INHERITANCE BETWEEN FEEDFORWARD AND CONVOLUTIONAL NETWORKS VIA MODEL PROJECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural-network techniques are often transferred across architecture families by analogy, but such transfer is valid only when the assumptions required by a technique are preserved. We introduce this idea as *inheritance* between model classes. Using a unified node-level framework with tensor-valued activations, we prove that generalized feedforward networks (GFFNs) form a strict subset of generalized convolutional networks (GCNNs), so GCNN properties transfer directly to GFFNs. The reverse direction is not automatic: standard CNN nodes use spatial kernels, while FFN nodes use one scalar weight per input contribution. We introduce *model projection* to recover a restricted reverse inheritance path. Projection freezes each convolutional input-channel sub-function and learns one scalar coefficient for each input-output channel contribution, giving projected CNN nodes the GFFN-style trainable structure of scalar-weighted input recombination. This inherited structure leads naturally to transfer learning. Across multiple ImageNet-pretrained CNN backbones and downstream image-classification datasets, model projection is competitive with standard transfer learning baselines and provides an effective initialization for subsequent full fine-tuning.

1 INTRODUCTION

Techniques and intuitions developed for one neural-network family are often reused in another, but this transfer is rarely made precise. FFNs and CNNs are a common example: ideas developed for one are frequently applied to the other, even though the two families are usually formalized with different input structures and node operations. We study this issue through the lens of *inheritance*: the transfer of a property between model classes through an explicit structural relationship that preserves the assumptions required by that property.

This perspective is especially useful for FFNs and CNNs because the two classes differ in their trainable input contributions. A traditional FFN node has one scalar weight per input contribution, whereas a standard CNN node has a spatial kernel for each input-output channel contribution. Thus, techniques whose assumptions depend on a linear combination in which each input contribution is controlled by a single scalar weight apply naturally to FFN nodes, but not automatically to ordinary CNN nodes. This motivates our central question: *when can CNNs and FFNs inherit properties from one another, and can such inheritance be used for transfer learning?*

A unified formalization. We first introduce a unified node-level formalization with tensor-valued activations. In this framework, generalized feedforward network (GFFN) nodes perform scalar-weighted summation across input channels while preserving tensor structure, whereas generalized convolutional network (GCNN) nodes apply channel-wise convolutional transformations and then sum across channels. This lets us compare the two model classes directly. We prove that GFFNs form a strict subset of GCNNs: a convolution with kernel size one in every spatial dimension reduces to scalar multiplication of each input channel. Therefore, any property that holds for all GCNNs also holds for the corresponding GFFN subclass.

Model projection. The reverse direction does not hold by subclass inclusion. Arbitrary CNNs are not GFFNs, because their input-output channel contributions are spatial kernels rather than scalar weights. To recover a restricted reverse direction, we introduce *model projection*. Projection uses the fact that CNN node functions are separable by input channel before the nonlinearity. It freezes each convolutional input-channel sub-function and learns one scalar coefficient for each input-output

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

channel contribution. The resulting projected node has the same scalar-weighted trainable summation structure as a GFFN node, possibly with inhomogeneous inputs.

Transfer learning benefit. This gives the precise inheritance statement needed for our application: projected CNN nodes inherit the FFN-style trainable structure of one scalar trainable weight per input contribution. When applied to convolutional layers, this inherited property directly yields a transfer learning method. Instead of adapting an input-output channel contribution by training an entire spatial kernel, model projection freezes the pretrained kernel and trains only a scalar coefficient. Model projection therefore preserves pretrained spatial feature extractors while allowing every convolutional layer to adapt through scalar channel-contribution weights rather than full spatial kernels (See Fig. 1).

Empirical evaluation. We evaluate model projection as a transfer learning method for ImageNet-pretrained CNN backbones. Across multiple downstream image classification datasets and architectures, projection is competitive with logistic regression, full fine-tuning, and established parameter efficient fine tuning (PEFT) baselines for transfer learning. Empirically, projection often combines the stability of partial tuning with the continued improvement of adapting the full model. In two-stage training, projection also provides a strong initialization for subsequent full fine-tuning.

Contributions. Our main contributions are:

- We introduce an inheritance framework between FFNs and CNNs. There is currently no inheritance framework between common deep learning models. A formally defined framework allows researchers to understand when a result on one type of model automatically works on another type as well. This can remove the need to test on both types of model. We prove that one direction of inheritance is always valid, and introduce projection to recover conditional inheritance in the other direction.
- We apply projection for transfer learning for a medium range of dataset sizes. For very small datasets, LR is usually best, while for large enough datasets FT is usually best. There are a range of dataset sizes that fall between these two ideals. We fill this gap with model projection, and compare it with PEFT methods that could also potentially fill this gap by adjusting the base model with fewer parameters.
- We provide an open-source implementation of projection and demonstrate its practical value for transfer learning across multiple public datasets, showing that projection is competitive across transfer learning benchmarks.

1.1 Related Work

Formalizations and inheritance across model families. Formal treatments of neural-network computations are often specialized to particular architectures and input shapes rather than expressed in a unified, tensor-level language. Classical discussions of feedforward networks typically assume vector-valued scalar inputs (Hastie et al., 2009), while standard formulations of convolutional networks are usually presented for specific spatial cases, most commonly two-dimensional inputs (LeCun et al., 2010). This separation makes it harder to state precise relationships between model classes, and therefore to identify when results or techniques from one class preserve the assumptions needed to apply to another. Although ideas are frequently transferred informally between FFNs and CNNs, explicit structural accounts of such transfer appear limited. This motivates a framework that makes the preserved node-level structure explicit.

Transfer learning Transfer learning adapts a pre-trained model to a downstream task, often in low-data regimes where full fine tuning can overfit or be difficult to optimize, so reducing the total number of parameters tuned during training is often done to avoid overfitting (Chollet, 2021; Han et al., 2024a; Jia et al., 2022; Wang et al., 2023). The main method of reducing the number of parameters in CNNs is to perform a partial fine-tuning of the model (Kornblith et al., 2019; Chollet, 2021; Han et al., 2024a; Jia et al., 2022; Wang et al., 2023). One drawback of this is that large parts of the pre-trained model will not get adapted to the new task or domain. PEFT methods aim to adapt or train a model using few parameters with the intent of improving compute efficiency, but can also reduce overfitting when adapting a model to a downstream task (Han et al., 2024b). Some PEFT methods can be easily applied to CNNs for transfer learning. BatchNorm adaptation updates affine normalization parameters (Ioffe & Szegedy, 2015), but these act after convolutional input

contributions have already been summed; projection instead rescales each fixed input-output channel contribution before aggregation. BitFit tunes bias terms (Zaken et al., 2022), but biases shift node outputs rather than selectively reweighting individual contributions. Other PEFT methods include scale-shift methods (Lian et al., 2022), residual and convolutional adapters (Rebuffi et al., 2017; Chen et al., 2024; Luo et al., 2023), low-rank adaptation (Hu et al., 2022), including convolution-specific variants (Ding et al., 2024), and sensitivity-aware methods (He et al., 2023).

