Universal Neural Functionals

Allan Zhou 12 Chelsea Finn 2 James Harrison 3

Abstract

A challenging problem in many modern machine learning tasks is to process weight-space features, i.e., to transform or extract information from the weights and gradients of a neural network. Recent works have developed promising weight-space models that are equivariant to the permutation symmetries of simple feedforward networks. However, they are not applicable to general architectures, since the permutation symmetries of a weight space can be complicated by recurrence or residual connections. work proposes an algorithm that automatically constructs permutation equivariant models, which we refer to as universal neural functionals (UNFs), for any weight space. Among other applications, we demonstrate how UNFs can be substituted into existing learned optimizer designs, and find promising improvements over prior methods when optimizing small image classifiers and language models. Our results suggest that learned optimizers can benefit from considering the (symmetry) structure of the weight space they optimize. We open-source our library for constructing UNFs at https: //github.com/AllanYangZhou/

universal_neural_functional.

1. Introduction

Many problems in machine learning require handling weight-space features, such as the weights, gradients, or sparsity masks of neural networks. For example, optimizers iteratively map the current weights and gradient history to updated weights. Taking this perspective, researchers have proposed a variety of data-driven methods that train a neural network to process these weight-space features. Examples applications of these neural functionals (Zhou et al., 2023a) include training neural networks to predict classifier

generalization from weights (Eilertsen et al., 2020), to optimize other networks (Metz et al., 2022), and to classify or edit implicit neural representations (INRs) (De Luigi et al., 2023).

Until recently, researchers lacked a unifying and principled framework for designing neural functionals, and would implement a custom model for their particular weightspace task. A significant recent advance was the development of weight-space models that are permutation equivariant (Navon et al., 2023; Zhou et al., 2023a). Neuron permutation symmetries arise in a neural network's weight space because re-ordering hidden neurons has no effect on the network's function (Hecht-Nielsen, 1990). A permutation equivariant neural functional can guarantee that under a neuron permutation of its input, its output permutes accordingly.

Navon et al. (2023) showed that permutation equivariance significantly improves performance on weight-space tasks, but their models only apply to the weight spaces of simple feedforward multilayer perceptrons (MLPs). Permutation equivariant neural functionals (Zhou et al., 2023a) added the ability to process weights from simple feedforward convolutional networks (CNNs). However, in practice we may deal with the weight spaces of complex networks that have residual connections, recurrence, normalization layers, and so on. Extending existing approaches to each possible weight space would be tedious and challenging.

We propose an approach that automatically constructs permutation equivariant models for any collection of tensors whose dimensions can permute according to a shared set of permutations. This naturally encompasses the permutation equivariance we might desire for any given weight space. We show that our algorithm constructs the most general linear layer that operates on a given weight space while guaranteeing equivariance to the specified permutation symmetries. Stacking multiple such layers with pointwise nonlinearities produces a deep permutation equivariant model, which we refer to as a universal neural functional.

To evaluate the empirical effectiveness of UNFs, we apply them to tasks involving a variety of weight spaces, including those of recurrent neural networks (RNNs) and Transformers. We incorporate UNFs into the implementation of learned optimizers and use them to optimize small im-

¹Work done at Google DeepMind ²Stanford University ³Google DeepMind. Correspondence to: Allan Zhou <ayz@cs.stanford.edu>, James Harrison <jamesharrison@google.com>.

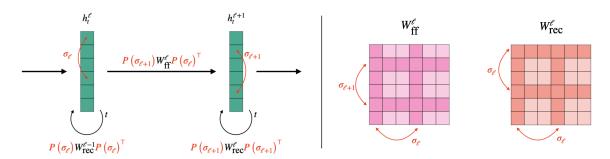


Figure 1. Illustration of the permutation symmetries in the weight space of a recurrent neural network (Example 2.2). Left: Each layer contains feedforward (ff) weights mapping between different layer's activations, and recurrent (rec) weights transforming activations over time. We can permute the hidden activations as illustrated without changing the final outputs h_t^L . Right: Permuting the hidden activations induces a permutation on the weights. Here, the rows and columns of the feedforward weights are permuted by $(\sigma_{\ell+1}, \sigma_{\ell})$, while the recurrent weights are permuted by $(\sigma_{\ell}, \sigma_{\ell})$. Our algorithm automatically constructs permutation equivariant models for any collection of weight tensors given a description of its symmetries (Appendix A).

age classifiers and language models, observing promising improvements over prior methods. In a generalization prediction task, we use UNF to predict the performance of sequence-to-sequence RNN models from their weights. Our experiments show that universal neural functionals are flexible, can be easily applied to different weight spaces, and improve upon prior weight-space methods.

2. Preliminaries

We largely follow or extend the notation and naming of Zhou et al. (2023a). Given a fixed neural network architecture, there is a **weight space** \mathcal{W} of possible parameters (weights, biases, normalization scalings, etc.). We refer to all such parameters as "weights". A particular set of weights $W = (W^{(1)}, \cdots, W^{(L)})$ contains multiple "tensors", or multidimensional arrays. Depending on the architecture, \mathcal{W} contains numerous symmetries (Hecht-Nielsen, 1990; Godfrey et al., 2022), i.e., transformations on the weight space that do not affect the network's behavior. Following prior work (Navon et al., 2023; Zhou et al., 2023a), this work focuses only on the permutation symmetries, which are called *neuron permutations*.

Neuron permutations correspond to re-arranging the neurons within (hidden) layers, which have no canonical ordering. We make the simplifying assumption that *all* layers can be re-arranged—this assumption can be later corrected using positional encodings (Zhou et al., 2023a). Assuming there are N independently permutable layers of neurons, the neuron permutation $group\ \mathcal{S}$ is the direct product:

$$S = S_{n_1} \times \dots \times S_{n_N}, \tag{1}$$

where n_i is the number of neurons being permuted in each layer.

In general, each weight is a "tensor" (multi-dimensional

array) of real numbers. Using $M(a,b,\cdots)$ to denote arrays $\mathbb{R}^{a \times b \times \cdots}$, consider a rank- D_ℓ tensor

$$W^{(\ell)} \in M\left(n_{d_1^{\ell}}, \cdots, n_{d_{D_{\ell}}^{\ell}}\right). \tag{2}$$

Each dimension d_i^{ℓ} is permuted by $\sigma_{d_i^{\ell}}$. That is, the action of σ on the indices of the weight tensor is:

$$\sigma(i_1, \cdots, i_{D_\ell}) := \left(\sigma_{d_1^\ell}(i_1), \cdots, \sigma_{d_{D_\ell}^\ell}(i_{D_\ell})\right). \tag{3}$$

Defining the multi-index $\vec{i} := (i_1, \dots, i_{D_\ell})$, the action on the weight tensor is to permute the entries:

$$\left[\sigma W^{(\ell)}\right]_{\vec{i}} := W^{(\ell)}_{\sigma^{-1}(\vec{i})},\tag{4}$$

and the action on W is:

$$\sigma W := \left(\sigma W^{(1)}, \cdots, \sigma W^{(L)}\right). \tag{5}$$

We now elaborate on the definition of the group and action in several common cases.

