Orthogonal Over-Parameterized Training

Weiyang Liu[1,2,*] Rongmei Lin[3,*] Zhen Liu[4] James M. Rehg[5] Liam Paull[4] Li Xiong[3] Le Song[5] Adrian Weller[1,6]

[1]University of Cambridge [2]Max Planck Institute for Intelligent Systems [3]Emory University
[4]Mila, Université de Montréal [5]Georgia Institute of Technology [6]Alan Turing Institute [*]Equal Contribution

Abstract

The inductive bias of a neural network is largely determined by the architecture and the training algorithm. To achieve good generalization, how to effectively train a neural network is of great importance. We propose a novel orthogonal over-parameterized training (OPT) framework that can provably minimize the hyperspherical energy which characterizes the diversity of neurons on a hypersphere. By maintaining the minimum hyperspherical energy during training, OPT can greatly improve the empirical generalization. Specifically, OPT fixes the randomly initialized weights of the neurons and learns an orthogonal transformation that applies to these neurons. We consider multiple ways to learn such an orthogonal transformation, including unrolling orthogonalization algorithms, applying orthogonal parameterization, and designing orthogonality-preserving gradient descent. For better scalability, we propose the stochastic OPT which performs orthogonal transformation stochastically for partial dimensions of neurons. Interestingly, OPT reveals that learning a proper coordinate system for neurons is crucial to generalization. We provide some insights on why OPT yields better generalization. Extensive experiments validate the superiority of OPT over the standard training.

1. Introduction

The inductive bias encoded in a neural network is generally determined by two major aspects: how the neural network is structured (*i.e.*, network architecture) and how the neural network is optimized (*i.e.*, training algorithm). For the same network architecture, using different training algorithms could lead to a dramatic difference in generalization performance [36, 60] even if the training loss is close to zero, implying that different training procedures lead to different inductive biases. Therefore, how to effectively train a neural network that generalize well remains an open challenge.

Recent theories [16, 15, 34, 45] suggest the importance of over-parameterization in linear neural networks. For example, [16] shows that optimizing an underdetermined quadratic objective over a matrix $\boldsymbol{M}$ with gradient descent on a factorization of $\boldsymbol{M}$ leads to an implicit regularization that may improve generalization. There is also strong empirical evidence [11, 51] that over-parameterzing the convolutional filters under some regularity is beneficial to generalization. Our paper aims to leverage the power of over-parameterization and explore more intrinsic structural priors in order to train a well-performing neural network.

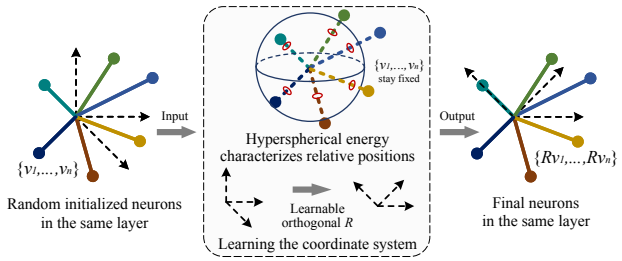


Figure 1: Overview of the orthogonal over-parameterized training framework. OPT learns an orthogonal transformation for each layer in the neural network, while keeping the randomly initialized neuron weights fixed.

Motivated by this goal, we propose a generic orthogonal over-parameterized training (OPT) framework for neural networks. Different from conventional neural training, OPT over-parameterizes a neuron $\boldsymbol{w}\in\mathbb{R}^d$ with the multiplication of a learnable layer-shared orthogonal matrix $\boldsymbol{R}\in\mathbb{R}^{d\times d}$ and a fixed randomly-initialized weight vector $\boldsymbol{v}\in\mathbb{R}^d$, and it follows that the equivalent weight for the neuron is $\boldsymbol{w}=\boldsymbol{R}\boldsymbol{v}$. Once each element of the neuron weight $\boldsymbol{v}$ has been randomly initialized by a zero-mean Gaussian distribution [20, 14], we fix them throughout the entire training process. Then OPT learns a layer-shared orthogonal transformation $\boldsymbol{R}$ that is applied to all the neurons (in the same layer). An illustration of OPT is given in Fig. 1. In contrast to standard neural training, OPT decomposes the neuron into an orthogonal transformation $\boldsymbol{R}$ that learns a proper coordinate system, and a weight vector $\boldsymbol{v}$ that controls the specific position of the neuron. Essentially, the weights $\{\boldsymbol{v}_1,\cdots,\boldsymbol{v}_n\in\mathbb{R}^d\}$ of different neurons determine the relative positions, while the layer-shared orthogonal matrix $\boldsymbol{R}$ specifies the coordinate system. Such a decoupled parameterization enables strong modeling flexibility.

Another motivation of OPT comes from an empirical observation that neural networks with lower *hyperspherical*

energy generalize better [49]. Hyperspherical energy quantifies the diversity of neurons on a hypersphere, and essentially characterizes the relative positions among neurons via this form of diversity. [49] introduces hyperspherical energy as a regularization in the network but do not guarantee that the hyperspherical energy can be effectively minimized (due to the existence of data fitting loss). To address this issue, we leverage the property of hyperspherical energy that it is independent of the coordinate system in which the neurons live and only depends on their relative positions. Specifically, we prove that, if we randomly initialize the neuron weight $\boldsymbol{v}$ with certain distributions, these neurons are guaranteed to attain minimum hyperspherical energy in expectation. It follows that OPT maintains the minimum energy during training by learning a coordinate system (*i.e.*, layer-shared orthogonal matrix) for the neurons. Therefore, OPT is able to provably minimize the hyperspherical energy.

We consider several ways to learn the orthogonal transformation. First, we unroll different orthogonalization algorithms such as Gram-Schmidt process, Householder reflection and Löwdin's symmetric orthogonalization. Different unrolled algorithms yield different implicit regularizations to construct the neuron weights. For example, symmetric orthogonalization guarantees that the new orthogonal basis has the least distance in the Hilbert space from the original non-orthogonal basis. Second, we consider to use a special parameterization (*e.g.*, Cayley parameterization) to construct the orthogonal matrix, which is more efficient in training. Third, we consider an orthogonality-preserving gradient descent to ensure that the matrix $\boldsymbol{R}$ stays orthogonal after each gradient update. Last, we relax the original optimization problem by making the orthogonality constraint a regularization for the matrix $\boldsymbol{R}$. Different ways of learning the orthogonal transformation may encode different inductive biases. We note that OPT aims to utilize orthogonalization as a tool to learn neurons that maintain small hyperspherical energy, rather than to study a specific orthogonalization method. Furthermore, we propose a refinement strategy to reduce the hyperspherical energy for the randomly initialized neuron weights $\{\boldsymbol{v}_1, \cdots, \boldsymbol{v}_n\}$. In specific, we directly minimize the hyperspherical energy of these random weights as a preprocessing step before training them on actual data.

To improve scalability, we further propose the stochastic OPT that randomly samples neuron dimensions to perform orthogonal transformation. The random sampling process is repeated many times such that each dimension of the neuron is sufficiently learned. Finally, we provide some theoretical insights and discussions to justify the effectiveness of OPT. The advantages of OPT are summarized as follows:

- OPT is a generic neural network training framework with strong flexibility. There are many different ways to learn the orthogonal transformations and each one imposes a unique inductive bias. Our paper compares how different orthogonalizations may affect generalization in OPT.
- OPT is the first training framework where the hyperspherical energy is provably minimized (in contrast to [49]), leading to better empirical generalization. OPT reveals that learning a proper coordinate system is crucial to generalization, and the hyperspherical energy is sufficiently expressive to characterize relative neuron positions.
- There is no extra computational cost for the OPT-trained neural network in inference. In the testing stage, it has the same inference speed and model size as the normally trained network. Our experiments also show that OPT performs well on a diverse class of neural networks and therefore is agnostic to different neural architectures.
- Stochastic OPT can greatly improve the scalability of OPT while enjoying the same guarantee to minimize hyperspherical energy and having comparable performance.

2. Related Work

Orthogonality in Neural Networks. Orthogonality is widely adopted to improve neural networks. [4, 54, 7, 26, 78] use orthogonality as a regularization for neurons. [27, 42, 3, 75, 58, 31] use principled orthogonalization methods to guarantee the neurons are orthogonal to each other. In contrast to these works, OPT does not encourage orthogonality among neurons. Instead, OPT utilizes principled orthogonalization for learning orthogonal transformations for (not necessarily orthogonal) neurons to minimize hyperspherical energy.

Parameterization of Neurons. There are various ways to parameterize a neuron for different applications. [11] overparameterizes a 2D convolution kernel by combining a 2D kernel of the same size and two additional 1D asymmetric kernels. The resulting convolution kernel has the same effective parameters during testing but more parameters during training. [51] constructs a neuron with a bilinear parameterization and regularizes the bilinear similarity matrix. [79] reparameterizes the neuron matrix with an adaptive fastfood transform to compress model parameters. [30, 48, 73] employ sparse and low-rank structures to construct convolution kernels for a efficient neural network.

Hyperspherical Learning. [54, 52, 72, 10, 71, 47, 50] propose to learn representations on a hypersphere and show that the angular information, in contrast to magnitude information, preserves the most semantic meaning. [49] define the hyperspherical energy that quantifies the diversity of neurons on a hypersphere and shows that the small hyperspherical energy generally improves empirical generalization.

3. Orthogonal Over-Parameterized Training

3.1. General Framework

OPT parameterizes the neuron as the multiplication of an orthogonal matrix $\boldsymbol{R} \in \mathbb{R}^{d\times d}$ and a neuron weight vector $\boldsymbol{v} \in \mathbb{R}^d$, and the equivalent neuron weight becomes $\boldsymbol{w} = \boldsymbol{R}\boldsymbol{v}$. The output $\hat{y}$ of this neuron can be represented by $\hat{y} = (\boldsymbol{R}\boldsymbol{v})^\top \boldsymbol{x}$

where $\boldsymbol{x}\in\mathbb{R}^d$ is the input vector. In OPT, we typically fix the randomly initialized neuron weight $\boldsymbol{v}$ and only learn the orthogonal matrix $\boldsymbol{R}$. In contrast, the standard neuron is directly formulated as $\hat{y}=\boldsymbol{v}^\top\boldsymbol{x}$, where the weight vector $\boldsymbol{v}$ is learned via back-propagation in training.

As an illustrative example, we consider a linear MLP with a loss function $\mathcal{L}$ (*e.g.*, the least squares loss: $\mathcal{L}(e_1,e_2)=(e_1-e_2)^2$). Specifically, the learning objective of the standard training is $\min_{\{\boldsymbol{v}_i,u_i,\forall i\}}\sum_{j=1}^m\mathcal{L}\big(y,\sum_{i=1}^n u_i\boldsymbol{v}_i^\top\boldsymbol{x}_j\big)$, while differently, our OPT is formulated as

$$\min_{\{\boldsymbol{R},u_i,\forall i\}}\sum_{j=1}^m\mathcal{L}\big(y,\sum_{i=1}^n u_i(\boldsymbol{R}\boldsymbol{v}_i)^\top\boldsymbol{x}_j\big)\quad\text{s.t. }\boldsymbol{R}^\top\boldsymbol{R}=\boldsymbol{R}\boldsymbol{R}^\top=\boldsymbol{I}\quad(1)$$

where $\boldsymbol{v}_i\in\mathbb{R}^d$ is the i-th neuron in the first layer, and $\boldsymbol{u}=\{u_1,\cdots,u_n\}\in\mathbb{R}^n$ is the output neuron in the second layer. In OPT, each element of $\boldsymbol{v}_i$ is usually sampled from a zero-mean Gaussian distribution (*e.g.*, both Xavier [14] and Kaiming [20] initializations belong to this class), and is fixed throughout the entire training process. In general, OPT learns an orthogonal matrix that is applied to all the neurons instead of learning the individual neuron weight. Note that, we usually do not apply OPT to neurons in the output layer (*e.g.*, $\boldsymbol{u}$ in this MLP example, and the final linear classifiers in CNNs), since it makes little sense to fix a set of random linear classifiers. Therefore, the central problem is how to learn these layer-shared orthogonal matrices.

3.2. Hyperspherical Energy Perspective

One of the most important properties of OPT is its invariance to hyperspherical energy. Based on [49], the hyperspherical energy of n neurons is defined as $\boldsymbol{E}(\hat{\boldsymbol{v}}_i|_{i=1}^n)=\sum_{i=1}^n\sum_{j=1,j\neq i}^n\|\hat{\boldsymbol{v}}_i-\hat{\boldsymbol{v}}_j\|^{-1}$ in which $\hat{\boldsymbol{v}}_i=\frac{\boldsymbol{v}_i}{\|\boldsymbol{v}_i\|}$ is the i-th neuron weight projected onto the unit hypersphere $\mathbb{S}^{d-1}=\{\boldsymbol{v}\in\mathbb{R}^d|\,\|\boldsymbol{v}\|=1\}$. Hyperspherical energy is used to characterize the diversity of n neurons on a unit hypersphere. Assume that we have n neurons in one layer, and we have learned an orthogonal matrix $\boldsymbol{R}$ for these neurons. The hyperspherical energy of these n OPT-trained neurons is

$$\begin{aligned}\boldsymbol{E}(\hat{\boldsymbol{R}}\hat{\boldsymbol{v}}_i|_{i=1}^n)&=\sum_{i=1}^n\sum_{j=1,j\neq i}^n\|\boldsymbol{R}\hat{\boldsymbol{v}}_i-\boldsymbol{R}\hat{\boldsymbol{v}}_j\|^{-1}\\(\text{since }\|\boldsymbol{R}\|^{-1}=1)\quad&=\sum_{i=1}^n\sum_{j=1,j\neq i}^n\|\hat{\boldsymbol{v}}_i-\hat{\boldsymbol{v}}_j\|^{-1}=\boldsymbol{E}(\hat{\boldsymbol{v}}_i|_{i=1}^n)\end{aligned}\quad(2)$$

which verifies that the hyperspherical energy does not change in OPT. Moreover, [49] proves that minimum hyperspherical energy corresponds to the uniform distribution over the hypersphere. As a result, if the initialization of the neurons in the same layer follows the uniform distribution over the hypersphere, then we can guarantee that the hyperspherical energy is minimal in a probabilistic sense.

Theorem 1. *For the neuron $\boldsymbol{h}=\{h_1,\cdots,h_d\}$ where $h_i,\forall i$ are initialized i.i.d. following a zero-mean Gaussian distribution (i.e., $h_i\sim N(0,\sigma^2)$), the projections onto a unit hypersphere $\hat{\boldsymbol{h}}=\boldsymbol{h}/\|\boldsymbol{h}\|$ where $\|\boldsymbol{h}\|=(\sum_{i=1}^d h_i^2)^{1/2}$ are uniformly distributed on the unit hypersphere $\mathbb{S}^{d-1}$. The neurons with minimum hyperspherical energy attained asymptotically approach the uniform distribution on $\mathbb{S}^{d-1}$.*

Theorem 1 proves that, as long as we initialize the neurons in the same layer with zero-mean Gaussian distribution, the resulting hyperspherical energy is guaranteed to be small (*i.e.*, the expected energy is minimal). It is because the neurons are uniformly distributed on the unit hypersphere and hyperspherical energy quantifies the uniformity on the hypersphere in some sense. More importantly, prevailing neuron initializations such as [14] and [20] are zero-mean Gaussian distribution. Therefore, our neurons naturally have low hyperspherical energy from the beginning. Appendix L gives geometric properties of the random initialized neurons.

3.3. Unrolling Orthogonalization Algorithms

In order to learn the orthogonal transformation, we unroll classic orthogonalization algorithms and embed them into the neural network such that the training can be performed in an end-to-end fashion. We need to make every step of the orthogonalization algorithm differentiable, as shown in Fig. 2.

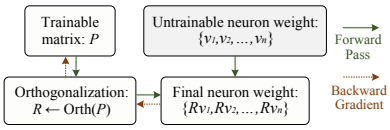


Figure 2: Unrolled orthogonalization.

Gram-Schmidt Process. This method takes a linearly independent set and eventually produces an orthogonal set based on it. The Gram-Schmidt Process (GS) usually takes the following steps to orthogonalize a set of vectors $\{\boldsymbol{u}_1,\cdots,\boldsymbol{u}_n\}\in\mathbb{R}^{n\times n}$ and obtain an orthonormal set $\{\boldsymbol{e}_1,\cdots,\boldsymbol{e}_i,\cdots,\boldsymbol{e}_n\}\in\mathbb{R}^{n\times n}$. First, when $i=1$, we have $\boldsymbol{e}_1=\frac{\tilde{\boldsymbol{e}}_1}{\|\tilde{\boldsymbol{e}}_1\|}$ where $\tilde{\boldsymbol{e}}_1=\boldsymbol{u}_1$. Then, when $n\geq i\geq 2$, we have $\boldsymbol{e}_i=\frac{\tilde{\boldsymbol{e}}_i}{\|\tilde{\boldsymbol{e}}_i\|}$ where $\tilde{\boldsymbol{e}}_i=\boldsymbol{u}_i-\sum_{j=1}^{i-1}\text{Proj}_{\boldsymbol{e}_j}(\boldsymbol{u}_i)$. Note that, $\text{Proj}_{\boldsymbol{b}}(\boldsymbol{a})=\frac{\langle\boldsymbol{a},\boldsymbol{b}\rangle}{\langle\boldsymbol{b},\boldsymbol{b}\rangle}\boldsymbol{b}$ is defined as the projection operator.

Householder Reflection. A Householder reflector is defined as $\boldsymbol{H}=\boldsymbol{I}-2\frac{\boldsymbol{u}\boldsymbol{u}^\top}{\|\boldsymbol{u}\|^2}$ where $\boldsymbol{u}$ is perpendicular to the reflection hyperplane. In QR factorization, Householder reflection (HR) is used to transform a (non-singular) square matrix into an orthogonal matrix and an upper triangular matrix. Given a matrix $\boldsymbol{U}=\{\boldsymbol{u}_1,\cdots,\boldsymbol{u}_n\}\in\mathbb{R}^{n\times n}$, we consider the first column vector $\boldsymbol{u}_1$. We use Householder reflector to transform $\boldsymbol{u}_1$ to $\boldsymbol{e}_1=\{1,0,\cdots,0\}$. Specifically, we construct an orthogonal matrix $\boldsymbol{H}_1$ with $\boldsymbol{H}_1=\boldsymbol{I}-2\frac{(\boldsymbol{u}_1-\|\boldsymbol{u}_1\|\boldsymbol{e}_1)(\boldsymbol{u}_1-\|\boldsymbol{u}_1\|\boldsymbol{e}_1)^\top}{\|\boldsymbol{u}_1-\|\boldsymbol{u}_1\|\boldsymbol{e}_1\|^2}$ The first column of $\boldsymbol{H}_1\boldsymbol{U}$ becomes $\{\|\boldsymbol{u}_1\|,0,\cdots,0\}$. At the k-th step, we can view the sub-matrix $\boldsymbol{U}_{(k:n,k:n)}$ as a new $\boldsymbol{U}$, and use the same procedure to construct the Householder transformation $\tilde{\boldsymbol{H}}_k\in\mathbb{R}^{(n-k)\times(n-k)}$. We construct the final Householder transformation as $\boldsymbol{H}_k=\text{Diag}(\boldsymbol{I}_k,\tilde{\boldsymbol{H}}_k)$. Now we can gradually transform $\boldsymbol{U}$ to an upper triangular matrix with n Householder reflections. Therefore, we have that $\boldsymbol{H}_n\cdots\boldsymbol{H}_2\boldsymbol{H}_1\boldsymbol{U}=\boldsymbol{R}^{\text{up}}$ where $\boldsymbol{R}^{\text{up}}$ is an upper triangular matrix and the obtained

orthogonal set is $\boldsymbol{Q}^\top = \boldsymbol{H}_n \cdots \boldsymbol{H}_2 \boldsymbol{H}_1$.

Löwdin's Symmetric Orthogonalization. Let the matrix $\boldsymbol{U} = \{\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n\} \in \mathbb{R}^{n \times n}$ be a given set of linearly independent vectors in an n-dimensional space. A non-singular linear transformation $\boldsymbol{A}$ can transform the basis $\boldsymbol{U}$ to an orthogonal basis $\boldsymbol{R}$: $\boldsymbol{R} = \boldsymbol{U}\boldsymbol{A}$. The matrix $\boldsymbol{R}$ will be orthogonal if $\boldsymbol{R}^\top \boldsymbol{R} = (\boldsymbol{U}\boldsymbol{A})^\top \boldsymbol{U}\boldsymbol{A} = \boldsymbol{A}^\top \boldsymbol{M}\boldsymbol{A} = \boldsymbol{I}$ where $\boldsymbol{M} = \boldsymbol{U}^\top \boldsymbol{U}$ is the Gram matrix of the given set $\boldsymbol{U}$. We obtain a general solution to the orthogonalization problem via the substitution: $\boldsymbol{A} = \boldsymbol{M}^{-\frac{1}{2}} \boldsymbol{B}$ where $\boldsymbol{B}$ is an arbitrary unitary matrix. The specific choice $\boldsymbol{B} = \boldsymbol{I}$ gives the Löwdin's symmetric orthogonalization (LS): $\boldsymbol{R} = \boldsymbol{U}\boldsymbol{M}^{-\frac{1}{2}}$. We can analytically obtain the symmetric orthogonalization from the singular value decomposition: $\boldsymbol{U} = \boldsymbol{W}\boldsymbol{\Sigma}\boldsymbol{V}^\top$. Then LS gives $\boldsymbol{R} = \boldsymbol{W}\boldsymbol{V}^\top$ as the orthogonal set for $\boldsymbol{U}$. LS has a unique property which the other orthogonalizations do not have. The orthogonal set resembles the original set in a nearest-neighbour sense. More specifically, LS guarantees that $\sum_i \|\boldsymbol{R}_i - \boldsymbol{U}_i\|^2$ (where $\boldsymbol{R}_i$ and $\boldsymbol{U}_i$ are the i-th column of $\boldsymbol{R}$ and $\boldsymbol{U}$, respectively) is minimized. Intuitively, LS indicates the gentlest pushing of the directions of the vectors in order to get them orthogonal to each other.

Discussion. These orthogonalization algorithms are fully differentiable and end-to-end trainable. For accurate orthogonality, these algorithms can be used repeatedly and unrolled with multiple steps. Empirically, one-step unrolling already works well. Givens rotations can also construct the orthogonal matrix, but it requires traversing all lower triangular elements in the original set $\boldsymbol{U}$, which takes $\mathcal{O}(n^2)$ complexity and is too costly. Interestingly, each orthogonalization encodes a unique inductive bias to the neurons by imposing implicit regularizations (*e.g.*, least distance in Hilbert space for LS). Details about these orthogonalizations are in Appendix A. Unrolling orthogonalization has been considered in different scenarios [27, 69, 56]. More orthogonalization methods [41] can be applied in OPT, but exhaustively applying them to OPT is out of the scope of this paper.