Pre-defined CNNs. Model projection also differs from pre-defined CNNs, where a set of fixed filters produce outputs that are then passed to a layer of 1×1 convolutional nodes (Linse et al., 2023). In projection, the pretrained convolutional filter channels are not first aggregated into a shared intermediate representation. Instead, each output node rescales its own fixed input-channel contributions before summation. We elaborate on this distinction in Remark B.1.

2 Background and Problem

2.1 Problem

Our goal is to answer the following questions: *Under what conditions, if any, do CNNs naturally inherit results from FFNs, and vice versa*, and, *does this inheritance aid in transfer learning?* For example, traditional feed forward neural networks train with fewer parameters per node. When fine tuning a CNN foundation model for a downstream task, can we inherit this property to efficiently train an entire neural network with fewer parameters?

Before introducing the relevant background, we first formalize what is meant by a neural network model. We define a network model $M := M(\theta)$, where $\theta \in \mathbb{R}^p$, to be some function composed of $L \in \mathbb{N}$ sequential layers. That is, M can be expressed as a composition of L subfunctions or *layers* $M(\theta) = f_L \circ \ldots \circ f_1$. Given input Z_i and parameters W_i, each layer f_i produces a sequence of at least two outputs $a_{ij},\ j = 1, \ldots, J_i, J_i \geq 2$. Each output a_{ij} corresponds to a node f_{ij} and subset of parameters W_{ij}. We may refer to a_{ij} as the jth output *channel* of layer i. Precisely, each layer can be written as

$$\begin{aligned} f_i(Z_i, W_i) &= [f_{i1}(Z_i, W_{i1}), \ldots, f_{iJ_i}(Z_i, W_{iJ_i})] = [a_{i1}, \ldots, a_{iJ_i}] \\ &= [Z_{(i+1,1)}, \ldots, Z_{(i+1,J_i)}] = Z_{i+1}. \end{aligned}$$

2.2 Traditional Neural Networks

Traditional neural networks (FFNs) are a type of network model, where data flows along edges, and calculations are done in nodes. We summarize the description from Hastie et al. (2009). Traditionally, these calculations are linear regressions, followed by a non-linear function σ. Let subset W_{jk} be the subset of parameters W_j that are associated with input channel Z_k. Given an input $Z \in \mathbb{R}^d$, a parameter set W_j containing a vector of weights $W_{jk} \in \mathbb{R}^d$, and a bias $b_j \in \mathbb{R}$, the node will have the following form:

$$f_j(Z, W_j) = \sigma(\sum_{k=1}^{d} Z_k W_{jk} + b_j). \tag{1}$$

This formulation requires that the input and output data are vectors of scalars. While more general versions have been implemented in code (Chollet, 2021), to the best of our knowledge, all formal treatments have so far required non-scalar data to be vectorized first.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are another type of network model, but where some nodes perform a convolution. For simplicity, we describe the 2-dimensional case from LeCun et al. (2010), from which the more general case extends naturally. Given an input $Z \in \mathbb{R}^{d \times n_1 \times n_2}$, and a parameter set containing a filter $F_j \in \mathbb{R}^{d \times l_1 \times l_2}$ with filter channels $F_{jk} \in \mathbb{R}^{l_1 \times l_2}$, and a bias $B_j \in \mathbb{R}^{m_1 \times m_2}$, a convolutional node f_j will have the following form, where $*$ denotes the convolution operation:

$$f_j(Z, W_j) = \sigma(\sum_{k} F_{jk} * Z_k + B_j). \tag{2}$$

F_{jk} connects input channel Z_k to output a_j. We consider the node f_j to be the collection of operations and parameters connecting Z to output a_j, plus the bias. While CNNs often operate on 1-, 2-, or even 3-dimensional data, the common formulations often only specify the 2-dimensional case (LeCun et al., 2010; Chollet, 2021; Goodfellow et al., 2016).

3 Inheritance between CNNs and FFNs

3.1 Node Generalization and Model Subclasses

We can now address our earlier question of inheritance through *model subclasses* and *model projections*. We start by generalizing equations for commonly used neural network models. In order to do so, we define a generalization of the dot product that outputs a tensor called the tensor dot product. Let a tensor be a multidimensional array with shape D. Let Γ_D be a tensor of shape D where all values are 1.

Definition 3.1. For $Z \in \mathbb{R}^{d \times D}$, and $W_{jk} \in \mathbb{R}^d$ the simplified tensor dot product, denoted $\odot_t$, is $Z \odot_t W_{jk} = \sum_{k=1}^{d} Z_k W_{jk}$.

The simplified tensor dot product takes a vector Z, of d tensors, $Z_k \in \mathbb{R}^D$, of uniform size and shape D, as well as a vector of d scalars, W_{jk}, and outputs a tensor of size and shape D. Here, $Z_k W_{jk}$ is scalar multiplication. If the size of Z_k is 1 in all dimensions, $\odot_t$ reduces to the traditional dot product.

Remark 3.2. For ease of exposition, we introduced a simpler version of the tensor dot product, namely W_{jk} is assumed to be a vector of scalars, whereas commonly used implementations use a more general tensor dot product for such nodes, such as in dense layers in Tensorflow Abadi et al. (2015); Chollet (2021), and linear layers in Pytorch Ansel et al. (2024).

We now generalize FFN nodes to apply to structures of data other than vectors of scalars, and define a general convolutional node for a tensor of shape D. Let Γ_D be a tensor of shape D where all values are equal to one.

Definition 3.3. For $Z \in \mathbb{R}^{d \times D_{in}}$, $b_j \in \mathbb{R}, \Gamma_{D_{out}} \in \mathbb{R}^{D_{out}}$, and $W_{jk} \in \mathbb{R}^d$ the GFFN node is:

$$f_j(Z, W_j) = \sigma(Z \odot_t W_{jk} + b_j \Gamma_{D_{out}}). = \sigma(\sum_{k=1}^{d} Z_k W_{jk} + b_j \Gamma_{D_{out}}). \tag{3}$$

To generalize an FFN node from Equation 1, we use the tensor dot product and replace $b_j \in \mathbb{R}$ with $b_j \Gamma_{D_{out}} \in \mathbb{R}^{D_{out}}$. In other words, the bias is now a tensor with all values being b_j. Traditional neural network nodes are a special case of GFFN nodes where the input data is a vector of scalars. If the 3-dimensional input is a sequence of 2D image channels, then a GFFN node acts as a 1×1 CNN node. To act as a fully connected layer in the traditional sense, the input can be flattened first. We now define GCNN nodes.