Example 2.1 (Multilayer perceptron). A multilayer perceptron (MLP) with L+1 layers has activations:

$$h^{\ell+1} = s\left(W^{(\ell)}h^{\ell} + b^{(\ell+1)}\right), \quad h^1 := x, y := h^{L+1}$$
(6)

where each $h^{\ell} \in \mathbb{R}^{n_{\ell}}$. The weights and biases are rank-2 and rank-1 tensors:

$$W^{(\ell)} \in M(n_{\ell+1}, n_{\ell})$$
 and $b^{(\ell)} \in M(n_{\ell})$. (7)

Then we have a neuron permutation group $\mathcal{S} = S_{n_1} \times \cdots \times S_{n_{L+1}}$, and $\sigma \in \mathcal{S}$ can be written $\sigma = (\sigma_\ell)_{\ell=1}^{L+1}$. The action on the weights and biases is:

$$W^{(\ell)} \mapsto P\left(\sigma_{\ell+1}\right) W^{(\ell)} P\left(\sigma_{\ell}\right)^{\top}, \tag{8}$$

$$b^{(\ell)} \mapsto P\left(\sigma_{\ell}\right) b^{(\ell)} \tag{9}$$

where $P\left(\sigma_{\ell}\right)$ is the $n_{\ell} \times n_{\ell}$ permutation matrix corresponding to σ_{ℓ} . This corresponds exactly to the "NP" setting in Zhou et al. (2023a).

Example 2.2 (Recurrent neural network). Consider a deep recurrent neural network (RNN) (Elman, 1990) without biases. We follow the presentation of Wang et al. (2023):

$$h_t^{\ell+1} = s \left(W_{\text{rec}}^{\ell+1} h_{t-1}^{\ell+1} + W_{\text{ff}}^{\ell} h_t^{\ell} \right),$$

$$h_t^1 := x_t, y_t := h_t^{L+1}$$
(10)

where x_t, y_t are the inputs and outputs at each timestep and h_0^{ℓ} is initialized to 0. The weight space consists of feedforward (ff) and recurrent (rec) weights:

$$W_{\mathrm{ff}}^{\ell} \in M\left(n_{\ell+1}, n_{\ell}\right) \quad \text{ and } \quad W_{\mathrm{rec}}^{\ell} \in M\left(n_{\ell}, n_{\ell}\right).$$
 (11)

We again define the neuron permutation group $S := S_{n_1} \times \cdots \times S_{n_{L+1}}$, but the action of the group on the weight space is now different. Here, re-arranging the neurons corresponds to transforming the weights:

$$W_{\text{ff}}^{\ell} \mapsto P\left(\sigma_{\ell+1}\right) W_{\text{ff}}^{\ell} P\left(\sigma_{\ell}\right)^{\top},$$

$$W_{\text{rec}}^{\ell} \mapsto P\left(\sigma_{\ell}\right) W_{\text{rec}}^{\ell} P\left(\sigma_{\ell}\right)^{\top}.$$
(12)

As illustrated by Figure 1, the feedforward weights transform just as in the MLP case (Eq. 8), but the recurrent weights' rows and columns must be transformed by the *same* permutation.

Example 2.3 (Convolutional neural network). Consider a 1D convolutional neural network (CNN) without biases, with activations:

$$h^{\ell+1} = s\left(W^{(\ell)} \star h^{\ell}\right), \quad h^1 := x, y := h^{L+1}, \quad (13)$$

where \star is cross-correlation. If each filter has spatial dimension k_ℓ and each h^ℓ has n_ℓ channels, then we have rank-3 weight tensors $W^{(\ell)} \in M(n_{\ell+1},n_\ell,k_\ell)$ and neuron permutation group:

$$S = \prod_{\ell=1}^{L} S_{n_{\ell}} \times S_{k_{\ell}}.$$
 (14)

Looking at how each dimension of $W^{(\ell)}$ permutes, we would have $\sigma_{n_{\ell+1}} \in S_{n_{\ell+1}}$ permute the first dimension (output channels), $\sigma_{n_{\ell}} \in S_{n_{\ell}}$ permute the second dimension (input channels), and $\sigma_{k_{\ell}} \in S_{k_{\ell}}$ permute the third dimension (spatial).

We note that permutating the spatial dimensions of a convolution filter would change the CNN's behavior and is not a true symmetry of the weight space. This is a notable difference between how our framework handles convolutional weight spaces compared to NFNs (Zhou et al., 2023a), where the action of the neuron permutation group does not

affect the spatial dimensions at all. Assuming that all dimensions of each weight tensor can permute simplifies the development of our framework, and undesired symmetry can be broken (if desired) by positional encodings of the input (Zhou et al., 2023a; Lim et al., 2023).

Equivariance and invariance. We are interested in functions $T: \mathcal{W} \to \mathcal{W}$ that are equivariant, meaning that it doesn't matter whether we apply a neuron permutation to the input or the output. We define $\mathbb{L}_{\mathcal{S}}(\mathcal{W}, \mathcal{W})$ as the space of equivariant linear maps, i.e., those T satisfying:

$$T(\sigma W) = \sigma T(W), \forall \sigma \in \mathcal{S}, W \in \mathcal{W}.$$
 (15)

Our goal is to design a layer (i.e., a parameterized space of functions) that is equivalent to $\mathbb{L}_{\mathcal{S}}(\mathcal{W},\mathcal{W})$.

In some applications, we may instead desire invariance, that is a function P satisfying

$$P(\sigma W) = P(W), \forall \sigma \in \mathcal{S}, W \in \mathcal{W}.$$
 (16)

Following prior work (Navon et al., 2023; Zhou et al., 2023a), we can build invariant neural functionals by composing several equivariant layers with an invariant pooling layer, e.g., one that sums over every dimension of each weight tensor and concatenates the results.

3. Universal neural functionals

Since equivariance is preserved under composition, and pointwise non-linearities are already permutation equivariant, we can build deep equivariant models as long as we have an equivariant linear layer. Additionally, composing equivariant layers with an invariant pooling operation produces a deep invariant model. This section introduces a method for producing equivariant weight-space layers for any given weight space, which enables the flexible construction of *universal neural functionals*.

3.1. Decomposing equivariant weight-space maps

The weight space is a direct sum of individual weight subspaces,

$$\mathcal{W} = \mathcal{W}^{(1)} \oplus \cdots \oplus \mathcal{W}^{(L)}. \tag{17}$$

The problem of defining an equivariant layer on W can be decomposed into defining equivariant layers between each pair of weight subspaces $W^{(m)}$ and $W^{(\ell)}$, for all ℓ and m (Navon et al., 2023).

