{1, ..., n_{\ell-1}\}$ in the previous layer, weighted by the edges from $h_{\ell-1,\alpha}$ to $h_{\ell,j}$ for each α. Now, given this "fully connected"[2] structure, in which each hidden layer node has an incoming edge from each node in the previous layer, and also an outgoing edge to each node in the next layer, we can imagine the following graph isomorphism: in any fixed layer, permute the nodes within each layer, preserving all edges (that is, edges move with their endpoints, rather than staying in a rigid layout with only nodes being shuffled). Since shuffling the nodes only rearranges the terms in any linear combination of those nodes, permuting nodes within a layer has no effect on the values of the nodes in any following layer.

Thus we can define the analogous map to $\mathcal{R}$ for graphs

$$\mathcal{R}_G : G_\Theta \to C(\mathbb{R}^m, \mathbb{R}^k)$$
$$G_\theta \mapsto f_\theta$$

where G_Θ is the set of all graphs corresponding to neural network parameterizations in Θ. Now if

$$G_\theta(h_1, h_2, ..., h_{d-1})$$

[1]Here, "representation" is used in the common sense, rather than in the sense of representation theory - although the notion underlying the technical term is not entirely dissimilar to that of the present usage.

[2]We can assume fully-connectedness fairly generally, since missing edges can simply be thought of as edges with weight 0.

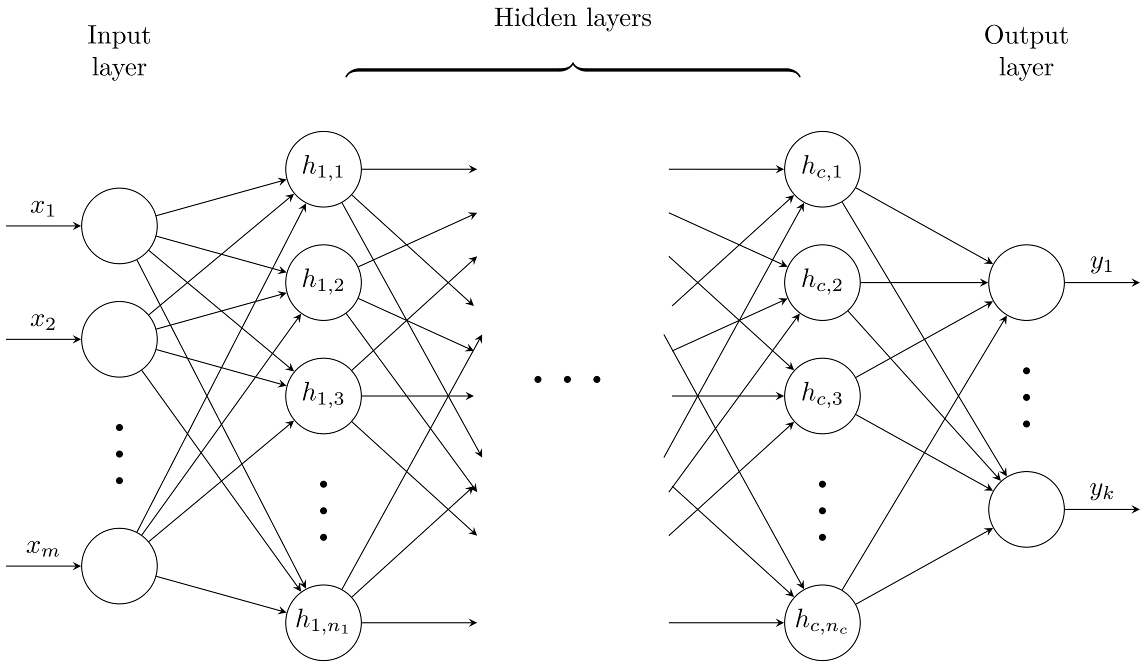

Figure 1: A fully-connected feedforward neural network with m inputs, k outputs, and $c = d - 1$ (notated this way to keep the node labels in the figure tidy) hidden layers, in which each layer $i \in \{1, ..., d-1\}$ contains n_i nodes. Layers 0 and d are known as the input and output layers, respectively, while layers $1, ..., d-1$ are known as hidden layers.