Definition 3.4. For $F_j \in \mathbb{R}^{d \times D_{F_j}}$, $Z \in \mathbb{R}^{d \times D_{in}}$, and $b_j \Gamma_{D_{out}} \in \mathbb{R}^{D_{out}}$ the GCNN node is:

$$f_j(Z, W_j) = \sigma(Z * F_j + b_j \Gamma_{D_{out}}). = \sigma(\sum_{k=1}^{d} Z_k * F_{jk} + b_j \Gamma_{D_{out}}). \tag{4}$$

That is, to generalize a CNN node from Equation 2, we replace $F_j \in \mathbb{R}^{d \times l_1 \times l_2}$ with $F_j \in \mathbb{R}^{d \times D_{F_j}}$, we replace $Z \in \mathbb{R}^{d \times n_1 \times n_2}$ with $Z \in \mathbb{R}^{d \times D_{in}}$, and we replace $B_j \in \mathbb{R}^{m_1 \times m_2}$ with $b_j \Gamma_{D_{out}} \in \mathbb{R}^{D_{out}}$. Colloquially, the filters and data have the same number of dimensions in each channel, but the number of dimensions is not specified globally for the network. Instead of a vector of 2D images, we now have a vector of tensors. The bias is now a tensor with all values being b_j.

Now that we have formalized general definitions, we can prove that the class of traditional feed forward neural networks (FFNs) is a subclass of the class of convolutional neural networks (CNNs).

Theorem 3.5. *GFFNs are a strict subset of GCNNs.*

The proof of this result can be found in Appendix A. To prove Theorem 3.5, we establish a bijection between the set of GFFNs and a subset of GCNNs, namely the set of GCNNs with kernels of size one in all dimensions. An important consequence of Theorem 3.5 is the following.

Corollary 3.6. *Since the class of GFFNs is contained in the class of GCNNs, any property that holds for all GCNNs also holds for the corresponding subclass of GFFNs.*

Corollary 3.6 says that any results or techniques that apply to all GCNNs automatically apply to all GFFNs. Thus, any techniques and results from GCCNs carry over to GFFNs.

3.2 Model Projection

Of course, CNNs are not a subclass of FNNs. As a consequence, we now introduce *model projection*: a method to project CNNs and other non-FFN nodes, onto FFN nodes. This allows inheritance of many properties exclusive to FFNs, such as training a node with one weight per input. We first define a node function.

Definition 3.7. The *node function* f'_j associated with a node f_j is defined by

$$f_j(Z, W_j) = \sigma\big(f'_j(Z, W_j)\big),$$

that is, f'_j is the pre-activation (inner) function of the node.

We now define *separable by input*, a property that a node function is required to have in order for the node to be projected.

Definition 3.8. A node function f'_j is separable by input if, for an input Z with d elements, f'_j takes the following form: $f'_j(Z, W_j) = \sum_{k=1}^{d} f'_{jk}(Z_k, W_{jk}) + b_j \Gamma_{D_{out}}$, where each f'_{jk} depends only on the kth input component Z_k and a corresponding subset $W_{jk} \subset W_j$ of parameters.

"Separable by input" means that the node function decomposes into a sum of node sub-functions f'_{jk} plus a bias, where each f'_{jk} depends on only one input channel. For example, the function $g_1(x_1, x_2) = a_1 x_1^2 + a_2 x_2^2 + a_3 x_1 + a_4 x_2$, where $a_1, \dots a_4 \in \mathbb{R}$ is separable by input, since we can write

$$g_1(x_1, x_2) = (a_1 x_1^2 + a_3 x_1) + (a_2 x_2^2 + a_4 x_2) = g_{11}(x_1) + g_{12}(x_2).$$

On the other hand, the function $g_2(x_1, x_2) = a_1 x_1^2 + a_2 x_2^2 + a_3 x_1 + a_4 x_2 + x_1 x_2$ is not separable by input, because of the $x_1 x_2$ term. GCNN node functions are separable by input, see Lemma A.1 in the Appendix. Node functions that are separable by input can be "projected."

Definition 3.9. Given a node function that is separable by input, its *projected node* is defined as $\hat{f}_j(Z, W_j, \gamma_j) = \sigma(\sum_{k=1}^{d} \gamma_{jk} f'_{jk}(Z_k, W_{jk}) + b_j \Gamma_{D_{out}})$. A model's *GFFN projection* is the model that results from projecting all non-GFFN nodes that are separable by input and leaving the rest of the network unchanged.

Here, the idea is that if we view the weights W_{jk} as fixed, and consider training the new weights γ_{jk} for a downstream task, then the projected model is essentially a GFFN with node-specific pre-processing and inherits all properties, including training techniques, for a GFFN that do not rely on homogeneous inputs, which are discussed next.

We now introduce a property of a layer called *homogeneous inputs*.

Definition 3.10. A layer has *homogeneous inputs* if all nodes in the layer have the same input, otherwise it has inhomogeneous inputs. A node has homogeneous inputs if it is part of a layer with homogeneous inputs, otherwise it has inhomogeneous inputs.

Our next result says that projected nodes are GFFN nodes with inhomogeneous inputs.

Theorem 3.11. *Projected nodes are GFFN nodes with inhomogeneous inputs.*

t]

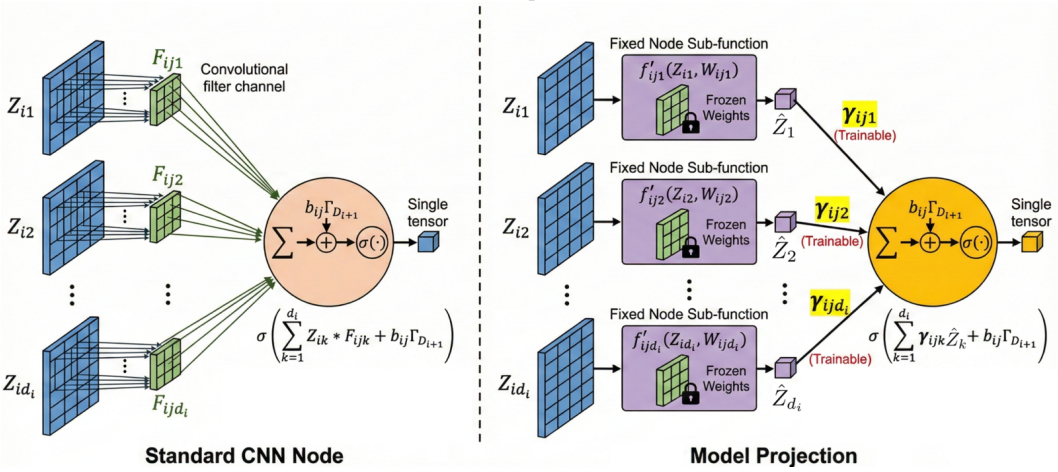

Figure 1: (left) Standard CNN Node: Each input channel contributes through multiple learned weights due to the spatial extent of the kernels. (right) Model projection: The node has exactly one trainable weight per input channel. Spatial structure is preserved by fixed sub-functions, while channel interaction is reduced to a linear weighted sum. The resulting computation is structurally identical to an FFN node with pre-processed tensors as inputs.