3.4. Orthogonal Parameterization

A convenient way to ensure orthogonality while learning the matrix $\boldsymbol{R}$ is to use a special parameterization that inherently guarantees orthogonality. The exponential parameterization use $\boldsymbol{R} = \exp(\boldsymbol{W})$ (where $\exp(\cdot)$ denotes the matrix exponential) to represent an orthogonal matrix from a skew-symmetric matrix $\boldsymbol{W}$. The Cayley parameterization (CP) is a Padé approximation of the exponential parameterization, and is a more natural choice due to its simplicity. CP uses the following transform to construct an orthogonal matrix $\boldsymbol{R}$ from a skew-symmetric matrix $\boldsymbol{W}$: $\boldsymbol{R} = (\boldsymbol{I} + \boldsymbol{W})(\boldsymbol{I} - \boldsymbol{W})^{-1}$ where $\boldsymbol{W} = -\boldsymbol{W}^\top$. We note that CP only produces the orthogonal matrices with determinant 1, which belong to the special orthogonal group and thus $\boldsymbol{R} \in SO(n)$. Specifically, it suffices to learn the upper or lower triangular of the matrix $\boldsymbol{W}$ with unconstrained optimization to obtain a desired orthogonal matrix $\boldsymbol{R}$. Cayley parameterization does not cover the entire orthogonal group and is less flexible in terms of representation power, which serves as an explicit regularization for the neurons.

3.5. Orthogonality-Preserving Gradient Descent

An alternative way to guarantee orthogonality is to modify the gradient update for the matrix $\boldsymbol{R}$. The idea is to initialize $\boldsymbol{R}$ with an arbitrary orthogonal matrix and then ensure each gradient update is to apply an orthogonal transformation to $\boldsymbol{R}$. It is essentially conducting gradient descent on the Stiefel manifold [44, 74, 75, 42, 3, 22, 33]. Given a matrix $\boldsymbol{U}_{(0)} \in \mathbb{R}^{n \times n}$ that is initialized as an orthogonal matrix, we aim to construct an orthogonal transformation as the gradient update. We use the Cayley transform to compute a parametric curve on the Stiefel manifold $\mathcal{M}_s = \{\boldsymbol{U} \in \mathbb{R}^{n \times n} : \boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}\}$ with a specific metric via a skew-symmetric matrix $\boldsymbol{W}$ and use it as the update rule:

$$\boldsymbol{Y}(\lambda) = (\boldsymbol{I} - \frac{\lambda}{2}\boldsymbol{W})^{-1}(\boldsymbol{I} + \frac{\lambda}{2}\boldsymbol{W})\boldsymbol{U}_{(i)},\ \boldsymbol{U}_{(i+1)} = \boldsymbol{Y}(\lambda) \quad (3)$$

where $\hat{\boldsymbol{W}} = \nabla f(\boldsymbol{U}_{(i)})\boldsymbol{U}_{(i)}^\top - \frac{1}{2}\boldsymbol{U}_{(i)}(\boldsymbol{U}_{(i)}^\top \nabla f(\boldsymbol{U}_{(i)}\boldsymbol{U}_{(i)}^\top)$ and $\boldsymbol{W} = \hat{\boldsymbol{W}} - \hat{\boldsymbol{W}}^\top$. $\boldsymbol{U}_{(i)}$ denotes the orthogonal matrix in the i-th iteration. $\nabla f(\boldsymbol{U}_{(i)})$ denotes the original gradient of the loss function *w.r.t.* $\boldsymbol{U}_{(i)}$. We term this gradient update as orthogonal-preserving gradient descent (OGD). To reduce the computational cost of the matrix inverse in Eq. 3, we use an iterative method [44] to approximate the Cayley transform without matrix inverse. We arrive at the fixed-point iteration:

$$\boldsymbol{Y}(\lambda) = \boldsymbol{U}_{(i)} + \frac{\lambda}{2}\boldsymbol{W}\left(\boldsymbol{U}_{(i)} + \boldsymbol{Y}(\lambda)\right) \quad (4)$$

which converges to the closed-form Cayley transform with a rate of $o(\lambda^{2+n})$ (n is the iteration number). In practice, two iterations suffice for a reasonable approximation accuracy.

3.6. Relaxation to Orthogonal Regularization

Alternatively, we also consider relaxing the original optimization with an orthogonality constraint to an unconstrained optimization with orthogonality regularization (OR). Specifically, we remove the orthogonality constraint, and adopt an orthogonality regularization for $\boldsymbol{R}$, *i.e.*, $\|\boldsymbol{R}^\top \boldsymbol{R} - \boldsymbol{I}\|_F^2$. However, OR cannot guarantee the energy stays unchanged. Taking Eq. 1 as an example, the objective becomes

$$\min_{\boldsymbol{R}, u_i, \forall i} \sum_{j=1}^{m} \mathcal{L}\big(y, \sum_{i=1}^{n} u_i (\boldsymbol{R}\boldsymbol{v}_i)^\top \boldsymbol{x}_j\big) + \beta \|\boldsymbol{R}^\top \boldsymbol{R} - \boldsymbol{I}\|_F^2 \quad (5)$$

where β is a hyperparameter. This serves as an relaxation of the original OPT objective. Note that, OR is imposed to $\boldsymbol{R}$ instead of neurons and is quite different from the existing orthogonality regularization on neurons [54, 4, 27, 78, 7].

3.7. Refining the Initialization as Preprocessing

Minimizing the energy beforehand. Because we randomly initialize the neurons $\{\boldsymbol{v}_1, \cdots, \boldsymbol{v}_n\}$, there exists a variance that makes the hyperspherical energy deviate from

the minima even if the hyperspherical energy is minimal in a probabilistic sense. To further reduce the hyperspherical energy, we propose to refine the random initialization by minimizing its hyperspherical energy as a preprocessing step before the OPT training. Specifically, before feeding these neurons to OPT, we first minimize the hyperspherical energy of the initialized neurons with gradient descent (without fitting the training data). Moreover, since the randomly initialized neurons cannot guarantee to get rid of the collinearity redundancy as shown in [49] (*i.e.*, two neurons are on the same line but have opposite directions), we can perform the half-space hyperspherical energy minimization [49].

Normalizing the neurons. The norm of the randomly initialized neurons may have some influence on OPT, serving a role similar to weighting the importance of different neurons. Moreover, the norm makes the hyperspherical energy less expressive to characterize the diversity of neurons, as discussed in Section 5.3. To make the coordinate frame (i.e. the rotation matrix R) truly independent of the relative positions of the neurons, we propose to normalize the neuron weights such that each neuron has unit norm. Because the weights of the neurons $\{\boldsymbol{v}_1, \cdots, \boldsymbol{v}_n\}$ are fixed during training and orthogonal matrices will not change the norm of the neurons, we only need to normalize the randomly initialized neuron weights as a preprocessing before the OPT training.

We have comprehensively evaluated both refinement strategies in Section 6.2 and verified their effectiveness. Note that the effectiveness of OPT is not dependent on these refinements. Our experiments do not use these refinements by default and the results show that OPT still performs well.

4. Towards Better Scalablity for OPT

If the dimension of neurons becomes extremely large, then the orthogonal matrix to transform the neurons will also be large. Therefore, it may take large GPU memory and time to train the neural networks with the original OPT. To address this, we propose a scalable variant – stochastic OPT (S-OPT). The key idea of S-OPT is to randomly select some dimensions from the neurons in the same layer and construct a small orthogonal matrix to transform these dimensions together. The selection of dimensions is stochastic in each outer iteration, so a small orthogonal matrix is sufficient to cover all the neuron dimensions. S-OPT aims to approximate a large orthogonal transformation for all the neuron dimensions with many small orthogonal transformations for random subsets of these dimensions, which shares similar spirits with Givens rotation. The approximation will be more accurate when the procedure is randomized over many times. Fig. 3 compares the size of the orthogonal matrix in OPT and S-OPT. The orthogonal matrix in OPT is of size $d \times d$, while the orthogonal matrix in S-OPT is of size $p \times p$ where p is usually much smaller than d. Most importantly, S-OPT can still preserve the low hyperspherical energy of neurons because of the following result.

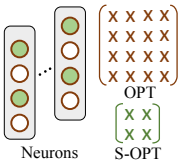


Figure 3: S-OPT.

Theorem 2. *For n d-dimensional neurons, selecting any p ($p \leq d$) dimensions and applying an shared orthogonal transformation ($p \times p$ orthogonal matrix) to these p dimensions of all neurons will not change the hyperspherical energy.*

A description of S-OPT is given in Algorithm 1. S-OPT has outer and inner iterations. In each inner iteration, the training is almost the same as OPT, except that the orthogonal matrix transforms a subset of the dimensions and the learnable orthogonal matrix has to be re-initialized to an identity matrix. The selection of neuron dimension is randomized in every outer iteration such that all neuron dimensions can be sufficiently covered as the number of outer iterations increases. Therefore, given sufficient number of iterations, S-OPT will perform comparably to OPT, as empirically verified in Section 6.3. As a parallel direction to improve the scalability, we further propose a parameter-efficient OPT in Appendix I. This OPT variant explores structure priors in $\boldsymbol{R}$ to improve parameter efficiency.

Algorithm 1 Stochastic OPT

for $i = 1, 2, \cdots, N_{\text{out}}$ **do**
  for $j = 1, 2, \cdots, N_{\text{in}}$ **do**
    1. Randomly select p dimensions from d-dimensional neurons in the same layer.
    2. Construct an orthogonal matrix $\boldsymbol{R}_p \in \mathbb{R}^{p \times p}$ and initialize it as identity matrix.
    3. Update $\boldsymbol{R}_p$ by applying OPT with one iteration.
  end
  4. Multiply $\boldsymbol{R}_p$ back to the p-dim sub-vectors from the d-dim neurons to transform these neurons.
end

5. Intriguing Insights and Discussions

5.1. Local Landscape

We follow [43] to visualize the loss landscapes of both standard training and OPT in Fig. 4. For standard training, we perturb the parameter space of all the neurons (*i.e.*, filters). For OPT, we perturb the parameter space of all the trainable matrices (*i.e.*, $\boldsymbol{P}$ in Fig. 2), because OPT does not directly learn neuron weights. The general idea is to use two random vectors (*e.g.*, normal distribution) to perturb the parameter space and obtain the loss value with the perturbed network parameters. Details and full results about the visualization are given in Appendix E. The loss landscape of standard training has extremely sharp minima. The red region is very flat, leading to small gradients. In contrast, the loss landscape of OPT is much more smooth and convex with flatter minima, well matching the finding that flat minimizers generalize well [23, 8, 29]. Additional loss landscape visualization results in Appendix F (with uniform perturbation distributions) also support the same argument.

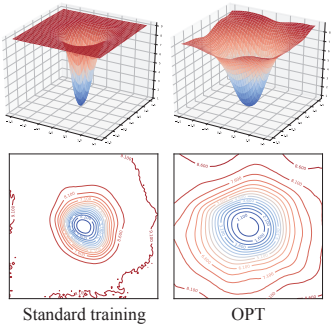


Figure 4: Training loss landscapes.

We also show the landscape of testing error on CIFAR-100 in Fig. 5. Full results and details are in Appendix E. Compared to standard training, the testing error of OPT increases more slowly and smoothly while the network parameters move away from the minima, which indicates that the parameter space of OPT yields better robustness to perturbations.

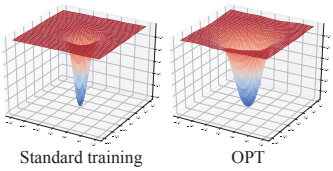


Figure 5: Testing error landscapes.

5.2. Optimization and Generalization

We discuss why OPT may improve optimization and generalization. On one hand, [77] proves that once the neurons are hyperspherically diverse enough in a one-hidden-layer network, the training loss is on the order of the square norm of the gradient and the generalization error will have an additional term $\tilde{\mathcal{O}}(1/\sqrt{m})$ where m is the number of samples. This suggests that SGD-optimized networks with minimum hyperspherical energy (MHE) attained have no spurious local minima. Since OPT is guaranteed to achieve MHE in expectation, OPT-trained networks enjoy the inductive bias induced by MHE. On the other hand, [34, 1, 12, 45, 16] shows that over-parameterization in neural networks improves the first-order optimization, leads to better generalization, and imposes implicit regularizations. In the light of this, OPT also introduces over-parameterization to each neuron, which shares similar spirits with [46]. Specifically, one d-dimensional neuron has d^2+d parameters in OPT (with d^2 being layer-shared), compared to d parameters in a standard neuron. Although OPT uses more parameters for a neuron in training, the equivalent number of parameters for a neuron stays unchanged and it will not affect testing speed.

5.3. Discussions

Over-parameterization. We delve deeper into the over-parameterization in the context of OPT. Its definition varies in different cases. OPT is over-parameterized in terms of training in the following sense. Although OPT-trained networks have the same effective number of parameters as the standard networks in testing, the OPT neuron is decomposed into two sets of parameters in training: orthogonal matrix and neuron weights. It means that the same set of parameters in a neural network can be represented by different sets of training parameters in OPT (*i.e.*, different combinations of orthogonal matrices and neuron weights can lead to the same neural network). OPT is still over-parameterized even if we only count the number of learnable parameters. For a layer of n d-dimensional neurons, the number of learnable parameters in vanilla OPT is $\frac{d(d-1)}{2}$ in contrast to nd in standard training. In prevailing architectures (*e.g.*, ResNet [21]), the neuron dimension is far larger than the number of neurons.

Coordinate system and relative position. OPT shows that learning the coordinate system yields better generalization than learning neuron weights directly. This implies that the coordinate system is crucial to generalization. However, the relative position does not matter only when the hyperspherical energy is sufficiently low, indicating that the neurons need to be diverse enough on the unit hypersphere.

The effects of neuron norm. Because we will normalize the neuron norm when computing the hyperspherical energy, the effects of neuron norm will not be taken into consideration. Moreover, simply learning the orthogonal matrices will not change the neuron norm. Therefore, the neuron norm may affect the training. We use an extreme example to demonstrate the effects. Assume that one of the neurons has norm 1000 and the other neurons have norm 0.01. Then no matter what orthogonal matrices we have learned, the final performance will be bad. In this case, the hyperspherical energy can still be minimized to a very low value, but it can not capture the norm distribution. Fortunately, such an extreme case is unlikely to happen, because we are using zero-mean Gaussian distribution to initialize the neuron and every neuron also has the same expected value for the norm. To eliminate the effects of norms, we can normalize the neuron weights in training, as proposed in Section 3.7.

6. Applications and Experimental Results

We put all the experimental settings and many additional results in Appendix D and Appendix I,K, respectively.

6.1. Ablation Study and Exploratory Experiments

Orthogonality. We evaluate whether orthogonality in OPT is necessary. We use 6-layer and 9-layer CNN (Appendix D) on CIFAR-100. Then we compare OPT with unconstrained over-parameterized training (UPT) which learns an unconstrained matrix $\boldsymbol{R}$ (with weight decay) using the same network. In Table 1, "FN" denotes whether the randomly initialized neuron weights are fixed in training. "LR" denotes whether the learnable matrix $\boldsymbol{R}$ is unconstrained ("U") or orthogonal ("GS" for Gram-Schmidt process). Table 1 shows that without orthogonality, UPT performs much worse than OPT.

Method	FN	LR	CNN-6	CNN-9
Baseline	-	-	37.59	33.55
UPT	✗	U	48.47	46.72
UPT	✓	U	42.61	39.38
OPT	✗	GS	37.24	32.95
OPT	✓	GS	**33.02**	**31.03**

Table 1: Error (%) on C-100.

Fixed or learnable weights. From Table 1, we can see that using fixed neuron weights is consistently better than learnable neuron weights in both UPT and OPT. It indicates that fixing the neuron weights can well maintain low hyperspherical energy and is beneficial to empirical generalization.

Refining initialization. We evaluate two refinement methods in Section 3.7 for neuron initialization. First, we consider the hyperspherical energy minimization as a preprocessing for the neuron weights. Our experiment uses CNN-6 on CIFAR-100. Specifically, we run gradient descent for 5k

Method	Original	MHE	HS-MHE	CoMHE
OPT (GS)	33.02	32.99	32.78	**32.69**
OPT (LS)	34.48	34.43	34.37	**34.15**
OPT (CP)	33.53	33.50	33.42	**33.27**
Energy	3.5109	3.5003	3.4976	**3.4954**

Table 2: Refining initialized energy.

iterations to minimize the objective of MHE/HS-MHE [49] or CoMHE [47] before the training starts. Table 2 shows the hyperspherical energy before and after the preprocessing. All methods start with the same random initialization, so all hyperspherical energies start at 3.5109. Testing errors (%) in Table 2 show that the refinement well improves OPT. Although using advanced regularizations such as CoMHE as pre-processing can improve the performance significantly, we do not use them in the other experiments in order to keep our comparison fair and clean. More different ways to minimize the hyperspherical energy can also be considered [50].

We evaluate the second refinement strategy, *i.e.*, neuron weight normalization. Section 5.3 has explained why normalizing the neuron weights may be useful. After initialization, we normalize all the neuron weights to 1. Since OPT does not change the neuron norm, the neuron will keep the norm as 1. More importantly, the hyperspherical energy will not be affected by the neuron normalization. We conduct classification with CNN-6 on CIFAR-100. Testing errors in Table 3 show that normalizing the neurons greatly improves OPT, validating our previous analysis. Note that, these two refinements are not used by default in other experiments.

Method	w/o Norm	w/ Norm
Baseline	37.59	**36.05**
OPT (GS)	33.02	**32.54**
OPT (HR)	35.67	**35.30**
OPT (LS)	34.48	**32.11**
OPT (CP)	33.53	**32.49**
OPT (OGD)	33.37	**32.70**
OPT (OR)	34.70	**33.27**

Table 3: Normalization (%).

High vs. low energy. We validate that high hyperspherical energy corresponds to inferior empirical generalization. To initialize high energy neurons, we use [20] and set the mean as 1e-3, 1e-2, 2e-2, and 3e-2. We experiment on CIFAR-100 with CNN-6. Table 4 ("N/C" denotes not converged) show that higher energy generalizes worse and also leads to difficulty in convergence. We see that a small change in energy can lead to a dramatic generalization gap.

Mean	Energy	Error (%)
0	**3.5109**	**32.49**
1e-3	3.5117	33.11
1e-2	3.5160	39.51
2e-2	3.5531	53.89
3e-2	3.6761	N/C

Table 4: Initial energy.

No BatchNorm. We evaluate how OPT performs without BatchNorm (BN) [28]. We perform classification on CIFAR-100 with CNN-6. In Table 5, we see that all OPT variants consistently outperform both the baseline and HS-MHE [49] by a significant margin, validating that OPT can work well without BN. CP achieves the best error with more than 4% lower than standard training.

Method	Error (%)
Baseline	38.95
HS-MHE	36.90
OPT (GS)	35.61
OPT (HR)	37.51
OPT (LS)	35.83
OPT (CP)	**34.88**
OPT (OGD)	35.38

Table 5: No BN.

6.2. Empirical Evaluation on OPT

Multi-layer perceptrons. We evaluate OPT on MNIST with a 3-layer MLP. Appendix D gives specific settings. Table 6 shows the testing error with normal initialization (MLP-N) or Xavier initialization [14] (MLP-X). GS/HR/LS denote different orthogonalization unrolling. CP denotes Cayley parameterization. OGD denotes orthogonal-preserving gradient descent. OR denotes relaxed orthogonal regularization. All OPT variants outperform the others by a large margin.

Method	MNIST		CIFAR-100			
	MLP-N	MLP-X	CNN-6	CNN-9	ResNet-20	ResNet-32
Baseline	6.05	2.14	37.59	33.55	31.11	30.16
Orthogonal [7]	5.78	1.93	36.32	33.24	31.06	30.05
SRIP [4]	-	-	34.82	32.72	30.89	29.70
HS-MHE [49]	5.57	1.88	34.97	32.87	30.98	29.76
OPT (GS)	**5.11**	**1.45**	**33.02**	**31.03**	30.49	29.34
OPT (HR)	5.31	1.60	35.67	32.75	30.73	29.56
OPT (LS)	5.32	1.54	34.48	31.22	30.51	29.42
OPT (CP)	5.14	1.49	33.53	31.28	**30.47**	**29.31**
OPT (OGD)	5.38	1.56	33.33	31.47	30.50	29.39
OPT (OR)	5.41	1.78	34.70	32.63	30.66	29.47

Table 6: Testing error (%) of OPT for MLPs and CNNs.

Convolutional networks. We evaluate OPT with 6/9-layer plain CNNs and ResNet-20/32 [21] on CIFAR-100. Detailed settings are in Appendix D. All neurons (*i.e.*, convolution kernels) are initialized by [20]. BatchNorm is used by default. Table 6 shows that all OPT variants outperform both baseline and HS-MHE by a large margin. HS-MHE puts the hyperspherical energy into the loss function and naively minimizes it along with the CNN. We observe that OPT (HR) performs the worse among all OPT variants partially because of its intensive unrolling computation. OPT (GS) achieves the best testing error on CNN-6/9, while OPT (CP) achieves the best testing error on ResNet-20/34, implying that different OPT encodes different inductive bias.

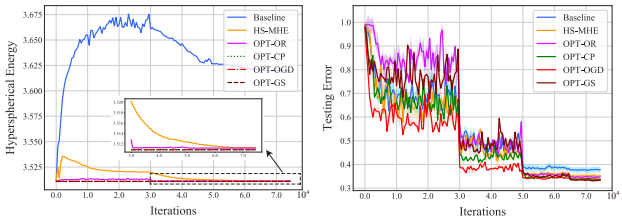


Figure 6: Training dynamics on CIFAR-100. Left: Hyperspherical energy vs. iteration. Right: Testing error vs. iteration.

Training dynamics. We look into how hyperspherical energy and testing error changes in OPT. Fig. 6 shows that the energy of the baseline will increase dramatically at the beginning and then gradually go down, but it still stays in a high value in the end. HS-MHE well reduces the energy at the end of the training. In contrast, OPT variants always maintain very small energy in training. OPT with GS, CP and OGD keep exactly the same energy as the random initialization, while OPT (OR) slightly increases the energy due to relaxation. All OPT variants converge efficiently and stably.

Large-scale learning. To see how OPT performs in large-scale settings, we evaluate OPT on the large-scale ImageNet-2012 [62]. Specifically, we use OPT with OGD and CP to train a plain 10-layer CNN (Appendix D) on ImageNet. Note that, our purpose is to validate the superiority of OPT over the corresponding baseline rather than achieving state-of-the-art results. Table 7 shows that OPT (CP) reduces top-1 and top-5 error for the baseline by $\sim$0.7% and $\sim$0.9%, respectively.

Method	Top-1	Top-5
Baseline	44.32	21.13
Orthogonal [7]	44.13	20.97
HS-MHE [49]	43.92	20.85
OPT (OGD)	43.81	20.49
OPT (CP)	**43.67**	**20.26**

Table 7: ImageNet (%).

Few-shot recognition. For evaluating OPT on cross-task generalization, we perform the few-shot recognition on Mini-ImageNet, following the same setup as [9]. Appendix D gives more detailed settings. We apply OPT with CP to train the baseline and baseline++ in [9], and immediately obtain improvements. Therefore, OPT-trained networks generalize well in this challenging scenario.

Method	5-shot Acc. (%)
MAML [13]	62.71 ± 0.71
MatchingNet [70]	63.48 ± 0.66
ProtoNet [65]	64.24 ± 0.72
Baseline [9]	62.53 ± 0.69
Baseline w/ OPT	**63.27 ± 0.68**
Baseline++ [9]	66.43 ±0.63
Baseline++ w/ OPT	**66.82 ± 0.62**

Table 8: Few-shot learning.

Geometric learning. We apply OPT to graph convolution network (GCN) [37] and point cloud network (PointNet) [57] for graph node and point cloud classification, respectively. The training of GCN and PointNet is conceptually similar to MLP, and the detailed training procedures are given in Appendix D. For GCN, we evaluate OPT on Cora and Pubmed datsets [63]. For PointNet, we conduct experiments on ModelNet-40 dataset [76]. Table 9 shows that OPT effectively improves both GCN and PointNet.