is the graph corresponding to the neural network parameterization θ such that $h_1 = (h_{1,1}, ..., h_{1,n_1}), ..., h_{d-1} = (h_{d-1,1}, ..., h_{d-1,n_{d-1}})$ are the arrangements of hidden layer nodes, and

$$G_\theta(\pi_1(h_1), \pi_2(h_2), ..., \pi_{d-1}(h_{d-1}))$$

is the same graph as $G_\theta(h_1, h_2, ..., h_{d-1})$, except with the nodes of hidden layer ℓ permuted according to the permutation π_ℓ (on n_ℓ elements), then

$$\mathcal{R}_G(G_\theta(h_1, h_2, ..., h_{d-1})) = \mathcal{R}_G(G_\theta(\pi_1(h_1), \pi_2(h_2), ..., \pi_d(h_{d-1}))). \tag{3}$$

Note that if nodes within layers of $G_\theta(h_1, h_2, ..., h_{d-1})$ are rearranged, then this corresponds to a rearrangement of rows and columns of the matrices $((W_1, b_1,), ..., (W_d, b_d))$ that are specified by the parameterization θ. This is then a different parameterization $\theta' = ((W'_1, b'_1,), ..., (W'_d, b'_d))$, such that W'_ℓ, b'_ℓ for $\ell \in \{1, ..., d\}$ are some permuted form of W_ℓ, b_ℓ. That is, for any $\theta \in \Theta$ there exists some $\theta' \in \Theta$ such that

$$G_\theta(\pi_1(h_1), \pi_2(h_2), ..., \pi_{d-1}(h_{d-1})) = G_{\theta'}(h_1, h_2, ..., h_{d-1}),$$

which suggests the following definition: let $\pi = (\pi_1, ..., \pi_{d-1}) \in \prod_{\ell=1}^{d-1} S_{n_\ell}$ and P be the map

$$P : \Theta \times \prod_{\ell=1}^{d-1} S_{n_\ell} \to \Theta$$
$$\theta, \pi \mapsto \theta'$$

that sends a neural network parameterization θ and a sequence specifying a permutation of nodes at each hidden layer of the graph $G_\theta(h_1, h_2, ..., h_{d-1})$ to the parameterization θ' whose matrices are permutations of those of θ in such a way that the graph $G_{\theta'}(h_1, h_2, ..., h_{d-1})$ realizes the permutations π on layers of G_θ. As a slight abuse of notation, but one that makes for a cleaner and more intuitive symbolism, if $\pi = (\pi_1, ..., \pi_{d-1})$ we can write

$$\pi(\theta) = P(\pi, \theta) = \theta', \tag{4}$$

so that

$$G_{\pi(\theta)}(h_1, h_2, ..., h_{d-1}) = G_\theta(\pi_1(h_1), \pi_2(h_2), ..., \pi_{d-1}(h_{d-1})).$$

These permutations on nodes can be realized in terms of θ as a sequence of matrices as follows: if we think of each π_ℓ, $\ell \in \{1, ..., d-1\}$ as a permutation matrix (so that left multiplication by π_ℓ permutes rows accordingly), then using the notation of Equation 4, we have

$$\begin{aligned}\pi(\theta) &= \pi((W_1, b_1,), ..., (W_d, b_d)) \\ &= \big((W_1\pi_1^{-1}, b_1\pi_1^{-1}), (\pi_1 W_2 \pi_2^{-1}, \pi_1 b_2 \pi_2^{-1}), ..., (\pi_{d-1}W_d, \pi_{d-1}b_d)\big) \\ &= \prod_{\ell=1}^{d} (\pi_{\ell-1}W_\ell\pi_\ell^{-1}, \pi_{\ell-1}b_\ell\pi_\ell^{-1}),\end{aligned} \tag{5}$$

where $\pi_0 = \pi_d = 1$ (the identity matrix).