Proof. Let $\hat{Z}_{jk}$ denote the output of the node sub-function with fixed weights $f'_{jk}(Z_k, W_{jk})$. Then we can rewrite the equation for a projected node as:

$$\begin{aligned}\hat{f}_j(Z, W_j, \gamma_j) &= \sigma(\sum_{k=1}^{d} \gamma_{jk} f'_{jk}(Z_k, W_{jk}) + b_j \Gamma_{D_{out}}) \\ &= \sigma(\sum_{k=1}^{d} \gamma_{jk} \hat{Z}_{jk} + b_j \Gamma_{D_{out}}) \\ &= \sigma(\hat{Z}_j \odot_t \gamma_j + b_j \Gamma_{D_{out}}).\end{aligned}$$

The final expression above is in the form of Equation 3, with $\hat{Z}_j$ replacing Z_j, and γ_j replacing W_j. □

A direct consequence of Theorem 3.11 is the following.

Corollary 3.12. *Since projected nodes are GFFN nodes with inhomogeneous inputs, properties that apply to GFFN nodes with inhomogeneous inputs also apply to projected nodes.*

Corollary 3.12 says that projected layers inherit properties that apply to GFFN layers with inhomogeneous inputs. In particular, given that GCNN node functions are separable by input, see Lemma A.1 in the Appendix, we can project GCNN layers. This is especially useful for transfer learning. For example, GFFN nodes can be trained with one weight per input channel. This is true even in layers with inhomogeneous inputs. Therefore, a projected GCNN node can be trained with one weight per input channel, potentially greatly reducing the total number of parameters being trained.

In order to test the usefulness of model projection for transfer learning, we implemented model projection for 2D convolutional layers as a modified version of a 2D convolutional layer. The main difference is that it also includes a new set of parameters, γ_{jk}, one for each channel, k, in each convolutional node, j. We also fix the filter weights, while the new parameters and biases are trained. Each γ_{jk} is initialized to one, so that the first forward pass through the network will be the same as before model projection. Projected nodes learn a linear regression before σ is applied, but unlike a GFFN node, projected nodes also perform an extra pre-processing step to their input. Intuitively, our specific design choice is intended to implement Theorem 3.11, allowing us to turn a CNN into a kind of FFN. While other freezing techniques can reduce total trainable parameters, ours keeps the trainable part in a FFN node design, which is already known to be successful. This should not only allow us to benefit from our demonstrated property, but also benefit from other strengths of FFNs.

As stated earlier, one advantage of projecting nodes is that it allows a node to be trained with fewer parameters, reducing the amount of data needed. We explore this idea in Section 4.

4 EXPERIMENTS

We now demonstrate the utility of inheritance via projection for transfer learning: training nodes with only one parameter per input channel. We focus on transfer learning large pre-trained foundation CNN models onto downstream tasks. Normally, CNN nodes can have many parameters per input channel. By projecting these models, we can train an entire model using many fewer parameters, which should be advantageous when there is not enough data to fully train these large networks.

We conduct two sets of experiments. The first compares model projection with full fine-tuning and logistic regression , with an emphasis on analyzing training behavior. The second evaluates model projection on standard benchmarks and compares its performance to standard baselines and state-of-the-art PEFT methods on transfer learning.

4.1 FIRST EXPERIMENT: UNDERSTANDING MODEL PROJECTION BEHAVIOR

We conduct transfer learning experiments on seven benchmark datasets: Food-101 (Bossard et al., 2014), CIFAR-10 (Krizhevsky, 2009), CIFAR-100 (Krizhevsky, 2009), Stanford Dogs (Khosla et al., 2011), Oxford-IIIT Pets (Parkhi et al., 2012), Caltech-101 (Fei-Fei et al., 2004), and Oxford 102 Flowers (Nilsback & Zisserman, 2008). These datasets range in size from about 2,000 to about 75,000 training examples and span both coarse-grained and fine-grained image classification tasks. See Appendix Table 2. Our focus is on showing a proof of concept of our method rather than pushing state-of-the-art results, so we intentionally use simple training setups without hyperparameter optimization.

General setup. We perform experiments using three different convolutional bases, with weights pre-trained on ImageNet(Deng et al., 2009): VGG16(Simonyan & Zisserman, 2015), ResNet50(He et al., 2016), and DenseNet121(Huang et al., 2017). After each convolutional base, we add a global average pooling layer, a dropout layer set to 0.5, and finally a single dense layer for predictions. We use simple data augmentations: horizontal flip with probability 0.5, and random rotation and zoom both set to 0.1. Input images were set to 224 x 224 pixels. We do not shuffle data or address class imbalances. Nor do we address noisy labels or perform any other type of regularization. Any validation sets are absorbed into training sets. We evaluate two training setups:

Single-stage training. For each model, we run three variants: Logistic regression (LR), where only the final classifier is trained; Full fine-tuning (FT), where all parameters are trained; and model projection, where convolutional kernels are frozen and all other parameters, including projection parameters, are trained. All runs use Adam (Kingma, 2015) with default hyperparameters for 20 epochs.

Two-stage training. This setup uses the same models but trains in two stages. Stage 1 runs for 7 epochs with Adam using default hyperparameters. Stage 2 runs for 13 epochs with SGD, learning rate 10^{-4}, momentum 0.9, following the Keras Applications fine-tuning example (Chollet et al., 2015). Under this setup, we evaluate three combinations: Logistic regression in Stage 1, then full fine-tuning in Stage 2 (LR+FT); model projection in Stage 1, then full fine-tuning in Stage 2 (Model projection + FT); and model projection in both stages (2 step model Projection).

We perform the second setup because we observed full fine-tuning occasionally failed to learn in single-stage training, particularly for VGG16. Two-stage training provides a stronger full fine-tuning baseline and allows us to study projection as an alternative warm start to logistic regression training and/or an initialization before full fine-tuning.

Results. Results are summarized in Figure 2. Overall, model projection consistently outperforms logistic regression and full fine-tuning across both large and small datasets, and remains robust under older architectures and a simple training setup, while still performing well on newer models. Logistic regression typically achieves higher initial accuracy but shows limited improvement over epochs, indicating underfitting. Its performance is competitive on smaller datasets but degrades on larger ones, and it remains relatively robust on older models. Full fine-tuning starts from lower