Method	GCN		PointNet
	Cora	Pubmed	MN-40
Baseline	81.3	79.0	87.1
OPT (GS)	81.9	79.4	87.23
OPT (CP)	82.0	79.4	87.81
OPT (OGD)	**82.3**	**79.5**	**87.86**

Table 9: Geometric networks.

6.3. Empirical Evaluation on S-OPT

Method	CIFAR-100				ImageNet	
	CNN-6	Params	Wide CNN-9	Params	ResNet-18	Params
Baseline	37.59	258K	28.03	2.99M	32.95	11.7M
HS-MHE [49]	34.97	258K	25.96	2.99M	32.50	11.7M
OPT (GS)	**33.02**	1.36M	OOM	16.2M	OOM	46.5M
S-OPT (GS)	33.70	**90.9K**	**25.59**	**1.04M**	**32.26**	**3.39M**

Table 10: OPT vs. S-OPT on CIFAR-100 & ImageNet.

Convolutional networks. S-OPT is a scalable OPT variant, and we evaluate its performance in terms of number of *trainable parameters* and testing error. Training parameters are learnable variables in training, and are different from model parameters in testing. In testing, all methods have the same number of model parameters. We perform classification on CIFAR-100 with CNN-6 and wide CNN-9. We also evaluate S-OPT with standard ResNet-18 on ImageNet. Detailed settings are in Appendix D. For S-OPT, we set the sampling dimension as 25% of the original neuron dimension in each layer. Table 10 shows that S-OPT achieves a good trade-off between accuracy and scalability. More importantly, S-OPT can be applied to large neural networks, making OPT more useful in practice. Additionally, Appendix I discusses an efficient parameter sharing for OPT.

Sampling dimensions. We study how the sampling dimension p affect the performance by performing classification with wide CNN-9 on CIFAR-100. In Table 11, $p=d/4$ means that we randomly sample $1/4$ of the original neuron dimension in each layer, so p may vary in different layer. $p=16$ means that we sample 16 dimensions in each layer. Note that there are 25.6K parameters used for the final classification layer, which can not be saved in S-OPT. Table 11 shows that S-OPT can achieve highly competitive accuracy with a reasonably large p.

$p=$	Error (%)	Params
d	OOM	16.2M
$d/4$	**25.59**	1.04M
$d/8$	28.61	278K
$d/16$	32.52	88.7K
16	33.03	27.0K
3	45.22	26.0K
0	60.64	**25.6K**

Table 11: Sampling dim.

6.4. Large Categorical Training

Previously, OPT is not applied to the final classification layer, since it makes little sense to fix random classifiers and learn an orthogonal matrix to transform them. However, learning the classification layer can be costly with large number of classes. The number of trainable parameters of the classification layer grows linearly with the number of classes. To address this, OPT can be used to learn the classification layer, because its number of trainable parameters only depends on the classifier dimension. To be fair, we *only* learn the last classification layer with OPT and the other layers are normally learned (CLS-OPT). The oracle learns the entire network normally. Experimental details are in Appendix D.

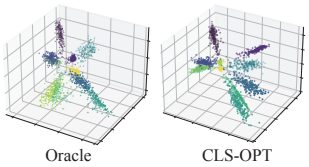


Figure 7: Feature visualization.

We intuitively compare the oracle and CLS-OPT by visualizing the deep MNIST features following [53]. The features are the direct outputs of CNN by setting the output dimension as 3. Fig. 7 show that even if CLS-OPT fixes randomly initialized classifiers, it can still learn discriminative and separable deep features.

Method	ResNet-18A		ResNet-18B	
	Error	Params	Error	Params
Oracle	**18.08**	64.0K	12.12	512K
CLS-OPT	21.12	**8.13K**	**12.05**	**131K**

Table 12: CLS-OPT on ImageNet.

We evaluate its performance on ImageNet with 1K classes. We use ResNet-18 with different output dimensions (A:128, B:512). Table 12 gives the top-5 test error (%) and "Params" denotes the number of trainable parameters in the classification layer. CLS-OPT performs well with far less trainable parameters.

Method	512 Dim.		1024 Dim.	
	Error	Params	Error	Params
Oracle	**95.7**	5.41M	**96.4**	10.83M
CLS-OPT	94.9	**131K**	95.8	**524K**

Table 13: Verification (%) on LFW.

Since face datasets usually contain large number of identities [17], it is natural to apply CLS-OPT to learn face embeddings. We train on CASIA [80] which has 0.5M face images of 10,572 identities, and test on LFW [25]. Since the training and testing sets do not overlap, the task well evaluates the generalizability of learned features. All methods use CNN-20 [52] and standard softmax loss. We set the output feature dimension as 512 or 1024. Table 13 validates CLS-OPT's effectiveness.

7. Concluding Remarks

We propose a novel training framework for neural networks. By parameterizing neurons with weights and a shared orthogonal matrix, OPT can provably achieve small hyperspherical energy and yield superior generalizability.

Acknowledgements. Weiyang Liu and Adrian Weller are supported by DeepMind and the Leverhulme Trust via CFI. Adrian Weller acknowledges support from the Alan Turing Institute under EPSRC grant EP/N510129/1 and U/B/000074. Rongmei Lin and Li Xiong are supported by NSF under CNS-1952192, IIS-1838200.

References

[1] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. *arXiv preprint arXiv:1811.03962*, 2018. 6, 36

[2] Ramesh Naidu Annavarapu. Singular value decomposition and the centrality of löwdin orthogonalizations. *American Journal of Computational and Applied Mathematics*, 3(1):33–35, 2013. 16

[3] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *ICML*, 2016. 2, 4

[4] Nitin Bansal, Xiaohan Chen, and Zhangyang Wang. Can we gain more from orthogonality regularizations in training deep cnns? In *NeurIPS*, 2018. 2, 4, 7

[5] Johann S Brauchart, Alexander B Reznikov, Edward B Saff, Ian H Sloan, Yu Guang Wang, and Robert S Womersley. Random point sets on the sphere—hole radii, covering, and separation. *Experimental Mathematics*, 27(1):62–81, 2018. 42

[6] Anna Breger, Martin Ehler, and Manuel Gräf. Points on manifolds with asymptotically optimal covering radius. *Journal of Complexity*, 48:1–14, 2018. 42

[7] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Neural photo editing with introspective adversarial networks. In *ICLR*, 2017. 2, 4, 7

[8] Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. Entropy-sgd: Biasing gradient descent into wide valleys. In *ICLR*, 2017. 5

[9] Wei-Yu Chen, Yen-Cheng Liu, Zsolt Kira, Yu-Chiang Frank Wang, and Jia-Bin Huang. A closer look at few-shot classification. *arXiv preprint arXiv:1904.04232*, 2019. 8, 20

[10] Jiankang Deng, Jia Guo, Niannan Xue, and Stefanos Zafeiriou. Arcface: Additive angular margin loss for deep face recognition. In *CVPR*, 2019. 2

[11] Xiaohan Ding, Yuchen Guo, Guiguang Ding, and Jungong Han. Acnet: Strengthening the kernel skeletons for powerful cnn via asymmetric convolution blocks. In *ICCV*, 2019. 1, 2

[12] Simon S Du, Jason D Lee, Yuandong Tian, Barnabas Poczos, and Aarti Singh. Gradient descent learns one-hidden-layer cnn: Don't be afraid of spurious local minima. *arXiv preprint arXiv:1712.00779*, 2017. 6, 36

[13] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, 2017. 8

[14] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010. 1, 3, 7

[15] Suriya Gunasekar, Jason D Lee, Daniel Soudry, and Nati Srebro. Implicit bias of gradient descent on linear convolutional networks. In *NeurIPS*, 2018. 1

[16] Suriya Gunasekar, Blake E Woodworth, Srinadh Bhojanapalli, Behnam Neyshabur, and Nati Srebro. Implicit regularization in matrix factorization. In *NeurIPS*, 2017. 1, 6

[17] Yandong Guo, Lei Zhang, Yuxiao Hu, Xiaodong He, and Jianfeng Gao. Ms-celeb-1m: A dataset and benchmark for large-scale face recognition. In *ECCV*, 2016. 8

[18] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016. 40

[19] DP Hardin and EB Saff. Minimal riesz energy point configurations for rectifiable d-dimensional manifolds. *Advances in Mathematics*, 193(1):174–204, 2005. 18

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015. 1, 3, 7

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 6, 7, 20, 21

[22] Mikael Henaff, Arthur Szlam, and Yann LeCun. Recurrent orthogonal networks and long-memory tasks. *arXiv preprint arXiv:1602.06662*, 2016. 4

[23] Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 1997. 5

[24] Walter Hoffmann. Iterative algorithms for gram-schmidt orthogonalization. *Computing*, 41(4):335–348, 1989. 14

[25] Gary B Huang, Marwan Mattar, Tamara Berg, and Eric Learned-Miller. Labeled faces in the wild: A database forstudying face recognition in unconstrained environments. *Technical Report*, 2008. 8

[26] Lei Huang, Li Liu, Fan Zhu, Diwen Wan, Zehuan Yuan, Bo Li, and Ling Shao. Controllable orthogonalization in training dnns. In *CVPR*, 2020. 2

[27] Lei Huang, Xianglong Liu, Bo Lang, Adams Wei Yu, Yongliang Wang, and Bo Li. Orthogonal weight normalization: Solution to optimization over multiple dependent stiefel manifolds in deep neural networks. In *AAAI*, 2018. 2, 4

[28] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015. 7

[29] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. In *UAI*, 2018. 5

[30] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *BMVC*, 2014. 2

[31] Kui Jia, Shuai Li, Yuxin Wen, Tongliang Liu, and Dacheng Tao. Orthogonal deep neural networks. *TPAMI*, 2019. 2

[32] Xu Jia, Bert De Brabandere, Tinne Tuytelaars, and Luc V Gool. Dynamic filter networks. In *NeurIPS*, 2016. 40

[33] Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnns. In *ICML*, 2017. 4

[34] Kenji Kawaguchi. Deep learning without poor local minima. In *NeurIPS*, 2016. 1, 6, 36

[35] Kenji Kawaguchi, Bo Xie, and Le Song. Deep semi-random features for nonlinear function approximation. In *AAAI*, 2018. 37

[36] Nitish Shirish Keskar and Richard Socher. Improving generalization performance by switching from adam to sgd. *arXiv preprint arXiv:1712.07628*, 2017. 1

[37] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 8

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012. 21

[39] N.S. Landkof. *Foundations of modern potential theory*. Springer-Verlag, 1972. 18

[40] Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. Gradient descent only converges to minimizers. In *COLT*, 2016. 36

[41] Mario Lezcano-Casado. Trivializations for gradient-based optimization on manifolds. In *NeurIPS*, 2019. 4

[42] Mario Lezcano-Casado and David Martínez-Rubio. Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group. In *ICML*, 2019. 2, 4

[43] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *NeurIPS*, 2018. 5, 23, 30

[44] Jun Li, Fuxin Li, and Sinisa Todorovic. Efficient riemannian optimization on the stiefel manifold via the cayley transform. In *ICLR*, 2020. 4

[45] Yuanzhi Li, Tengyu Ma, and Hongyang Zhang. Algorithmic regularization in over-parameterized matrix sensing and neural networks with quadratic activations. In *COLT*, 2018. 1, 6

[46] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013. 6, 40

[47] Rongmei Lin, Weiyang Liu, Zhen Liu, Chen Feng, Zhiding Yu, James M. Rehg, Li Xiong, and Le Song. Regularizing neural networks via minimizing hyperspherical energy. In *CVPR*, 2020. 2, 7

[48] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *CVPR*, 2015. 2

[49] Weiyang Liu, Rongmei Lin, Zhen Liu, Lixin Liu, Zhiding Yu, Bo Dai, and Le Song. Learning towards minimum hyperspherical energy. In *NeurIPS*, 2018. 2, 3, 5, 7, 8, 23, 30, 36, 43

[50] Weiyang Liu, Rongmei Lin, Zhen Liu, Li Xiong, Bernhard Schölkopf, and Adrian Weller. Learning with hyperspherical uniformity. In *AISTATS*, 2021. 2, 7

[51] Weiyang Liu, Zhen Liu, James M Rehg, and Le Song. Neural similarity learning. In *NeurIPS*, 2019. 1, 2, 40

[52] Weiyang Liu, Yandong Wen, Zhiding Yu, Ming Li, Bhiksha Raj, and Le Song. Sphereface: Deep hypersphere embedding for face recognition. In *CVPR*, 2017. 2, 8, 22

[53] Weiyang Liu, Yandong Wen, Zhiding Yu, and Meng Yang. Large-margin softmax loss for convolutional neural networks. In *ICML*, 2016. 8

[54] Weiyang Liu, Yan-Ming Zhang, Xingguo Li, Zhiding Yu, Bo Dai, Tuo Zhao, and Le Song. Deep hyperspherical learning. In *NeurIPS*, 2017. 2, 4, 21

[55] Arun Mallya, Dillon Davis, and Svetlana Lazebnik. Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In *ECCV*, 2018. 37

[56] Zakaria Mhammedi, Andrew Hellicar, Ashfaqur Rahman, and James Bailey. Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *ICML*, 2017. 4

[57] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *CVPR*, 2017. 8, 21

[58] Haozhi Qi, Chong You, Xiaolong Wang, Yi Ma, and Jitendra Malik. Deep isometric learning for visual recognition. *arXiv preprint arXiv:2006.16992*, 2020. 2

[59] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *NeurIPS*, 2008. 37

[60] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019. 1

[61] Matthias Reitzner. Stochastical approximation of smooth convex bodies. *Mathematika*, 51(1-2):11–29, 2004. 42

[62] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *IJCV*, 2015. 7

[63] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 2008. 8, 21

[64] Ian H Sloan and Robert S Womersley. Extremal systems of points and numerical integration on the sphere. *Advances in Computational Mathematics*, 21(1-2):107–125, 2004. 43

[65] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *NeurIPS*, 2017. 8

[66] Daniel Soudry and Yair Carmon. No bad local minima: Data independent training error guarantees for multilayer neural networks. *arXiv preprint arXiv:1605.08361*, 2016. 36

[67] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *JMLR*, 2014. 37

[68] Vipin Srivastava. A unified view of the orthogonalization methods. *Journal of Physics A: Mathematical and General*, 33(35):6219, 2000. 16

[69] Yun Tang, Jing Huang, Guangtao Wang, Xiaodong He, and Bowen Zhou. Orthogonal relation transforms with graph context modeling for knowledge graph embedding. In *ACL*, 2020. 4

[70] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *NeurIPS*, 2016. 8

[71] Feng Wang, Weiyang Liu, Haijun Liu, and Jian Cheng. Additive margin softmax for face verification. *arXiv preprint arXiv:1801.05599*, 2018. 2

[72] Hao Wang, Yitong Wang, Zheng Zhou, Xing Ji, Dihong Gong, Jingchao Zhou, Zhifeng Li, and Wei Liu. Cosface: Large margin cosine loss for deep face recognition. In *CVPR*, 2018. 2

[73] Min Wang, Baoyuan Liu, and Hassan Foroosh. Factorized convolutional neural networks. In *ICCV Workshops*, 2017. 2

[74] Zaiwen Wen and Wotao Yin. A feasible method for optimization with orthogonality constraints. *Mathematical Programming*, 2013. 4

[75] Scott Wisdom, Thomas Powers, John Hershey, Jonathan Le Roux, and Les Atlas. Full-capacity unitary recurrent neural networks. In *NeurIPS*, 2016. 2, 4

[76] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *CVPR*, 2015. 8, 21

[77] Bo Xie, Yingyu Liang, and Le Song. Diverse neural network learns true target functions. In *AISTATS*, 2017. 6, 36, 37

[78] Di Xie, Jiang Xiong, and Shiliang Pu. All you need is beyond a good init: Exploring better solution for training extremely deep convolutional neural networks with orthonormality and modulation. In *CVPR*, 2017. 2, 4

[79] Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. In *ICCV*, 2015. 2

[80] Dong Yi, Zhen Lei, Shengcai Liao, and Stan Z Li. Learning face representation from scratch. *arXiv preprint arXiv:1411.7923*, 2014. 8

Appendix

Table of Contents

A. Details of Unrolled Orthogoanlization Algorithms

A.1. Gram-Schmidt Process

Gram-Schmidt Process. GS process is a method for orthonormalizing a set of vectors in an inner product space, *i.e.*, the Euclidean space $\mathbb{R}^n$ equipped with the standard inner product. Specifically, GS process performs the following operations to orthogonalize a set of vectors $\{\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n\} \in \mathbb{R}^{n\times n}$:

$$
\begin{aligned}
&\text{Step 1: } \tilde{\boldsymbol{e}}_1 = \boldsymbol{u}_1, \quad \boldsymbol{e}_1 = \frac{\tilde{\boldsymbol{e}}_1}{\|\tilde{\boldsymbol{e}}_1\|} \\
&\text{Step 2: } \tilde{\boldsymbol{e}}_2 = \boldsymbol{u}_2 - \text{Proj}_{\tilde{e}_1}(\boldsymbol{u}_2), \quad \boldsymbol{e}_2 = \frac{\tilde{\boldsymbol{e}}_2}{\|\tilde{\boldsymbol{e}}_2\|} \\
&\text{Step 3: } \tilde{\boldsymbol{e}}_3 = \boldsymbol{u}_3 - \text{Proj}_{\tilde{\boldsymbol{e}}_1}(\boldsymbol{u}_3) - \text{Proj}_{\tilde{\boldsymbol{e}}_2}(\boldsymbol{u}_3), \quad \boldsymbol{e}_3 = \frac{\tilde{\boldsymbol{e}}_3}{\|\tilde{\boldsymbol{e}}_3\|} \\
&\text{Step 4: } \tilde{\boldsymbol{e}}_4 = \boldsymbol{u}_4 - \text{Proj}_{\tilde{\boldsymbol{e}}_1}(\boldsymbol{u}_4) - \text{Proj}_{\tilde{\boldsymbol{e}}_2}(\boldsymbol{u}_4) - \text{Proj}_{\tilde{\boldsymbol{e}}_3}(\boldsymbol{u}_4), \quad \boldsymbol{e}_4 = \frac{\tilde{\boldsymbol{e}}_4}{\|\tilde{\boldsymbol{e}}_4\|} \\
&\qquad \vdots \\
&\text{Step n: } \tilde{\boldsymbol{e}}_n = \boldsymbol{u}_n - \text{Proj}_{\tilde{\boldsymbol{e}}_1}(\boldsymbol{u}_n) - \text{Proj}_{\tilde{\boldsymbol{e}}_2}(\boldsymbol{u}_n) - \text{Proj}_{\tilde{\boldsymbol{e}}_3}(\boldsymbol{u}_n) - \cdots - \text{Proj}_{\tilde{\boldsymbol{e}}_{n-1}}(\boldsymbol{u}_n), \quad \boldsymbol{e}_n = \frac{\tilde{\boldsymbol{e}}_n}{\|\tilde{\boldsymbol{e}}_n\|}
\end{aligned}
\tag{6}
$$

where $\text{Proj}_{\boldsymbol{a}}(\boldsymbol{b}) = \frac{\langle \boldsymbol{a}, \boldsymbol{b}\rangle}{\langle \boldsymbol{a}, \boldsymbol{a}\rangle}\boldsymbol{a}$ denotes the projection of the vector $\boldsymbol{b}$ onto the vector $\boldsymbol{a}$. The set $\{\boldsymbol{e}_1, \boldsymbol{e}_2, \cdots, \boldsymbol{e}_n\}$ denotes the output orthonormal set. The algorithm flowchart can be described as follows:

Algorithm 2 Gram-Schmidt Process

Input: $\boldsymbol{U} = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n\} \in \mathbb{R}^{n\times n}$
Output: $\boldsymbol{R} = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \cdots, \boldsymbol{e}_n\} \in \mathbb{R}^{n\times n}$
$\boldsymbol{R} = \boldsymbol{0}$
for $j = 1, 2, \cdots, n$ **do**
  $\boldsymbol{q}_j = \boldsymbol{R}^\top \boldsymbol{u}_j$
  $\boldsymbol{t} = \boldsymbol{u}_j - \boldsymbol{R}\boldsymbol{q}_j$
  $\boldsymbol{q}_{jj} = \|\boldsymbol{t}\|_2$
  $\boldsymbol{e}_j = \frac{\boldsymbol{t}}{\boldsymbol{q}_{jj}}$
end

The vectors $\boldsymbol{q}_j, \forall j$ in the algorithm above are used to compute the QR factorization, which is not useful in orthogonalization and therefore does not need to be stored. When the GS process is implemented on a finite-precision computer, the vectors $\boldsymbol{e}_j, \forall j$ are often not quite orthogonal, because of rounding errors. Besides the standard GS process, there is a modified Gram-Schmidt (MGS) algorithm which enjoys better numerical stability. This approach gives the same result as the original formula in exact arithmetic and introduces smaller errors in finite-precision arithmetic. Specifically, GS computes the following formula:

$$
\begin{aligned}
\tilde{\boldsymbol{e}}_j &= \boldsymbol{u}_j - \sum_{k=1}^{j-1} \text{Proj}_{\tilde{\boldsymbol{e}}_k}(\boldsymbol{u}_j) \\
\boldsymbol{e}_j &= \frac{\tilde{\boldsymbol{e}}_j}{\|\tilde{\boldsymbol{e}}_j\|}
\end{aligned}
\tag{7}
$$

Instead of computing the vector $\boldsymbol{e}_j$ as in Eq. 7, MGS computes the orthogonal basis differently. MGS does not subtract the projections of the original vector set, and instead remove the projection of the previously constructed orthogonal basis.

Specifically, MGS computes the following series of formulas:

$$\begin{aligned}
\tilde{\boldsymbol{e}}_j^{(1)} &= \boldsymbol{u}_j - \text{Proj}_{\tilde{\boldsymbol{e}}_1}(\boldsymbol{u}_j) \\
\tilde{\boldsymbol{e}}_j^{(2)} &= \tilde{\boldsymbol{e}}_j^{(1)} - \text{Proj}_{\tilde{\boldsymbol{e}}_2}(\tilde{\boldsymbol{e}}_j^{(1)}) \\
&\vdots \\
\tilde{\boldsymbol{e}}_j^{(j-2)} &= \tilde{\boldsymbol{e}}_j^{(j-3)} - \text{Proj}_{\tilde{\boldsymbol{e}}_2}(\tilde{\boldsymbol{e}}_j^{(j-3)}) \\
\tilde{\boldsymbol{e}}_j^{(j-1)} &= \tilde{\boldsymbol{e}}_j^{(j-2)} - \text{Proj}_{\tilde{\boldsymbol{e}}_2}(\tilde{\boldsymbol{e}}_j^{(j-2)}) \\
\boldsymbol{e}_j &= \frac{\tilde{\boldsymbol{e}}_j^{(j-1)}}{\left\|\tilde{\boldsymbol{e}}_j^{(j-1)}\right\|}
\end{aligned} \tag{8}$$

where each step finds a vector $\tilde{\boldsymbol{e}}_j^{(i)}$ that is orthogonal to $\tilde{\boldsymbol{e}}_j^{(i-1)}$. Therefore, $\tilde{\boldsymbol{e}}_j^{(i)}$ is also orthogonalized against any errors brought by the computation of $\tilde{\boldsymbol{e}}_j^{(i-1)}$. In practice, although MGS enjoys better numerical stability, we find the empirical performance of GS and MGS is almost the same in OPT. However, MGS takes longer time to complete since the computation of each orthogonal basis is an iterative process. Therefore, we usually stick to classic GS for OPT.

Iterative Gram-Schmidt Process. Iterative Gram-Schmidt (IGS) process is an iterative version of the GS process. It is shown in [24] that GS process can be carried out iteratively to obtain a basis matrix that is orthogonal in almost full working precision. The IGS algorithm is given as follows:

Algorithm 3 Iterative Gram-Schmidt Process

Input: $\boldsymbol{U} = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n\} \in \mathbb{R}^{n\times n}$
Output: $\boldsymbol{R} = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \cdots, \boldsymbol{e}_n\} \in \mathbb{R}^{n\times n}$
$\boldsymbol{R} = \boldsymbol{0}$
for $j = 1, 2, \cdots, n$ **do**
  $\boldsymbol{q}_j = \boldsymbol{0}$
  $\boldsymbol{t} = \boldsymbol{u}_j$
  while $\boldsymbol{t} \perp \text{span}(\boldsymbol{e}_1, \cdots, \boldsymbol{e}_{j-1})$ *is False* **do**
    $\boldsymbol{p} = \boldsymbol{t}$
    $\boldsymbol{s} = \boldsymbol{R}^\top \boldsymbol{p}$
    $\boldsymbol{v} = \boldsymbol{R}\boldsymbol{s}$
    $\boldsymbol{t} = \boldsymbol{p} - \boldsymbol{v}$
    $\boldsymbol{q}_j \leftarrow \boldsymbol{q}_j + \boldsymbol{s}$
  end
  $\boldsymbol{q}_{jj} = \|\boldsymbol{t}\|_2$
  $\boldsymbol{e}_j = \frac{\boldsymbol{t}}{\boldsymbol{q}_{jj}}$
end

The vectors $\boldsymbol{q}_j, \forall j$ in the algorithm above are used to compute the QR factorization, which is not useful in orthogonalization and therefore does not need to be explicitly computed. The while loop in IGS is an iterative procedure. In practice, we can unroll a fixed number of steps for the while loop in order to improve the orthogonality. The resulting $\boldsymbol{q}_j$ in the j-th step corresponds to the solution of the equation $\tilde{\boldsymbol{R}}^\top \tilde{\boldsymbol{R}} \boldsymbol{q}_j = \tilde{\boldsymbol{R}}^\top \boldsymbol{u}_j$ where $\tilde{\boldsymbol{R}} = \{\boldsymbol{e}_1, \cdots, \boldsymbol{e}_{j-1}\}$. The IGS process corresponds to the Gauss-Jacobi iteration for solving this equation.

Both GS and IGS are easy to be embedded in the neural networks, since they are both differentiable. In our experiments, we find that the performance gain of unrolling multiple steps in IGS over GS is not very obvious (partially because GS has already achieved nearly perfect orthogonality), but IGS costs longer training time. Therefore, we unroll the classic GS process by default.

A.2. Householder Reflection

Let $\boldsymbol{v} \in \mathbb{R}^n$ be a non-zero vector. A matrix $\boldsymbol{H} \in \mathbb{R}^{n\times n}$ of the form

$$\boldsymbol{H} = \boldsymbol{I} - \frac{2\boldsymbol{v}\boldsymbol{v}^\top}{\boldsymbol{v}^\top \boldsymbol{v}} \tag{9}$$

is a Householder reflection. The vector $\boldsymbol{v}$ is the Householder vector. If a vector $\boldsymbol{x}$ is multiplied by the matrix $\boldsymbol{H}$, then it will be reflected in the hyperplane $\text{span}(\boldsymbol{v})^{\perp}$. Householder matrices are symmetric and orthogonal.

For a vector $\boldsymbol{x} \in \mathbb{R}^n$, we let $\boldsymbol{v} = \boldsymbol{x} \pm \|\boldsymbol{x}\|_2 \boldsymbol{e}_1$ where $\boldsymbol{e}_1$ is a vector of $\{1, 0, \cdots, 0\}$ (the first element is 1 and the remaining elements are 0). Then we construct the Householder reflection matrix with $\boldsymbol{v}$ and multiply it to $\boldsymbol{x}$:

$$\boldsymbol{H}\boldsymbol{x} = \left(\boldsymbol{I} - 2\frac{\boldsymbol{v}\boldsymbol{v}^\top}{\boldsymbol{v}^\top\boldsymbol{v}}\right)\boldsymbol{x} = \mp\|\boldsymbol{x}\|_2 \boldsymbol{e}_1 \tag{10}$$

which indicates that we can make any non-zero vector become $\alpha\boldsymbol{e}_1$ where α is some constant by using Householder reflection. By left-multiplying a reflection we can turn a dense vector $\boldsymbol{x}$ into a vector with the same length and with only a single nonzero entry. Repeating this n times gives us the Householder QR factorization, which also orthogonalizes the original input matrix. Householder reflection orthogonalizes a matrix $\boldsymbol{U} = \{\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n\}$ by triangularizing it:

$$\boldsymbol{U} = \boldsymbol{H}_1\boldsymbol{H}_2\cdots\boldsymbol{H}_n\boldsymbol{R} \tag{11}$$

where $\boldsymbol{R}$ is a upper-triangular matrix in the QR factorization. $\boldsymbol{H}_j, j \geq 2$ is constructed by $\text{Diag}(\boldsymbol{I}_{j-1}, \tilde{\boldsymbol{H}}_{n-j+1})$ where $\tilde{\boldsymbol{H}}_{n-j+1} \in \mathbb{R}^{(n-j+1)\times(n-j+1)}$ is the Householder reflection that is performed on the vector $\boldsymbol{U}_{(j:n,j)}$. The algorithm flowchart is given as follows:

Algorithm 4 Householder Reflection Orthogonalization

Input: $\boldsymbol{U} = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n\} \in \mathbb{R}^{n\times n}$

Output: $\boldsymbol{U} = \boldsymbol{Q}\boldsymbol{R}$, where $\boldsymbol{Q} = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \cdots, \boldsymbol{e}_n\} \in \mathbb{R}^{n\times n}$ is the orthogonal matrix and $\boldsymbol{R} \in \mathbb{R}^{n\times n}$ is a upper triangular matrix

for $j = 1, 2, \cdots, n-1$ **do**
  $\{\boldsymbol{v}, \beta\} = \texttt{Householder}(\boldsymbol{U}_{j:n,j})$
  $\boldsymbol{U}_{j:n,j:n} \leftarrow \boldsymbol{U}_{j:n,j:n} - \beta\boldsymbol{v}(\boldsymbol{v}^\top\boldsymbol{U}_{j:n,j:n})$
  $\boldsymbol{U}_{j+1:n,j} \leftarrow \boldsymbol{v}_{(2:\text{end})}$
end
function $\{\boldsymbol{v}, \beta\} = \texttt{Householder}(\boldsymbol{x})$
  $\sigma^2 = \|\boldsymbol{x}_{2:\text{end}}\|_2^2$
  $\boldsymbol{v} \leftarrow \begin{bmatrix} 1 \\ \boldsymbol{x}_{2:\text{end}} \end{bmatrix}$
  if $\sigma^2 = 0$ **then**
    $\beta = 0$
  else
    if $\boldsymbol{x}_1 \leq 0$ **then**
      $\boldsymbol{v}_1 = \boldsymbol{x}_1 - \sqrt{\boldsymbol{x}_1^2 + \sigma^2}$
    else
      $\boldsymbol{v}_1 = -\dfrac{\sigma^2}{\boldsymbol{x}_1 + \sqrt{\boldsymbol{x}_1^2 + \sigma^2}}$
    end
    $\beta = \frac{2\boldsymbol{v}_1^2}{\sigma^2 + \boldsymbol{v}_1^2}$
    $\boldsymbol{v} \leftarrow \frac{\boldsymbol{v}}{\boldsymbol{v}_1}$
  end
end function

The algorithm follows the Matlab notation where $\boldsymbol{U}_{j:n,j:n}$ denotes the submatrix of $\boldsymbol{U}$ from the j-th column to the n-th column and from the j-th row to the n-th row. Note that, there are a number of variants for the Householder reflection orthogonalization, such as the implicit variant where we do not store each reflection $\boldsymbol{H}_j$ explicitly. Here $\boldsymbol{Q}$ is the final orthogonal matrix we need.

A.3. Löwdin's Symmetric Orthogonalization

Let $\boldsymbol{U} = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n\}$ be a set of linearly independent vectors in a n-dimensional space. We define a general non-singular linear transformation $\boldsymbol{A}$ that can transform the basis $\boldsymbol{U}$ to a new basis $\boldsymbol{R}$:

$$\boldsymbol{R} = \boldsymbol{U}\boldsymbol{A} \tag{12}$$

where the basis $\boldsymbol{R}$ will be orthonormal if (the transpose will become conjugate transpose in complex space)

$$\boldsymbol{R}^\top \boldsymbol{R} = (\boldsymbol{U}\boldsymbol{A})^\top (\boldsymbol{U}\boldsymbol{A}) = \boldsymbol{A}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{A} = \boldsymbol{A}^\top \boldsymbol{M}\boldsymbol{A} = \boldsymbol{I} \tag{13}$$

where $\boldsymbol{M} = \boldsymbol{U}^\top \boldsymbol{U}$ is the gram matrix of the given basis $\boldsymbol{U}$.

A general solution to this orthogonalization problem can be obtained via the substitution:

$$\boldsymbol{A} = \boldsymbol{M}^{-1}\boldsymbol{B} \tag{14}$$

in which $\boldsymbol{B}$ is an arbitrary orthogonal (or unitary) matrix. When $\boldsymbol{B} = \boldsymbol{I}$, we will have the symmetric orthogonalization, namely

$$\boldsymbol{R} := \boldsymbol{\Phi} = \boldsymbol{U}\boldsymbol{M}^{-\frac{1}{2}} \tag{15}$$

When $\boldsymbol{B} = \boldsymbol{V}$ in which $\boldsymbol{V}$ diagonalizes $\boldsymbol{M}$, then we have the canonical orthogonalization, namely

$$\boldsymbol{\Lambda} = \boldsymbol{U}\boldsymbol{V}\boldsymbol{d}^{-\frac{1}{2}}. \tag{16}$$

Because $\boldsymbol{V}$ diagonalizes $\boldsymbol{M}$, we have that $\boldsymbol{M} = \boldsymbol{V}\boldsymbol{d}\boldsymbol{V}^\top$. Therefore, we have the $\boldsymbol{M}^{-\frac{1}{2}}$ transformation as $\boldsymbol{M}^{-\frac{1}{2}} = \boldsymbol{V}\boldsymbol{d}^{-\frac{1}{2}}\boldsymbol{V}^\top$. This is essentially an eigenvalue decomposition of the symmetric matrix $\boldsymbol{M} = \boldsymbol{U}^\top \boldsymbol{U}$.

In order to compute the Löwdin's symmetric orthogonalized basis sets, we can use singular value decomposition. Specifically, SVD of the original basis set $\boldsymbol{U}$ is given by

$$\boldsymbol{U} = \boldsymbol{W}\boldsymbol{\Sigma}\boldsymbol{V}^\top \tag{17}$$

where both $\boldsymbol{W} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{U} \in \mathbb{R}^{n\times n}$ are orthogonal matrices. $\boldsymbol{\Sigma}$ is the diagonal matrix of singular values. Therefore, we have that

$$\begin{aligned}\boldsymbol{R} &= \boldsymbol{U}\boldsymbol{M}^{-\frac{1}{2}} \\ &= \boldsymbol{W}\boldsymbol{\Sigma}\boldsymbol{V}^\top \boldsymbol{V}\boldsymbol{d}^{-\frac{1}{2}}\boldsymbol{V}^\top \\ &= \boldsymbol{W}\boldsymbol{\Sigma}\boldsymbol{d}^{-\frac{1}{2}}\boldsymbol{V}^\top\end{aligned} \tag{18}$$

where we have $\boldsymbol{\Sigma} = \boldsymbol{d}^{\frac{1}{2}}$ due to the connections between eigenvalue decomposition and SVD. Therefore, we end up with

$$\boldsymbol{R} = \boldsymbol{W}\boldsymbol{V}^\top \tag{19}$$

which is the output orthogonal matrix for Löwdin's symmetric orthogonalization.

An interesting feature of the symmetric orthogonalization is to ensure that

$$\boldsymbol{R} = \arg\min_{\boldsymbol{P}\in \text{orth}(\boldsymbol{U})} \sum_i \|\boldsymbol{P}_i - \boldsymbol{U}_i\| \tag{20}$$

where $\boldsymbol{P}_i$ and $\boldsymbol{U}_i$ are the i-th column vectors of $\boldsymbol{P} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{U}$, respectively. $\text{orth}(\boldsymbol{U})$ denotes the set of all possible orthonormal sets in the range of $\boldsymbol{U}$. This means that the symmetric orthogonalization functions $\boldsymbol{R}_i$ (or $\boldsymbol{\Phi}_i$) are the least distant in the Hilbert space from the original functions $\boldsymbol{U}_i$. Therefore, symmetric orthogonalization indicates the gentlest pushing of the directions of the vectors in order to make them orthogonal.

More interestingly, the symmetric orthogonalized basis sets has unique geometric properties [68, 2] if we consider the Schweinler-Wigner matrix in terms of the sum of squared projections.

B. Proof of Theorem 1

To be more specific, neurons with each element initialized by a zero-mean Gaussian distribution are uniformly distributed on a hypersphere. We show this argument with the following theorem.

Theorem 3. *The normalized vector of Gaussian variables is uniformly distributed on the sphere. Formally, let $x_1, x_2, \cdots, x_n \sim \mathcal{N}(0,1)$ and be independent. Then the vector*

$$\boldsymbol{x} = \left[\frac{x_1}{z}, \frac{x_2}{z}, \cdots, \frac{x_n}{z}\right] \tag{21}$$

follows the uniform distribution on $\mathbb{S}^{n-1}$, where $z = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$ is a normalization factor.

Proof. A random variable has distribution $\mathcal{N}(0,1)$ if it has the density function

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}. \tag{22}$$

A n-dimensional random vector $\boldsymbol{x}$ has distribution $\mathcal{N}(0,1)$ if the components are independent and have distribution $\mathcal{N}(0,1)$ each. Then the density of $\boldsymbol{x}$ is given by

$$f(x) = \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}\langle x, x\rangle}. \tag{23}$$

Then we introduce the following lemma (Lemma 1) about the orthogonal-invariance of the normal distribution.

Lemma 1. *Let $\boldsymbol{x}$ be a n-dimensional random vector with distribution $\mathcal{N}(0,1)$ and $\boldsymbol{U} \in \mathbb{R}^{n\times n}$ be an orthogonal matrix ($\boldsymbol{U}\boldsymbol{U}^\top = \boldsymbol{U}^\top\boldsymbol{U} = \boldsymbol{I}$). Then $\boldsymbol{Y} = \boldsymbol{U}\boldsymbol{x}$ also has the distribution of $\mathcal{N}(0,1)$.*

Proof. For any measurable set $A \subset \mathbb{R}^n$, we have that

$$\begin{aligned} P(Y \in A) &= P(X \in U^\top A) \\ &= \int_{U^\top A} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}\langle x, x\rangle} \\ &= \int_A \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}\langle Ux, Ux\rangle} \\ &= \int_A \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}\langle x, x\rangle} \end{aligned} \tag{24}$$

because of orthogonality of U. Therefore the lemma holds. □

Because any rotation is just a multiplication with some orthogonal matrix, we know that normally distributed random vectors are invariant to rotation. As a result, generating $\boldsymbol{x} \in \mathbb{R}^n$ with distribution $\mathbb{N}(0,1)$ and then projecting it onto the hypersphere $\mathbb{S}^{n-1}$ produces random vectors $U = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}$ that are uniformly distributed on the hypersphere. Therefore the theorem holds. □

Then we show the normalized vector $\boldsymbol{y}$ where each element follows a zero-mean Gaussian distribution with some constant variance σ^2:

$$\boldsymbol{y} = \left[\frac{y_1}{r}, \frac{y_2}{r}, \cdots, \frac{y_n}{r}\right] \tag{25}$$

where $r = \sqrt{y_1^2 + y_2^2 + \cdots + y_n^2}$. Because we have that $\frac{y_i}{\sigma} \sim \mathcal{N}(0,1)$, we can rewrite $\boldsymbol{y}$ as the following random vector:

$$\boldsymbol{y} = \left[\frac{y_1/\sigma}{r/\sigma}, \frac{y_2/\sigma}{r/\sigma}, \cdots, \frac{y_n/\sigma}{r/\sigma}\right] \tag{26}$$

where $r/\sigma = \sqrt{(y_1/\sigma)^2 + (y_2/\sigma)^2 + \cdots + (y_n/\sigma)^2}$. Therefore, we directly can apply Theorem 3 and conclude that $\boldsymbol{y}$ also follows the uniform distribution on $\mathbb{S}^{n-1}$. Now we obtain that any random vector with each element following a zero-mean Gaussian distribution with some constant variance follows the uniform distribution on $\mathbb{S}^{n-1}$.

Then we show that the minimum hyperspherical energy asymptotically corresponds to the uniform distribution over the unit hypersphere. We first write down the hyperspherical energy of N neurons $\{\boldsymbol{w}_1, \cdots, \boldsymbol{w}_N \in \mathbb{R}^{d+1}\}$ (we also define that $\hat{\boldsymbol{w}}_i = \frac{\boldsymbol{w}_i}{\|\boldsymbol{w}_i\|} \in \mathbb{S}^d$):

$$\boldsymbol{E}_{s,d}(\hat{\boldsymbol{w}}_i|_{i=1}^N) = \sum_{i=1}^{N} \sum_{j=1, j\neq i}^{N} f_s\big(\|\hat{\boldsymbol{w}}_i - \hat{\boldsymbol{w}}_j\|\big) = \begin{cases} \sum_{i\neq j} \|\hat{\boldsymbol{w}}_i - \hat{\boldsymbol{w}}_j\|^{-s}, & s > 0 \\ \sum_{i\neq j} \log\big(\|\hat{\boldsymbol{w}}_i - \hat{\boldsymbol{w}}_j\|^{-1}\big), & s = 0 \end{cases} \tag{27}$$

where s is a hyperparameter that controls the behavior of hyperspherical energy. We then define a N-point minimal hyperspherical s-energy over $\boldsymbol{A}$ with

$$\varepsilon_{s,d}(\boldsymbol{A}, \hat{\boldsymbol{W}}_N) := \inf_{\hat{\boldsymbol{W}}_N \subset \boldsymbol{A}} \boldsymbol{E}_{s,d}(\hat{\boldsymbol{w}}_i|_{i=1}^N) \tag{28}$$

where we denote that $\hat{\boldsymbol{W}}_N = \{\hat{\boldsymbol{w}}_1, \cdots, \hat{\boldsymbol{w}}_N\}$. Typically, we will assume that $\boldsymbol{A}$ is compact. Based on [19], we discuss the asymptotic behavior (as $N \to \infty$) of $\varepsilon_{s,d}(\boldsymbol{A}, \hat{\boldsymbol{W}}_N)$ in three different scenarios: (1) $0 < s < d$; (2) $s = d$; and $s > d$. The reason behind is the behavior of the following energy integral:

$$I_s(\mu) = \iint_{\mathbb{S}^d \times \mathbb{S}^d} \|\boldsymbol{u} - \boldsymbol{v}\|^{-s} d\mu(\boldsymbol{u}) d\mu(\boldsymbol{v}), \tag{29}$$

is quite different under these three scenarios. In scenario (1), Eq. 29 that is taken over all probability measures μ supported on $\mathcal{S}^d$ will be minimal for normalized Lebesgue measure $\frac{\mathcal{H}_d(\cdot)|_{\mathbb{S}^d}}{\mathcal{H}_d(\mathbb{S}^d)}$ on $\mathbb{S}^d$. In the case of $s \geq d$, we will have that $I_s(\mu)$ is positive infinity for all such measures μ. Therefore, the behaviour of the minimum hyperspherical energy is different in these three cases. In general, as the parameter s increases, there is a transition from the global effects to the more local influences (from nearest neighbors). The transition happens when $s = d$. However, we typically have $0 < s < d$ in the neural networks. Therefore, we will mostly study the case of $0 < s < d$ and the theoretical asymptotic behavior is quite standard results from the potential theory [39]. From the classic potential theory, we have the following known lemma:

Lemma 2. *If $0 < s < d$, we have that*

$$\lim_{N\to\infty} \frac{\varepsilon_{s,d}(\boldsymbol{S}^d, \hat{\boldsymbol{W}}_N)}{N^2} = I_s\left(\frac{\mathcal{H}_d(\cdot)|_{\mathbb{S}^d}}{\mathcal{H}_d(\mathbb{S}^d)}\right) \tag{30}$$

Moreover, any sequence of s-energy configuration of minimal hyperspherical energy ($(\hat{\boldsymbol{W}}_N^*)_2^\infty \subset \mathbb{S}^d$) is asymptotically uniformly distributed in the sense that for the weak-star topology of measures,

$$\frac{1}{N} \sum_{\boldsymbol{v} \in \hat{\boldsymbol{W}}_N^*} \delta_{\boldsymbol{v}} \to \frac{\mathcal{H}_d(\cdot)|_{\mathbb{S}^d}}{\mathcal{H}_d(\mathbb{S}^d)} \quad \text{as } N \to \infty \tag{31}$$

where $\delta_{\boldsymbol{v}}$ denotes the unit point mass at $\boldsymbol{v}$.

The lemma above concludes that the neuron configuration with minimal hyperspherical energy asymptotically corresponds to the uniform distribution on $\mathbb{S}^d$ when $0 < s < d$. From [19], we also have the following lemma that shows the same conclusion holds for the the case of $s = d$ and $s > d$:

Lemma 3. *Let $\mathcal{B}^d := \bar{B}(0, 1)$ denote the closed unit ball in $\mathbb{R}^d$. For the case of $s = d$, we have that*

$$\lim_{N\to\infty} \frac{\varepsilon_{s,d}(\boldsymbol{S}^d, \hat{\boldsymbol{W}}_N)}{N^2 \log N} = \frac{\mathcal{H}_d(\mathcal{B}^d)}{\mathcal{H}_d(\mathbb{S}^d)} = \frac{1}{d} \frac{\Gamma(\frac{d+1}{2})}{\sqrt{\pi}\Gamma(\frac{d}{2})} \tag{32}$$

and any sequence $(\hat{\boldsymbol{W}}_N^*) \subset \mathbb{S}^d$ of minimal s-energy configurations satisfies Eq. 31.

The lemma above shows that the same conclusion holds for $s = d$. For the case of $s > d$, the theoretical analysis is more involved, but the conclusion that the neuron configuration with minimal hyperspherical energy asymptotically corresponds to the uniform distribution on $\mathbb{S}^d$ still holds. Note that, we usually will not have the case of $s > d$ in our applications.

C. Proof of Theorem 2

We consider a set of n d-dimensional neurons $\boldsymbol{W} = \{\boldsymbol{w}_1, \cdots, \boldsymbol{w}_n\} \in \mathbb{R}^{d \times n}$. The hyperspherical energy of the original set of neurons can be written as:

$$\boldsymbol{E}(\boldsymbol{w}_i|_{i=1}^n) = \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} \left\| \frac{\boldsymbol{w}_i}{\|\boldsymbol{w}_i\|_2} - \frac{\boldsymbol{w}_j}{\|\boldsymbol{w}_j\|} \right\|^{-1} \tag{33}$$

which means that if the pairwise angle between any two neurons stays unchanged, then the hyperspherical energy will also stay unchanged. Now we consider the cosine value of the angle $\theta_{(\boldsymbol{w}_i, \boldsymbol{w}_j)}$ between any two neuron $\boldsymbol{w}_i$ and $\boldsymbol{w}_j$:

$$\cos(\theta_{(\boldsymbol{w}_i, \boldsymbol{w}_j)}) = \frac{\boldsymbol{w}_i^\top \boldsymbol{w}_j}{\|\boldsymbol{w}_i\| \cdot \|\boldsymbol{w}_j\|} = \frac{\sum_{k=1}^{d} w_{ik} \cdot w_{jk}}{\|\boldsymbol{w}_i\| \cdot \|\boldsymbol{w}_j\|} \tag{34}$$

where w_{ik} is the k-th element of the neuron $\boldsymbol{w}_i$. From the equation above, we can observe that permuting the order of the elements in the neurons together will not change the angle. For example, switching the i-th and j-th element in all the neurons will not change the hyperspherical energy. Assume that we randomly select p dimensions from the d dimensions and denote the set of p dimension as $\boldsymbol{s} = \{s_1, \cdots, s_p\} \in \mathbb{R}^p$. Therefore we can construct a new set of neurons $\tilde{\boldsymbol{W}} = \{\tilde{\boldsymbol{w}}_1, \cdots, \tilde{\boldsymbol{w}}_n\}$ by permuting the p dimensions in $\boldsymbol{s}$ to become the first p elements for all the neurons. Essentially, we use permutation to make $\tilde{\boldsymbol{w}}_i = \boldsymbol{w}_{s_i}$ for $i \in [1, p]$. Therefore, we can have the following equation:

$$\boldsymbol{E}(\boldsymbol{w}_i|_{i=1}^n) = \boldsymbol{E}(\tilde{\boldsymbol{w}}_i|_{i=1}^n) \tag{35}$$

Then we consider an orthogonal matrix $\boldsymbol{R}_p \in \mathbb{R}^{d \times d}$ that is used to transform the p dimension in the neurons. The equivalent orthogonal transformation for the d-dimensional neurons $\tilde{\boldsymbol{W}}$ is

$$\tilde{\boldsymbol{R}} = \begin{bmatrix} \boldsymbol{R}_p & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_{n-p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}_p & 0 & \cdots & 0 \\ 0 & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix} \tag{36}$$

where $\boldsymbol{I}_{n-p}$ is an identity matrix of size $(n-p) \times (n-p)$. It is easy to verify that $\tilde{\boldsymbol{R}}$ is also an orthogonal matrix: $\tilde{\boldsymbol{R}}^\top \tilde{\boldsymbol{R}} = \boldsymbol{I}_n$. Then we permute the order of $\tilde{\boldsymbol{W}}$ back to the original neuron set $\boldsymbol{W}$ and obtain a new set of neurons $\boldsymbol{W}^t = \{\boldsymbol{w}_1^t, \cdots, \boldsymbol{w}_n^t\}$. $\boldsymbol{W}^t$ is in fact the result of directly performing orthogonal transformation to the p dimensions in $\boldsymbol{W}$. Because any order permutation of elements in neurons does not change the hyperspherical energy, we have the following equation

$$\boldsymbol{E}(\boldsymbol{w}_i|_{i=1}^n) = \boldsymbol{E}(\tilde{\boldsymbol{w}}_i|_{i=1}^n) = \boldsymbol{E}(\tilde{\boldsymbol{R}}\tilde{\boldsymbol{w}}_i|_{i=1}^n) = \boldsymbol{E}(\tilde{\boldsymbol{R}}\boldsymbol{w}_i^t|_{i=1}^n) \tag{37}$$

which concludes our proof.

D. Experimental Settings

Layer	CNN-6 (CIFAR-100)	CNN-9 (CIFAR-100)	CNN-10 (ImageNet-2012)
Conv1.x	[3×3, 64]×2	[3×3, 64]×3	[7×7, 64], Stride 2 3×3, Max Pooling, Stride 2 [3×3, 64]×3
Pool1	2×2 Max Pooling, Stride 2		
Conv2.x	[3×3, 64]×2	[3×3, 64]×3	[3×3, 128]×3
Pool2	2×2 Max Pooling, Stride 2		
Conv3.x	[3×3, 64]×2	[3×3, 64]×3	[3×3, 256]×3
Pool3	2×2 Max Pooling, Stride 2		
Fully Connected	64	64	256

Table 14: Our plain CNN architectures with different convolutional layers. Conv1.x, Conv2.x and Conv3.x denote convolution units that may contain multiple convolution layers. E.g., [3×3, 64]×3 denotes 3 cascaded convolution layers with 64 filters of size 3×3.

Layer	ResNet-20 (CIFAR-100)	ResNet-32 (CIFAR-100)
Conv1.x	[3×3, 16]×1 $\begin{bmatrix} 3\times3, 16 \\ 3\times3, 16 \end{bmatrix} \times 3$	[3×3, 16]×1 $\begin{bmatrix} 3\times3, 16 \\ 3\times3, 16 \end{bmatrix} \times 5$
Conv2.x	$\begin{bmatrix} 3\times3, 32 \\ 3\times3, 32 \end{bmatrix} \times 3$	$\begin{bmatrix} 3\times3, 32 \\ 3\times3, 32 \end{bmatrix} \times 5$
Conv3.x	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 5$
	Average Pooling	

Table 15: Our ResNet architectures with different convolutional layers. Conv0.x, Conv1.x, Conv2.x, Conv3.x and Conv4.x denote convolution units that may contain multiple convolutional layers, and residual units are shown in double-column brackets. Conv1.x, Conv2.x and Conv3.x usually operate on different size feature maps. These networks are essentially the same as [21], but some may have a different number of filters in each layer. The downsampling is performed by convolutions with a stride of 2. E.g., [3×3, 64]×4 denotes 4 cascaded convolution layers with 64 filters of size 3×3, S2 denotes stride 2.

Reported Results. For all the experiments on MLPs and CNNs (except CNNs in the few-shot learning), we report testing error rates. For the few-shot learning experiment, we report testing accuracy. For all the experiments on both GCNs and PointNets, we report testing accuracy. All results are averaged over 10 runs of the model.

Multilayer perceptron. We conduct digit classification task on MNIST with a three-layer multilayer perceptron following this repository[1] . The input dimension of each MNIST digit is 28×28, which is 784 dimensions after flattened. Our two hidden layers have 256 output dimensions, *i.e.*, 256 neurons. The output layer will output 10 logits for classification. We use a cross-entropy loss with softmax function. For the optimization, we use a momentum SGD with learning rate 0.01, momentum 0.9 and batch size 100. The training stops at 100 epochs.

Convolutional neural networks. The network architectures used in the paper are elaborated in Table 14 and Table 15. For CIFAR-100, we use 128 as the mini-batch size. We use momentum SGD with momentum 0.9 and the learning rate starts with 0.1, divided by 10 when the performance is saturated. For ImageNet-2012, we use batch size 128 and start with learning rate 0.1. The learning rate is divided by 10 when the performance is saturated, and the training is terminated at 700k iterations. For ResNet-20 and ResNet-32 on CIFAR-100, we use exactly the same architecture used on CIFAR-10 as [21]. The rotation matrix is initialized with random normal distribution (mean is 0 and variance is 1). Note that, for all the compared methods, we always use the best possible hyperparameters to make sure that the comparison is fair. The baseline has exactly the same architecture and training settings as the one that OPT uses. If not otherwise specified, standard ℓ_2 weight decay ($5e-4$) is applied to all the neural network including baselines and the networks that use OPT training.

Few-shot learning. The network architecture (Table 16) we used for few-shot learning experiments is the same as that used in [9]. In our experiments, we show comparison of our OPT training with standard training on 'baseline' and 'baseline++' settings in [9]. In 'baseline' setting, a standard CNN model is pretrained on the whole meta-train dataset (standard non-MAML supervised training) and later only the classifier layer is finetuned on few-shot dataset. 'baseline++' differs from 'baseline' on the classifier: in 'baseline', each output dimension of the classifier is computed as the inner product between weight w and input x, i.e. $w \cdot x$; while in 'baseline++' it becomes the scaled cosine distance $c\frac{w \cdot x}{\|w\|\|x\|}$ where c is a positive scalar. Following [9], we set $c = 2$.

During pretraining, the model is trained for 200 epochs on the meta-train set of mini-ImageNet with an Adam optimzer (learning rate $1e-3$, weight decay $5e-4$) and the classifier is discarded after pretraining. The model is later finetuned, with

[1] https://github.com/hwalsuklee/tensorflow-mnist-MLP-batch_normalization-weight_initializers

Layer	CNN-4
Conv1	3×3, 64
Pool1	2×2 Max Pooling, Stride 2
Conv2	3×3, 64
Pool2	2×2 Max Pooling, Stride 2
Conv3	3×3, 64
Pool3	2×2 Max Pooling, Stride 2
Conv4	3×3, 64
Pool4	2×2 Max Pooling, Stride 2
Linear Classifier	number of classes

Table 16: Architecture for few-shot learning. The number of classes is different for pretraining and finetuning.

Layer	Wide CNN-9 (CIFAR-100)	ResNet-18 (ImageNet-2012)
Conv0.x	N/A	[7×7, 64], Stride 2 3×3, Max Pooling, Stride 2
Conv1.x	[3×3, 64]×3 2×2 Max Pooling, Stride 2	$\begin{bmatrix} 3\times 3, 64 \\ 3\times 3, 64 \end{bmatrix} \times 2$
Conv2.x	[3×3, 128]×3 2×2 Max Pooling, Stride 2	$\begin{bmatrix} 3\times 3, 128 \\ 3\times 3, 128 \end{bmatrix} \times 2$
Conv3.x	[3×3, 256]×3 2×2 Max Pooling, Stride 2	$\begin{bmatrix} 3\times 3, 256 \\ 3\times 3, 256 \end{bmatrix} \times 2$
Conv4.x	N/A	$\begin{bmatrix} 3\times 3, 512 \\ 3\times 3, 512 \end{bmatrix} \times 2$
Final	256-dim Fully Connected	Average Pooling

Table 17: Our wide CNN-9 and wide ResNet-18 architectures with different convolutional layers.

a new classifier, on the few-shot samples (5 way, support size 5) with a momentum SGD optimizer (learning rate $1e-2$, momentum 0.9, dampening 0.9, weight decay $1e-3$, batch size 4) for 100 epochs. We re-initialize the classifier for each few-shot sample.

Graph neural networks. We implement the OPT training for GCN in the official repository[2]. The experimental settings also follow the official repository to ensure a fair comparison. For OPT (CP) method, we use the original hyperparameters and experimental setup except the added rotation matrix. For OPT (OGD) method, we use our own OGD optimizer in Tensorflow to train the rotation matrix in order to maintain orthogonality and use the original optimizer to train the other variables.

Training a GCN with OPT is not that straightforward. Specifically, the forward model of GCN is $\boldsymbol{Z}=\text{Softmax}\big(\hat{\boldsymbol{A}}\cdot \text{ReLU}(\hat{\boldsymbol{A}}\cdot\boldsymbol{X}\cdot\boldsymbol{W}_0)\cdot\boldsymbol{W}_1\big)$ where $\hat{\boldsymbol{A}}=\tilde{\boldsymbol{D}}^{\frac{1}{2}}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{\frac{1}{2}}$. We note that $\boldsymbol{A}$ is the adjacency matrix of the graph, $\tilde{\boldsymbol{A}}=\boldsymbol{A}+\boldsymbol{I}$ ($\boldsymbol{I}$ is an identity matrix), and $\tilde{\boldsymbol{D}}=\sum_j \tilde{\boldsymbol{A}}_{ij}$. $\boldsymbol{X}\in\mathbb{R}^{n\times d}$ is the feature matrix of n nodes in the graph (feature dimension is d). $\boldsymbol{W}_1$ is the weights of the classifiers. $\boldsymbol{W}_1$ is the weights of the classifiers. $\boldsymbol{W}_0$ is the weight matrix of size $d\times h$ where h is the dimension of the hidden space. We treat each column vector of $\boldsymbol{W}_0$ as a neuron, so there are h neurons in total. Then we apply OPT to train these h neurons of dimension d in GCN. We conduct experiments on Cora and Pubmed datsets [63]. We aim to verify the effectiveness of OPT on GCN instead of achieving state-of-the-art performance on this task.

Point cloud recognition. To simplify the comparison and remove all the bells and whistles, we use a vanilla PointNet (without T-Net) as our backbone network. We apply OPT to train the MLPs in PointNet. We follow the same experimental settings as [57] and evaluate on the ModelNet-40 dataset [76]. We exactly follow the same setting in the original paper [57] and the official repositories[3]. Specifically, we multiply the rotation matrix to the original fixed neurons in all the 1×1 convolution layers and the fully connected layer except the final classifier. All the rotation matrix is initialized with random normal distribution. For the experiments, we use point number 1024, batch size 32 and Adam optimizer with initial learning rate 0.001. The learning rate will decay by 0.7 every 200k iterations, and the training is terminated at 250 epochs.

Experimental settings for S-OPT. For the experiment of S-OPT, the architecture of wide CNN-9 and wide ResNet-18 is given in Table 17. CNN-6 is the same as the one in Table 14. We use standard data augmentation for CIFAR-100, following [54]. For ImageNet-2012, we use the same data augmentation in [38, 54]. This data augmentation does not contain as many transformation as the one in [21], so the final performance may be worse than [21]. However, all the compared methods use the same data augmentation in our experiments, so the experiment is still a fair comparison. For CIFAR-100, we use $N_{\text{out}}=300$ and $N_{\text{in}}=750$. For ImageNet, we use $N_{\text{out}}=700$ and $N_{\text{in}}=1000$. For S-OPT, we directly use the original OPT for the first layer, as its neuron dimension is typically very small. We decrease the learning rate by a factor of 10 when the performance is saturated in the outer iteration.

[2] https://github.com/tkipf/gcn

[3] https://github.com/charlesq34/pointnet

Layer	MNIST Visualization	ResNet-18A	ResNet-18B	CNN-20 for Face Embeddings
Conv0.x	N/A	[7×7, 64], Stride 2 3×3, Max Pooling, Stride 2	[7×7, 64], Stride 2 3×3, Max Pooling, Stride 2	N/A
Conv1.x	[3×3, 32]× 2 3×3, Max Pooling, Stride 2	$\begin{bmatrix} 3\times 3, 64 \\ 3\times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3, 64 \\ 3\times 3, 64 \end{bmatrix} \times 2$	[3×3, 64]×1, Stride 2 $\begin{bmatrix} 3\times 3, 64 \\ 3\times 3, 64 \end{bmatrix} \times 1$
Conv2.x	[3×3, 64]× 2 3×3, Max Pooling, Stride 2	$\begin{bmatrix} 3\times 3, 128 \\ 3\times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3, 128 \\ 3\times 3, 128 \end{bmatrix} \times 2$	[3×3, 128]×1, Stride 2 $\begin{bmatrix} 3\times 3, 128 \\ 3\times 3, 128 \end{bmatrix} \times 2$
Conv3.x	[3×3, 128]× 2 3×3, Max Pooling, Stride 2	$\begin{bmatrix} 3\times 3, 128 \\ 3\times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3, 256 \\ 3\times 3, 256 \end{bmatrix} \times 2$	[3×3, 256]×1, Stride 2 $\begin{bmatrix} 3\times 3, 256 \\ 3\times 3, 256 \end{bmatrix} \times 4$
Conv4.x	N/A	$\begin{bmatrix} 3\times 3, 128 \\ 3\times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3, 512 \\ 3\times 3, 512 \end{bmatrix} \times 2$	[3×3, 512]×1, Stride 2 $\begin{bmatrix} 3\times 3, 512 \\ 3\times 3, 512 \end{bmatrix} \times 1$
Final	Fully Connected (3-dim)	Average Pooling (128-dim)	Average Pooling (512-dim)	Fully Connected (512-dim)

Table 18: Our network architectures for large categorical training.

Experimental settings for large categorical training. All the network architectures used in the large categorical training are specified in Table 18. For the visualization on MNIST, we simply set the output dimension as 3 and directly plot the 3-dimensional features. In Fig. 7, each color denotes a class of digits, and each dot point denotes 3-dimensional features for a digit image. The experiments on ImageNet follows the same setting as the previous section. For the open-set face recognition experiments, we generally follow the same training configuration as SphereFace [52]. For all the methods used in face recognition, we use the 20-layer residual network as described in Table 18. Since OPT is originally implemented in TensorFlow, we re-implement the CNN-20 for deep face recognition in TensorFlow, which yields an accuracy gap compared to [52]. This is due to some mis-match in data augmentation and optimizations. However, since both the baseline and our CLS-OPT use the same network implementation in TensorFlow and achieving state-of-the-art results is not our major focus, it is still a fair and valid comparison. We expect CLS-OPT can also be generally useful for large categorical training of deep face recognition.

E. Loss Landscape Visualization (Normal Distribution Perturbation)

E.1. Visualization Procedure

We generally follow the visualization procedure in [43]. However, since OPT has a different training process, we use a modified visualization method but still make it comparable to the baseline.

Specifically, if we want to plot the loss landscape of a neural neural with loss $\mathcal{L}(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is the learnable model parameters, we need to first choose pretrained model parameters $\boldsymbol{\theta}^*$ as a center point. Then we choose two random direction vectors $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$. The 2D plot $f(\alpha, \beta)$ is defined as

$$f(\alpha, \beta) = \mathcal{L}(\boldsymbol{\theta}^* + \alpha\boldsymbol{\delta} + \beta\boldsymbol{\eta}) \tag{38}$$

which can be used as a 2D surface visualization. Note that, after we randomly initialize the direction vectors $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$ (with normal distribution), we need to perform the filter normalization [43]. Specifically, we normalize each filter in $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$ to have the same norm as the corresponding filter in $\boldsymbol{\theta}^*$. The loss landscape of our baseline is plotted using this visualization approach.

In contrast, the learnable parameters in OPT are no longer the weights of neurons. Instead, the learnable parameters are the orthogonal matrices. More precisely, the trainable matrices are used to perform orthogonalization in the neural networks (*i.e.*, $\boldsymbol{P}$ in Fig. 2). We denote the combination of all the trainable matrices as $\tilde{\boldsymbol{R}}$, and the corresponding pretrained matrices as $\tilde{\boldsymbol{R}}^*$. Then the 2D visualization of OPT is

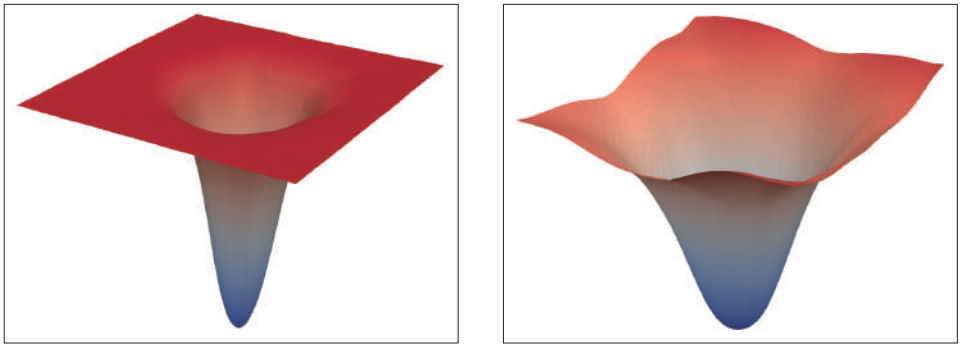
$$f(\alpha, \beta) = \mathcal{L}(\tilde{\boldsymbol{R}}^* + \alpha\boldsymbol{\gamma} + \beta\boldsymbol{\kappa}) \tag{39}$$

where $\boldsymbol{\gamma}$ and $\boldsymbol{\kappa}$ are two random direction vectors (which follow the normal distribution) to perturb $\tilde{\boldsymbol{R}}^*$. The visualization procedures of baseline and OPT are essentially the same except that the trainable variables are different. Therefore, their loss landscapes are comparable.

E.2. Experimental Details

In Fig. 4, we vary α and β from -1.5 to 1.5 for both baseline and OPT, and then plot the surface of 2D function f. We use the CNN-6 (as specified in Appendix D) on CIFAR-100. We use the same data augmentation as [49]. We train the network with SGD with momentum 0.9 and batch size 128. We start with learning rate 0.1, divide it by 10 at 30k, 50k and 64k iterations, and terminate training at 75k iterations. The training details basically follows [49]. We mostly use CP for OPT due to efficiency. Note that, the other orthogonalization methods in OPT yields similar loss landscapes in general. The pretrained model for standard training yields 37.59% testing error on CIFAR-100, while the pretrained model for OPT yields 33.53% error. This is also reported in Section 6.

E.3. Full Visualization Results for the Main Paper



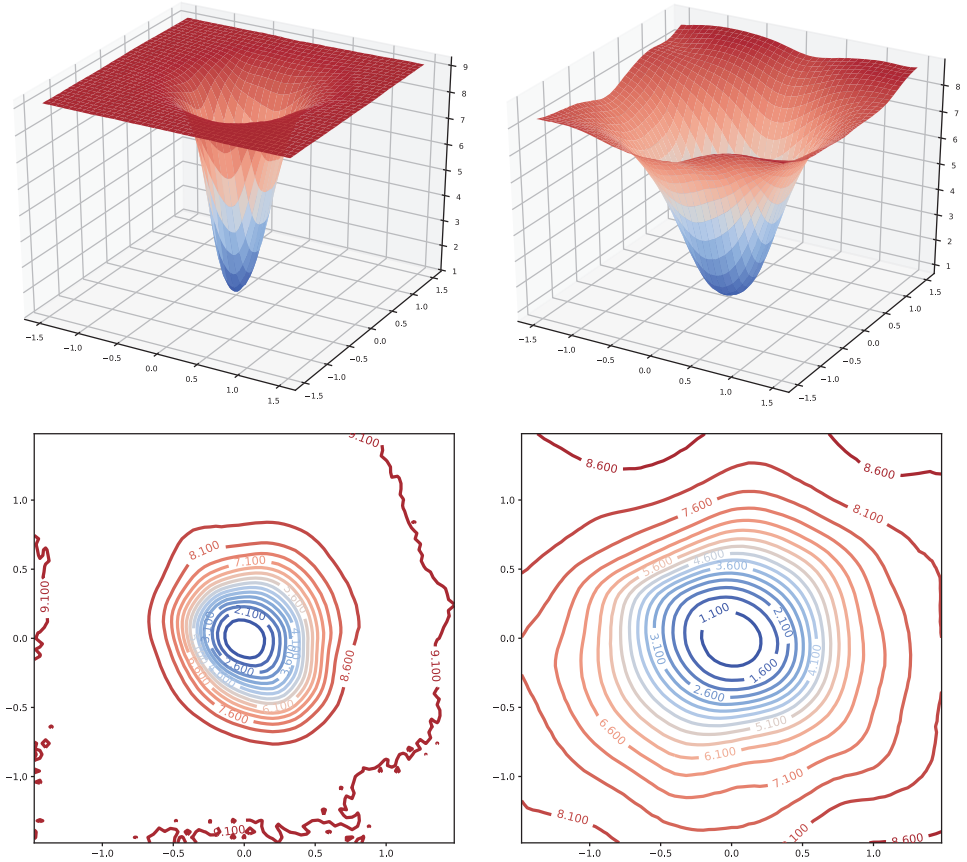
(a) Standard training (b) OPT

Figure 8: High-quality rendered loss landscapes of standard training and OPT.