We can now state the following

Lemma 2.0.2. Let $A = (n_0, n_1, ..., n_d)$ be a neural network architecture; and let θ be a parameterization and $[\theta]$ its equivalence class of realizations. Then

$$|[\theta]| \geq \prod_{\ell=1}^{d-1} n_\ell! \tag{6}$$

Proof. Each layer ℓ containing n_ℓ nodes has $n_\ell!$ distinct permutations, each of which corresponds to a distinct parameterization. Thus the total number of distinct parameterizations achievable by permuting hidden layer nodes is the product of $n_\ell!$ across all hidden layers $\ell \in \{1, ..., d-1\}$. By Equation 3, each of these parameterizations map to the same realization $\mathcal{R}(\theta) = f_\theta \in C(\mathbb{R}^{n_0}, \mathbb{R}^{n_d})$, thus permutations of θ alone account for $\prod_{\ell=1}^{d-1} n_\ell!$ equivalent parameterizations.

There may be many more equivalent parameterizations than permutations, however, since matrix products can in general factor in numerous different ways, yielding different sequences of weights and biases across layers that would map to the same realization. □

This directly implies the following more general

Theorem 2.0.3. Let $A = (n_0, n_1, ..., n_d)$ be a neural network architecture. Then (a point in) its parameter space has $\prod_{\ell=1}^{d-1} n_\ell!$ symmetries.

This also implies the following corollary, which is of more obvious relevance in deep learning.

Corollary 2.0.4. Let $A = (n_0, n_1, ..., n_d)$ be a neural network architecture. Then any optimum of a loss function over the corresponding parameter space has $\prod_{\ell=1}^{d-1} n_\ell!$ redundancies - that is, distinct optima that do not correspond to meaningfully distinct parameterizations - in the loss surface.

Proof. Because loss functions are defined over neural network predictions, they are functions of realizations, rather than of parameterizations. Thus if θ^* is a local optimum in the loss surface, any parameterization $\pi(\theta^*) \in [\theta^*]$ in its equivalence class is also a local optimum with identical loss value. Moreover, all such equivalent local optima share identical local neighborhoods - that is, not only is the value of the loss the same at every point in $[\theta^*]$, but also the entire loss surface near any point in $[\theta^*]$ is identical to that at any other point in $[\theta^*]$.

These optima are "redundant" because they occur at distinct points in the parameter space, but do not actually correspond to different parameterizations that are not simply permutations of one another. □

The practical implication of Corollary 2.0.4 is that training multiple neural networks with different random initializations may not result in finding truly different (that is, non-symmetric) optima, even if these optima as points are far apart in the loss surface. The probability that a collection of randomly chosen local optima are symmetries of one another is highly problem-specific, as it is dictated by the loss function over a particular parameter space and corresponding to a particular dataset. However, the notion that the number of symmetries in the loss surface increases quite dramatically with network size may shed some light on why, in any given learning task, increasing the size of a network beyond some critical threshold does little to improve predictive performance - if network size exceeds some "natural" point at which the capacity of the network in some sense matches the complexity of the learning problem, then additional network layers or nodes within layers may only increase the number of symmetries in parameter space and the loss surface over it, without substantially increasing the number of truly different optimal parameterizations.

3 Collapsing symmetries

That symmetry exists among neural network parameterizations will come as no surprise to deep learning researchers. Indeed, redundancy in network parameterization was addressed even at the outset of research into loss surfaces in the seminal paper of [Choromanska et al., 2014] (although they addressed it from a statistical perspective), which has since become an important foundation for the study of loss surfaces in deep learning. However, although this redundancy has been acknowledged, little work has been done to systematically identify symmetric parameterizations.

What we would like is to be able to say that two parameterizations θ and θ' are equivalent if they differ by permutation. But how are we to check this? We certainly cannot simply generate all permutations of θ and check if θ' is equal to one of them - this would be outrageously computationally inefficient, but also, even if we were able to find permutations, because the weights are real-valued, it is overwhelmingly unlikely that θ' would be exactly equal any permutation of θ, even if they did in practice correspond to symmetric optima in the loss surface. This is simply because if θ and θ' are found independently in training two differently-initialized networks, then even if they did indeed correspond to equivalent parameterizations, the training methods themselves are unlikely to have sufficiently high precision find points that are exact permutations of one another.