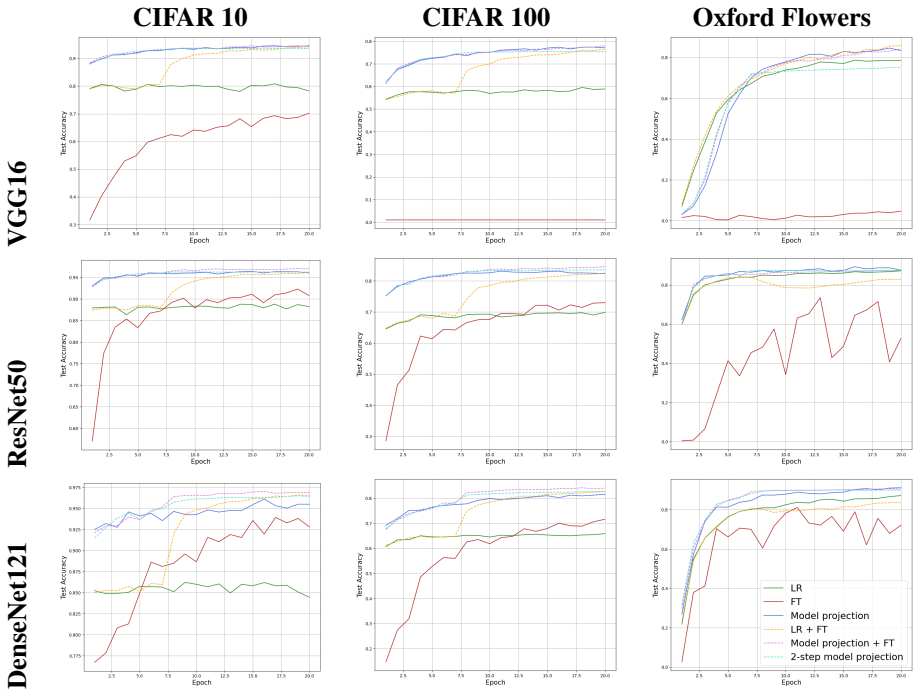

Figure 2: Select results from the first experiments. The solid lines show the single stage setup, while the dashed lines show the two stage setup. Each row corresponds to a particular convolutional base, organized from oldest to most recent. The columns show the results on CIFAR 10, CIFAR 100, and Oxford flowers respectively. In all charts, the x-axis is epochs, and the y-axis is test accuracy.

Table 1: Average accuracy and (standard deviation) of our method on the CIFAR 10, CIFAR 100, and Flowers datasets, compared to results from other recent methods, all using ResNet50. Values taken from other papers (LoRA-C (Ding et al., 2024) and PaRA, LoRA-r8, LoRA-r32 (Hedegaard et al., 2024)) are marked with *. The best averages are bolded, along with any averages that are within two standard deviations of the best result.

Method	C10	C100	Flowers
LR	87.90 (0.621)	69.73 (0.137)	**87.22** (0.161)
Batch Normalization	95.41 (0.152)	81.21 (0.099)	**87.81** (0.275)
BitFit	95.49 (0.168)	81.28 (0.173)	**88.31** (0.835)
FT	91.67 (0.692)	72.36 (0.807)	63.60 (8.961)
LR + FT	96.18 (0.166)	82.57 (0.248)	83.26 (0.253)
Model projection	96.13 (0.118)	82.58 (0.180)	**87.98** (1.049)
Model projection + FT	**96.83** (0.111)	**84.06** (0.335)	**87.69** (0.306)
2-step model projection	96.35 (0.193)	**83.53** (0.054)	**87.17** (0.386)
LoRA-C	96.59* (N/A*)	82.98* (N/A*)	N/A* (N/A*)
LoRA-r32	95.32* (0.13*)	79.47* (N/A*)	78.57* (5.93*)
LoRA-r8	95.35* (0.10*)	79.22* (N/A*)	80.96* (5.83*)
PaRA	93.52* (N/A*)	79.31* (N/A*)	80.65* (N/A*)

accuracy and improves over training, sometimes surpassing logistic regression but often struggling on smaller datasets or failing to learn for older architectures such as VGG16. In contrast, model projection combines favorable properties of both approaches: it starts from relatively high accuracy and continues to improve throughout training, achieving the strongest overall performance across experiments.

These results align with what we expect from projection inheritance: strong results from learning an entire neural network, allowing frequent outperformance of logistic regression and greater robustness to dataset size compared to full fine tuning due to inheriting fewer trainable parameters. Model projection retains the expressiveness of end-to-end learning while improving robustness to dataset size and optimization instability. While we expect full fine-tuning to outperform model projection given sufficiently large datasets, and logistic regression training to dominate in extremely low-data regimes (Kornblith et al., 2019), our results suggest that model projection performs best across a broad intermediate regime that covers most practical transfer learning scenarios.

In the two-stage setting, the first stage behaves similarly to the corresponding single-stage experiments, as expected. Notably, performance in the second stage is consistently higher when model projection is used in the first stage, regardless of whether the second stage applies model projection or full fine-tuning. This suggests that model projection provides a more effective initialization for subsequent fine-tuning. Overall, model projection yields stronger performance and emerges as the most effective strategy in the majority of cases. Further details and full results can be found in Appendix C.

4.2 Second Experiment: Model projection Against transfer learning Baselines

To evaluate model projection, we compare to standard transfer learning baselines and recent PEFT methods on pretrained ResNet50 architecture. We include PEFT methods that can also adapt the base model while training fewer parameters. We specifically focus on their transfer learning performances, which should also benefit from reduced overfitting in cases of little data due to fewer parameters. Our baselines include logistic regression (training only the final classifier) and full fine-tuning, as well as partial-tuning baselines that update only BatchNorm parameters (BN) (Ioffe & Szegedy, 2015) along with the final layer, or only bias terms (inspired by BitFit) (Zaken et al., 2022) along with BN and the final layer. This gives a set of methods with gradually increasing trainable parameters on the same model. We additionally compare to low-rank adaptation methods LoRA-r8 and LoRA-r32 (Hu et al., 2022), the Parallel Residual Adapter (PaRA) family (Rebuffi et al., 2017), and the convolutional variant LoRA-C (Ding et al., 2024). For LoRA-C we report results from Ding et al. (2024), and for LoRA-r8/LoRA-r32 and PaRA we use the ResNet50 results reported by Hedegaard et al. (2024). We include all two-stage variants considered previously. We report the average accuracy across 5 seeds, along with the standard deviation.

Results. Table 1 summarizes the results on CIFAR-10, CIFAR-100, and Oxford Flowers. Both single-stage and two-stage model projection are highly competitive across all datasets when compared to the baselines. In particular, the two-stage setup that uses model projection in the first stage, either with full fine tuning or model projection in the second stage, achieves the best performance on CIFAR-10 and CIFAR-100, while all projected models are within two standard deviations of the best performance on Flowers. These results support the view that model projection provides an effective initialization, especially for subsequent full fine-tuning, by guiding optimization toward more favorable regions of the loss landscape. The extended results on the other datasets can be found in Table 6 in the Appendix. Initial results with hyperparameter tuning can also be found in the Appendix, see Table 7.

Conclusions.

In this paper we formalized a general version of the operations of nodes from traditional neural networks and convolutional neural networks. We used this formalization to prove that FFNs can inherit from CNNs, but that CNNs cannot inherit from FFNs. We proposed *model projection*, a method to project a CNN node onto an FFN node, and proved that this method allows a CNN node to inherit any results or techniques from FFN nodes that do not require homogeneous inputs. We then demonstrated a proof of concept of our proposed projection method in a transfer learning example. Our method had strong results in a simple setting, reinforcing our theoretical findings. Our example is a good application of projection that we expect will be useful for simple home made models, and may also be useful in more heavily engineered settings as well. There are a number of potential directions for future research. Extending analysis of inheritance between more model types such as graph neural networks and transformer models is particularly interesting. More results from FFNs can be inherited to CNNs, and exploration of some of these methods such as lasso artificial neural networks Ma et al. (2022) would be interesting to experiment with.