Following the same experimental settings in Appendix E.2, we render the 3D loss landscapes with some color and lighting effects for Fig. 8. The visualization data is exactly the same as Fig. 4, and we simply use ParaView to plot the figure. The rendered loss landsacpe better reflects that OPT yields a much more smooth loss geometry.

We also give the large and full version of Fig. 4(b) (in the main paper) in the following figure. Fig. 9 is identical to Fig. 4(b) in the main paper except that Fig. 9 has larger size. From Fig. 9, we can better observe the dramatically different loss landscape

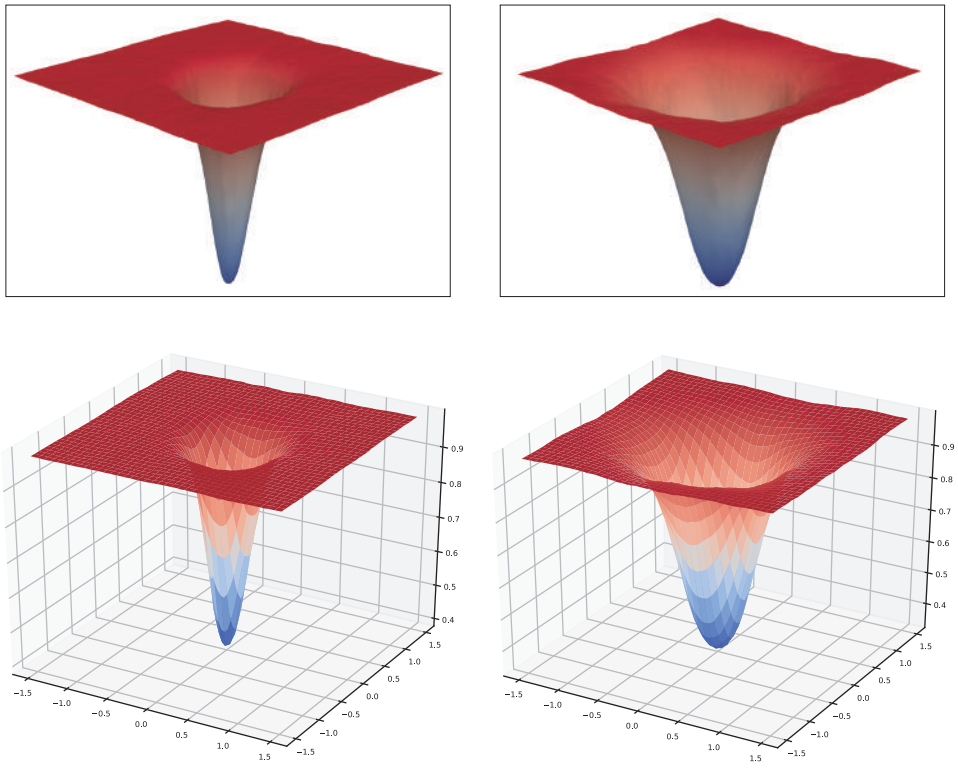
between standard training and OPT. From the contour plots, we can better see that the red region of standard training is extremely flat and is highly non-convex around the edge. In comparison to standard training, OPT has very smooth and relatively convex contour shape.



(a) Standard training (b) OPT

Figure 9: Comparison of loss landscapes between standard training and OPT (full results of Fig. 4(b) in the main paper). Top row: loss landscape visualization with Cartesian coordinate system; Bottom row: loss contour visualization.

Then we visualize the testing error landscape. Additionally, we include a high-quality visualization using ParaView. We render the plot with lighting and color effects in order to better demonstrate the difference between OPT and standard training. The visualization results are given in Fig. 10. The comparison of testing error landscape shows that the parameter space of OPT is more robust than standard training, because the testing error of OPT is increased in a slower speed while the model is perturbed away from the pretrained parameters. In other words, the parameter space of OPT is more robust to the random perturbation than standard training. Combining the visualization of loss landscape, it is well justified that OPT can significantly alleviate training difficulty and improve generalization.



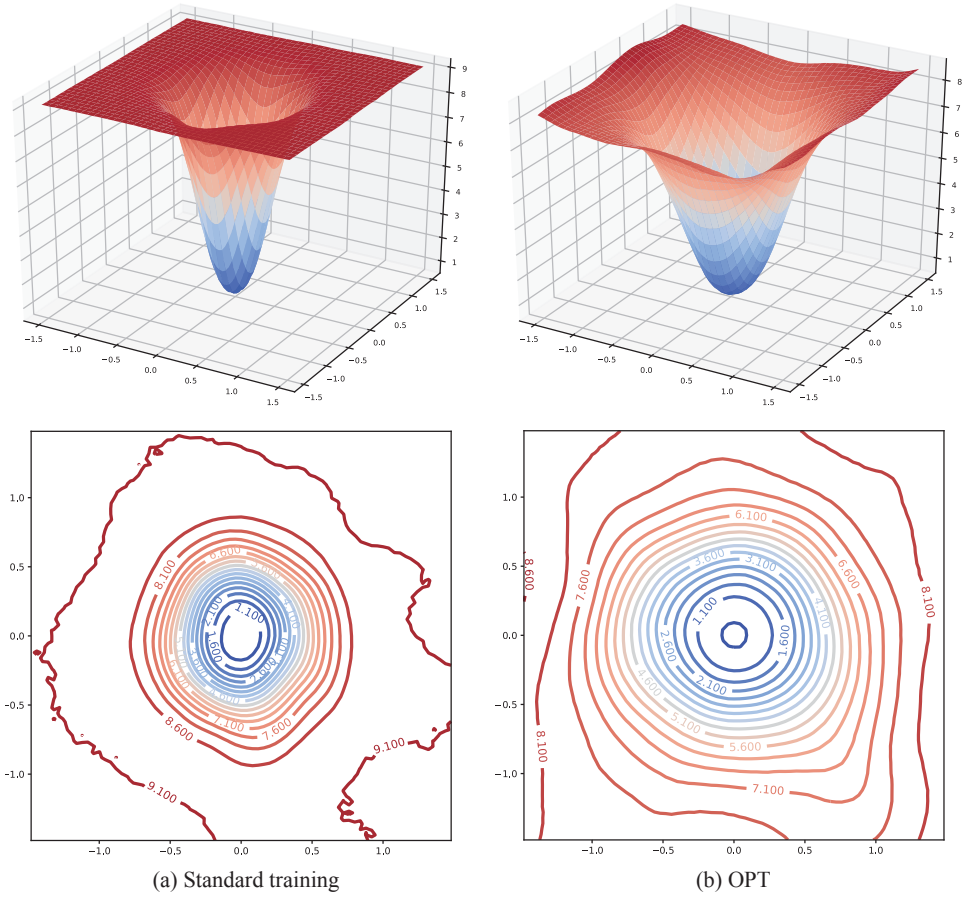
(a) Standard training (b) OPT

Figure 10: Comparison of testing error landscapes between standard training and OPT. Top row: high-quality rendered testing error landscape visualization with lighting effects; Bottom row: testing error landscape visualization with Cartesian coordinate system.

.

E.4. Loss Landscape Visualization of Different Neural Network

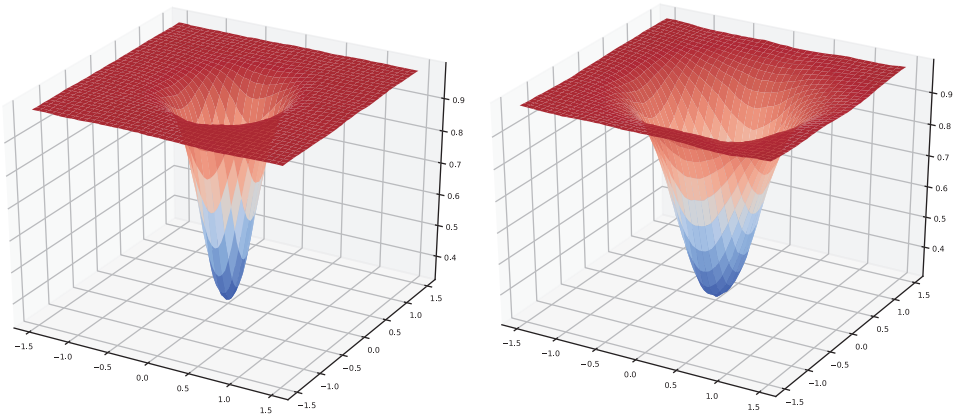
In order to show that the loss landscape of OPT is quite general and consistent across different neural network architectures, we also visualize the loss landscape using a deep network network (CNN-9 as specified in Appendix D). The experiments are conducted on CIFAR-100. The results are given in Fig. 11. We can see that the loss landscape of OPT is much more smooth than standard training, similar to the observation for CNN-6. Therefore, the loss landscape difference between OPT and standard training is consistent across different network architectures, and OPT consistently shows better optimization landscape.



(a) Standard training (b) OPT

Figure 11: Comparison of loss landscapes between standard training and OPT on CIFAR-100 (CNN-9). Top row: loss landscape visualization with Cartesian coordinate system; Bottom row: loss contour visualization.

Then we show the landscape of testing error for CNN-9 on CIFAR-100. The results are given in Fig. 12. Similar to CNN-6, the testing error landscape of OPT is more smooth and convex than standard training. Moreover, OPT has a more flat local minima of testing error, while standard training has a sharp local minima. The testing error landscape in Fig. 12 generally follows the same pattern as the loss landscape in Fig. 11. The visualization further verifies the superiority of OPT is very consistent across different network architectures.

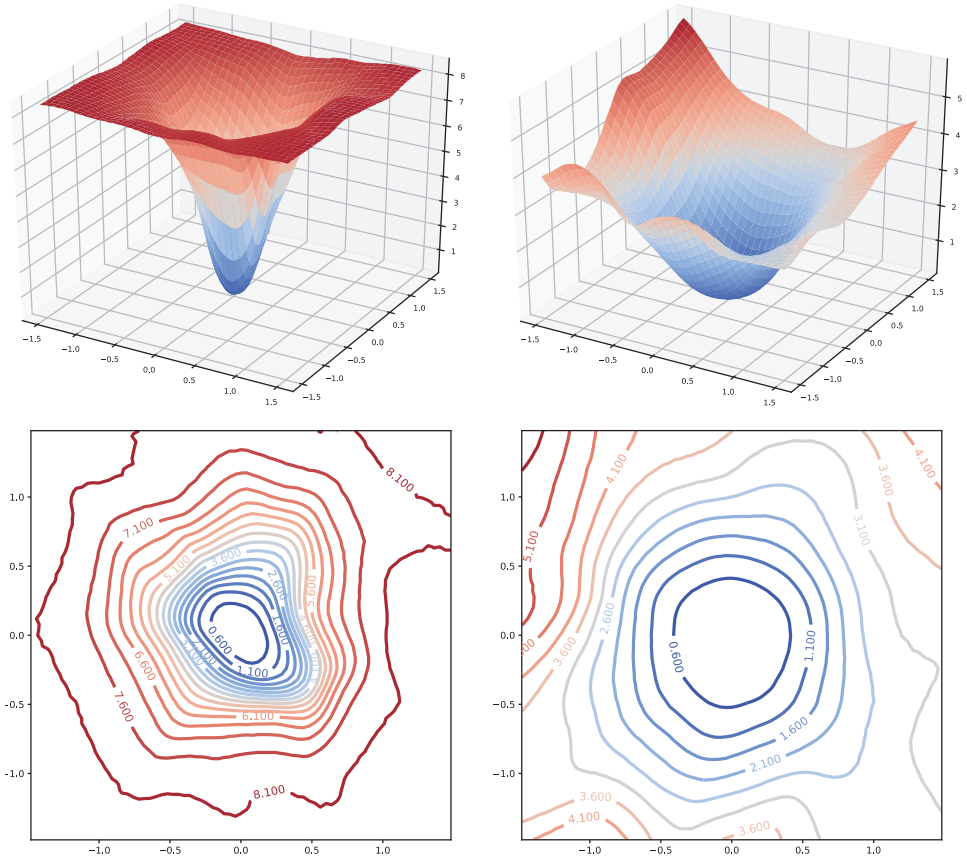


(a) Standard training (b) OPT

Figure 12: Comparison of testing error landscapes between standard training and OPT on CIFAR-100 (CNN-9).

E.5. Loss Landscape Visualization of Different Dataset

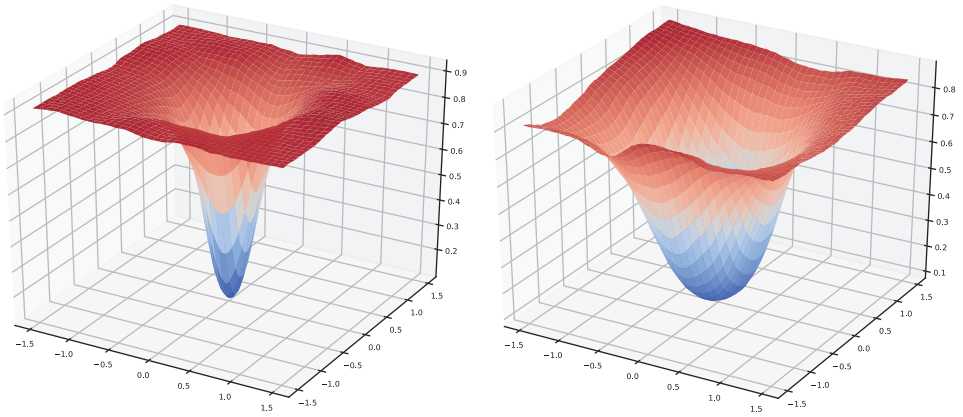
Similar to Appendix F.5, we also visualize the loss landscape of OPT and standard training on a different dataset (CIFAR-10) with CNN-6. The loss landscape visualization is given in Fig. 13. Although the loss landscape on CIFAR-10 is quite different from the one on CIFAR-100, we can still observe that the loss landscape of OPT has a very flat local minima and the loss values are increasing smoothly and slowly. In contrast, the loss landscape of standard training has a sharp local minima and the loss values quickly increase to a large value. The red region of standard training will lead to very small gradient, potentially affecting the training. From the contour plots, the comparison apparently shows that the loss landscape of OPT is much more smooth than standard training.



(a) Standard training (b) OPT

Figure 13: Comparison of loss landscapes between standard training and OPT on CIFAR-10 (CNN-6). Top row: loss landscape visualization with Cartesian coordinate system; Bottom row: loss contour visualization.

We also visualize the landscape of testing error in Fig. 14. The testing error landscape generally follows the pattern in the loss landscape. One can easily observe that the parameter space of standard training is very sensitive to perturbations. A small perturbation can make the model parameters completely fail (*i.e.*, the testing error dramatically increase). Differently, the parameter space of OPT is much more robust to perturbations. The model parameter can still work well with a small perturbation. Both Fig. 13 and Fig. 14 validate that the superiority of OPT is consistent across different training datasets.



(a) Standard training (b) OPT

Figure 14: Comparison of testing error landscapes between standard training and OPT on CIFAR-10 (CNN-6).

F. Loss Landscape Visualization (Uniform Distribution Perturbation)

F.1. Visualization Procedure

Different from Appendix E, we choose two random direction vectors $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$ based on $[0, 1]$ uniform distribution to further justify the effectiveness of OPT. The 2D plot $f(\alpha, \beta)$ is defined as

$$f(\alpha, \beta) = \mathcal{L}(\boldsymbol{\theta}^* + \alpha\boldsymbol{\delta} + \beta\boldsymbol{\eta}) \tag{40}$$

which can be used as a 2D surface visualization. Note that, after we randomly initialize the direction vectors $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$ with $[0, 1]$ normal distribution, we need to perform the filter normalization [43]. Specifically, we normalize each filter in $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$ to have the same norm as the corresponding filter in $\boldsymbol{\theta}^*$. The loss landscape of our baseline is plotted using this visualization approach.

In contrast, the learnable parameters in OPT are no longer the weights of neurons. Instead, the learnable parameters are the orthogonal matrices. More precisely, the trainable matrices are used to perform orthogonalization in the neural networks (*i.e.*, $\boldsymbol{P}$ in Fig. 2). We denote the combination of all the trainable matrices as $\tilde{\boldsymbol{R}}$, and the corresponding pretrained matrices as $\tilde{\boldsymbol{R}}^*$. Then the 2D visualization of OPT is

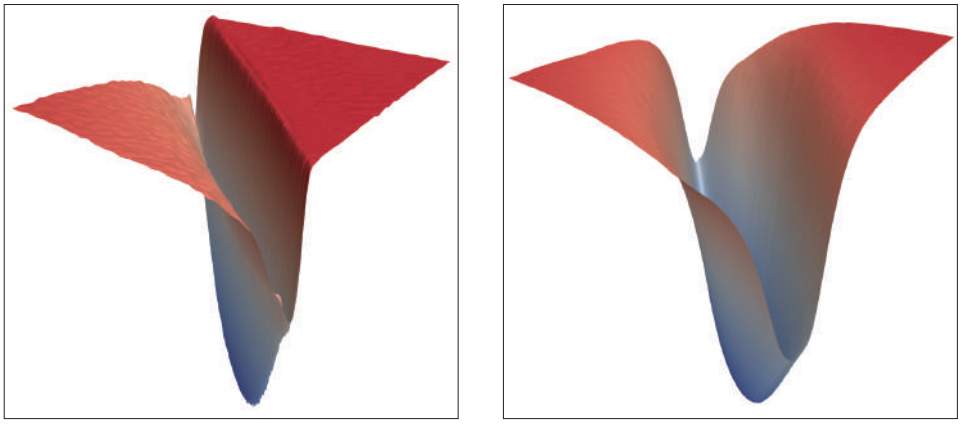
$$f(\alpha, \beta) = \mathcal{L}(\tilde{\boldsymbol{R}}^* + \alpha\boldsymbol{\gamma} + \beta\boldsymbol{\kappa}) \tag{41}$$

where $\boldsymbol{\gamma}$ and $\boldsymbol{\kappa}$ are two random direction vectors (which follow the $[0, 1]$ uniform distribution) to perturb $\tilde{\boldsymbol{R}}^*$. The visualization procedures of baseline and OPT are essentially the same except that the trainable variables are different. Therefore, their loss landscapes are comparable.

F.2. Experimental Details

In Fig. 4, we vary α and β from -1 to 1 for both baseline and OPT, and then plot the surface of 2D function f. We use the CNN-6 (as specified in Appendix D) on CIFAR-100. We use the same data augmentation as [49]. We train the network with SGD with momentum 0.9 and batch size 128. We start with learning rate 0.1, divide it by 10 at 30k, 50k and 64k iterations, and terminate training at 75k iterations. The training details basically follows [49]. We mostly use CP for OPT due to efficiency. Note that, the other orthogonalization methods in OPT yields similar loss landscapes in general. The pretrained model for standard training yields 37.59% testing error on CIFAR-100, while the pretrained model for OPT yields 33.53% error. This is also reported in Section 6.

F.3. Full Visualization Results

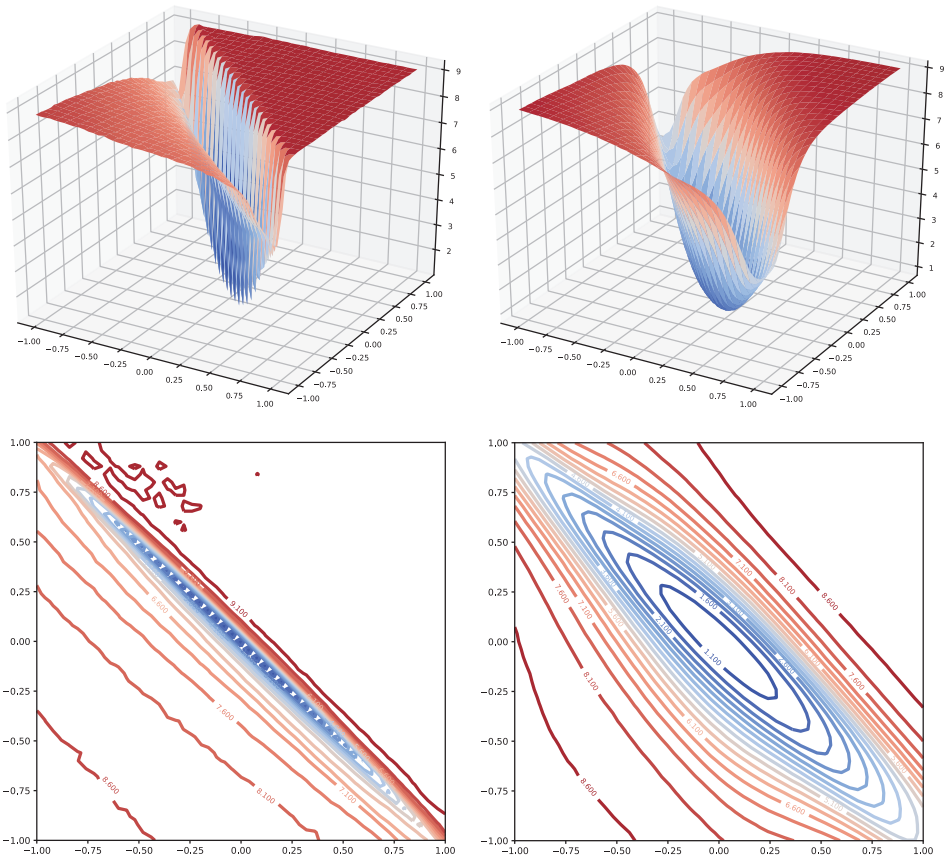


(a) Standard training (b) OPT

Figure 15: High-quality rendered loss landscapes of standard training and OPT.

Following the same experimental settings in Appendix F.2, we render the 3D loss landscapes with some color and lighting effects for Fig. 15. We first use ParaView to plot a high-qualify loss landscape comparison between standard training and OPT. As expected, the loss landscape of OPT is much more smooth than standard training. Note that, for the flat red region in standard training, we can still observe numerous small local minima, while the red region of OPT is very smooth. Fig. 15 better validates our analysis and discussion in Section 5.1, and also shows the superiority of OPT.

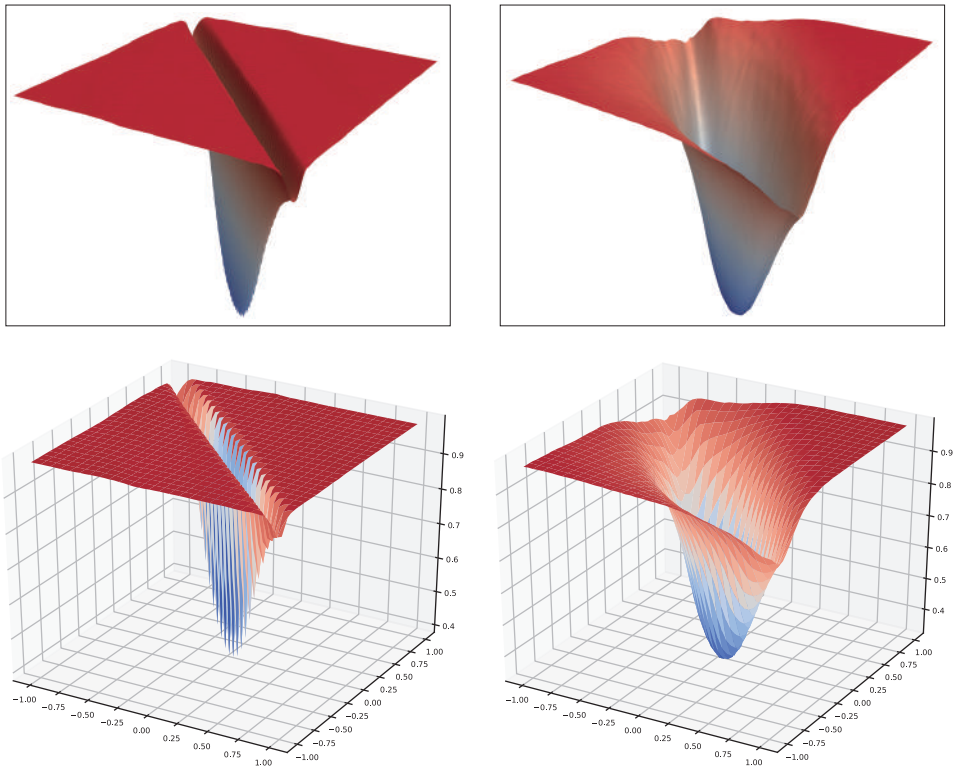
We provide the visualization results in the rest of the subsection. From Fig. 16, we can better observe the dramatically different loss landscape between standard training and OPT.