Now, this latter issue of precision can be reduced somewhat by relaxing the check that

$$\theta' = \pi(\theta), \qquad \text{for some permutation } \pi$$

to

$$\theta' \approx \pi(\theta),$$

or equivalently,

$$|\theta' - \pi(\theta)| < \epsilon$$

for some permutation π and some $\epsilon > 0$. This still, however, does not address the computational issue of finding which permutation π might give $\pi(\theta) = \theta'$.

Part of the difficulty of making these identifications is that there is no *a priori* notion of a canonical choice of representative for the equivalence class of parameterizations under permutation. That is, we would like to be able to say that two parameterizations θ and θ' are equivalent if they differ by permutation - but clearly we can only determine this if we can write them in the same order of permutation. More generally, the only way to compare any two parameterizations that may or may not be permutations of one another is to have some fixed, "canonical" permutation according to which we write all parameterizations. Without such a choice of "universal basis", so to speak, we have no way to compare any two parameterizations.

3.1 Sorted order as a canonical permutation

Fortunately, there is indeed a natural choice of permutation over numbers, in terms of which we can compare parameterizations: the ordered permutation. Recalling the graph representations

$$G_\theta(h_1, h_2, ..., h_{d-1}), \quad G_{\pi(\theta)}(h_1, h_2, ..., h_{d-1}) = G_\theta(\pi_1(h_1), \pi_2(h_2), ..., \pi_{d-1}(h_{d-1}))$$

of parameterizations θ and $\pi(\theta)$, note that sorting the nodes of each layer in each graph results in two identical graphs. That is, if $\tau_1, ..., \tau_{d-1}$ are permutations on layers $\ell \in \{1, ..., d-1\}$ such that $\tau_\ell(h_\ell)$ is an ordered sequence of nodes in layer ℓ (for some notion of order on nodes), and $\sigma_1, ..., \sigma_{d-1}$ are permutations on layers $\ell \in \{1, ..., d-1\}$ such that $\sigma_\ell(\pi_\ell(h_\ell))$ is an ordered sequence of nodes in layer ℓ, then

$$G_\theta(\tau_1(h_1), \tau_2(h_2), ..., \tau_{d-1}(h_{d-1})) = G_\theta(\sigma_1(\pi_1(h_1)), \sigma_2(\pi_2(h_2)), ..., \sigma_{d-1}(\pi_{d-1}(h_{d-1}))).$$

Thus if we could implement a map

$$\text{NodeSort} : \Theta \to \Theta \tag{7}$$

that maps an input θ to some permutation of its values that corresponds to ordering the nodes in each layer, then

$$\text{NodeSort}(\theta) = \text{NodeSort}(\pi(\theta))$$

for any permutation $\pi(\theta)$ of θ.

3.2 An order on nodes and a sorting algorithm

Of course, implementing an algorithm to sort nodes would be trivial to do if nodes corresponded to real values. Unfortunately, not only are nodes in a neural network not real-valued, but in fact they do not have, as such, values at all. As stated in Equation 2, the j-th node $h_{\ell,j}$ of layer ℓ represents a linear combination of the outputs of the nodes in layer $\ell - 1$, with coefficients given by the j-th column of matrix W_ℓ. That is, the node $h_{\ell,j}$ is a functional

$$\begin{aligned} h_{\ell,j} : \mathbb{R}^{n_{\ell-1}} &\to \mathbb{R} \\ (x_1, ..., x_{n_{\ell-1}}) &\mapsto \sum_{\alpha=1}^{n_{\ell-1}} x_\alpha W_{\ell,(\alpha,j)} + b_{\ell,j}. \end{aligned}$$

In practice, "sorting nodes in layer ℓ" corresponds to sorting rows in the matrix W_ℓ whose entries are coefficients of functionals. We can define an order as follows: given an $n \times m$ matrix M and two rows M_i, M_j, let

$$M_i > M_j \iff \langle M_i, P\rangle > \langle M_j, P\rangle, \qquad \text{where } P = (p^0, p^1, ..., p^m),\ p \in \mathbb{R}, p > 1. \tag{8}$$

For example, if M_i and M_j have digit entries in $\{0, 1, ..., 9\}$, and $p = 10$, then $\langle M_i, P\rangle$ and $\langle M_j, P\rangle$ write the entries in M_i and M_j in reverse order, then interprets the resulting string as an integer. More generally, this method is simply an efficient way to implement lexicographial ordering of vectors (that is, first sorting by the first coordinate, then sorting by the second, etc.) that attaches a value to each node in a layer. Thus this order does uniquely and correctly sort rows of a matrix.