AI USE STATEMENT

We have not used generative AI tools for any tasks with required disclosure.

Additionally, we used generative AI tools for language editing, including rephrasing, polishing, and grammar checking. We used generative AI tools to produce an early version of our figure, which we then reviewed and hand edited for correctness. Generative AI tools were also used to produce code for empty latex tables, which we then edited and filled by hand. We also used generative AI tools to produce code for empty nested for loops that we used to iterate through some supplemental experiments and group those results. This code was then edited by hand to include our methods, and further modified and verified for correctness. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

For reproducibility of code, experiments, and method implementations, please see the code files in the supplementary materials. For reproducibility of proofs and theoretical claims, please see Section 3 and Appendix A.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024. doi: 10.1145/3620665.3640366. URL `https://docs.pytorch.org/assets/pytorch2-2.pdf`.

Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – mining discriminative components with random forests. In *European Conference on Computer Vision*, 2014.

Hao Chen, Ran Tao, Han Zhang, Yidong Wang, Xiang Li, Wei Ye, Jindong Wang, Guosheng Hu, and Marios Savvides. Conv-adapter: Exploring parameter efficient transfer learning for convnets. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 1551–1561, 2024.

François Chollet et al. Keras. `https://keras.io`, 2015.

Francois Chollet. *Deep Learning with Python*. Manning, 2021.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Chuntao Ding, Xu Cao, Jianhang Xie, Linlin Fan, Shangguang Wang, and Zhichao Lu. Lora-c: Parameter-efficient fine-tuning of robust cnn for iot devices. *arXiv preprint arXiv:2410.16954*, 2024.

Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer Vision and Pattern Recognition Workshop*, 2004.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Cheng Han, Qifan Wang, Yiming Cui, Wenguan Wang, Lifu Huang, Siyuan Qi, and Dongfang Liu. Facing the elephant in the room: Visual prompt tuning or full finetuning? In *The Twelfth International Conference on Learning Representations*, 2024a.

Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey, 2024b. URL https://arxiv.org/abs/2403.14608.

Trevor Hastie, Robert Tibshirani, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

Haoyu He, Jianfei Cai, Jing Zhang, Dacheng Tao, and Bohan Zhuang. Sensitivity-aware visual parameter-efficient fine-tuning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 11825–11835, 2023.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Lukas Hedegaard, Aman Alok, Juby Jose, and Alexandros Iosifidis. Structured pruning adapters. *Pattern Recognition*, 156:110724, 2024. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2024.110724. URL https://www.sciencedirect.com/science/article/pii/S0031320324004758.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. pmlr, 2015.

Menglin Jia, Luming Tang, Bor-Chun Chen, Claire Cardie, Serge Belongie, Bharath Hariharan, and Ser-Nam Lim. Visual prompt tuning. In *European conference on computer vision*, pp. 709–727. Springer, 2022.

Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Li Fei-Fei. Novel dataset for fine-grained image categorization. In *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*, Colorado Springs, CO, June 2011.

Diederik P Kingma. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR 2015)*, 2015.

Simon Kornblith, Jonathon Shlens, and Quoc V Le. Do better imagenet models transfer better? In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2661–2671, 2019.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional networks and applications in vision. In *Proceedings of 2010 IEEE international symposium on circuits and systems*, pp. 253–256. IEEE, 2010.

Dongze Lian, Daquan Zhou, Jiashi Feng, and Xinchao Wang. Scaling & shifting your features: A new baseline for efficient model tuning. *Advances in Neural Information Processing Systems*, 35: 109–123, 2022.

Christoph Linse, Erhardt Barth, and Thomas Martinetz. Convolutional neural networks do work with pre-defined filters. In *2023 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2023.

Christoph Linse, Beatrice Brückner, and Thomas Martinetz. Enhancing generalization in convolutional neural networks through regularization with edge and line features. In *International Conference on Artificial Neural Networks*, pp. 432–446. Springer, 2024.

Gen Luo, Minglang Huang, Yiyi Zhou, Xiaoshuai Sun, Guannan Jiang, Zhiyu Wang, and Rongrong Ji. Towards efficient visual adaption via structural re-parameterization. *arXiv preprint arXiv:2302.08106*, 2023.

Xiaoyu Ma, Sylvain Sardy, Nick Hengartner, Nikolai Bobenko, and Yen Ting Lin. A phase transition for finding needles in nonlinear haystacks with lasso artificial neural networks. *Statistics and Computing*, 32(6):99, 2022.

M-E. Nilsback and A. Zisserman. Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, Dec 2008.

O. M. Parkhi, A. Vedaldi, A. Zisserman, and C. V. Jawahar. Cats and dogs. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. Learning multiple visual domains with residual adapters. *Advances in neural information processing systems*, 30, 2017.

K Simonyan and A Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations (ICLR 2015)*. Computational and Biological Learning Society, 2015.

Dequan Wang, Xiaosong Wang, Lilong Wang, Mengzhang Li, Qian Da, Xiaoqiang Liu, Xiangyu Gao, Jun Shen, Junjun He, Tian Shen, et al. A real-world dataset and benchmark for foundation model adaptation in medical image classification. *Scientific Data*, 10(1):574, 2023.

Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1–9, 2022.

A SUPPLEMENTARY PROOFS

Proof of Theorem 3.5. We will prove the theorem by showing a bijection between GFFNs and a subset of GCNNs. Suppose that we have a GCNN node where F_{jk} has size 1 in all dimensions. This means F_{jk} is a scalar, and F_j is a vector of scalars. As a result, $Z_k * F_{jk} = Z_k F_{jk}$. Using this fact, and the definition of GCNN nodes, we can write:

$$f_j(Z, W_j) = \sigma(Z * F_j + b_j \Gamma_{D_{out}}) = \sigma(\sum_{k=1}^{d} Z_k * F_{jk} + b_j \Gamma_{D_{out}}) = \sigma(\sum_{k=1}^{d} Z_k F_{jk} + b_j \Gamma_{D_{out}}).$$

Note that $\sum_{k=1}^{d} Z_k F_{jk} = Z \odot_t F_j$ by definition. As a result, we have that

$$\sigma(\sum_{k=1}^{d} Z_k F_{jk} + b_j \Gamma_{D_{out}}) = \sigma(Z \odot_t F_j + b_j \Gamma_{D_{out}}).$$

Observe that the right-hand side is exactly the definition of GFFN node. Every GCNN node with F_{jk} of size 1 in all dimensions maps to exactly one GFFN node. We now show the inverse. Suppose we have a GFFN node, viz.

$$f_j(Z, W_j) = \sigma(Z \odot_t W_j + b_j \Gamma_{D_{out}}) = \sigma(\sum_{k=1}^{d} Z_k W_{jk} + b_j \Gamma_{D_{out}}).$$

Note that since W_{jk} is a scalar, $\sum_{k=1}^{d} Z_k W_{jk} = \sum_{k=1}^{d} Z_k * W_{jk}$. As a result, we have that

$$\sigma(\sum_{k=1}^{d} Z_k W_{jk} + b_j \Gamma_{D_{out}}) = \sigma(\sum_{k=1}^{d} Z_k * W_{jk} + b_j \Gamma_{D_{out}}).$$

Observe that the right hand side is exactly the definition of a GCNN node, where the kernels have size 1 in all dimensions. Every GFFN node maps to exactly one GCNN node with kernels of size 1.