(a) Standard training (b) OPT

Figure 16: Comparison of loss landscapes between standard training and OPT (full results of Fig. 4(a) in the main paper). Top row: loss landscape visualization with Cartesian coordinate system; Bottom row: loss contour visualization.

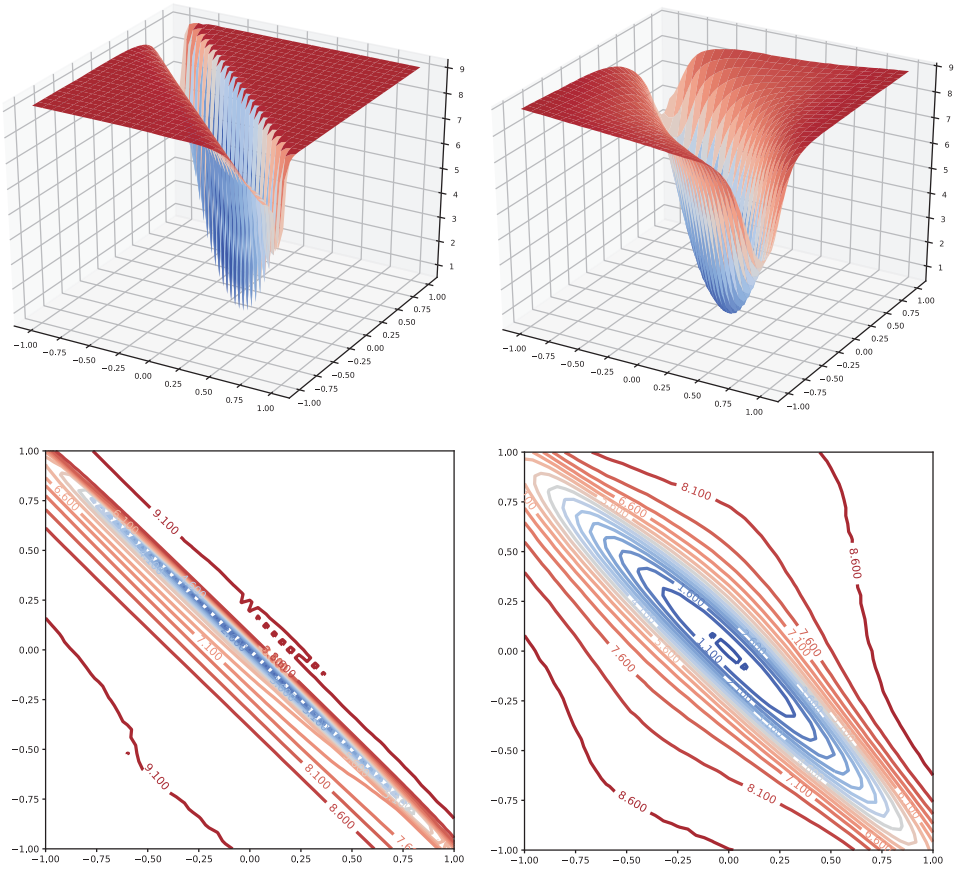
To better understand the difference of the training dynamics between standard training and OPT, we also plot the testing error landscapes in Fig. 17 for both methods. The testing error is computed on the testing set of CIFAR-100 with the perturbed pretrained model (α and β are the perturbation parameters). From the testing error landscape comparison in Fig. 17, we can see that once the baseline pretrained model is slightly perturbed, the testing error will immediately increase to 99.99% which is random selection-level testing error (because we have 100 balanced classes in total, randomly picking a class leads to 0.01% accuracy). In contrast, the testing error landscape of OPT is much more smooth. Even if we perturb the OPT pretrained model, we still end up with a reasonably low testing error, show that the parameter space of OPT is more smooth and continuous. All these evidences suggest that OPT is a better training framework for neural networks and can significantly alleviate the optimization difficulty. In this following subsections, we aim to show that the loss and testing error landscape difference between standard training and OPT is not a coincidence. We will show that the improvement of OPT on the loss and testing error landscape is both dataset-agnostic and architecture-agnostic.



(a) Standard training (b) OPT

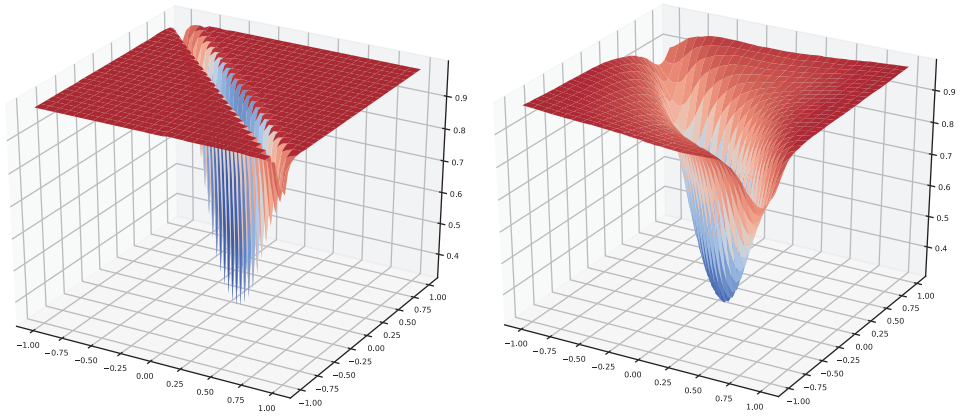
Figure 17: Comparison of testing error landscapes between standard training and OPT. Top row: high-quality rendered testing error landscape visualization with lighting effects; Bottom row: testing error landscape visualization with Cartesian coordinate system.

F.4. Loss Landscape Visualization of Different Neural Network



(a) Standard training (b) OPT

Figure 18: Comparison of loss landscapes between standard training and OPT on CIFAR-100 (CNN-9). Top row: loss landscape visualization with Cartesian coordinate system; Bottom row: loss contour visualization.



(a) Standard training (b) OPT

Figure 19: Comparison of testing error landscapes between standard training and OPT on CIFAR-100 (CNN-9).

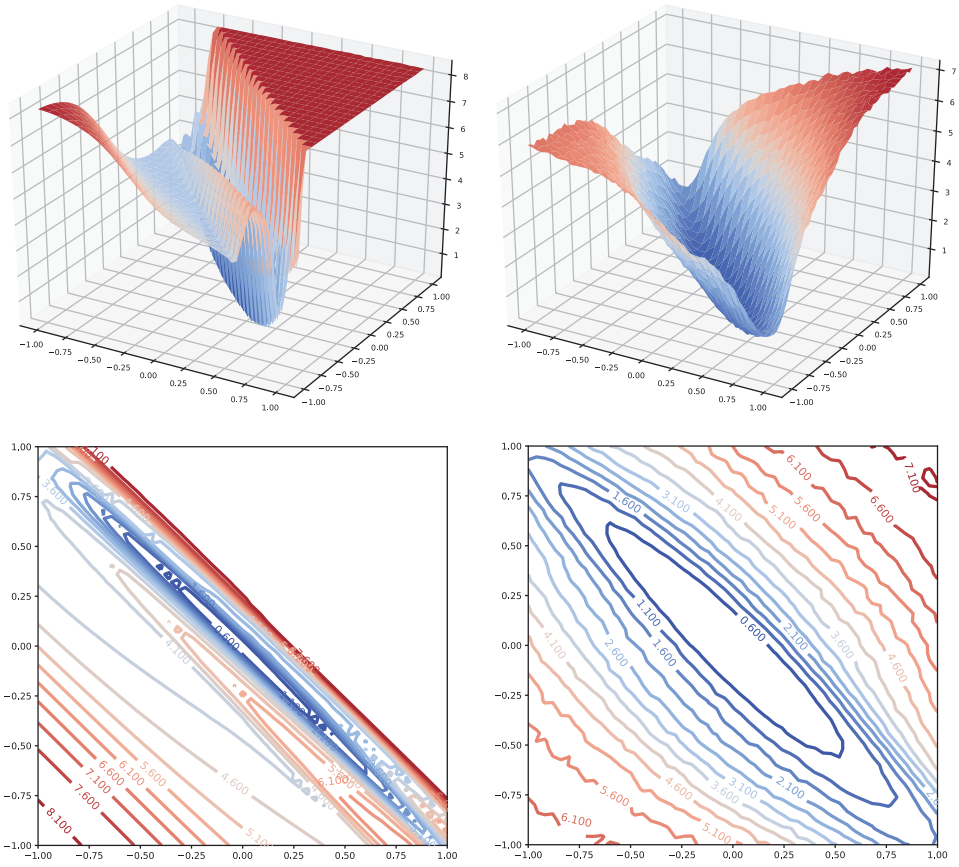
To show that the difference of the loss landscape between standard training and OPT is consistent across different neural networks. We use a deeper CNN-9 (as specified in Appendix D) to visualize the loss landscape. The experimental settings generally follow Appendix F.2. We use CNN-9 as our backbone architecture and train it on CIFAR-100. The visualization of

the loss landscapes is given in Fig. 18. We observe that the loss landscapes of OPT with CNN-9 and CNN-6 are very similar. In general, the conclusion that OPT yields better loss landscape still holds for a deeper neural network, showing that the effectiveness of OPT is not architecture-dependent.

Moreover, we also visualize the landscape of testing error in Fig. 19. We can observe that the testing error landscapes are somewhat similar to the loss landscape in Fig. 18. The results further validate the superiority of OPT. We can observe in Fig. 19 that OPT has more smooth testing error landscape and can make the training parameter space of the neural network less sensitive to perturbations.

F.5. Loss Landscape Visualization on Different Dataset

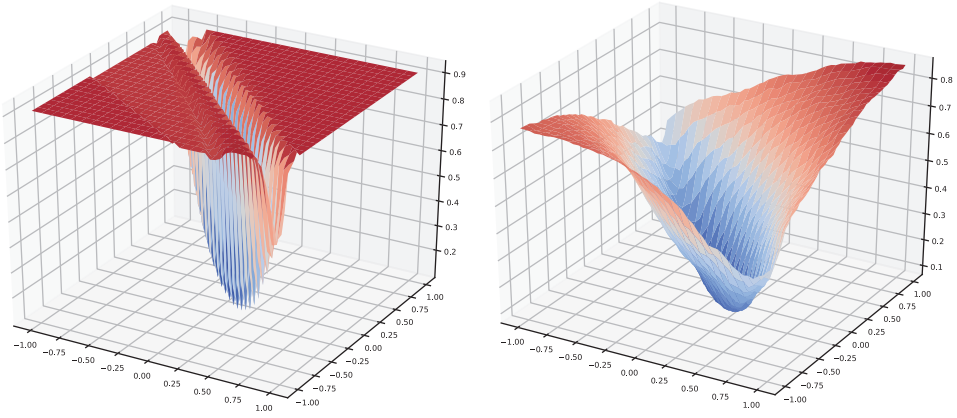
We also perform the same loss and testing error landscape visualization on CIFAR-10. The training details basically follows Appendix F.2. For CIFAR-10, we use the same data augmentation as in Appendix D. The results are given in Fig. 20. From Fig. 20, we can observe even more dramatic difference of the loss landscape between standard training and OPT. In standard training, the loss landscape exhibits highly non-convex and non-smooth behavior. There are countless local minima in the loss landscape. Different from the results in Fig. 18, the loss landscape of standard training on CIFAR-10 has some huge local minima that are hard to escape from. In contrast, the loss landscape of OPT on CIFAR-10 does not show obvious and huge local minima and is far more convex and smooth than standard training. The contour maps show more significant difference between standard training and OPT. The contour map of OPT shows the shape of a single symmetric and convex valley, while the contour map of standard training presents the shape of multiple highly irregular valleys. The visualization further validate that the improvement of OPT on optimization landscape is very consistent across different training datasets.



(a) Standard training (b) OPT

Figure 20: Comparison of loss landscapes between standard training and OPT on CIFAR-10 (CNN-6). Top row: loss landscape visualization with Cartesian coordinate system; Bottom row: loss contour visualization.

Then we also visualize the testing error landscape of standard training and OPT in Fig. 21. As expected, the testing error landscape of standard training shows that changing the pretrained model parameters with a very small perturbation could lead to a dramatic increase in testing error. It indicates that the parameter space of standard training is very sensitive to even a small perturbation. In comparison, the testing error landscape of OPT shows similar shape to the training loss landscape which is the shape of a single regular, smooth and convex valley. We can conclude that OPT has huge advantages over standard training in terms of the optimization landscape. Although the conclusion is drawn from a simple visualization method, it can still shed some light on why OPT yields better training dynamics and generalization ability.



(a) Standard training (b) OPT

Figure 21: Comparison of testing error landscapes between standard training and OPT on CIFAR-10 (CNN-6).

G. Theoretical Discussion on Optimization and Generalization

The key problem we discuss in this section is why OPT may lead to easier optimization and better generalization. We have already shown that OPT can guarantee the minimum hyperspherical energy (MHE) in a probabilistic sense. Although empirical evidences [49] have shown significant and consistent performance gain by minimizing hyperspherical energy, why lower hyperspherical energy will lead to better generalization is still unclear. We argue that OPT leads to better generalization from two aspects: how OPT may affect the training and generalization, and why minimum hyperspherical energy serves as a good inductive bias. We note that rigorously proving that OPT generalizes better is out of the scope of this paper and remains our future work. The section serves as a very preliminary discussion for this topic, and hopefully the discussion can inspire more theoretical studies about OPT.

Our goal here is to leverage and apply existing theoretical results [34, 77, 66, 40, 12, 1] to explain the role that MHE plays rather than proving sharp and novel generalization bounds. We emphasize that our paper is *NOT* targeted as a theoretical one that proves novel generalization bounds.

We simply consider one-hidden-layer networks as the hypothesis class:

$$\mathcal{F} = \{f(x) = \sum_{j=1}^{n} v_j \sigma(\boldsymbol{w}_j^\top \boldsymbol{x}) : v_j \in \{\pm 1\}, \sum_{j=1}^{n} \|\boldsymbol{w}_j\| \leq C_w\} \tag{42}$$

where $\sigma(\cdot) = \max(0, \cdot)$ is ReLU. Since the magnitude of v_j can be scaled into $\boldsymbol{w}_j$, we can restrict v_j to be ± 1. Given a set of *i.i.d.* training sample $\{\boldsymbol{x}_i, y_i\}_{i=1}^m$ where $x \in \mathbb{R}^d$ is drawn uniformly from the unit hypersphere, we minimize the least square loss $\mathcal{L} = \frac{1}{2m}\sum_{i=1}^m (y_i - f(\boldsymbol{x}_i))^2$. The gradient *w.r.t.* $\boldsymbol{w}_i$ is

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_j} = \frac{1}{m}\sum_{i=1}^{m} \big(f(\boldsymbol{x}_i) - y_i\big) v_j \sigma'(\boldsymbol{w}_j^\top \boldsymbol{x}_i)\boldsymbol{x}_i. \tag{43}$$

Let $\boldsymbol{W} := \{\boldsymbol{w}_1^\top, \cdots, \boldsymbol{w}_n^\top\}^\top$ be the column concatenation of neuron weights. We aim to identify the conditions under which there are no spurious local minima. We rewrite that

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \big(\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_1}^\top, \cdots, \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_n}^\top\big)^\top = \boldsymbol{D}\boldsymbol{r} \tag{44}$$

where $\boldsymbol{r} \in \mathbb{R}^m$ $\boldsymbol{r}_i = \frac{1}{m} f(\boldsymbol{x}_i) - y_i$, $\boldsymbol{D} \in \mathbb{R}^{n \times m}$, and $\boldsymbol{D}_{ij} = v_i \sigma'(\boldsymbol{w}_i^\top \boldsymbol{x}_j)\boldsymbol{x}_j$. Therefore, we can obtain the following inequality:

$$\|\boldsymbol{r}\| \leq \frac{1}{s_m(\boldsymbol{D})} \left\| \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} \right\| \tag{45}$$

where $\|\boldsymbol{r}\|$ is the training error and $s_m(\boldsymbol{D})$ is the minimum singular value of $\boldsymbol{D}$. If we need the training error to be small, then we have to lower bound $s_m(\boldsymbol{D})$ away from zero. Therefore, the essential problem now becomes the relationship between MHE and the lower bound of $s_m(\boldsymbol{D})$. We have the following result from [77]:

Lemma 4. *With probability larger than* $1 - m\exp(-m\gamma_m/8) - 2m^2\exp(-4\log^2 d) - \delta$, *we will have that* $s_m(\boldsymbol{D})^2 \geq \frac{1}{2}nm\gamma_m - cn\rho(\boldsymbol{W})$ *where*

$$\begin{aligned} \rho(\boldsymbol{W}) \leq & \frac{\log d}{\sqrt{d}}\sqrt{2L_2(\boldsymbol{W})}m\big(\frac{4}{m}\log\frac{1}{\delta}\big)^{1/4} \\ & + \frac{2\log d}{\sqrt{d}}m\sqrt{\frac{4}{3m}\log\frac{1}{\delta}} + \frac{\log d}{\sqrt{d}}mL_2(\boldsymbol{W}) + 2, \end{aligned} \tag{46}$$

and $L_2(\boldsymbol{W}) = \frac{1}{n^2}\sum_{i,j=1}^n k(\boldsymbol{w}_i, \boldsymbol{w_j})^2 - \mathbb{E}_{\boldsymbol{u},\boldsymbol{v}}[k(u,v)^2]$. *The kernel function* $k(\boldsymbol{u}, \boldsymbol{v})$ *is* $\frac{1}{2} - \frac{1}{2\pi}\arccos(\frac{\langle \boldsymbol{u}, \boldsymbol{v}\rangle}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|})$.

Once MHE is achieved, the neurons will be uniformly distributed on the unit hypersphere. From Lemma 4, we can see that if the neurons are uniformly distributed on the unit hypersphere, $L_2(\boldsymbol{W})$ will be very small and close to zero. Then $\rho(\boldsymbol{W})$ will also be small, leading to large lower bound for $s_m(\boldsymbol{D})$. Therefore, MHE can result in small training error once the gradient norm $\left\|\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}\right\|$ is small. The result implies no spurious local minima if we use OPT for training.

Furthermore, suppose that $\|\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}\|^2 \leq \epsilon$, [77] also proves a training error bound $\tilde{\mathcal{O}}(\epsilon)$ and a generalization bound bound $\tilde{\mathcal{O}}(\epsilon + \frac{1}{\sqrt{m}})$ based on the assumption that $\boldsymbol{W}$ belongs to a specific set $\mathcal{G}_{\boldsymbol{W}}$ (for the definition of $\mathcal{G}_{\boldsymbol{W}}$, please refer to [77]). Therefore, MHE is also connected to the training and generalization error. Note that, the analysis here is highly simplified and the purpose here is to give some justifications rather than rigorously proving any bound.

We further argue that MHE induced by OPT serves as an important inductive bias for neural networks. As the standard regularizer for neural networks, weight decay controls the norm of the neuron weights, regularizing essentially one dimension of the weight. In contrast, MHE completes an important missing pieces by regularizing the remaining dimensions of the weight. MHE encourages minimum hyperspherical redundancy between neurons. In the linear classifier case, MHE impose a prior of maximal inter-class separability.

H. More Discussions

Semi-randomness. OPT fixes the randomly initialized neuron weight vectors and simply learns layer-shared orthogonal matrices, so OPT naturally imposes strong randomness to the neurons. OPT well combines the good generalizability from randomness and the strong approximation power from neural networks. Such randomness suggests that the specific configuration of relative position among neurons does not matter that much, and the coordinate system is more crucial for generalization. [35, 59, 67] also show that randomness can be beneficial to generalization.

Flexible training. First, OPT can used in multi-task training [55] where each set of orthogonal matrices represent one task. OPT can learn different set of orthogonal matrices for different tasks with the neuron weights remain the same. Second, we can perform progressive training with OPT. For example, after learning a set of orthogonal matrices on a large coarse-grained dataset (*i.e.*, pretraining), we can multiple the orthogonal matrices back to the neuron weights and construct a new set of neuron weights. Then we can use the new neuron weights as a starting point and apply OPT to train on a small fine-grained dataset (*i.e.*, finetuning).

Limitations and open problems The limitations of OPT include more GPU memory consumption and heavy computation during training, more numerical issues when ensuring orthogonality and weak scalability for ultra wide neural networks. Therefore, there will be plenty of open problems in OPT, such as scalable and efficient training. Most significantly, OPT opens up a new possibility for studying theoretical generalization of deep networks. With the decomposition to hyperspherical energy and coordinate system, OPT provides a new perspective for future research.

I. On Parameter-Efficient OPT

I.1. Formulation

Since OPT over-parameterizes the neurons, it will consume more GPU memory in training (note that, the number of parameters will not increase in testing). For a d-dimensional neuron, OPT will learn an orthogonal matrix of size $d \times d$ that applies to the the neuron. Therefore, we will need d^2 extra parameters for one layer of neurons, making the training more expensive in terms of the GPU memory. Although the extra training overhead in OPT will not affect the inference speed of the trained neural networks, we still desire to achieve better parameter efficiency in OPT. To this end, we discuss some design possibilities for the *parameter-efficient OPT* (PE-OPT) in this section.

Original OPT over-parameterize a neuron $\boldsymbol{v} \in \mathbb{R}^{n \times n}$ with $\boldsymbol{R}\boldsymbol{v}$ where $\boldsymbol{R}$ is a layer-shared orthogonal matrix of size $d \times d$. We aim to reduce the effective parameters of this $d \times d$ orthogonal matrix. We incorporate a block-diagonal structure to the orthogonal matrix $\boldsymbol{R}$. Specifically, we formulate $\boldsymbol{R}$ as $\text{Diag}(\boldsymbol{R}^{(1)}, \boldsymbol{R}^{(2)}, \cdots, \boldsymbol{R}^{(k)})$ where $\boldsymbol{R}^{(i)}$ is an orthogonal matrix with size $d_i \times d_i$ (it is easy to see that we need $d = \sum_i d_i$). As an example, we only consider the case where all $\boldsymbol{R}^{(i)}$ are of the same size (*i.e.*, $d_1 = d_2 = \cdots = d_k = \frac{d}{k}$). It is also obvious that as long as each block is an orthogonal matrix, then the overall matrix $\boldsymbol{R}$ remains an orthogonal matrix.