We re-state this result in our own notation. For any ℓ, m pair we define $\mathbb{L}_{\mathcal{S}}\left(\mathcal{W}^{(m)}, \mathcal{W}^{(\ell)}\right)$ as the space of equivariant maps between the two weight subspaces. It contains all $T^{\ell m}: \mathcal{W}^{(m)} \to \mathcal{W}^{(\ell)}$ satisfying

$$T^{\ell m}\left(\sigma W^{(m)}\right) = \sigma T^{\ell m}\left(W^{(m)}\right) \quad \forall \sigma, W^{(m)}, \quad (18)$$

noting that the action on the left and right hand sides of the equivariance condition are not, in general, the same.

Assume that we already have a basis \mathcal{B}^{sp} for $\mathbb{L}_{\mathcal{S}}(\mathcal{W}^{(p)},\mathcal{W}^{(s)})$. A basis function $E \in \mathcal{B}^{sp}$ can be extended to $\bar{E}: \mathcal{W} \to \mathcal{W}$ by defining:

$$\bar{E}(W)^{\ell} := \begin{cases} E\left(W^{(p)}\right) & \ell = s \\ 0 & \text{otherwise} \end{cases}, \tag{19}$$

where $\bar{E}(W) := (\bar{E}^1(W), \cdots, \bar{E}^L(W)).$

Theorem 3.1 (Navon et al. (2023)). Let $\{\mathcal{B}^{\ell m}\}$ be bases for each $\mathbb{L}_{\mathcal{S}}(\mathcal{W}^{(m)},\mathcal{W}^{(\ell)})$. Then the union of these bases (extended by Eq. 19) is a basis for linear equivariant maps on \mathcal{W} . That is,

$$\mathcal{B} = \bigcup_{\ell, m \in [\![L]\!]^2} \left\{ \bar{E} \mid E \in \mathcal{B}^{\ell m} \right\}$$
 (20)

is a basis for $\mathbb{L}_{\mathcal{S}}(\mathcal{W}, \mathcal{W})$.

This result tells us that we can construct an equivariant basis \mathcal{B} for $\mathbb{L}_{\mathcal{S}}(\mathcal{W},\mathcal{W})$ by simply combining the equivariant bases $\{\mathcal{B}^{\ell m}\}$ for each pair of weight subspaces.

3.2. Equivariant layers between tensors

Since weights are tensors, our decomposed problem involves finding bases for permutation equivariant maps between tensors. Variants of this problem have been studied by numerous prior works—in particular, Maron et al. (2018) theoretically characterize a basis for equivariant maps between arbitrary-rank tensors, and provide a concrete implementation of the basis functions in the rank-2 case. Here, we describe a *general* algorithm that automatically constructs a basis for permutation equivariant maps between arbitrary-rank tensors. Concretely, it implements each basis function in terms of simple array operations that are amenable to efficient computation with modern deep learning frameworks.

Functions in $\mathbb{L}_{\mathcal{S}}\left(\mathcal{W}^{(m)},\mathcal{W}^{(\ell)}\right)$ take input tensors indexed by $\{i_1,\cdots,i_{D_m}\}$ and produces output tensors indexed by $\{o_1,\cdots,o_{D_\ell}\}$. We can construct a basis $\mathcal{B}^{\ell m}$ for this space where each element is identified by a **valid partition** \mathcal{P} of these indices. Recall that the indices (i_1,i_2,\cdots) of $W^{(m)}$ are permuted by $\left(\sigma_{d_1^m},\sigma_{d_2^m},\cdots\right)$. We say that two indices i_1 and i_2 "permute simultaneously" if $d_1^m=d_2^m$.

Definition 1. A **valid partition** is a partition \mathcal{P} of the output and input indices $\mathcal{I} = \{o_1, \cdots, o_{D_\ell}, i_1, \cdots, i_{D_m}\}$ into non-empty subsets, such that each subset only contains indices that are permuted simultaneously.

Example 3.1 $(W^{(m)} = W^{(\ell)} = \mathbb{R}^{n_1 \times n_2})$. Here the output and input indices are $\{o_1, o_2, i_1, i_2\}$. The partition $\{\{o_1, o_2\}, \{i_1, i_2\}\}$ is **not** valid because o_1, o_2 are permuted by σ_1, σ_2 , so they do not permute simultaneously. On the other hand, $\{\{o_1, i_1\}, \{o_2, i_2\}\}$ is a valid partition.

Example 3.2 ($W^{(m)} = W^{(\ell)} = \mathbb{R}^{n_1 \times n_1}$). This time, the partition $\{\{o_1, o_2\}, \{i_1, i_2\}\}$ is valid because o_1, o_2 are both permuted by σ_1 , as are i_1, i_2 .

To construct the equivariant basis, we enumerate all valid partitions and then map each partition $\mathcal P$ to a basis function $E_{\mathcal P}$. Concretely, we label each subset of $\mathcal P$ with a distinct character $\alpha,\beta,\gamma,\cdots$ and then remap each of our original indices $\{\,o_1,\cdots,o_{D_\ell},i_1,\cdots,i_{D_m}\,\}$ to a a character based on which subset the index was in. This mapping is best illustrated by continuing our previous example.

Example 3.3 $(W^{(m)} = W^{(\ell)} = \mathbb{R}^{n_1 \times n_2})$. Here input and output are both matrices, with combined indices $\{o_1, o_2, i_1, i_2\}$. We have two permutations $(\sigma_1, \sigma_2) \in S_{n_1} \times S_{n_2}$ that can act on the rows and columns of the input and output matrices. There are four valid partitions:

$$\mathcal{P}_{1} = \{ \{ o_{1}, i_{1} \}, \{ o_{2}, i_{2} \} \},
\mathcal{P}_{2} = \{ \{ o_{1}, i_{1} \}, \{ o_{2} \}, \{ i_{2} \} \},
\mathcal{P}_{3} = \{ \{ o_{1} \}, \{ i_{1} \}, \{ o_{2}, i_{2} \} \},
\mathcal{P}_{4} = \{ \{ o_{1} \}, \{ o_{2} \}, \{ i_{1} \}, \{ i_{2} \} \}.$$
(21)

Consider \mathcal{P}_2 —we assign a character to each subset:

$$\mathcal{P}_2 = \left\{ \underbrace{\left\{ o_1, i_1 \right\}}_{\alpha}, \underbrace{\left\{ o_2 \right\}}_{\beta}, \underbrace{\left\{ i_2 \right\}}_{\gamma} \right\}. \tag{22}$$

which tells us to remap the output indices $(o_1, o_2) \mapsto (\alpha, \beta)$ and the input indices $(i_1, i_2) \mapsto (\alpha, \gamma)$, producing the basis function $E_{\mathcal{P}_2}$ defined:

$$E_{\mathcal{P}_2}\left(W^{(m)}\right)_{\alpha\beta} := \sum_{\gamma} W_{\alpha\gamma},\tag{23}$$

where summation over γ can be inferred because it only contains an input index.

Repeating this index-remapping process for each valid partition will generate a total of four basis functions $E_{\mathcal{P}_1}, \cdots, E_{\mathcal{P}_4}$ for $\mathbb{L}_{\mathcal{S}}\left(\mathcal{W}^{(m)}, \mathcal{W}^{(\ell)}\right)$. Our equivariant $\mathcal{W}^{(m)} \to \mathcal{W}^{(\ell)}$ layer will be defined as a linear combination of them:

$$T^{\ell m}\left(W^{(m)};\lambda\right) := \sum_{k=1}^{4} \lambda_k \cdot E_{\mathcal{P}_k}\left(W^{(m)}\right),\qquad(24)$$

which is the layer introduced in Hartford et al. (2018).

To generalize the previous example, for each valid partition of the indices \mathcal{P} we label its subsets with characters $\alpha, \beta, \gamma, \cdots$ and then construct a basis function:

$$E(W^{(m)})_{c[o_1],\cdots,c[o_{D_\ell}]} = \sum_{\mathcal{R}} W^{(m)}_{c[i_1],\cdots,c[i_{D_m}]}, \quad (25)$$

where $c[\cdot]$ maps each index to the subset of \mathcal{P} that contains it. We sum over the characters in \mathcal{R} , which is the (possibly

empty) subset of characters that only contain input indices (i.e., only appear on the right-hand side). Entries that are not explicitly assigned by the left-hand side are 0. Algorithm 1 gives a formal description of the complete process for generating $\mathcal{B}^{\ell m}$.

Algorithm 1 Basis for equivariant $\mathcal{W}^{(m)} \to \mathcal{W}^{(\ell)}$ layer

Require: $\mathcal{W}^{(m)}$, $\mathcal{W}^{(\ell)}$

1: Initialize basis $\mathcal{B}^{\ell m} \leftarrow \{ \}$

2: Define indices $\mathcal{I} \leftarrow \{o_1, \cdots, o_{D_\ell}, i_1, \cdots, i_{D_m}\}$

3: for \mathcal{P} in ValidPartitions (\mathcal{I}) do

Label each subset $s_p \in \mathcal{P}$ by unique character $CHAR(s_p)$

for $\alpha \in \mathcal{I}$ do 5:

Map index $c[\alpha] \leftarrow CHAR(s_p)$ where $\alpha \in s_p$ 6:

7:

 $E_{\mathcal{P}}(X)_{c[o_1],\cdots,c_{o[D_\ell]}} := \sum_{\mathcal{R}} X_{c[i_1],\cdots,c[i_{D_m}]}$ $\mathcal{B}^{\ell m} \leftarrow \mathcal{B}^{\ell m} \cup \{ E_{\mathcal{P}} \}$ 8:

10: **end for**

11: return $\mathcal{B}^{\ell m}$

Once Algorithm 1 has generated a basis of equivariant functions $\mathcal{B}^{\ell m}$, we can implement an equivariant layer

$$T^{\ell m}\left(W^{(m)}; \lambda^{\ell m}\right) := \sum_{b=1}^{|\mathcal{B}^{\ell m}|} \lambda_b^{\ell m} \cdot E_{\mathcal{P}_b}\left(W^{(m)}\right), \quad (26)$$

where $\lambda^{\ell m} \in \mathbb{R}^{|\mathcal{B}^{\ell m}|}$ is a vector of learned coefficients, one per basis function.

3.3. Equivariant layers on weight spaces

Theorem 3.1 now tells us that we may now construct the equivariant weight-space layer by combining the bases $\{\mathcal{B}^{\ell m}\}\$ into a basis \mathcal{B} of functions on \mathcal{W} . The weight-space layer $T(\cdot, \lambda)$ could then be defined by a linear combination of the basis functions with learned coefficients λ .

Equivalently, we may instead combine the layers $\{T^{\ell m}\}$:

$$T^{1}\left(W,\lambda^{1,:}\right) = \sum_{m=1}^{L} T^{1m}\left(W^{(m)},\lambda^{1m}\right)$$

$$\vdots$$

$$T^{L}(W; \lambda^{L,:}) = \sum_{m=1}^{L} T^{Lm}(W^{(m)}, \lambda^{Lm})$$
 (27)

where $\lambda^{\ell,:} = \{ \lambda^{\ell m} \mid \ell = 1, \cdots, L \}$. Then the full weightspace layer is defined:

$$T(W,\lambda) = \left(T^{1}(W,\lambda^{1,:}), \cdots, T^{L}(W,\lambda^{L,:})\right), \quad (28)$$

parameterized by $\lambda = (\lambda^{1,:}, \dots, \lambda^{L,:}).$

Appendix A provides a concrete description of how we specify the weight space in code and how the algorithm is then used to automatically construct an equivariant weight space layer. Our open-source implementation is compatible with most JAX (Bradbury et al., 2018) neural network libraries.

Theorem 3.2. The weight-space layer defined by Eqs. 27-28 is S-equivariant, and can express any linear equivariant function on W.

Proof. Each $T^{\ell m}$ is a linear combination of basis functions in $\mathcal{B}^{\ell m}$ by coefficients in $\lambda^{\ell m}$. Then, as described by Thm 3.1, Eq. 27 is a linear combination of basis functions for $\mathbb{L}_{\mathcal{S}}(\mathcal{W},\mathcal{W})$ by the coefficients in λ . Hence the set of functions $\{T(\cdot,\lambda)\}_{\lambda}$ that the layer can express is exactly $\mathbb{L}_{\mathcal{S}}(\mathcal{W},\mathcal{W}).$

For an MLP weight space with neuron permutation group defined as in Example 2.1, this approach will generate the exact same layer as NFN_{NP} (Zhou et al., 2023a). This is because the layers each parameterize all possible linear maps equivariant to the same symmetry group, and hence can express the same set of functions.

3.4. Multiple feature channels

In practice, we may be interested in simultaneously processing multiple weight-space features, such as the weights and a history of gradients. These features can be stacked into a "channel" dimension analogous to the channels of convolutional networks. In that case, we must consider direct sums of weight spaces of the form $\mathcal{W}^c = \bigoplus_{k=1}^c \mathcal{W}$, with elements that can be written as¹:

$$W = (W[1], \dots, W[c]), \quad W[k] \in \mathcal{W}.$$
 (29)

Then the action is $\sigma W := (\sigma W[1], \cdots, \sigma W[c])$ for $\sigma \in \mathcal{S}$, extending the (single channel) definition given in Eq. 5. The definition of equivariance can then be extended to layers of the form $T(\cdot): \mathcal{W}^{c_i} \to \mathcal{W}^{c_o}$, where c_i, c_o are the number of input and output channels.