Therefore, there is a bijection between the subset of GCNNs with kernel size 1 in all dimensions, and the set of GFFNs. Therefore, GFFNs are a subset of GCNNs. □

A.1 PROJECTION

Lemma A.1. *GCNN node functions are separable by input.*

Proof. From Definition 3.4 we have that GCNN node functions can be written as:

$$\sum_{k=1}^{d} Z_k * F_{jk} + b_j \Gamma_{D_{out}} = \sum_{k=1}^{d} f'_{jk}(Z_k, W_{jk}) + b_j \Gamma_{D_{out}}.$$

Here, f'_{jk} is a convolution, and W_{jk} are the weights in filter channel F_{jk}. □

Lemma A.2. *If all node functions are separable by input, and all node sub-functions, f'_{jk}, are associative for scalar multiplication, then γ_{jk} can be applied either before, or after f'_{jk} acts on Z_k.*

Proof. Each node is separable by input, therefore, we can just consider each node sub-function. Since all node sub-functions, f'_{jk}, are associative for scalar multiplication, then $f'_{jk}(\gamma_{jk} Z_k) = \gamma_{jk} f'_{jk}(Z_k)$. □

Proposition A.3. *In a GCNN node, γ_{jk} can be applied either before, or after f'_{jk} acts on Z_k.*

Proof. Convolutions are associative for scalar multiplication, therefore, this follows from Lemmas A.1 and A.2. □

Table 2: Benchmark datasets.

Dataset	Classes	Train	Test
Food 101	101	75750	25250
CIFAR 10	10	50000	10000
CIFAR 100	100	50000	10000
Stanford dogs	120	12000	8580
Oxford IIIT pets	37	3680	3669
Caltech 101	102	3060	6084
Oxford 102 flowers	102	2040	6149

B SUPPLEMENTARY REMARKS

Remark B.1. A projected CNN differs from a pre-defined CNN Linse et al. (2023), where a set of fixed filters produce outputs that are then sent to a layer of 1 x 1 convolutional nodes. Instead, each projected node receives a unique input from a single fixed set of filter channels that have not aggregated their channels. Specifically, a Pre-defined Filter Module is similar to a depthwise separable convolution, except they fix the filters in the depthwise part Linse et al. (2024). A depthwise separable convolution has a depthwise convolution layer (a single filter per input channel) followed by BN and RELU, then a 1x1 convolution layer, and another BN and RELU Howard et al. (2017). So the PDFM consists of 6 steps: a fixed depthwise convolution, BN, RELU, 1x1 convolution, BN, RELU. In contrast, our method has 2 steps: projected convolution, RELU. There is no BN, and no intermediary RELU. Also, beyond this, PDFMs only use pre-defined edge and line filters, which we do not. Furthermore, each 1x1 convolutional node of the depthwise separable receives the same input, whereas each projected node receives a unique input. A depthwise layer has one filter per input channel, where we have one filter for each input channel - output channel pair.

C SUPPLEMENTARY RESULTS

C.1 FIRST EXPERIMENT

We now discuss the results from the first set of single stage experiments, seen in Figure 3, with summarizing information presented in Tables 3, 4, and 5. Figure 3 shows that model projection consistently had strong results, while training was relatively smooth and fast. Table 3 shows the ranking of each method across each experiment. Model projection significantly outperformed both the logistic regression and full fine tuning in this simple setup, producing the best model about 81% of the time, and the second best about 19% of the time.

Table 4 shows the rankings achieved by model projection organized by model. Model projection had consistently strong results on each model. Table 5 shows the rankings for model projection grouped by dataset. Model projection had the best scores on all five datasets that were not exclusively pet related, but did not do as well on the two datasets that were exclusively pet related. Interestingly, model projection did well on datasets both larger and smaller than the pet ones, suggesting the issue is not solely dataset size.

Table 3: Ranking of each training method in the 21 experiments from the first experimental setup. In a particular experiment (setup, convolutional base, and dataset combination), the method producing the highest test accuracy at the end of the last epoch was designated '1st', the second highest '2nd', and so on.

Method	1st	2nd	3rd
Model projection	17	4	0
Logistic regression	4	11	6
Full fine tuning	0	6	15

Table 4: Ranking of Model projection in the 21 experiments from the first experimental setup, organized by convolutional base. In a particular experiment (setup, convolutional base, and dataset combination), the method producing the highest test accuracy at the end of the last epoch was designated '1st', the second highest '2nd', and so on.

Model	**1st**	**2nd**	**3rd**
VGG16	6	1	0
ResNet50	6	1	0
DenseNet121	5	2	0

Table 5: Ranking of Model projection in the 21 experiments from the first experimental setup, organized by dataset. In a particular experiment (setup, convolutional base, and dataset combination), the method producing the highest test accuracy at the end of the last epoch was designated '1st', the second highest '2nd', and so on.

Dataset	**1st**	**2nd**	**3rd**
Food 101	3	0	0
CIFAR 10	3	0	0
CIFAR 100	3	0	0
Stanford dogs	1	2	0
Oxford IIIT pets	1	2	0
Caltech 101	3	0	0
Oxford 102 flowers	3	0	0

C.2 Second Experiment Supplementary Results

The results on the remaining datasets from the second experiment can be found in Table C.2. These results are consistent with the results in Table 1 in the main paper. Model projection + FT had the highest score in each of these four datasets. The performances of the methods using model projection was particularly higher than methods that did not use it in the food dataset, which is the largest.

Table 6: Average accuracy and (standard deviation) of our method on the remaining datasets, compared to results from other recent methods, all using ResNet50. The best averages are bolded, along with any averages that are within two standard deviations of the best result.

Method	**Food**	**Stanford Dogs**	**Caltech**	**Oxford Pets**
LR	29.68 (0.82)	76.86 (0.51)	89.70 (0.54)	**86.92** (0.67)
Batch Normalization	52.33 (0.37)	75.81 (0.29)	91.13 (0.28)	86.34 (0.58)
BitFit	52.56 (0.62)	75.78 (0.33)	90.79 (0.09)	86.31 (0.59)
FT	53.84 (3.67)	42.39 (6.17)	68.73 (8.11)	52.27 (6.51)
LR + FT	57.79 (0.69)	77.41 (0.29)	90.18 (0.23)	**86.69** (0.33)
Model projection	**65.17** (0.36)	74.72 (0.68)	**91.50** (0.47)	85.92 (1.04)
Model projection + FT	**65.59** (0.53)	**78.60** (0.38)	**92.00** (0.33)	**87.40** (0.36)
2-step model projection	63.82 (0.40)	**78.05** (0.38)	**91.72** (0.43)	**86.98** (0.39)

C.3 Hyperparameter tuning Experiment

In addition to our two sets of experiments, we also ran a set of experiments with hyperparameter tuning. Apart from the hyperparameters being tuned, the settings for these experiments are identical to those in the second set of experiments. Specifically, we tune the amount of each type of augmentation, the amount of dropout, and the initial learning rate. We used a validation set for the tuning, which consisted of half the training data. For the Oxford flowers dataset we used the pre-defined split, and for the other two we randomly selected half. We used the Optuna library and tuned for 20 trials. The code for these experiments can also be found in the supplementary materials. Due to compute restraints, we only include results from the smaller datasets, but intend to add more in the accepted paper.

The results of our hyperparameter experiments can be seen in Table C.3. While each performs better with hyperparameter tuning, methods using Model projection have the highest scores in each dataset, continuing to show strong results when tuning is done.

Table 7: Initial results with performing hyperparameter tuning. Average accuracy and (standard deviation) of our method on the Caltech, Oxford Pets, and Flowers datasets, compared to results from other recent methods, all using ResNet50. The best averages are bolded.

Method	**Caltech**	**Pets**	**Flowers**
LR	91.05 (0.43)	87.38 (0.87)	87.90 (0.72)
Batch Normalization	91.79 (0.28)	86.29 (0.43)	89.40 (0.74)
BitFit	91.26 (0.54)	86.38 (0.52)	89.31 (0.76)
FT	86.00 (1.15)	77.14 (2.84)	84.61 (2.22)
LR + FT	90.68 (0.38)	87.07 (0.81)	89.17 (0.46)
Model projection	91.89 (0.48)	87.96 (0.98)	**89.44** (1.85)
Model projection + FT	**92.12** (0.54)	**88.21** (0.52)	89.30 (0.98)
2-step model projection	91.72 (0.41)	87.94 (0.52)	88.70 (0.99)

Figure 3: Results from the first set of experiments. The blue lines show the performance of model projection. The green and red lines show the performances of the single layer logistic regression, and the full fine tuning respectively. Each column corresponds to a particular convolutional base, organized from oldest to most recent. Each row shows the results on a particular dataset, organized from largest to smallest. In all charts, the x-axis is epochs, and the y-axis is test accuracy.

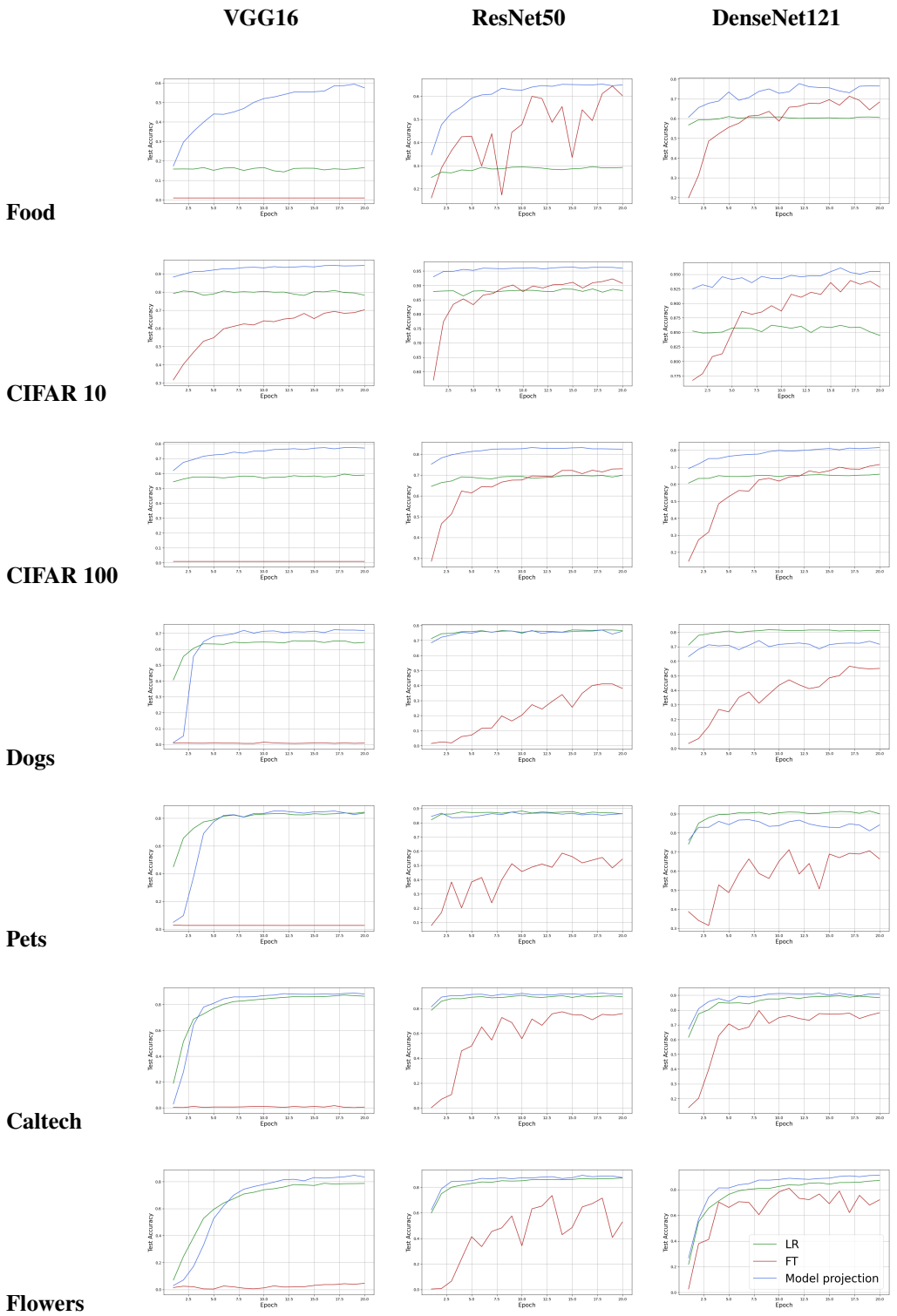

Figure 4: Results from the second set of experiments. The orange lines show the performance of 2-step fine tuning. The pink lines show the performance of 2-step fine tuning using projection in the first step, and the light blue lines represent the performance of the 2-steps using projection in both steps. Each column corresponds to a particular convolutional base, organized from oldest to most recent. Each row shows the results on a particular dataset, organized from largest to smallest. In all charts, the x-axis is epochs, and the y-axis is test accuracy.