Given $\theta = ((W_1, b_1,), ..., (W_d, b_d))$, we can implement the NodeSort map from Equation 7 as follows, where we seek an output of the form as in Equation 5:

Algorithm 1: NodeSort

```
Input: $\theta = ((W_1, b_1,), ..., (W_d, b_d))$
// iterate backward through layers
for layer $\ell \in \{d-1, ..., 1\}$ do
    $P = (p^0, p^1, ..., p^{n_\ell})$; // for some $p > 1$
    $\pi_\ell = \text{argsort}(P \cdot W_{\ell+1})^{-1}$;
end
return $\big((W_1\pi_1^{-1}, b_1\pi_1^{-1}), (\pi_1 W_2 \pi_2^{-1}, \pi_1 b_2 \pi_2^{-1}), ..., (\pi_{d-1}W_d, \pi_{d-1}b_d)\big)$
```

An implementation in Python is as follows:

Listing 1: NodeSort

```python
import numpy as np

def nodeSort(theta):

  row_orders = [0]*len(theta) # initialize empty list of length the number of layers

  ordered_theta = [0]*len(theta)

  for i, layer in reversed(list(enumerate(theta))): # assume layer = [weights, bias]
    weights, bias = layer
    num_cols = weights.shape[1]
    p = np.power([base] * num_cols, range(num_cols), dtype='float128')
    row_order = np.argsort(weights @ p) # '@' denotes the matrix product
    row_orders[i] = row_order

    if i == len(theta)-1:
      ordered_theta[i] = [weights[row_order[i]], bias]
    elif i == 0:
      ordered_theta[i] = [weights[:,row_order[i-1]], bias[row_order[i-1]]]
    else:
      ordered_theta[i] = [weights[row_order[i]:,row_order[i-1]], bias[row_order[i-1]]]

  return(ordered_theta)
```

Note that in order to maintain the correct mapping of weights from layer to layer while we permute nodes, the columns of $W_{\ell-1}$ must be permuted in the same order as are the rows of W_ℓ. This corresponds to the terms of the form $\pi_{\ell-1}W_\ell\pi_\ell^{-1}$ of Equation 5.

Now, when W_ℓ is a matrix that is part of a neural network parameterization, its entries are (in general) < 1 - thus we can use as a base for P any $p > 1$ and write rows of W_ℓ in "base p". Practically speaking, it is crucial that we can make p as close to 1 as possible, while still being greater than 1, since powers of p for p even as large as 2 quickly exceed the number limits in computers. Still, for very large networks, it is possible that a choice of p that is sufficiently large that it is compatible with the precision of the entries in W_ℓ and such that an inner product $\langle W_{\ell,i}, P\rangle$ of a row of W_ℓ with the resulting P is free of roundoff errors due to memory limits in computers, will inevitably cause overflow when taken to such high powers.

4 Experiments

Experiments were done using simple feedforward neural networks on MNIST data, in TensorFlow. The network architecture used was of the form

$$A = (784, 512, 10)$$

Distances between parameterizations				
	W_1		W_2	
	Unsorted	Sorted	Unsorted	Sorted
Min	82.12292	81.64965	15.79031	14.64832
Max	83.69969	83.78498	16.77915	15.66162
Mean	82.89585	82.65826	16.28864	15.16187

Table 1: Distances between weight matrices W_1, W_2, of size 784×512 and 512×10, respectively, taken as points in $\mathbb{R}^n$, before and after sorting nodes.

with input size 784, output size 10, and a single hidden layer of 512 nodes, for a total of 407050 trainable parameters of the form

$$((W_1, b_1), (W_2, b_2)), \qquad W_1 \in \mathbb{R}^{784\times 512}, b_1 \in \mathbb{R}^{512}, W_2 \in \mathbb{R}^{512\times 10}, b_2 \in \mathbb{R}^{10}.$$

Among 1000 trained networks, none were found to have distance 0 after sorting nodes. However, unsurprisingly, distance between all networks in general decreased, as shown in Table 1. That the decrease in distance was not particularly substantial, however, reflects both the sparsity of the space and the relative uniformity in scale across the weights.