Extending equivariant layers to the multi-channel setting is quite common in the geometric deep learning literature and simply involves taking linear combinations along the channel dimension (Cohen & Welling, 2016; Ravanbakhsh et al., 2017). That is, we modify the equivariant layer between subspaces as:

$$T^{\ell m} \left(W^{(m)}; \lambda^{\ell m} \right) [k']$$

$$:= \sum_{b=1}^{|\mathcal{B}^{\ell m}|} \sum_{k=1}^{c_i} \lambda_b^{\ell m} [k', k] \cdot E_{\mathcal{P}_b} \left(W^{(m)} \right) [k], \quad (30)$$

¹In the multichannel setting we overload notation and use Wto refer to elements of \mathcal{W}^c , not \mathcal{W} .

where each $\lambda_b^{\ell m}$ is now a learned $c_o \times c_i$ matrix instead of a scalar. Again assembling these subspace layers according to Eq. 27 results in an equivariant weight-space layer $T(\cdot; \lambda): \mathcal{W}^{c_i} \to \mathcal{W}^{c_o}$.

3.5. Deep models

The previous sections describes the construction of S-equivariant layers that operate operate on weight-space features in \mathcal{W}^c . We construct *universal neural functionals* by stacking multiple such layers (interleaved with pointwise non-linearities) into a deep, permutation equivariant model that can process weights. To construct a permutation invariant model, we can add an invariant pooling layer after the equivariant layers, as in prior work (Navon et al., 2023; Zhou et al., 2023a).

4. Experiments

In this section, we refer to weight-space models constructed using our algorithm as **universal neural functionals (UNFs)**. We compare their performance to prior methods on two types of weight-space tasks: predicting the generalization of recurrent sequence-to-sequence models, and training learned optimizers for a variety of architectures and datasets.

4.1. RNN generalization prediction

Method	Test $ au$		
STATNN	0.8839 ± 0.0007		
UNF (Ours)	0.8968 ± 0.0006		

Table 1. Rank correlation between predicted and actual success rates of RNNs in the Tiny RNN Zoo. The goal is to predict the success rate of a Seq2Seq model on an arithmetic task by only looking at its weights. Implementing predictors with UNFs significantly outperforms STATNN (Unterthiner et al., 2020).

One promising application of neural functionals is in predicting the generalization of neural network models from their weights (Eilertsen et al., 2020). We construct **Tiny RNN Zoo**², a dataset of recurrent neural networks trained to do arithmetic by completing given questions character-by-character. For example, given the input string "15+20=" the correct completion would be "35<EOS>". To construct the dataset, we train 10⁴ sequence-to-sequence (Sutskever et al., 2014) models on example problems with input numbers up to five input digits. Both encoder and decoder RNNs contain a single GRU cell (Chung et al., 2014) with hidden size 128. Each model is trained with a distinct learning rate and batch size, and it's test success rate (SR) is recorded. The learning rate is sampled from a log-uniform distribution over

 $[10^-4, 10^-2]$, and the batch size is sampled uniformly from $\{64, 128, 256\}$. With the goal of predicting test SR from weights, we split the Tiny RNN Zoo into 8000/1000/1000 training, validation, and test examples.

The success rate of each RNN model is clearly invariant under permutation symmetries of its weights, so invariance is a natural inductive bias for any generalization predictor. We evaluate STATNN (Unterthiner et al., 2020) and a UNF-based predictor (note that NFNs are not applicable to the weights of recurrent networks). STATNN is operates on basic statistical features³ of the weights, and has been shown to be a very strong baseline on previous generalization prediction tasks (Unterthiner et al., 2020). On the other hand, UNF operates on raw weight inputs and may be able to extract more nuanced signals than STATNN, as was shown (for CNN classifiers) in Zhou et al. (2023a).

In particular, STATNN computes the mean, variance, and (0,25,50,75,100)-percentiles of each weight tensor in the RNN and feeds them into a six-layer MLP with hidden width 600. UNF is a permutation invariant model, implemented using a three-layer equivariant backbone (16 hidden channels) followed by invariant pooling and a three-layer MLP (512 hidden neurons). We train each predictor with binary cross entropy loss (since the target SR is in [0,1]), using the Adam optimizer with learning rate 0.001, batch size 10, and training for up to 10 epochs. We use the validation data only for early stopping, and assess the performance of each predictor on the test inputs using Kendall's τ , the rank correlation between predicted and actual success rate.

Results. Table 1 shows the performance of each predictor on held out weight inputs. Our UNF-based predictor achieves significantly higher rank correlation between predicted and actual success rate, suggesting that the equivariant layers are able to extract more informative features from the raw weights compared to STATNN.

4.2. Learned optimizers

Choosing the optimizer is a key step in training any modern neural network. Though most popular optimizers are variants of stochastic descent, the non-convexity of neural network training leaves few rigorous guidelines for ideal optimizer design. This has led some researchers to propose *training* good optimizers using some form of metalearning (Bengio et al., 1990; 2013; Andrychowicz et al., 2016; Wichrowska et al., 2017; Metz et al., 2019).

Common optimizers today (including the learned ones) are equivariant to any permutation of the weights. This is because permuting the weights also permutes the gradients, so stochastic gradient descent and similar optimizers will

²Inspired by the Tiny CNN Zoo (Unterthiner et al., 2020).

³Notably, it computes statistics that are invariant to permutations of the weights.

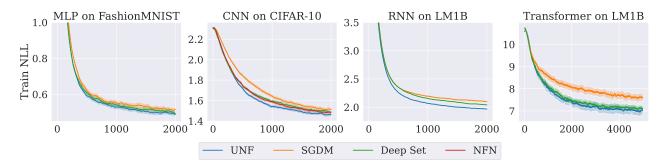


Figure 2. Training loss (negative log-likelihood) curves for different tasks and architectures using meta-learned optimizers. We implement learned optimizers with either universal neural functionals (UNFs), NFNs (Zhou et al., 2023a), or Deep Sets (Zaheer et al., 2017). Deep Sets are the current standard choice for implementing learned optimizers. Note that NFN is identical to UNF in the MLP case, different for CNN case, and not applicable to RNNs or Transformers. All loss curves are smoothed and averaged over 5 random initializations (3 for Transformer), with shaded regions showing standard error.

produce permuted updates. However, equivariance to *any* permutation ignores the actual symmetry structure of the optimized neural network. Arguably the more appropriate constraint is to only require equivariance to the *neuron permutation group*, which enables more expressive optimizers while still respecting the symmetries of the weight space. As we will see, this can be achieved by using UNFs to implement a learned optimizer.

Training learned optimizers that generalize well is extremely compute-intensive (Metz et al., 2022), so we conduct our experiments in several smaller settings to analyze the impact of architecture choice on learned optimizer performance. In each setting, an optimizer is meta-trained to optimize an architecture type on a task from random initializations. Following Harrison et al. (2022), our learned optimizers have the form:

$$m_t^{\gamma} \leftarrow \gamma m_{t-1} + \nabla_t$$

$$W_{t+1} \leftarrow W_t - \alpha \left(m_t^{\gamma_0} + \beta f \left(W_t, \nabla_t, \left\{ m_t^{\gamma_i} \right\}_i, t \right) \right).$$

$$(31)$$

Here $\alpha m_t^{\gamma_0}$ is a "nominal term" that biases the learned optimizer to behave like stochastic gradient descent with momentum coefficient γ_0 . The neural functional $f(\cdot)$ ingests weights W_t , gradients ∇_t , momentum terms at several coefficients $\{m_t^{\gamma_i}\}_i$, and the iteration t.

During meta-training, we optimize network f and scalars α, β, γ_0 to minimize the task training loss after a fixed number of training steps T, the "inner training horizion." To avoid the issue of backpropagating through an optimization process, we estimate meta-gradients using persistent evolutionary strategies (Vicol et al., 2021).

Comparisons. The default architecture choice for $f(\cdot)$ in prior work is **Deep Sets** (Zaheer et al., 2017), which offers equivariance to *any* permutation symmetry. We study the effect of replacing Deep Sets by **UNFs**. We also try the **NFN**_{NP} architecture (Zhou et al., 2023a) where applicable,

though it cannot be used on the RNN and Transformer experiments. Finally, we consider stochastic gradient descent with momentum (**SGDM**), which is equivalent to fixing $\beta=0$ in Eq. 31. The SGDM baseline is also meta-trained to tune the learning rate α and momentum decay rate γ_0 . We compare the different learned optimizers in four tasks:

MLP on FashionMNIST. Each optimizer trains an MLP classifier on a downsized and flattened version of the FashionMNIST dataset (Xiao et al., 2017). We note that for MLP weight spaces, UNF are identical to NFN_{NP} (Zhou et al., 2023a).

CNN on CIFAR-10. Each optimizer trains a convolutional classifier on a downsized 16×16 CIFAR-10. In this setting our algorithm produces a UNF that is *different* to NFN_{NP} (see Example 2.3).

RNN on LM1B. Each optimizer trains a character-level RNN-based language model (LM) on the One Billion Word Language Model Benchmark (LM1B) dataset (Chelba et al., 2013).

Transformer on LM1B. Each optimizer trains a Transformer LM on LM1B, this time predicting tokens instead of characters.

We use an inner training horizon $T=2{,}000$ for the first three tasks and $T=5{,}000$ for the Transformer task, since it takes longer to train. When implementing $f(\cdot)$ for each method, we use a network with four layers, 32 hidden channels, and ReLU nonlinearities. The Deep Set optimizer uses exclusively Deep Set layers (Zaheer et al., 2017, Eq. 4), while the UNF and NFN optimizers uses three Deep Set layers followed by a single UNF or NFN layer. See Appendix B.1-B.2 for full descriptions of the tasks and meta-training.

Results. Figure 2 shows the training curves produced by each of the meta-trained optimizers in each experiment.

Learned optimizers with deep architectures (UNF, Deep Set, or NFN) outperform SGDM, even after tuning SGDM's learning rate and momentum decay. UNF typically learns fastest and achieves the lowest training loss across all methods, though Deep Set and NFN can be comparable in some settings. One interesting observation is that UNF outperforms NFN in the CNN experiment. As noted in Example 2.3, UNFs make the stronger assumption that all tensor dimensions—including the spatial dimensions of the convolution filter—are permutable, while NFNs do not. Although the UNF assumption is technically incorrect, the stronger assumption leads to a lower parameter count (see Table 3 in the appendix) which may be easier for meta-optimization.

Overall, our results show the promise of using UNFs to create more expressive learned optimizers that utilize the specific symmetry structure of the weight spaces they optimize. Further work could investigate their capacity for generalization to new tasks and architectures, for example by meta-training on diverse tasks (Metz et al., 2022). Moreover, as Table 3 in the appendix shows, a necessary trade-off of UNFs being more expressive is that they require more parameters for an equivalent number of layers and hidden channels. Since learned optimizers are still much smaller than the networks they could optimize, this may not be a significant computational constraint in practice. Still, it could be a challenge to meta-optimization, since evolutionary strategies are known to struggle in higher dimensions. Hence, further work on efficient high-dimensional metagradient estimators would complement the development of expressive weight-space models like UNF.

5. Related Work

There is a long history of neural network architectures that are equivariant to various symmetry groups (LeCun et al., 1995; Cohen & Welling, 2016; Ravanbakhsh et al., 2017; Kondor & Trivedi, 2018; Cohen et al., 2018). Existing frameworks for automatically constructing equivariant models (Finzi et al., 2021) produce equivariant matrices, which would be intractable for our task. Our work constructs efficient equivariant basis functions for a particular class of permutation symmetries that arise in the weight spaces of neural networks. Permutation equivariant networks have been developed for sets (Zaheer et al., 2017), matrices whose rows and columns permute independently (Hartford et al., 2018), and tensors under higher-order permutation actions (Thiede et al., 2020; Pan & Kondor, 2022)-the latter may also be viewed as equivariant models on graphs or polytopes (Maron et al., 2018; Albooyeh et al., 2019). This work observes that a weight space is a collection of tensors under higher-order permutation symmetries, and develops equivariant models for that setting.

There has been significant interest in designing architec-

tures that that either optimize or generate neural network weights (Schmidhuber, 1993; Ha et al., 2016; Krueger et al., 2017; Kirsch & Schmidhuber, 2021; Peebles et al., 2022; Metz et al., 2022). Some works have identified the importance of respecting the relevant symmetries when implementing black box meta-learners (Kirsch et al., 2022). However, precise characterizations of equivariant models on neural weight spaces are relatively recent and were initially restricted to simple feedforward models (Navon et al., 2023; Zhou et al., 2023a;b).

A recent alternative approach has instead leveraged message passing neural networks (MPNNs) (Zhang et al., 2023) to process weights as edges of a graph. Concurrent to this work, Kofinas et al. (2024) demonstrated applications of MPNNs to learned optimization for MLPs and CNNs and Lim et al. (2023) extended MPNNs to process more general weight-spaces. Our approach gives maximally expressive equivariant linear layers for collections of tensors that describe (hyper)graphs with multiple node sets. Hence, UNFs can be viewed as a type of of equivariant graph network (Maron et al., 2018), in constrast to an MPNN-style approach.

6. Conclusion

We introduce a method for constructing permutationequivariant neural functionals that operate on arbitrary weight spaces, removing a major limitation of previous frameworks that were only applicable to the weight spaces of simple MLPs and CNNs. Our algorithm constructs maximally expressive equivariant linear layers for processing any collection of tensors given a description of their permutation symmetries, and implements these layers in terms of efficient array operations in standard deep learning frameworks. We empirically validate that the resulting universal neural functionals (UNFs) are effective at tasks that involve processing the weights and gradients of convolutional image classifiers, recurrent sequence-to-sequence models, and Transformer language models. In particular, we find that UNFs show promising improvements over existing learned optimizer designs in small scale experiments.

Limitations and future work. It remains to be demonstrated how UNFs can be applied to heterogenous weight-space inputs, e.g., to have a single UNF act as a learned optimizer for any input architecture. Moreover, our experimental results only validate the promise of UNF-based learned optimizers in relatively limited settings, and more work would needed to test generalization across arbitrary tasks. Finally, computational tractability may be a significant challenge for more complex architectures as the number of basis terms generated by Alg. 1 would grow rapidly for higher rank tensors with higher-order interactions. Resolving these challenges would further improve the scalability and applicability of neural functionals to weight-space tasks.