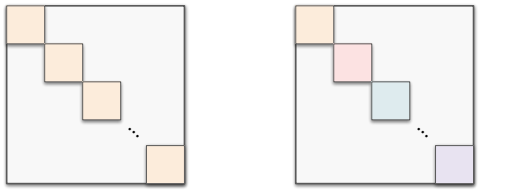


Figure 22: Comparison between the block-shared matrix $\boldsymbol{R}_s$ and the unconstrained block matrix $\boldsymbol{R}_u$.

First, we consider that all the block matrices on the diagonal of the orthogonal matrix $\boldsymbol{R}$ are shared, meaning that $\boldsymbol{R} = \text{Diag}(\boldsymbol{R}^{(1)}, \boldsymbol{R}^{(1)}, \cdots, \boldsymbol{R}^{(1)})$ (*i.e.*, $\boldsymbol{R}^{(1)} = \boldsymbol{R}^{(2)} = \cdots = \boldsymbol{R}^{(k)}$). Therefore, we have a block-diagonal matrix $\boldsymbol{R}_s$ with shared block $\boldsymbol{R}^{(1)}$ as the final orthogonal matrix for the neuron $\boldsymbol{v}$:

$$\boldsymbol{R}_s = \begin{bmatrix} \boldsymbol{R}^{(1)} & 0 & \cdots & 0 \\ 0 & \boldsymbol{R}^{(1)} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \boldsymbol{R}^{(1)} \end{bmatrix} \tag{47}$$

where $\boldsymbol{R}^{(1)} \in \mathbb{R}^{\frac{d}{k} \times \frac{d}{k}}$. The effective number of parameters for the orthogonal matrix $\boldsymbol{R}_s$ immediately reduces to $\frac{d^2}{k^2}$. The left figure in Fig. 22 gives an intuitive illustration for the block-shared matrix $\boldsymbol{R}_s$. Therefore, PE-OPT only needs to learn $\boldsymbol{R}^{(1)}$ in order to construct the orthogonal matrix of size $d \times d$.

Second, we consider that all the diagonal block matrices are independent, indicating that $\boldsymbol{R} = \text{Diag}(\boldsymbol{R}^{(1)}, \boldsymbol{R}^{(2)}, \cdots, \boldsymbol{R}^{(k)})$ where $\boldsymbol{R}^{(i)}, \forall i$ are different orthogonal matrices in general. We term such matrix $\boldsymbol{R}$ as unconstrained block matrix. Therefore, we have the unconstrained block diagonal matrix $\boldsymbol{R}_u$ as

$$\boldsymbol{R}_u = \begin{bmatrix} \boldsymbol{R}^{(1)} & 0 & \cdots & 0 \\ 0 & \boldsymbol{R}^{(2)} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \boldsymbol{R}^{(k)} \end{bmatrix} \tag{48}$$

where the orthogonal matrices $\boldsymbol{R}^{(i)}, \forall i$ will be learned independently. The effective number of parameters for the orthogonal matrix $\boldsymbol{R}_u$ is $\frac{d^2}{k}$, making it more flexible than the block-shared matrix $\boldsymbol{R}_s$.

Let's consider a convolution neuron (*i.e.*, convolution filter) $\boldsymbol{v} \in \mathbb{R}^{c_1 \times c_2 \times c_3}$ (*e.g.*, a typical convolution neuron is of size $3 \times 3 \times 64$) as an example. The orthogonal matrix $\boldsymbol{R}$ for the convolution neuron is of size $(c_1 c_2 c_3) \times (c_1 c_2 c_3)$. Typically, we will divide the neuron into k sub-neuron along the c_3-axis, each with size $c_1 \times c_2 \times \frac{c_3}{k}$. Then in order to learn a block-shared

orthogonal matrix $\boldsymbol{R}_s$, we will essentially learn a shared orthogonal matrix of size $(\frac{1}{k}c_1c_2c_3) \times (\frac{1}{k}c_1c_2c_3)$ that applies to each sub-neuron (there are k sub-neurons of size $c_1 \times c_2 \times \frac{c_3}{k}$ in total). For the case of learning a unconstrained block-diagonal orthogonal matrix $\boldsymbol{R}_u$, we simply learn different orthogonal matrices for different sub-neurons.

I.2. Experiments and Results

We conduct the image recognition experiments on CIFAR-100 with CNN-6 described in Table 14. The setting is exactly the same as Section 6.2. For the convolution filter, we use the size of $3 \times 3 \times 64$, *i.e.*, $c_1 = 3, c_2 = 3, c_3 = 64$. The results are given in Table 19 and Table 20. "# Parameters" in both tables denote the number of effective parameters for the orthogonal matrix $\boldsymbol{R}$ in a single layer. The baseline with fixed neurons is only to train the final classifiers with the randomly initialized neuron weights staying fixed. It means that this baseline basically removes the learnable orthogonal matrices but still fixes the neuron weights, so it only achieves 73.81% testing error. As expected, as the number of effective parameters goes down, the performance of PE-OPT generally decreases. One can also observe that using separate orthogonal matrices generally yields better performance than shared orthogonal matrices. $k = 2$ and $k = 4$ seems to be a reasonable trade-off between better accuracy and less parameters.

When k becomes larger (*i.e.*, the number of parameters become less) in the case of block-shared orthogonal matrices, we find that PE-OPT (LS) performs the best among all the variants. When k becomes larger (*i.e.*, the number of parameters become less) in the case of unconstrained block orthogonal matrices, we can see that both PE-OPT (GS) and PE-OPT (LS) performs better than the other variants.

Method	# Parameters	PE-OPT (CP)	PE-OPT (GS)	PE-OPT (HR)	PE-OPT (LS)	PE-OPT (OGD)
$c_3/k = 64$ $(k = 1)$ (*i.e.*, Original OPT)	331.7K	33.53	**33.02**	35.67	34.48	33.33
$c_3/k = 32$ $(k = 2)$	82.9K	34.93	**34.39**	35.83	34.50	35.06
$c_3/k = 16$ $(k = 4)$	20.7K	39.40	39.13	39.67	**37.58**	39.80
$c_3/k = 8$ $(k = 8)$	5.2K	47.77	46.65	46.69	**45.62**	47.43
$c_3/k = 4$ $(k = 16)$	1.3K	56.65	55.91	55.69	**54.75**	57.15
$c_3/k = 2$ $(k = 32)$	0.3K	63.46	62.65	62.38	**61.60**	62.46
$c_3/k = 1$ $(k = 64)$	0.1K	67.36	67.11	67.05	**66.61**	67.23
Baseline	-			37.59		
Baseline with fixed random neurons	-			73.81		

Table 19: Testing error (%) on CIFAR-100 with different settings of PE-OPT (with block-shared orthogonal matrix $\boldsymbol{R}_s$).

Method	# Parameters	PE-OPT (CP)	PE-OPT (GS)	PE-OPT (HR)	PE-OPT (LS)	PE-OPT (OGD)
$c_3/k = 64$ $(k = 1)$ (*i.e.*, Original OPT)	331.7K	33.53	**33.02**	35.67	34.48	33.33
$c_3/k = 32$ $(k = 2)$	165.9K	33.54	**33.15**	35.65	34.09	34.27
$c_3/k = 16$ $(k = 4)$	82.9K	34.77	**34.50**	35.71	34.96	35.97
$c_3/k = 8$ $(k = 8)$	41.5K	37.25	36.43	36.40	**36.17**	39.75
$c_3/k = 4$ $(k = 16)$	20.7K	40.74	**39.89**	39.98	39.93	43.43
$c_3/k = 2$ $(k = 32)$	10.4K	45.36	44.77	44.83	**44.61**	48.98
$c_3/k = 1$ $(k = 64)$	5.2K	50.94	**49.16**	49.57	49.23	54.93
Baseline	-			37.59		
Baseline with fixed random neurons	-			73.81		

Table 20: Testing error (%) on CIFAR-100 with different settings of PE-OPT (with unconstrained block orthogonal matrix $\boldsymbol{R}_u$).

J. On Generalizing OPT: Over-Parameterized Training with Constraint

OPT opens many new possibilities in training neural networks. We consider a simple generalization to OPT in this section to showcase the great potential of OPT. Instead of constraining the over-parameterization matrix $\boldsymbol{R} \in \mathbb{R}^{d\times d}$ in Eq. 1 to be orthogonal, we can use any meaningful structural constraints for this matrix, and even regularize it in a task-driven way. Furthermore, instead of a linear over-parameterization (*i.e.*, multiplying a matrix $\boldsymbol{R}$) to the neuron, we can also consider nonlinear mapping. We come up with the following straightforward generalization to OPT (the settings and notations exactly follow Eq. 1):

$$\begin{aligned}
\text{Standard:}\quad & \min_{\boldsymbol{v}_i,u_i,\forall i} \sum_{j=1}^{m} \mathcal{L}\big(y, \sum_{i=1}^{n} u_i \boldsymbol{v}_i^\top \boldsymbol{x}_j\big) \\
\text{Original OPT:}\quad & \min_{\boldsymbol{R},u_i,\forall i} \sum_{j=1}^{m} \mathcal{L}\big(y, \sum_{i=1}^{n} u_i (\boldsymbol{R}\boldsymbol{v}_i)^\top \boldsymbol{x}_j\big) \\
& \text{s.t.}\quad \boldsymbol{R}^\top\boldsymbol{R} = \boldsymbol{R}\boldsymbol{R}^\top = \boldsymbol{I} \\
\text{Generalized OPT:}\quad & \min_{\boldsymbol{R},u_i,\forall i} \sum_{j=1}^{m} \mathcal{L}\big(y, \sum_{i=1}^{n} u_i (\mathcal{T}(\boldsymbol{v}_i))^\top \boldsymbol{x}_j\big) \\
& \text{s.t.}\quad \text{Some constraints on } \mathcal{T}(\cdot)
\end{aligned} \tag{49}$$

where $\mathcal{T}(\cdot) : \mathbb{R}^d \to \mathbb{R}^d$ denotes some transformation (including both linear and nonlinear). Notice that the generalized OPT (G-OPT) no longer requires orthogonality. Such formulation of G-OPT can immediately inspire a number of instances. We will discuss some obvious ones here.

If we consider $\mathcal{T}(\cdot)$ to be a linear mapping, we may constrain $\boldsymbol{R}$ to be symmetric positive definite other than orthogonal. A simple way to achieve that is to use Cholesky factorization $\boldsymbol{L}\boldsymbol{L}^\top$ where $\boldsymbol{L}$ is a lower triangular matrix to parameterize the matrix $\boldsymbol{R}$. Essentially, we learn a lower triangular matrix $\boldsymbol{L}$ and use $\boldsymbol{L}\boldsymbol{L}^\top$ to replace $\boldsymbol{R}$ in OPT. The positive definiteness provides the transformation $\boldsymbol{R}$ with some geometric constraint. Specifically, a positive definite $\boldsymbol{R}$ only transforms the neuron weight $\boldsymbol{v}$ to the direction that has the angle less than $\frac{\pi}{2}$ to $\boldsymbol{v}$, because $\boldsymbol{v}^\top\boldsymbol{R}\boldsymbol{v} > 0$. Moreover, we can also require the transformation to have structural constraints on $\boldsymbol{R}$. For example, $\boldsymbol{R}$ can be upper (lower) triangular, banded, symmetric, skew-symmetric, upper (lower) Hessenberg, etc.

We can also consider $\mathcal{T}(\cdot)$ to be a nonlinear mapping. A obvious example is to use a neural network (*e.g.*, MLP, CNN) as $\mathcal{T}(\cdot)$. Then the nonlinear G-OPT will share some similarities with HyperNetworks [18] and Network-in-Network [46]. If we further consider $\mathcal{T}(\cdot)$ to be dependent on the input, then the nonlinear G-OPT will have close connections to dynamic neural networks [32, 51].

To summarize, OPT provides a novel and effective framework to train neural networks and may inspire many different threads of future research.

K. Hyperspherical Energy Training Dynamics of Individual Layers

We also plot the hyperspherical energy ($\boldsymbol{E}(\hat{\boldsymbol{v}}_i|_{i=1}^n) = \sum_{i=1}^n \sum_{j=1, j\neq i}^n \|\hat{\boldsymbol{v}}_i - \hat{\boldsymbol{v}}_j\|^{-1}$ in which $\hat{\boldsymbol{v}}_i = \frac{\boldsymbol{v}_i}{\|\boldsymbol{v}_i\|}$ is the i-th neuron weight projected onto the unit hypersphere.) in every layer of CNN-6 during training to show how these hyperspherical energies are being minimized. From Fig. 23, we can observe that OPT can always maintain the minimum hyperspherical energy during the entire training process, while the MHE regularization cannot. Moreover, the hyperspherical energy of the baseline will also decrease as the training proceeds, but it is still much higher than the OPT training.

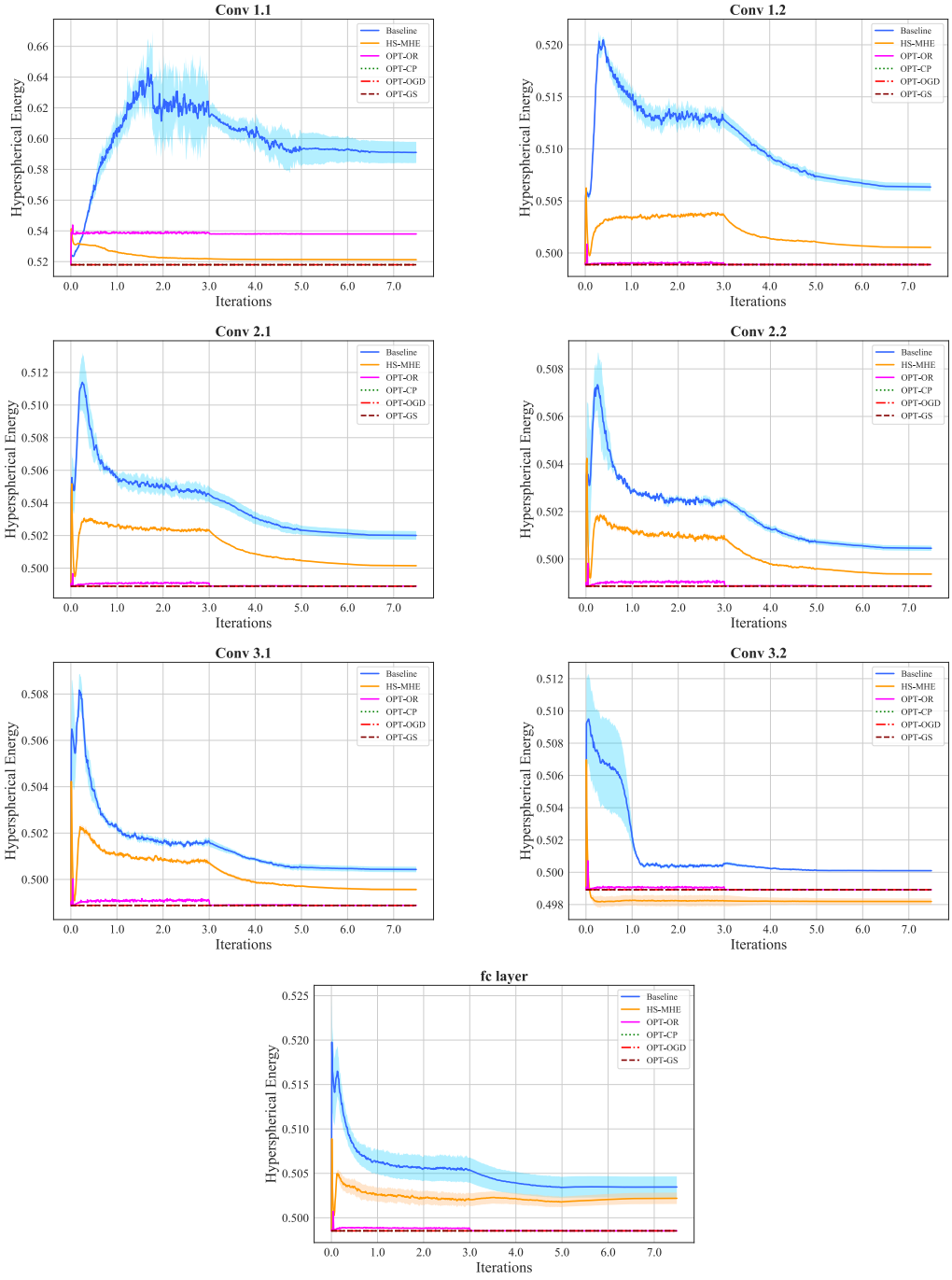


Figure 23: Training dynamics of hyperspherical energy in each layer of CNN-6. We average results with 10 runs.

L. Geometric Properties of Randomly Initialized Neurons

There are many interesting geometric properties [5, 6] of random points distributed independently and uniformly on the unit hypersphere. We summarize a few of them that make randomly initialized neurons distinct from any deterministic neuron configuration. Note that, there exist many deterministic neuron configurations that can also achieve very low hyperspherical energy, and this section aims to describe a few unique geometric properties of randomly initialized neurons.

There are two widely used geometric properties corresponding to a neuron configuration (*i.e.*, a set of neurons) $\hat{\boldsymbol{W}}_N = \{\hat{\boldsymbol{w}}_1, \cdots, \hat{\boldsymbol{w}}_N \in \mathbb{S}^d\}$. In the main paper, we define neurons on $\mathbb{S}^{d-1}$, but without loss of generality we define neurons on $\mathbb{S}^d$ here for convenience. The first one is the *covering radius*:

$$\alpha(\hat{\boldsymbol{W}}_N) := \alpha(\hat{\boldsymbol{W}}_N; \mathbb{S}^{d-1}) := \max_{\boldsymbol{u}\in\mathbb{S}^d} \min_{1\leq i\leq N} \arccos(\boldsymbol{u}, \hat{\boldsymbol{w}}_i) \tag{50}$$

which is the biggest geodesic distance from a neuron in $\mathbb{S}^d$ to the nearest point in $\hat{\boldsymbol{W}}_N$. The second one is the *separation distance*:

$$\psi(\hat{\boldsymbol{W}}_N) := \min_{1\leq i,j,\leq N, i\neq j} \arccos(\hat{\boldsymbol{w}}_i, \hat{\boldsymbol{w}}_j) \tag{51}$$

which gives the least geodesic distance between arbitrary two points in $\hat{\boldsymbol{W}}_N$. Random points (*i.e.*, randomly initialized neurons) typically have poor separation properties, since the separation is sensitive to the specific placement of points. [5] shows an example on $\mathbb{S}^1$ to illustrate this observation.

[5] considers a different but related quantity, *i.e.*, the sume of powers of the "hole radii". A set of neurons $\hat{\boldsymbol{W}}_N$ on $\mathbb{S}^d$ uniquely defines a convex polytope, which can be viewed as the convex hull of the neuron configuration. Each facet of the polytope defines a "hole". Such a hole denotes the maximal spherical cap for a facet that contains neurons of $\hat{\boldsymbol{W}}_N$ only on the boundary. It is easy to see that the geodesic radius of the largest hole is the covering radius $\alpha(\hat{\boldsymbol{W}}_N)$. We assume that for the set of neurons $\hat{\boldsymbol{W}}_N$, there are f_d holes (*i.e.*, facets) in total. Therefore, the i-th hole radius is defined as $\rho_i = \rho_i(\hat{\boldsymbol{W}}_N)$ which is the Euclidean distance in $\mathbb{R}^{d+1}$ between the center of the i-th spherical cap and the boundary. The i-th spherical cap is located on the sphere corresponding to the i-th facet. We have that $\rho_i = 2\sin(\frac{\alpha_i}{2})$ where α_i is the geodesic radius of the i-th spherical cap. We are interested in the sums of the p-th powers of the hole radii, *i.e.*,

$$\mathcal{P} = \sum_{i=1}^{f_d} (\rho_i)^p \tag{52}$$

where p is larger than zero. For large p, the largest hole dominates:

$$\lim_{p\to\infty} (\mathcal{P})^{\frac{1}{p}} = \lim_{p\to\infty} \left(\sum_{i=1}^{f_d} (\rho_i)^p \right)^{\frac{1}{p}} = \max_{1\leq i\leq f_d} \rho_i = 2\sin(\frac{\alpha(\hat{\boldsymbol{W}}_N)}{2}) \tag{53}$$

where $\rho(\hat{\boldsymbol{W}}_N) := \max_{1\leq i\leq f_d} \rho_i$. Then we introduce some useful notations to state the geometric properties. Let ψ_d be the surface area of $\mathbb{S}^d$, and we have that

$$\psi_d = \frac{2\pi^{\frac{d+1}{2}}}{\Gamma(\frac{d+1}{2})}, \tag{54}$$

and we also define the following quantities (with $\psi_0 = 2$):

$$\begin{aligned} \kappa_d &:= \frac{1}{d}\frac{\psi_{d-1}}{\psi_d} = \frac{1}{d}\frac{\Gamma(\frac{d+1}{2})}{\sqrt{\pi}\Gamma(\frac{d}{2})} \\ B_d &:= \frac{2}{d+1}\frac{\kappa_{d^2}}{(\kappa_d)^d} \end{aligned} \tag{55}$$

where κ_d can be alternatively defined with the recursion: $\kappa_1 = \frac{1}{\pi}$ and $\kappa_d = \frac{1}{2\pi d\kappa_{d-1}}$. [61] gives the expected number of facets constructed from N random neurons that are independently and uniformly distributed on the unit hypersphere $\mathbb{S}^d$:

$$\mathbb{E}[f_d] = B_d N\big(1 + o(1)\big) \tag{56}$$

where $N \to \infty$. Then we introduce the main results of [5] (asymptotics for the expected moments of the hole radii) in the following lemma:

Lemma 5. *If $p \geq 0$ and $\hat{\boldsymbol{w}}_1, \cdots \hat{\boldsymbol{w}}_N$ are N neurons on $\mathbb{S}^d$ that are independently and randomly distributed with respect to the normalized surface area measure σ_d on $\mathbb{S}^d$, then we have that*

$$\begin{aligned} \mathbb{E}[\mathcal{P}] &= B_d(\kappa_d)^{-\frac{p}{d}} \frac{\Gamma(d+\frac{p}{d})\Gamma(N+1)}{\Gamma(d)\Gamma(N+\frac{p}{d})}\big(1+\mathcal{O}(N^{-\frac{2}{d}})\big) \\ &= c_{d,p}N^{1-\frac{p}{d}}\big(1+\mathcal{O}(N^{-\frac{2}{d}})\big) \end{aligned} \tag{57}$$

as $N \to \infty$, where $\rho_i = \rho_{i,N}$ is the Euclidean hole radius associated with the i-th facet of the convex hull of $\hat{\boldsymbol{W}}_N$, $c_{d,p} := B_d B_{d,p}$, and $B_{d,p} := \frac{\Gamma(d+\frac{p}{d})}{\Gamma(d)}(\kappa_d)^{-\frac{p}{d}}$. The $\mathcal{O}$-terms above depend on d and p.

As we mentioned, there are many deterministic point (*i.e.*, neuron) configurations such as minimizing hyperspherical energy (*i.e.*, Riesz s-energy) [49] (as $s \to \infty$, the minimal s-energy points approach the best separation), maximizing the determinant for polynomial interpolation [64], Fibonacci points, spherical t-designs, minimizing covering radius (*i.e.*, best covering problem), maximizing the separation (*i.e.*, best packing problem) and maximizing the s-polarization, etc. We note that randomly initialized neurons are quite different from these deterministic neuron configurations and have unique geometric properties.