What the small experiment above shows is that even with a network as relatively small as that used on MNIST, the parameter space grows in dimension so rapidly that even after training thousands of networks, it is unlikely that any will be redundant parameterizations of one another. Indeed, this seems consistent with a basic intuition that a selection of size on the order of 10^3 of random points in a space of dimension on the order of 10^5, even when constraining the choice of values in each coordinate to $[-1, 1]$ (which in practice is generally the case), are still sufficiently distant that their images in the same symmetry region are not substantially closer to one another than the original points are.

In practice, this may be an indication that, when working with sufficiently large networks, we needn't worry too much about redundancy when training many networks with different initializations.

5 An application to averaging

Even if not particularly useful in finding permutations among many different neural network parameterizations, the ability to sort nodes and write network parameterizations in a "universal basis" corresponding to a chosen permutation can be useful in applications that require for well-definedness a choice of representative of an equivalence class of parameterizations, and moreover that require that points all lie in the same symmetry region.

It is a well-established fact that the performance of neural networks (and more generally essentially any "supervised learning" model) improves when *ensembled* - that is, when multiple models are trained independently, and then their predictions averaged to produce an "ensemble" result. Predictions of this ensemble nature have long been known to be, in general, more robust to noise and better at generalization than are predictions of any single model. More recently, as neural network optimization as finding certain points in loss surfaces has become better understood, interest has piqued in the notion of producing averages of optimal points in *parameter space*, rather than in prediction space (also known as *model space*), as is done in ensembles. The resulting network, with weights given by the average of points in the parameter space, would then be used to make a prediction. [Izmailov et al., 2018] introduced such a method of averaging weights, which they called Stochastic Weight Averaging (SWA), and which they showed to improve the performance of neural networks beyond those that could be trained using current standard SGD (stochastic gradient descent) optimization methods. Since then, SWA has been implemented as a part of the machine learning framework PyTorch.

SWA, however, does not provide a general notion of the average of two points in parameter space: it works only *locally*, on points very close in parameter space. To average two points $\theta = ((W_1, b_1,), ..., (W_d, b_d))$ and

$\theta' = ((W_1', b_1',), ..., (W_d', b_d'))$, they simply take the elementwise mean

$$\begin{aligned}\frac{\theta+\theta'}{2} &= \prod_{\ell=1}^{d}\left(\frac{1}{2}(W_\ell + W_\ell'), \frac{1}{2}(b_\ell + b_\ell')\right)\\ &= \prod_{\ell=1}^{d}\left(\sum_{i=1}^{n_{\ell-1}}\sum_{j=1}^{n_\ell}\frac{W_{\ell,(i,j)} + W'_{\ell,(i,j)}}{2}, \sum_{j=1}^{n_\ell}\frac{b_{\ell,(j)} + b'_{\ell,(j)}}{2}\right)\end{aligned}$$

where $W_{\ell,(i,j)}$ is the (i,j)-th matrix entry of W_ℓ and $b_{\ell,(j)}$ is the j-th vector element of b_ℓ. This is simply the Euclidean vector mean if we treat θ and θ' as “flattened” vectors (that is, as a concatenation of all arrays, each collapsed into one dimension) in $\mathbb{R}^{\sum_{\ell=1}^{d} n_\ell \cdot n_{\ell+1}}$. For points close in parameter space, this ordinary mean captures what we would like a notion of an average of neural networks parameterizations to capture: the average of points in parameter space should parameterize a network whose behavior - i.e., as predictions on test data - is something like the “average behavior” among the averaged networks' behavior. This works for points sufficiently close to each other because very locally, two points are almost certainly not permutations of one another (since permutations require making some form of “reflection” in $\mathbb{R}^N$ across the axes corresponding to the permuted coordinates).