7. Acknowledgements

We thank Jascha Sohl-Dickstein and Yiding Jiang for insightful general discussions about the project, and Louis Kirsch for helpful feedback on early drafts. AZ is supported by the NSF Graduate Research Fellowship Program. We are grateful to the TPU Research Cloud (TRC) for providing compute for some of the experiments.

References

- Albooyeh, M., Bertolini, D., and Ravanbakhsh, S. Incidence networks for geometric deep learning. *arXiv* preprint *arXiv*:1905.11460, 2019.
- Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and De Freitas, N. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.
- Bengio, S., Bengio, Y., Cloutier, J., and Gescei, J. On the optimization of a synaptic learning rule. In *Optimality in Biological and Artificial Networks?*, pp. 281–303. Routledge, 2013.
- Bengio, Y., Bengio, S., and Cloutier, J. *Learning a synaptic learning rule*. Université de Montréal, Département d'informatique et de recherche ..., 1990.
- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.
- Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T. One billion word benchmark for measuring progress in statistical language modeling. *arXiv* preprint arXiv:1312.3005, 2013.
- Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv* preprint arXiv:1412.3555, 2014.
- Cohen, T. and Welling, M. Group equivariant convolutional networks. In *International conference on machine learning*, pp. 2990–2999. PMLR, 2016.
- Cohen, T. S., Geiger, M., Köhler, J., and Welling, M. Spherical CNNs. *arXiv preprint arXiv:1801.10130*, 2018.
- De Luigi, L., Cardace, A., Spezialetti, R., Zama Ramirez, P., Salti, S., and Di Stefano, L. Deep learning on implicit neural representations of shapes. In *International Conference on Learning Representations (ICLR)*, 2023.

- Eilertsen, G., Jönsson, D., Ropinski, T., Unger, J., and Ynnerman, A. Classifying the classifier: dissecting the weight space of neural networks. *arXiv preprint arXiv:2002.05688*, 2020.
- Elman, J. L. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
- Finzi, M., Welling, M., and Wilson, A. G. A practical method for constructing equivariant multilayer perceptrons for arbitrary matrix groups. In *International Conference on Machine Learning*, pp. 3318–3328. PMLR, 2021.
- Godfrey, C., Brown, D., Emerson, T., and Kvinge, H. On the symmetries of deep learning models and their internal representations. *Advances in Neural Information Processing Systems*, 35:11893–11905, 2022.
- Ha, D., Dai, A., and Le, Q. V. Hypernetworks. *arXiv* preprint arXiv:1609.09106, 2016.
- Harrison, J., Metz, L., and Sohl-Dickstein, J. A closer look at learned optimization: Stability, robustness, and inductive biases. *Advances in Neural Information Processing Systems*, 35:3758–3773, 2022.
- Hartford, J., Graham, D., Leyton-Brown, K., and Ravanbakhsh, S. Deep models of interactions across sets. In *International Conference on Machine Learning*, pp. 1909– 1918. PMLR, 2018.
- Hecht-Nielsen, R. On the algebraic structure of feedforward network weight spaces. In *Advanced Neural Computers*, pp. 129–135. Elsevier, 1990.
- Kirsch, L. and Schmidhuber, J. Meta learning backpropagation and improving it. *Advances in Neural Information Processing Systems*, 34:14122–14134, 2021.
- Kirsch, L., Flennerhag, S., van Hasselt, H., Friesen, A., Oh, J., and Chen, Y. Introducing symmetries to black box meta reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 7202–7210, 2022.
- Kofinas, M., Knyazev, B., Zhang, Y., Chen, Y., Burghouts, G. J., Gavves, E., Snoek, C. G., and Zhang, D. W. Graph Neural Networks for Learning Equivariant Representations of Neural Networks. In 12th International Conference on Learning Representations (ICLR), 2024.
- Kondor, R. and Trivedi, S. On the generalization of equivariance and convolution in neural networks to the action of compact groups. In *International Conference on Machine Learning*, pp. 2747–2755. PMLR, 2018.

- Krueger, D., Huang, C.-W., Islam, R., Turner, R., Lacoste, A., and Courville, A. Bayesian hypernetworks. *arXiv* preprint arXiv:1710.04759, 2017.
- Le, Q. V., Jaitly, N., and Hinton, G. E. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.
- LeCun, Y., Bengio, Y., et al. Convolutional networks for images, speech, and time series. *The handbook of brain* theory and neural networks, 3361(10):1995, 1995.
- Lim, D., Maron, H., Law, M. T., Lorraine, J., and Lucas, J. Graph metanetworks for processing diverse neural architectures. *arXiv preprint arXiv:2312.04501*, 2023.
- Maron, H., Ben-Hamu, H., Shamir, N., and Lipman, Y. Invariant and equivariant graph networks. *arXiv* preprint *arXiv*:1812.09902, 2018.
- Metz, L., Maheswaranathan, N., Nixon, J., Freeman, D., and Sohl-Dickstein, J. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pp. 4556–4565. PMLR, 2019.
- Metz, L., Harrison, J., Freeman, C. D., Merchant, A., Beyer, L., Bradbury, J., Agrawal, N., Poole, B., Mordatch, I., Roberts, A., et al. Velo: Training versatile learned optimizers by scaling up. arXiv preprint arXiv:2211.09760, 2022.
- Navon, A., Shamsian, A., Achituve, I., Fetaya, E., Chechik, G., and Maron, H. Equivariant architectures for learning in deep weight spaces. arXiv preprint arXiv:2301.12780, 2023.
- Pan, H. and Kondor, R. Permutation equivariant layers for higher order interactions. In *International Conference* on Artificial Intelligence and Statistics, pp. 5987–6001. PMLR, 2022.
- Peebles, W., Radosavovic, I., Brooks, T., Efros, A. A., and Malik, J. Learning to learn with generative models of neural network checkpoints. *arXiv preprint arXiv:2209.12892*, 2022.
- Ravanbakhsh, S., Schneider, J., and Poczos, B. Equivariance through parameter-sharing. In *International conference on machine learning*, pp. 2892–2901. PMLR, 2017.
- Schmidhuber, J. A 'self-referential' weight matrix. In *ICANN'93: Proceedings of the International Conference on Artificial Neural Networks Amsterdam, The Netherlands 13–16 September 1993 3*, pp. 446–450. Springer, 1993.

- Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.
- Thiede, E. H., Hy, T. S., and Kondor, R. The general theory of permutation equivarant neural networks and higher order graph variational encoders. *arXiv* preprint *arXiv*:2004.03990, 2020.
- Unterthiner, T., Keysers, D., Gelly, S., Bousquet, O., and Tolstikhin, I. Predicting neural network accuracy from weights. *arXiv preprint arXiv:2002.11448*, 2020.
- Vicol, P., Metz, L., and Sohl-Dickstein, J. Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In *International Conference on Machine Learning*, pp. 10553–10563. PMLR, 2021.
- Wang, L., Zhang, K., Zhou, A., Simchowitz, M., and Tedrake, R. Fleet policy learning via weight merging and an application to robotic tool-use. *arXiv* preprint *arXiv*:2310.01362, 2023.
- Wichrowska, O., Maheswaranathan, N., Hoffman, M. W., Colmenarejo, S. G., Denil, M., Freitas, N., and Sohl-Dickstein, J. Learned optimizers that scale and generalize. In *International conference on machine learning*, pp. 3751–3760. PMLR, 2017.
- Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.
- Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R., and Smola, A. Deep sets. doi: 10.48550. *arXiv preprint ARXIV.1703.06114*, 2017.
- Zhang, D. W., Kofinas, M., Zhang, Y., Chen, Y., Burghouts, G. J., and Snoek, C. G. Neural networks are graphs! graph neural networks for equivariant processing of neural networks. 2023.
- Zhou, A., Yang, K., Burns, K., Jiang, Y., Sokota, S., Kolter, J. Z., and Finn, C. Permutation equivariant neural functionals. *arXiv preprint arXiv:2302.14040*, 2023a.
- Zhou, A., Yang, K., Jiang, Y., Burns, K., Xu, W., Sokota, S., Kolter, J. Z., and Finn, C. Neural functional transformers. *arXiv preprint arXiv:2305.13546*, 2023b.

A. Weight-space specifications

Here we discuss the concrete **specification** that precisely describes a weight space and must be provided as input to the algorithm before it can construct equivariant weight-space layers. Our open-sourced implementation⁴ is compatible with most JAX (Bradbury et al., 2018) neural network libraries.

Suppose we wish to process an MLP's weights that are stored in a (nested) Python dictionary:

```
params = {
    "layer1": {"weight": Array[64, 32], "bias": Array[64]},
    "layer2": {"weight": Array[64, 64], "bias": Array[64]},
}
```

Then a specification should match the nested dictionary structure but provide a string or integer name for each dimension of each array. The name tells the algorithm which permutation affects which dimensions of each array.

In this example, the specification closely follows the MLP description in Example 2.1, where $W^{(1)} \in M(n_2, n_1)$ is permuted as $W^{(1)} \mapsto P(\sigma_2) W^{(1)} P(\sigma_1)^{\top}$.

```
specification = {
    "layer1": {"weight": ("n2", "n1"), "bias": ("n2",)},
    "layer2": {"weight": ("n3", "n2"), "bias": ("n3",)},
}
```

Providing this specification object to our algorithm is sufficient for it to deduce the symmetry group, its action, and construct the corresponding equivariant layer.

Since most neural networks consist of repeating layers or blocks, the process of constructing the specification can be semi-automated by first defining a function that creates the specification for a single layer or block and then re-using that function for each block. Although we did not find this necessary for our experiments, it may also be possible to automatically deduce the specifications for a network in common deep learning frameworks by analyzing its computation graph.

B. Experimental details

B.1. Learned optimization tasks

Here we describe each of the experimental settings we evaluated the learned optimizers on. Across all experiments, the training loss is negative log-likelihood.

MLP on FashionMNIST. Train a three-layer MLP classifier on a downsized (8×8) and flattened version of the FashionMNIST dataset (Xiao et al., 2017). The MLP has a hidden size of 32 and ReLU activation function. We use a batch size of 128.

CNN on CIFAR-10. Train a convolutional classifier on a downsized 16×16 CIFAR-10. The classifier has two convolutional layers (16 and 32 channels), followed by global average pooling and a linear classification head, and is trained with a batch size of 128.

RNN on LM1B. Trains a character-level RNN-based language model (LM) on LM1B (Chelba et al., 2013). The RNN itself has one hidden layer with size 64, and uses identity-initialization (Le et al., 2015). An embedding layer with dimension 32 maps tokens to embeddings before feeding into the RNN, and an output layer produces token predictions from the RNN output. The LM is trained to predict the next token with teacher forcing at batch size 64, on sequences of length 16.

Transformer on LM1B. Train a Transformer LM on LM1B, this time predicting tokens instead of characters. The Transformer has two blocks with an embedding dimension of 32, and uses four self-attention heads. We train with a batch size of 8 on length-8 sequences.

⁴https://github.com/AllanYangZhou/universal_neural_functional

Figure 3. Number of parameters used by $f(\cdot)$ in each learned optimizer, for each task. Note that NFN and UNF are identical for the MLP task. This count does not include the other meta-learned scalars in Eq. 31, which are α , γ_0 , β .

Task	UNF	Deep Set	NFN
MLP on FashionMNIST	3,783	2,788	3,783
CNN on CIFAR-10	7,369	2,788	41,603
RNN on LM1B	8,043	2,788	N/A
Transformer on LM1B	64,168	2,788	N/A

B.2. Learned optimization meta-training

Call DS[c] a single equivariant Deep Set layer (Zaheer et al., 2017, Eq 4) with c output channels (similarly for UNF[c] and NFN[c]). Then $f(\cdot)$ in our learned optimizers (Eq. 31) is always implemented as a feedforward architecture:

```
DeepSetOpt = DS[32] -> ReLU -> DS[32] -> ReLU -> DS[32] -> ReLU -> DS[1]
UNFOpt = DS[32] -> ReLU -> DS[32] -> ReLU -> DS[32] -> ReLU -> UNF[1]
NFNOpt = DS[32] -> ReLU -> DS[32] -> ReLU -> DS[32] -> ReLU -> NFN[1]
```

For all methods, we initialize $\alpha=0.1$ and $\gamma_0=0.9$ before starting meta-training. For non-SGDM methods, we initialize $\beta=0.001$, and provide six momentum values $\{m_t^{\gamma_i}\}_i$ with coefficients $\gamma_i=0.1,0.5,0.9,0.99,0.999,0.9999$. The iteration number t is converted into an 11-dimensional sinusoidal encoding, and all inputs to $f(\cdot)$ are concatenated along the channel dimension. Concretely, this results in an input in \mathcal{W}^{19} . The output is in \mathcal{W}^{1} .

We meta-train for 50,000 steps using Adam, estimating meta-gradients over 16 parallel training runs using persistent evolutionary strategies (PES) (Vicol et al., 2021) with a truncation length of 50 and a noise standard deviation of 0.01. The meta-training objective is training loss at the end of the inner training horizon (T = 5,000 for the Transformer setting, and T = 2,000 otherwise), and we apply a gradient clipping of 1.0.

Size of each learned optimizer $f(\cdot)$. Since Deep Set layers are agnostic to the specific weight space being optimized, the Deep Set learned optimizer uses the same number of parameters in each task. The same is not true of UNF layers, where the number of parameters grows in proportion to the size of the bases generated by Algorithm 1. Table 3 lists the number of parameters in $f(\cdot)$ for each learned optimizer.