More generally, however, the Euclidean mean of two arbitrary parameterizations does not capture the desired notion of average, because the equivalence of permutations result in parameterizations with identical behavior (and thus whose “average” we would also like to have similar behavior) whose Euclidean elementwise mean as points yields an entirely different network with in general entirely different behavior. This problem can be resolved, however, by first writing all parameterizations in the “universal” or “standard basis” corresponding to the sorted order permutation. This standardization of the representative permutation for networks forces parameterizations as points in some real space to lie in the same symmetry region. As a simple example of what this means, consider the following plots of the points

$$r = (1.4, -3.0, 5.6), \quad g = (5.0, 1.2, -2.4), \quad b = (-3.2, 0.9, 6.0).$$

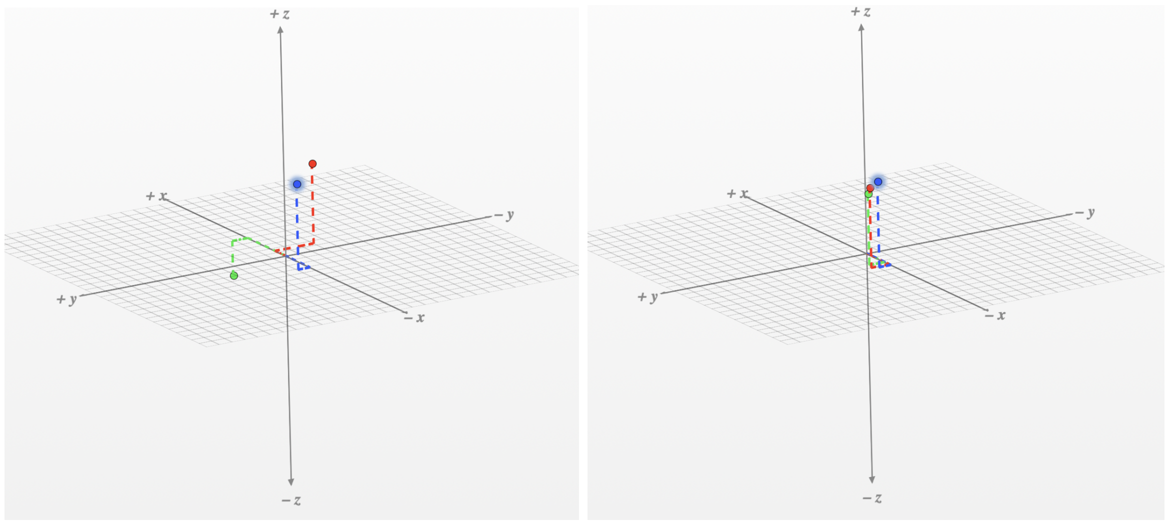

Figure 2: Left: points according to original coordinates; right: points with coordinates sorted in increasing order

On the left is plotted the original points, while on the right is plotted points

$$r' = (-3.0, 1.4, 5.6), \quad g' = (-2.4, 1.2, 5.0), \quad b = (-3.2, 0.9, 6.0),$$

whose coordinates are the sorted-order permutations of r, g and b, respectively. While the original points lie in distinct regions separated by the axes, the sorted-order points all lie in the same octant.

Similarly, parameterizations in their sorted-order permutations are forced to occupy the same symmetry region in parameter space, and thus an average between them is more likely to be a meaningful one - in particular, it eliminates the possibility that the average is the zero network, and reduces the probability that many of the weights of the average are zero.

In this way, establishing a fixed permutation as a basis in which to write neural network parameterizations establishes a more general notion of average for neural networks, not constrained to local points. Of course, because the loss surface is highly non-convex even within the symmetry region in which sorted-order points lie, this form of average does not fully capture the notion of average over network *behavior*. However, it may be a step toward a better understanding of how to combine neural networks at the level of weights, which is certainly a matter of great interest in current research.

References

[Berner et al., 2019] Berner, J., Elbrächter, D., and Grohs, P. (2019). How degenerate is the parametrization of neural networks with the relu activation function?

[Choromanska et al., 2014] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2014). The loss surfaces of multilayer networks.

[Izmailov et al., 2018] Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., and Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization.