What Do Neural Networks Learn When Trained With Random Labels?

Hartmut Maennel[⋆] hartmutm@google.com

Ibrahim Alabdulmohsin[⋆] ibomohsin@google.com

Ilya Tolstikhin tolstikhin@google.com

Robert J. N. Baldock rbaldock@google.com

Olivier Bousquet obousquet@google.com

Sylvain Gelly sylvaingelly@google.com

Daniel Keysers keysers@google.com

Google Research, Brain Team
Zürich, Switzerland

Abstract

We study deep neural networks (DNNs) trained on natural image data with entirely random labels. Despite its popularity in the literature, where it is often used to study memorization, generalization, and other phenomena, little is known about what DNNs learn in this setting. In this paper, we show analytically for convolutional and fully connected networks that an alignment between the principal components of network parameters and data takes place when training with random labels. We study this alignment effect by investigating neural networks pre-trained on randomly labelled image data and subsequently fine-tuned on disjoint datasets with random or real labels. We show how this alignment produces a *positive* transfer: networks pre-trained with random labels train faster downstream compared to training from scratch even after accounting for simple effects, such as weight scaling. We analyze how competing effects, such as specialization at later layers, may hide the positive transfer. These effects are studied in several network architectures, including VGG16 and ResNet18, on CIFAR10 and ImageNet.

1 Introduction

Over-parameterization helps deep neural networks (DNNs) to generalize better in real-life applications [53, 23, 29, 8], despite providing them with the capacity to fit almost any set of random labels [54]. This phenomenon has spawned a growing body of work that aims at identifying fundamental differences between real and random labels, such as in training time [4, 19, 55, 18], sharpness of the minima [39, 27], dimensionality of layer embeddings [3, 34, 11], and sensitivity [4, 40], among other complexity measures [6, 7, 38, 39]. While it is obvious that over-parameterization helps DNNs to interpolate any set of random labels, it is not immediately clear what DNNs *learn* when trained in this setting. The objective of this study is to provide a partial answer to this question.

There are at least two reasons why answering this question is of value. First, in order to understand how DNNs work, it is imperative to observe how they behave under "extreme" conditions, such as when trained with labels that are entirely random. Since the pioneering work of [54], several works

[⋆]Equal contribution.

Preprint. Under review.

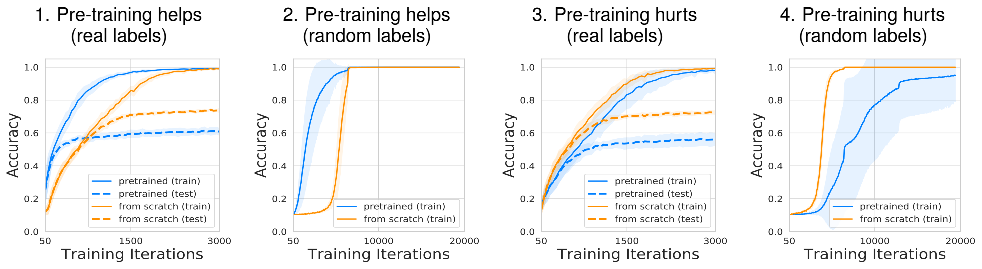


Figure 1: Pre-training on random labels may exhibit both positive (1 & 2) and negative (3 & 4) effects on the downstream fine-tuning depending on the setup. VGG16 models are pre-trained on CIFAR10 examples with random labels and subsequently fine-tuned on the fresh CIFAR10 examples with either real labels (1 & 3) or 10 random labels (2 & 4) using different hyperparameters.

have looked into the case of random labels. What distinguishes our work from others is that previous works aimed to demonstrate differences between real and random labels, highlighting the *negative* side of training on random labels. By contrast, this work provides insights into what properties of the data distribution DNNs learn when trained on random labels.

Second, observing DNNs trained on random labels can explain phenomena that have been previously noted, but were poorly understood. In particular, by studying what is learned on random labels, we offer new insights into: (1) why DNNs exhibit critical stages [1, 16], (2) how earlier layers in DNNs generalize while later layers specialize [3, 4, 10, 52], (3) why the filters learned by DNNs in the first layer seem to encode some useful structure when trained on random labels [4], and (4) why pre-training on random labels can accelerate training in downstream tasks [41]. We show that even when controlling for simple explanations like weight scaling (which was not always accounted for previously), such curious observations continue to hold. The main contributions of this work are:

- We investigate DNNs trained with random labels and fine-tuned on disjoint image data with real or random labels, demonstrating unexpected positive and negative effects.
- We provide explanations of the observed effects. We show analytically for convolutional and fully connected networks that an alignment between the principal components of the network parameters and the data takes place. We demonstrate experimentally how this effect explains why pre-training on random labels helps. We also show why, under certain conditions, pre-training on random labels can hurt the downstream task due to specialization at the later layers.
- We conduct experiments verifying that these effects are present in several network architectures, including VGG16 [45] and ResNet18-v2 [21], on CIFAR10 [30] and ImageNet ILSVRC-2012 [13], across a range of hyper-parameters, such as the learning rate, initialization, number of training iterations, width and depth.

In this work, we do not use data augmentation as it provides a (weak) supervisory signal. Moreover, we use the terms "positive" and "negative" to describe the impact of what is learned with random labels on the downstream training, such as faster/slower training. The networks reported throughout the paper are taken from a big set of experiments that we conducted using popular network architectures, datasets, and wide hyperparameter ranges. Experimental details are provided in Appendix A and B. We use boldface for random variables, small letters for their values, and capital letters for matrices.

1.1 Motivating example

Figure 1 shows learning curves of the VGG16 architecture [45] pre-trained on 20k CIFAR10 examples [30] with random labels (upstream) and fine-tuned on a disjoint subset of 25k CIFAR10 examples with either random or real labels (downstream). We observe that in this setup, pre-training a neural network on images with random labels accelerates training on a second set of images, both for real and random labels (positive effect). However, in the *same setting* but with a different initialization scale and number of random classes upstream, a negative effect can be observed downstream: training becomes slower. We also observe a lower final test accuracy for real labels in both cases, which we are not explicitly investigating in this paper (and which has been observed before, e.g. in [16]).

The fact that pre-training on random labels can accelerate training downstream has been observed previously, e.g. in [41]. However, there is a "simple" property that can explain improvements in the downstream task: Because the cross-entropy loss is scale-sensitive, training the network tends to increase the scale of the weights [39], which can increase the effective learning rate of the downstream task (see the gray curve in Figure 5). To eliminate this effect, in all experiments we re-scale the weights of the network after pre-training to match their ℓ_2 norms at initialization. We show that even after this correction, pre-training on random labels positively affects the downstream task. This holds for both VGG16 and ResNet18 trained on CIFAR10 and ImageNet (see Appendix B).

We show experimentally that some of the positive transfer is due to the second-order statistics of the network parameters. We prove that when trained on random labels the principal components of weights at the first layer are aligned with the principal components of data. Interestingly, this *alignment effect* implies that the model parameters learned at the first layer can be summarized by a one-dimensional mapping between the eigenvalues of the data and the eigenvalues of the network parameters. We study these mappings empirically and raise some new open questions. We also analyze how, under certain conditions, a competing effect of specialization at the later layers may hide the positive transfer of pre-training on random labels, which we show to be responsible for the negative effect demonstrated in Figure 1.

To the best of our knowledge, the alignment effect has not been established in the literature before. This paper proves the existence of this effect and studies its implications. Note that while these effects are established for training on random labels, we also observe them empirically for real labels.

1.2 Related work

A large body of work in the literature has developed techniques for mitigating the impact of *partial* label noise, such as [55, 24, 37, 26, 31, 32, 47]. Our work is distinct from this line of literature because we focus on the case of purely random labels.

The fact that positive and negative learning takes place is related to the common observation that earlier layers in DNNs learn general-purpose representations whereas later layers specialize [3, 4, 10, 52]. For random labels, it has been noted that memorization happens at the later layers, as observed by measuring the classification accuracy using activations as features [10] or by estimating the intrinsic dimensionality of the activations [3]. We show that specialization at the later layers has a negative effect because it exacerbates the inactive ReLU phenomenon. Inactive ReLUs have been studied in previous works, which suggest that this effect could be mitigated by either increasing the width or decreasing the depth [33], using skip connections [15], or using other activation functions, such as the leaky ReLU [35, 20] or the exponential learning unit (ELU) [9].

For transfer learning, it has been observed that pre-training on random labels can accelerate training on real labels in the downstream task [41]. However, prior works have not accounted for simple effects, such as the change in first-order weight statistics (scaling), which increases when using the scale-sensitive cross-entropy loss [39]. Changing the norm of the weights alters the effective learning rate. [42] investigated transfer from ImageNet to medical data and observed that the transfer of first-order weight statistics provided faster convergence. We show that even when taking the scaling effect into account, additional gains from second-order statistics are identified.

Other works have considered PCA-based convolutional filters either as a model by itself without training [17], as an initialization [43, 48], or to estimate the dimensionality of intermediate activations [11, 36]. Note that our results suggest an initialization by *sampling* from the data covariance instead of initializing the filters directly using the principal axes of the data. Ye et al. [51] show that a "deconvolution" of data at input and intermediate layers can be beneficial. This deconvolution corresponds to a whitening of the data distribution, therefore aligning data with an isotropic weight initialization, which is related to a positive effect of alignment we observe in this paper.

In addition, there is a large body of work on unsupervised learning. Among these, the *Exemplar-CNN* method [14] can be seen as the limiting case of using random labels with infinitely many classes (one label per image) and large-scale data augmentation. In our study we do *not* use data augmentation since it provides a supervisory signal to the neural network that can cause additional effects.

2 Covariance matrix alignment between network parameters and data

Returning to the motivating example in Figure 1, we observe that pre-training on random labels can improve training in the downstream tasks for both random and real labels. This improvement is in the form of *faster training*. In this section, we explain this effect. We start by considering the first layer in the neural network, and extend the argument to later layers in Section 2.5.

2.1 Preliminaries

Let $\mathcal{D}$ be the probability distribution over the instance space $\mathcal{X} \subseteq \mathbb{R}^d$ and $\mathcal{Y}$ be a finite target set. We fix a network architecture, a loss function, a learning rate/schedule, and a distribution of weights for random initialization. Then, "training on random labels" corresponds to the following procedure: We randomly sample i.i.d. instances $\mathbf{x}_1, ..., \mathbf{x}_N \sim \mathcal{D}$, and i.i.d. labels $\mathbf{y}_1, ..., \mathbf{y}_N \in \mathcal{Y}$ independently of each other. We also sample the initial weights of the neural network, and train the network on the data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1,...,N}$ for T training iterations using stochastic gradient descent (SGD). During training, the weights are *random variables* due to the randomness of the initialization and the training sample. Hence, we can speak of their statistical properties, such as their first and second moments.

In the following, we are interested in layers that are convolutional or fully connected. We assume that the output of the k-th neuron in the first layer can be written as: $f_k(x) = g(\langle w_k, x\rangle + b_k)$ for some activation function g. We write $\mu_x = \mathbb{E}[\mathbf{x}]$ and observe that since the covariance matrix $\Sigma_x = \mathbb{E}[(\mathbf{x} - \mu_x) \cdot (\mathbf{x} - \mu_x)^T]$ is symmetric positive semi-definite, there exists an orthogonal decomposition $\mathbb{R}^d = V_1 \oplus ... \oplus V_r$ such that V_i are eigenspaces to Σ_x with distinct eigenvalues σ_i^2.

Definition 1 (Alignment)**.** *A symmetric matrix A is said to be* **aligned** *with a symmetric matrix B if each eigenspace of B is a subset of an eigenspace of A. If A is aligned with B, we define for each eigenvalue of B with eigenspace $V \subseteq \mathbb{R}^d$ the* **corresponding** *eigenvalue of A as the one belonging to the eigenspace that contains V.*

If A and B's eigenspaces are all of dimension 1 (which is true except for a Lebesgue null set in the space of symmetric matrices), "A is aligned with B" becomes equivalent to the assertion that they share the same eigenvectors. However, the relation is not symmetric in general (e.g. only scalar multiples of the identity matrix I_d are aligned with I_d, but I_d is aligned with any symmetric matrix).

2.2 Alignment for centered Gaussian inputs

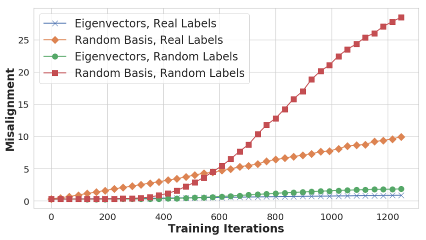
Proposition 1. *Assume the instances $\mathbf{x}$ are drawn i.i.d. from $\mathcal{N}(0, \Sigma_x)$ and the initial weights in the first layer are drawn from an isotropic distribution (e.g. the standard Gaussian). Let $\mathbf{w} \in \mathbb{R}^d$ be a random variable whose value is drawn uniformly at random from the set of weights in the first layer after training using SGD with random labels (see Section 2.1). Then: (1) $\mathbb{E}[\mathbf{w}] = 0$ and (2) $\Sigma_w = \mathbb{E}[\mathbf{w} \cdot \mathbf{w}^T]$ is aligned with the covariance matrix of data Σ_x.*

Proof. The proof exploits symmetries: The input, initialization, and gradient descent are invariant under reflections on hyperplanes spanned by eigenvectors of Σ_x so the distribution of weights must have the same invariance properties. The full proof is given in Appendix C. □

Proposition 1 says that independently of many settings (e.g. number of random labels, network architecture, learning rate or schedule), the eigenspaces of $\Sigma_w \in \mathbb{R}^{d\times d}$ are given by the eigenspaces of $\Sigma_x \in \mathbb{R}^{d\times d}$. Hence, the only information needed to fully determine Σ_w is a function f that maps the eigenvalues σ_i^2 of Σ_x to the corresponding eigenvalues τ_i^2 of Σ_w. Note that the argument of the proof of Proposition 1 also apply to the case of a single training run of an infinitely wide network in which the layers are given by weight vector distributions, see e.g. [46]. (For finite networks, in practice, one would only be able to approximate Σ_w based on several independent training runs.)

Next, we present experimental evidence that first-layer alignment actually takes place, not just for Gaussian input with random labels, but also in real image datasets with random labels, and even when training on real labels using convolutional networks. The intuition behind this result for real labels is that small patches in the image (e.g. 3×3) are nearly independent of the labels. Before we do that, we introduce a suitable measure of alignment we use in the experiments.

Figure 2: Plots of the misalignment scores of the filters of the first layer in a two-layer neural network (256 convolutional filters, 64 fully-connected nodes) when trained on CIFAR10 with either real or random labels. Throughout training, misalignment scores between the filters of the first layer and the data remain very small compared to those between filters and a random orthonormal basis.



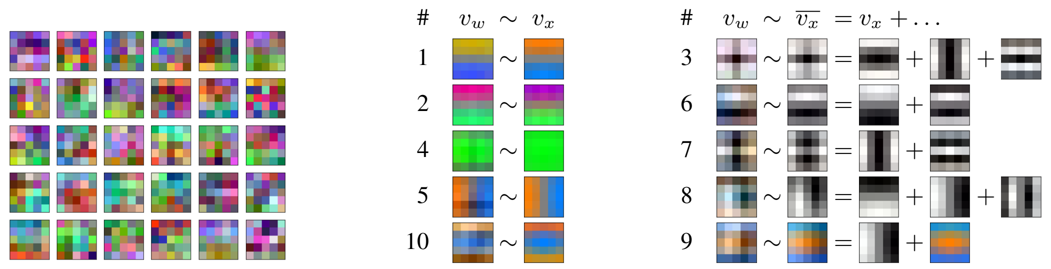


Figure 3: Visualization of covariance alignment. LEFT: Random selection of WRN-28-4 first-layer convolutional filters (CIFAR10, random labels). CENTER/RIGHT: Eigenvectors v_w of Σ_w with largest eigenvalues (rank in column '#') and eigenvectors v_x of Σ_x with $\langle v_x, v_w \rangle > 0.4$. CENTER: Cases where one v_x matches. RIGHT: Cases where multiple v_x and their weighted combination $\overline{v_x}$ match.

Definition 2. *For two positive definite matrices A, B, the "misalignment" $M(A, B)$ is defined as:*

$$M(A,B) := \inf_{\Sigma \succ 0 \text{ aligned with } A} \left\{ \tfrac{1}{2} \boldsymbol{tr}(\Sigma^{-1}B + B^{-1}\Sigma) - d \right\} \tag{1}$$

The rationale behind this definition of misalignment is presented in Appendix D. In particular, it can be shown that for any $A, B \succ 0$, we have $M(A, B) \geq 0$ with equality if and only if B is aligned with A. In addition, $M(A, B)$ is continuous in B and satisfies desirable equivariance and invariance properties and can be computed in closed form by $M(A,B) + d = \sum_{i=1}^{r} \sqrt{\mathbf{tr}(B|_{V_i}) \cdot \mathbf{tr}(B^{-1}|_{V_i})}$ where $V_1 \oplus \ldots \oplus V_r$ is the orthogonal decomposition of $\mathbb{R}^d$ into eigenspaces of A, and $B|_{V_i}$ is the linear map $V_i \to V_i, \mathbf{v} \mapsto pr_i(B(\mathbf{v}))$ with pr_i the orthogonal projection $\mathbb{R}^d \to V_i$.

Figure 2 displays the misalignment scores between the covariance of filters at the first layer with the covariance of the data (patches of images). For comparison, the misalignment scores with respect to some random orthonormal basis are plotted as well. As predicted by Proposition 1, the weight eigenvectors stay aligned to the data eigenvectors but not to an arbitrary random basis.

For image data, we can also visualize the alignment of Σ_w to Σ_x. Figure 3 shows results based on 70 wide ResNet models [53] trained on CIFAR10 with random labels. For better visualization, we use a 5×5 initial convolution here. The left part of the figure shows a random selection of some of the 70·64 convolution filters. From the filters we estimate Σ_w and compute the eigenvectors v_w, then visualize the ten v_w with the largest eigenvalues. From the image data we compute the patch data covariance Σ_x and its eigenvectors v_x. We show the data eigenvectors for which the inner product with the filter eigenvectors exceeds a threshold and the weighted sum $\overline{v_x}$ of these if there are multiple such v_x. The visual similarity between the v_w and the $\overline{v_x}$ illustrates the predicted covariance alignment. Note that this alignment is visually non-obvious when looking at individual filters as shown on the left.

2.3 Mapping of eigenvalues

As stated earlier, Proposition 1 shows that, on average, the first layer effectively learns a function which maps each eigenvalue of Σ_x to the *corresponding* eigenvalue of Σ_w (see Definition 1). In this section, we examine the shape of this function.

Since, in practice, we will only have *approximate* alignment due to the finiteness of the number of inputs and weights, non–Gaussian inputs, and correlations between overlapping patches, we extend

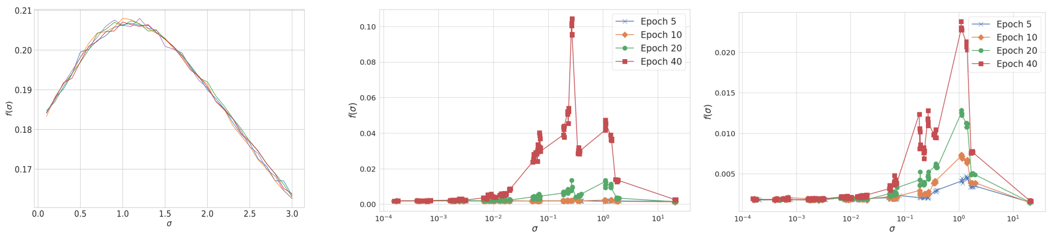


Figure 4: LEFT: $f(\sigma)$ for synthetic data $\mathcal{N}(0, \text{diag}(0.1, 0.2, \dots, 3.0))$ in fully-connected neural networks with two layers of size 256. The graph is approximately continuous and of a regular structure: increasing, then decreasing. CENTER,RIGHT: $f(\sigma)$ that results from training a 2-layer convolutional network (256 filters followed by 64 fully-connected nodes) on CIFAR10 for random (CENTER) and real labels (RIGHT) after 5, 10, 20, and 40 epochs ($\sim$195 training iterations per epoch).

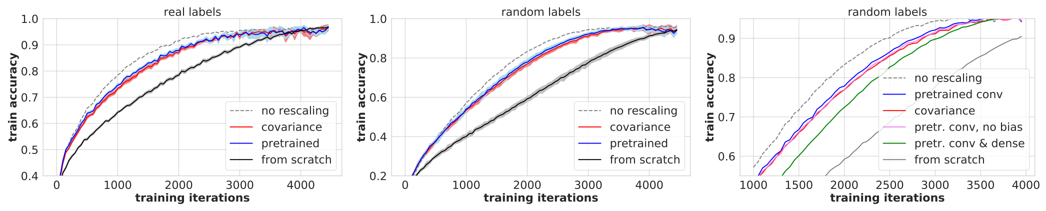


Figure 5: Training accuracy for learning real labels (LEFT) and random labels (MIDDLE, zoomed in: RIGHT) on CIFAR10 with a simple CNN (one 3×3 convolution, one hidden dense layer). Randomly initializing the convolutional filters from a learned covariance reproduces the effect of pre-training within measurement error (red and pink lines are almost indistinguishable in the right plot).

the definition of $f(\sigma)$. Such an extension is based on the following identity (2): Assume $\Sigma_x \in \mathbb{R}^{d \times d}$ has d distinct eigenvalues and let $\mathbf{v}_i$ be an eigenvector of length 1 with eigenvalue σ_i^2. If Σ_w is aligned with Σ_x, $\mathbf{v}_i$ is also an eigenvector of Σ_w and the corresponding eigenvalue τ_i^2 is:

$$\tau_i^2 = \mathbf{v}_i^T \Sigma_w \mathbf{v}_i = \mathbf{v}_i^T \mathbb{E}[(\mathbf{w} - \mu_w)(\mathbf{w} - \mu_w)^T] \mathbf{v}_i = \mathbb{E}[\langle \mathbf{w} - \mu_w, \mathbf{v}_i \rangle^2], \tag{2}$$

which is the variance of the projection of the weight vectors onto the principal axis $\mathbf{v}_i$. We can take this as the definition of τ_i in the general case, since this formulation can be applied even when we have an imperfect alignment between the eigenspaces.

Using this definition of $f(\sigma)$, we can now look at the shape of the function for concrete examples. Here, we train a simple fully connected network 50 times, collect statistics, and plot the corresponding mapping between each eigenvalue σ_i of Σ_x with the corresponding τ_i in (2). The result is shown in Figure 4 (LEFT). In general, $f(\sigma)$ on synthetic data looks smooth and exhibits a surprising structure: the function first increases before it decreases. Both the decreasing part or the increasing part may be missing depending on the setting (e.g. dimensionality of data and network architecture) but we observe the same shape of curves in all experiments. For real data (CIFAR10), Figure 4 (CENTER/RIGHT) shows that the function $f(\sigma)$ appears to have a similar shape (increasing, then decreasing, but less smooth) for training with both real and random labels. We interpret (without a formal argument) this surprisingly regular shape of $f(\sigma)$ to be the result of two effects: (1) Larger eigenvalues σ_i lead to larger effective learning rate in gradient descent, which leads in turn to larger corresponding τ_i, hence the increasing part of f. (2) Very large eigenvalues τ_i would dominate the output of the layer, masking the contribution of other components. Backpropagation compensates for this effect to capture more of the input signal. This leads to the decreasing part of f for higher σ_i.

2.4 Covariance alignment and eigenvalue mapping explains positive transfer experimentally

To connect the alignment effect to the faster learning downstream, we conduct the following experiment. Suppose that instead of pre-training on random labels, we sample from the Gaussian approximation of the filters in the first layer that were trained on random labels. In a simple CNN on

Table 1: Training and test accuracy on subsets of CIFAR10 of the initialization procedure described in Section 2.5 on the layers of a simple convolutional network. Both training and test accuracies improve with the number of layers that are initialized in this way.

		Convolutional layers sampled			
Iterations	Data	$\{\}$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$
100	Train	0.31	0.34	0.38	0.41
	Test	0.31	0.33	0.37	0.40
1000	Train	0.58	0.61	0.67	0.68
	Test	0.53	0.55	0.56	0.56

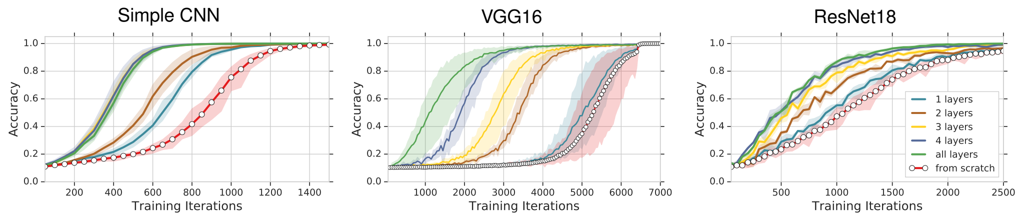


Figure 6: Transferring more layers improves downstream performance. Simple CNN architecture with 3 convolutional layers (LEFT), VGG16 (CENTER), and ResNet18 (RIGHT) pre-trained on CIFAR10 examples with random labels and subsequently fined-tuned on 25k fresh CIFAR10 examples with random labels. Lines with circular markers correspond to training from scratch. Error bars correspond to min/max over 3 runs. Plots for fine-tuning with real labels available in the appendix.

CIFAR10, consisting of one convolutional and one fully connected layers, the gain in downstream task is almost fully recovered, as shown in Figure 5. The gray curves show the raw effect of pre-training, but this contains the scaling effect. To eliminate the latter effect, we always re-scale the weights after pre-training to match their ℓ_2 norm at initialization (using ℓ_1 norm gives similar results). Recovering the transfer effect in this way implies that the positive transfer is mainly due to the second-order statistics of the weights in the first layer, which, by Proposition 1, are fully described by the alignment of principal components in combination with the shape of the function $f(\sigma)$.

Note that the combined results presented up to here indicate that both analytically and experimentally the following seems to be true: Given the training data of a neural network that is trained with random labels, we can predict the second-order statistics of its first layer weights up to a one-dimensional scaling function $f(\sigma)$ of the eigenvalues and this function has a surprisingly regular shape. If further investigation of f may lead us to understand its shape (which we regard as an interesting area of research), we could predict these after-training statistics perfectly by only gathering the data statistics.

2.5 Deeper layers

So far, we have only discussed effects in the first layer of a neural network. However, in Figure 6 we show that transferring more of the layers from the model pre-trained on random labels improves the effect considerably. For deeper networks transferring only the first layer gives a modest speedup for the training in the downstream phase. We have seen that we can reproduce the effect of training on random labels on the first layer by sampling from the corresponding covariance matrix, which in turn is given by the principal axes and a set of eigenvalues $\tau_1, ..., \tau_d$. So, if we can approximate the right (or good) eigenvalues, we can directly give an initialization that improves the training speed in the downstream phase. We can then iterate this procedure as well for the later layers in a fully connected or convolutional network. Given the filters for the earlier layers $L_1, ..., L_{k-1}$, for each training image we can compute the input to the layer L_k. Treating this as our input data, we determine the eigenvectors $e_1, e_2, ...$ of the covariance matrix. Then we compute the d most important directions and use $\tau_1 e_1, \tau_2 e_2, ...$ as our constructed filters. Applying this recipe to a CNN with three convolutional layers and one fully connected layer, we see that this indeed gives initializations that become better when we apply it to more layers, see Table 1. The performance after applying this approach to all three convolutional layers matches the performance of using the first three layers of a network trained on random labels. See Appendix E for details.

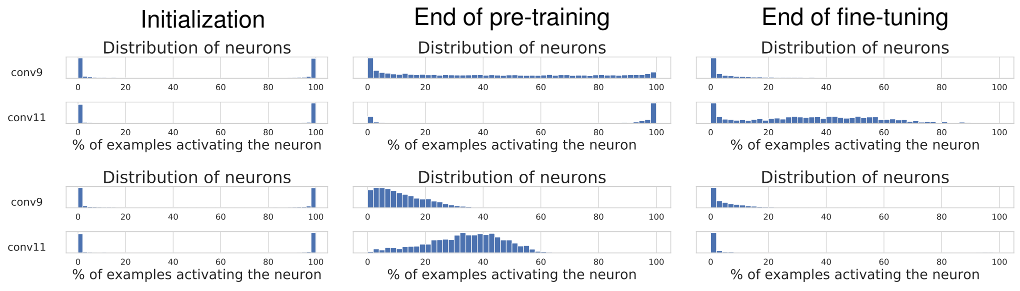


Figure 7: Activation plots for the two VGG16 models in Figure 1 at initialization (LEFT), after pre-training with random labels (CENTER), and after subsequently fine-tuning on fresh examples with random labels (RIGHT). Top row is for the positive transfer case; bottom row shows negative transfer. Histograms depict distributions of neurons over the fraction of *held out* examples that activate them. The two histograms in each subplot correspond to two different activation spaces.

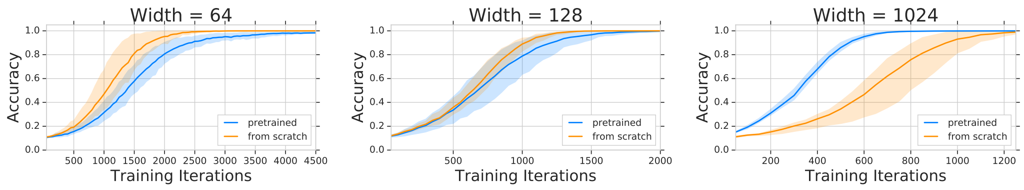


Figure 8: Increasing model width mitigates negative transfer. Simple CNN architectures with two convolutional layers and 64 (LEFT), 128 (CENTER), and 1024 (RIGHT) units in the dense layer.

3 Specializing neurons

Despite the alignment effect taking place at the earlier layers of the neural network when trained with random labels, negative transfer is sometimes observed when fine-tuning on a downstream task as shown in Figure 1. In this section, we show that this is likely due to a specialization at the later layer.

Figure 7 displays the distribution of neurons with respect to the number of held out images they are activated by for the settings of Figure 1 that exhibited positive (top row) and negative (bottom row) transfers. Comparing neural activations during initialization, end of pre-training, and end of fine-tuning, we note that neural activations are markedly diminished in the negative transfer case compared to the positive transfer case despite the fact that their neural activation distributions were identical during initialization. In Appendix F, we show that the significant drop in neural activation in the negative transfer case happens immediately after switching to the downstream task. As a result, the effective capacity available downstream is diminished. By contrast, neural activations are not severely impacted in the positive transfer setting. In Appendix F, we provide detailed figures describing this phenomenon across all layers of VGG16, which reveal that such a specialization effect becomes more prominent in the later layers compared to the earlier layers.

One way to mitigate the effect of the inactive ReLU units is to increase the width so that the capacity remains sufficiently large for the downstream task. Figure 8 shows that increasing the width can indeed mitigate the negative transfer effect. While increased width seems to have general performance advantages [53], it seems to be also particularly useful in the case of transfer learning [29].

4 Concluding remarks

The objective of this paper is to answer the question of what neural networks learn when trained on random labels. We provide a partial answer by proving an alignment effect of principal components of network parameters and data and studying its implications, particularly for transfer learning. One important consequence is that second-order statistics of the earlier layers can be reduced to a one-dimensional function, which exhibits a surprising, regular structure. It remains an open question what the "optimal" shape of such function is, or whether it can be described analytically.

The models used in this paper are taken from a large set of experiments that we conducted using popular network architectures and datasets, such as simple convolutional networks, VGG16, ResNet18-v2, CIFAR10, and ImageNet, with wide range of hyperparameter settings (Appendix B). These experiments show that pre-training on random labels very often accelerates training on downstream tasks compared to training from scratch with the *same hyperparameters* and rarely hurts the training speed.

By studying what is learned on random labels, we shed new insights into previous phenomena that have been reported in the literature. For instance, the alignment effect at the earlier layers explains the empirical observations of [41] that pre-training on random labels can accelerate training in downstream tasks and the observation of [4] that the filters learned on random labels seem to exhibit some useful structure. Also, our findings related to the inactive ReLU units at the later layers demonstrate how upper layers specialize early during training, which may explain why neural networks exhibit critical learning stages [1] and why increasing the width seems to be particularly useful in transfer learning [29]. Both alignment and specialization are in agreement with the observation that earlier layers generalize while later layers specialize, a conclusion that has been consistently observed in the literature when training on real labels [3, 4, 10, 52]. We show that it holds for random labels as well.

Acknowledgements

The authors are grateful to Alexander Kolesnikov, Alexandru Ţifrea, Larisa Markeeva, Lionel Ngoupeyou Tondji, Lucas Beyer, Philippe Gervais, and other members of the Brain Team for valuable discussions.

References

[1] A. Achille, M. Rovere, and S. Soatto. Critical learning periods in deep networks. In *ICLR*, 2019.

[2] S. Amari. *Information geometry and its applications*. Springer, 2016.

[3] A. Ansuini, A. Laio, J. H. Macke, and D. Zoccolan. Intrinsic dimension of data representations in deep neural networks. In *NeurIPS*, 2019.

[4] D. Arpit, S. Jastrzębski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio, and S. Lacoste-Julien. A closer look at memorization in deep networks. In *ICML*, 2017.

[5] N. Ay, J. Jost, H. Vân Lê, and L. Schwachhöfer. *Information geometry*. Springer, 2017.

[6] P. L. Bartlett. The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network. *IEEE Trans. Information Theory*, 44 (2):525–536, 1998.

[7] P. L. Bartlett, D. J. Foster, and M. Telgarsky. Spectrally-normalized margin bounds for neural networks. In *NeurIPS*, 2017.

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv:2005.14165*, 2020.

[9] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). *arXiv:1511.07289*, 2015.

[10] G. Cohen, G. Sapiro, and R. Giryes. DNN or k-NN: That is the generalize vs. memorize question. *arXiv:1805.06822*, 2018.

[11] E. Collins, S. A. Bigdeli, and S. Süsstrunk. Detecting memorization in ReLU networks. *arXiv:1810.03372*, 2018.

[12] S. I. Costa, S. A. Santos, and J. E. Strapasson. Fisher information distance: a geometrical reading. *Discrete Applied Mathematics*, 197:59–69, 2015.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[14] A. Dosovitskiy, J. T. Springenberg, M. Riedmiller, and T. Brox. Discriminative unsupervised feature learning with convolutional neural networks. In *NeurIPS*, 2014.

[15] S. C. Douglas and J. Yu. Why ReLU units sometimes die: Analysis of single-unit error backpropagation in neural networks. In *52nd ACSSC*, 2018.

[16] J. Frankle, D. J. Schwab, and A. S. Morcos. The early phase of neural network training. In *ICLR*, 2020.

[17] Y. Gan, J. Liu, J. Dong, and G. Zhong. A PCA-based convolutional network. *arXiv:1505.03703*, 2015.

[18] J. Gu and V. Tresp. Neural network memorization dissection, 2019.

[19] B. Han, Q. Yao, X. Yu, G. Niu, M. Xu, W. Hu, I. Tsang, and M. Sugiyama. Co-teaching: Robust training of deep neural networks with extremely noisy labels. In *NeurIPS*, 2018.

[20] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[21] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *ECCV*, 2016.

[22] H. v. Helmholtz. *Handbuch der Physiologischen Optik. 2.Auflage.* Leopold Voss, 1896.

[23] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, 2019.

[24] P. J. Huber. Robust statistics. In *International Encyclopedia of Statistical Science*, pages 1248–1251. Springer, 2011.

[25] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[26] L. Jiang, Z. Zhou, T. Leung, L.-J. Li, and L. Fei-Fei. Mentornet: Learning data-driven curriculum for very deep neural networks on corrupted labels. In *ICML*, 2018.

[27] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017.

[28] J. Kivinen and M. K. Warmuth. Exponentiated gradient versus gradient descent for linear predictors. *Information and Computation*, 132(1):1–63, 1997.

[29] A. Kolesnikov, L. Beyer, X. Zhai, J. Puigcerver, J. Yung, S. Gelly, and N. Houlsby. Big transfer (BiT): General visual representation learning. *arXiv:1912.11370*, 2019.

[30] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[31] Y. Li, J. Yang, Y. Song, L. Cao, J. Luo, and L.-J. Li. Learning from noisy labels with distillation. In *CVPR*, 2017.

[32] T. Liu and D. Tao. Classification with noisy labels by importance reweighting. *IEEE TPAMI*, 38(3):447–461, 2015.

[33] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis. Dying ReLU and initialization: Theory and numerical examples. *arXiv:1903.06733*, 2019.

[34] X. Ma, Y. Wang, M. E. Houle, S. Zhou, S. Erfani, S. Xia, S. Wijewickrema, and J. Bailey. Dimensionality-driven learning with noisy labels. In *ICML*, 2018.

[35] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *ICML*, 2013.

[36] G. Montavon, M. L. Braun, and K.-R. Müller. Kernel analysis of deep networks. *JMLR*, 12(78): 2563–2581, 2011.

[37] N. Natarajan, I. S. Dhillon, P. K. Ravikumar, and A. Tewari. Learning with noisy labels. In *NeurIPS*, 2013.

[38] B. Neyshabur, R. Tomioka, and N. Srebro. Norm-based capacity control in neural networks. In *CoLT*, 2015.

[39] B. Neyshabur, S. Bhojanapalli, D. McAllester, and N. Srebro. Exploring generalization in deep learning. In *NeurIPS*, 2017.

[40] R. Novak, Y. Bahri, D. Abolafia, J. Pennington, and J. Sohl-dickstein. Sensitivity and generalization in neural networks: an empirical study. In *ICLR*, 2018.

[41] V. Pondenkandath, M. Alberti, S. Puran, R. Ingold, and M. Liwicki. Leveraging random label memorization for unsupervised pre-training. *arXiv:1811.01640*, 2018.

[42] M. Raghu, C. Zhang, J. Kleinberg, and S. Bengio. Transfusion: Understanding transfer learning for medical imaging. In *NeurIPS*, 2019.

[43] X.-D. Ren, H.-N. Guo, G.-C. He, X. Xu, C. Di, and S.-H. Li. Convolutional neural network based on principal component analysis initialization for image classification. In *IEEE DSC*, 2016.

[44] A. Saxe, J. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *ICLR*, 2014.

[45] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[46] J. Sirignano and K. Spiliopoulos. Mean field analysis of neural networks: A central limit theorem. *arXiv:1808.09372*, 2018.

[47] S. Sukhbaatar and R. Fergus. Learning from noisy labels with deep neural networks. *arXiv:1406.2080*, 2014.

[48] R. Wagner, M. Thom, R. Schweiger, G. Palm, and A. Rothermel. Learning convolutional neural networks from few samples. In *IJCNN*, 2013.

[49] L. Wasserman. *All of statistics: a concise course in statistical inference*. Springer, 2004.

[50] Y. Wu and K. He. Group normalization. In *ECCV*, 2018.

[51] C. Ye, M. Evanusa, H. He, A. Mitrokhin, T. Goldstein, J. A. Yorke, C. Fermuller, and Y. Aloimonos. Network deconvolution. In *ICLR*, 2020.

[52] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *NeurIPS*, 2014.

[53] S. Zagoruyko and N. Komodakis. Wide residual networks. In *BMVC*, 2016.

[54] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. In *ICLR*, 2017.

[55] Z. Zhang and M. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. In *NeurIPS*, 2018.

A Experimental details

A.1 Upstream and downstream datasets

We use two datasets: CIFAR10 [30] and ImageNet ILSVRC-2012 [13].

For each run of a transfer experiment we

1. randomly sample `num_examples_upstream` examples uniformly from the training split of the dataset to form the (upstream) training set for pre-training;
2. randomly sample `num_examples_downstream` examples from the remainder of the training split of the dataset. They form the (downstream) training set for fine-tuning. This guarantees that the upstream and downstream examples do not intersect;
3. upstream training examples are labelled with `num_classes_upstream` classes randomly and uniformly;
4. when fine-tuning on random labels, we randomly and uniformly label the downstream examples with `num_classes_downstream` classes.

A.2 Neural network architectures

Throughout the main text we use three different architectures: a simple convolutional architecture "Simple CNN", VGG16, and ResNet18. The Simple CNN can be further configured by specifying the number of convolutional layers and units in the dense layer. We use the same architectures for both CIFAR10 and ImageNet datasets, by only adjusting the number of outputs (logits).

Simple CNN is a convolutional architecture consisting of:

1. `num_conv_layers` convolutional layers with 3×3 filters, each followed by the ReLU activation. Each convolutional layer contains `num_filters` filters (with biases) that are applied with stride 1.
2. The outputs of the final convolutional layer are flattened and passed to a dense layer (with biases) of `num_units` units, followed by the ReLU activation.
3. The classifier head, i.e. a dense layer with `num_output` units (logits).

The **VGG16** architecture that we use is "Configuration D" from Table 1 of [45] with two dense layers ("FC-4096") removed:

```
conv0:   64 filters
conv1:   64 filters
maxpool
conv2:  128 filters
conv3:  128 filters
maxpool
conv4:  256 filters
conv5:  256 filters
conv6:  256 filters
maxpool
conv7:  512 filters
conv8:  512 filters
conv9:  512 filters
maxpool
conv10: 512 filters
conv11: 512 filters
conv12: 512 filters
maxpool
dense layer with num_outputs units (classifier head)
```

All convolutional filters (with biases) are of size 3×3 and applied with `SAME` padding and stride 1. ReLU activation is applied after every convolutional layer. Max-pooling is performed over a 2×2 pixel window, with SAME padding and stride 2.

ResNet18 We use vanilla ResNet-v2 architecture [21] with batch normalization [25] replaced by the group normalization [50].

A.3 Training

All biases in the models are initialized with zeros, while all the other parameters (convolutional filters and weight matrices of the dense layers) are initialized using He normal algorithm [20] with `init_scale` scaling.

Model outputs (logits) are passed through the softmax function and we minimize the cross-entropy (i.e. the negative log-likelihood) during training. We use SGD with momentum 0.9 and batch size 256 to train our models. We start training with the specified `learning_rate` and divide it by 3 two times during the training: after $1/3\times$`total_steps` and $2/3\times$`total_steps` steps.

When training the models we report the accuracy on *the entire training set*, not on the mini-batch.

We do not use data augmentation in our experiments. We do not use weight decay, dropout, or any other regularization. For CIFAR10 we scale the inputs to the $[-1, 1]$ range. For ImageNet we resize the examples, take a 224×224 central crop, and scale inputs to the $[-1, 1]$ range.

A.4 Transferring the model from the upstream to the downstream task

After the upstream pre-training finishes, we replace the classifier head with a freshly initialized one.

Unless otherwise stated, for Simple CNN and VGG16 architectures we also re-scale the model parameters after the pre-training. We store the per-layer parameter ℓ_2 norms at the initialization and re-scale each layer of the trained model to match the stored ℓ_2 norms. Re-scaling does not affect the classifier accuracy and predictions, since it reduces to multiplying all parameters of a given layer by a strictly positive constant. However, re-scaling changes the logits and, therefore, the cross-entropy loss. It is not immediately obvious how to re-scale the residual architecture without changing its predictions so we decided not to re-scale ResNet18.

When fine-tuning downstream we use the same initial learning rate and schedule as in upstream pre-training.

A.5 Figure 1: Positive and negative transfer with VGG16 on CIFAR10

Experiments 1 and 2 use:

```
init_scale = 0.526
learning_rate = 0.01
num_classes_upstream = 5
num_examples_upstream = 20000
epochs_upstream = 120
```

Experiments 3 and 4 use:

```
init_scale = 0.612
learning_rate = 0.009
num_classes_upstream = 50
num_examples_upstream = 20000
epochs_upstream = 80
```

Error bars correspond to ± 1 std. over 12 independent runs.

A.6 Figure 2: Misalignment plots

To measure misalignment, we use the misalignment score in Definition 1 using the closed-from expression provided in Section 2.2. First, the eigenvectors of the 3×3 patches of images are computed, which reside in a space of dimension 27 due to the three input channels. Let v_i be the eigenvectors of data, which have distinct eigenvalues almost surely, the alignment score is then:

$$\sum_{i=1}^{d} \sqrt{(v_i^T \Sigma_w v_i) \cdot (v_i^T \Sigma_w^{-1} v_i)} - 1,$$

where the summation is taken over all 27 eigenvectors. To gain an intuition behind this formula, note that if v_i is itself an eigenvector of Σ_w, then $(v_i^T \Sigma_w v_i) \cdot (v_i^T \Sigma_w^{-1} v_i) = 1$. The covariance of weights Σ_w is estimated in a single run by computing the covariance of the filters in the first layer.

To ensure that alignment depends indeed on the eigenvectors of data, we also compute alignment in which the set of data eigenvectors $\{v_i\}$ is replaced by some random orthonormal basis of the plane. The network architecture is a 2-layer CNN: convolutional followed by a fully-connected layer. The experiment setup used to produce Figure 2 is:

```
num_conv_layers = 1
num_filters = 256
num_units = 64
learning_rate = 0.01
num_classes = 10
num_examples = 50000
epochs_upstream = 40
```

A.7 Figure 3: ResNet convolutional filter alignment

Figure 3 was produced from 70 runs of a wide ResNet [53] on CIFAR10 with random labels. We used 4 blocks per group and a width factor of 4. As mentioned in the main text, we replaced the initial 3×3 convolution with a 5×5 convolution for better visualization. Training was run with batch normalization [25] and the following parameters:

```
init_scale = 0.5
learning_rate = 0.005
num_examples = 50000
epochs = 1000
```

A.8 Figure 4: Plots of the function $f(\sigma)$

These figures were generated using the procedure described in Section 2.3. For every eigenvalue σ_i^2 of the data with eigenvector v_i, the corresponding τ_i^2 is computed using Eq. (2). The pairs (σ_i^2, τ_i^2) define a mapping from $\mathbb{R}^+$ to $\mathbb{R}^+$, which is plotted in Figure 4.

The network architecture is a 2-layer CNN: convolutional followed by a fully-connected layer. The experiment setup used to produce Figure 2 is:

```
num_conv_layers = 1
num_filters = 256
num_units = 64
learning_rate = 0.01
num_classes = 10
num_examples = 50000
epochs_upstream = 40
```

A.9 Figure 5: Explaining the positive transfer

Network: SimpleCNN

```
num_conv_layers = 1
num_filters = 64
num_units = 256
```

Training:

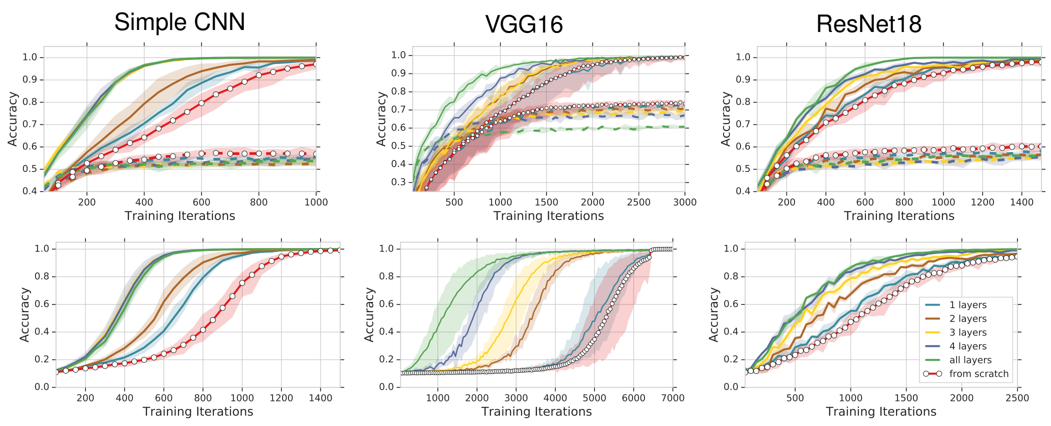


Figure 9: Transferring more layers improves the downstream performance. Simple CNN architecture with 3 conv. layers (LEFT), VGG16 (CENTER), and ResNet18 (RIGHT) pre-trained on CIFAR10 examples with random labels and subsequently fined-tuned on 25k fresh CIFAR10 examples with real labels (TOP) and 10 random labels (BOTTOM). Lines with circular markers correspond to training from scratch. Error bars correspond to min/max over 3 independent runs.

```
num_examples_upstream = 10000
num_classes_upstream = 10
num_examples_downstream = 10000
num_classes_downstream = 10
init_scaling = 1.0
learning_rate = 0.0005
total_steps = 10000
```

For all curves with the exception of the "no rescaling" curve, each layer is scaled down by a factor between pre-training and training to match the ℓ_2 norm of the weights at initialization.

For all runs, the head layer is re-initialized after pre-training. For "pretrained conv" and "pretrained conv, no bias" also the fully connected layer is re-initialized. For "pretraind conv, no bias" also the bias of the convolutional layer is reset to zero. This is the fairest comparison to "covariance": For "covariance" the filters in the convolutional layer are random samples from a Gaussian distribution with mean 0 and the covariance obtained from training on random labels. The bias is set to zero and the dense and head layers are initialized randomly.

Each line is the average of 12 runs that differ in the random initializations. In the left and center image, the "covariance", "pretrained", and "from scratch" curves are surrounded by a colored area indicating ± 1 standard deviations. The "no rescaling" curve is without this area, it would touch the "pretrained" curve below it. In the right image no error bounds are plotted since the curves are too close together.

A.10 Figure 6: Transferring more layers improves the downstream performance

Figure 9 is the extended version of Figure 6 that includes the models fine-tuned on real labels (apart from the models fine-tuned on the random labels).

The left column (Simple CNN architecture) uses:

```
init_scale = 0.518
learning_rate = 0.01
num_classes_upstream = 25
num_examples_upstream = 15000
epochs_upstream = 40
num_conv_layers = 3
num_filters = 16
num_units = 1024
```

The center column (VGG16) corresponds to the same setup as in Experiments 1 and 2 in Figure 1. It uses:

```
init_scale = 0.526
learning_rate = 0.01
num_classes_upstream = 5
num_examples_upstream = 20000
epochs_upstream = 120
```

The right column (ResNet18) uses:

```
init_scale = 0.671
learning_rate = 0.01
num_classes_upstream = 50
num_examples_upstream = 25000
epochs_upstream = 80
```

A.11 Figure 7: Neuron activation plots

Figures 10 and 11 are the extended versions of Figure 7. They include activation plots for *all intermediate layers* of VGG16 models (i) at initialization, (ii) in the end of the pre-training, (iii) in the end of fine-tuning on real labels, (iv) in the end of fine-tuning on 10 random labels. Figures 10 and 11 illustrate the negative and positive transfer examples respectively.

The VGG16 model from Figure 10 (top row in Figure 8) uses the same setup as in Experiments 3 and 4 in Figure 1:

```
init_scale = 0.612
learning_rate = 0.01
num_classes_upstream = 50
num_examples_upstream = 20000
epochs_upstream = 80
```

The VGG16 model from Figure 11 (bottom row in Figure 8) uses the same setup as in Experiments 1 and 2 in Figure 1:

```
init_scale = 0.526
learning_rate = 0.01
num_classes_upstream = 5
num_examples_upstream = 20000
epochs_upstream = 120
```

A.12 Figure 8: Increasing the width mitigates the negative transfer

The models from all three subplots use Simple CNN architecture and share the same parameters:

```
init_scale = 1.218
learning_rate = 0.012
num_classes_upstream = 2
num_examples_upstream = 10000
epochs_upstream = 100
num_conv_layers = 2
num_filters = 16
```

The models from LEFT, CENTER, and RIGHT subplots use `num_units` of 64, 128, and 1 024, respectively. Error bars correspond to max/min over 3 independent runs.

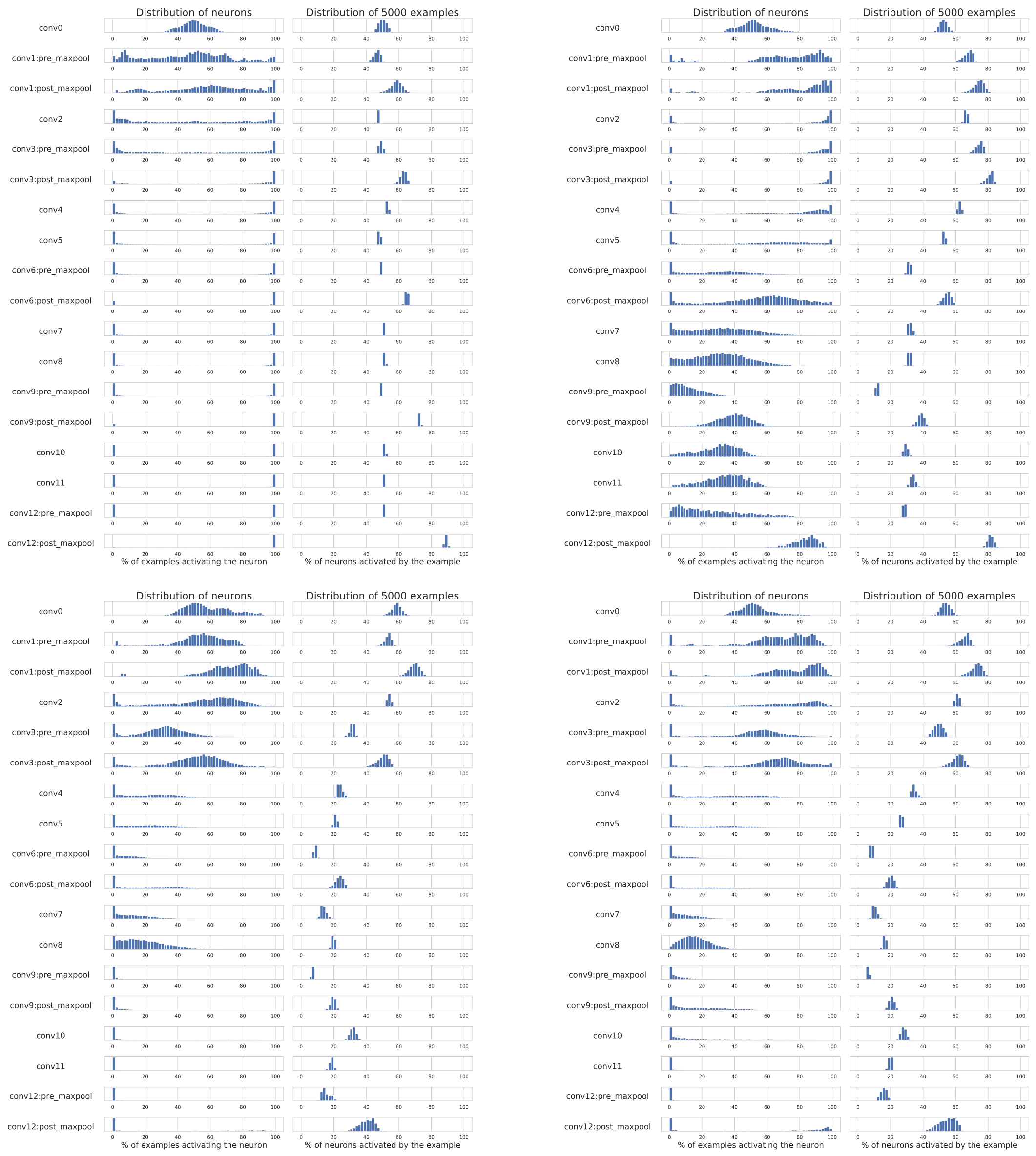


Figure 10: Neuron activations in case of the negative transfer. The VGG16 model pre-trained for 1 (TOP-LEFT) and 6240 (TOP-RIGHT) training iterations on 20k examples from CIFAR10 with 50 random classes and subsequently fine-tuned for 200 epochs on fresh 25k examples from CIFAR10 with real labels (BOTTOM-LEFT) and 10 random labels (BOTTOM-RIGHT). In each subplot, the left column depicts distributions of neurons over the fraction of input examples that activate them. The right column depicts distribution of the input examples over the fraction of neurons that are activated by them. The 5k input examples were taken from the holdout test split of CIFAR10.

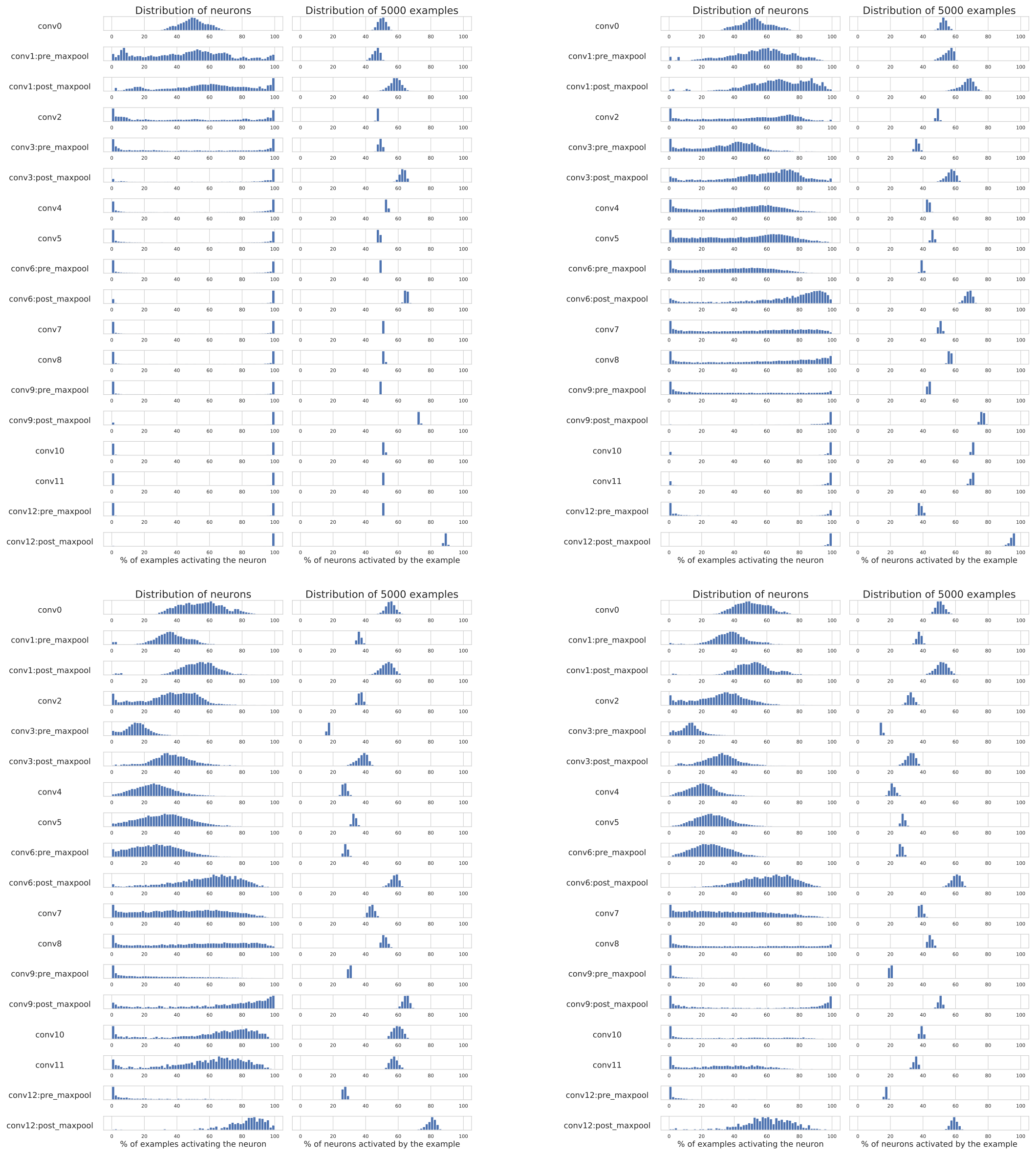


Figure 11: Neuron activations in case of the positive transfer. The VGG16 model pre-trained for 1 (TOP-LEFT) and 9360 (TOP-RIGHT) training iterations on 20k examples from CIFAR10 with 5 random classes and subsequently fine-tuned for 200 epochs on fresh 25k examples from CIFAR10 with real labels (BOTTOM-LEFT) and 10 random labels (BOTTOM-RIGHT). In each subplot, the left column depicts distributions of neurons over the fraction of input examples that activate them. The right column depicts distribution of the input examples over the fraction of neurons that are activated by them. The 5k input examples were taken from the holdout test split of CIFAR10.

B Empirical evidence with diverse real-world settings

We argued theoretically that the alignment effect holds under certain idealized conditions (Proposition 1) and demonstrated that it entirely explains the positive transfer in a real-world setting (Figure 5). In this appendix, we confirm that the positive transfer is reproducible and can be frequently observed in common real-world settings with various popular network architectures, datasets, and different hyperparameters.

Experimental setup We run multiple transfer experiments, repeating each run with several different seeds. We consider three network architectures: (a) a simple convolutional architecture with a configurable number of conv. layers, filters, and units in one dense layer, (b) VGG16 [45] with two final dense layers of width 4096 removed, and (c) ResNet-v2 architecture [21]. The *disjoint* upstream and downstream training sets are sampled randomly from one of the two datasets: CIFAR10 [30] or ImageNet ILSVRC-2012 [13]. We never transfer *between different* datasets. The models are pre-trained with random labels and fine-tuned for a fixed number of epochs with either real or random labels using SGD with momentum 0.9. We randomly explore various initial learning rates, initialization types and scales, numbers of random classes and examples upstream, duration of the pre-training, and configurations of the Simple CNN architecture.

We collect two sets of experiments for CIFAR10 (Experiments A and B) and two sets for ImageNet (Experiments A and B) reported below. Each set consists of multiple *groups* of experiments. Experiments are gathered in a *group* if they share same (1) architecture, (2) learning rate, (3) initialization type and scale, (4) number of examples upstream, (5) number of classes upstream, and (6) number of epochs upstream. For each configuration of these 6 parameters we explore *all possible combinations* of the following choices: (a) the random seed, (b) pre-train / train from scratch on downstream, (c) transfer one layer / all layers [optional], (d) train downstream with real / random labels. Gathering experiments in groups like this allows to compare the downstream performance of the pre-trained models to that of models trained from scratch *with same hyperparameters*.

Visualizing the experiments In order to visualize the experiments we summarize each group with two numbers: one characterizing the downstream performance of the pre-trained models, and one for the models trained from scratch. In order to capture the speed of training we use the *area under the curve* (AUC) to sketch the training with a single number. For instance, the pre-trained model in Experiment 2 of Figure 1 (blue line) trains faster than the one trained from scratch (orange line). Accordingly, the area under the blue curve is larger than the area under the orange curve. We can now visualize all the groups of experiments on a single scatter plot. Depending on what exact curves we summarize with AUC, we get three scatter plots: (1) training accuracy when using real labels downstream (areas under the solid lines in Experiments 1 and 3 in Figure 1), (2) test accuracy when using real labels downstream (areas under the dotted lines in Experiments 1 and 3 in Figure 1), (3) training accuracy when using random labels downstream (areas under the solid lines in Experiments 2 and 4 in Figure 1).

B.1 CIFAR10, Experiments A

This set of experiments counts 20 groups per architecture. We use 2 different random seeds, resulting in 16 experiments per group, or $3 \times 20 \times 16 = 960$ trained models in total.

The following parameters are shared between all runs:

```
num_examples_downstream = 25000
num_epochs_downstream = 200
init_algorithm = "he"
```

For each group of experiments we sample the following parameters randomly:

```
init_scale = random log_uniform(0.5, 1.35)
learning_rate = random log_uniform(0.008, 0.0125)
num_examples_upstream = random uniform([5000, 10000, 15000, 20000, 25000])
num_classes_upstream = random uniform([2, 5, 10, 25, 50, 100])
num_epochs_upstream = random uniform([20, 40, 80, 100, 120, 150, 200])
```

For the Simple CNN architecture we also randomly sample

```
num_conv_layers = random uniform([1, 2, 3, 4, 5])
num_units = random uniform([64, 128, 256, 512, 1024])
num_filters = random uniform([16, 32])
```

Exmeriments A are depicted on Figure 12. Two clusters of points in the CENTER row correspond to models with VGG16 architecture ($y > 0.7$) and models with SimpleCNN or ResNet18 architecture ($y \leq 0.7$). Surprisingly, several experiments with transferring one (first) layer of VGG16 when fine-tuning with real labels lead to improved test accuracy compared to the same models trained from scratch (blue squares, CENTER row).

B.2 CIFAR10, Experiments B

This set of experiments contains 20 groups per architecture. We use 2 different random seeds, resulting in 16 experiments per group, or $3 \times 20 \times 16 = 960$ trained models in total.

We use slightly wider hyperparameter ranges compared to the CIFAR10 Experiments A. Also, we include the orthogonal initialization algorithm [44]. The following parameters are shared between all runs:

```
num_examples_downstream = 25000
num_epochs_downstream = 200
```

For each group of experiments we sample the following parameters randomly:

```
init_algorithm = random uniform(["he", "orthogonal"])
init_scale = random log_uniform(0.5, 2.)
learning_rate = random log_uniform(0.005, 0.02)
num_examples_upstream = random uniform([5000, 10000, 15000, 20000, 25000])
num_classes_upstream = random uniform([2, 5, 10, 25, 50, 100])
num_epochs_upstream = random uniform([20, 40, 80, 100, 120, 150, 200])
```

For the Simple CNN architecture we also randomly sample

```
num_conv_layers = random uniform([1, 2, 3, 4, 5])
num_units = random uniform([512, 1024, 2048])
num_filters = random uniform([16, 32])
```

Exmeriments B are depicted on Figure 13. The models with $y > 0.7$ on CENTER row again correspond to the VGG16 architecture. This time we do not observe improvements in the test performance when pre-training.

B.3 ImageNet, Experiments A

This set of experiments counts 10 groups per architecture. We use 5 different random seeds. We always transfer all layers. This results in 20 experiments per group, or $3 \times 10 \times 20 = 600$ trained models in total.

The following parameters are shared between all runs:

```
num_examples_downstream = 200000
num_epochs_downstream = 100
num_epochs_upstream = 100
num_classes_upstream = 1000
init_algorithm = "he"
```

For each group of experiments we sample the following parameters randomly:

```
init_scale = random log_uniform(0.5, 1.5)
learning_rate = random log_uniform(0.008, 0.0125)
num_examples_upstream = random uniform([50000, 100000])
```

For the Simple CNN architecture we also randomly sample

```
num_conv_layers = random uniform([1, 2, 3])
num_units = random uniform([32, 64, 128, 256, 512, 1024])
num_filters = random uniform([16, 32, 64])
```

ImageNet Exmeriments A are depicted on Figure 14. All the ResNet18 runs result in positive transfer in terms of the downstream training performance, i.e. all the green points on TOP and BOTTOM plots are below the diagonal. This may be caused by the fact that we do not re-scale ResNet18 models after pre-training, which means the effective downstream learning rate of the pre-trained ResNets18 may be affected.

B.4 ImageNet, Experiments B

This set of experiments counts 10 groups per architecture. We use 5 different random seeds. We always transfer all layers. This results in 20 experiments per group, or $3 \times 10 \times 20 = 600$ trained models in total.

We use smaller upstream and downstream training sizes compared to ImageNet Experiments A.

The following parameters are shared between all runs:

```
num_examples_downstream = 50000
num_epochs_downstream = 100
num_epochs_upstream = 100
num_classes_upstream = 1000
init_algorithm = "he"
```

For each group of experiments we sample the following parameters randomly:

```
init_scale = random log_uniform(0.5, 1.5)
learning_rate = random log_uniform(0.008, 0.0125)
num_examples_upstream = random uniform([25000, 50000])
```

For the Simple CNN architecture we also randomly sample

```
num_conv_layers = random uniform([1, 2, 3])
num_units = random uniform([32, 64, 128, 256, 512, 1024])
num_filters = random uniform([16, 32, 64])
```

ImageNet Exmeriments B are depicted on Figure 15. We see again that all the ResNet18 runs result in positive transfer in terms of the downstream training performance. However, this time we do not observe the positive transfer with architectures other than ResNet18.

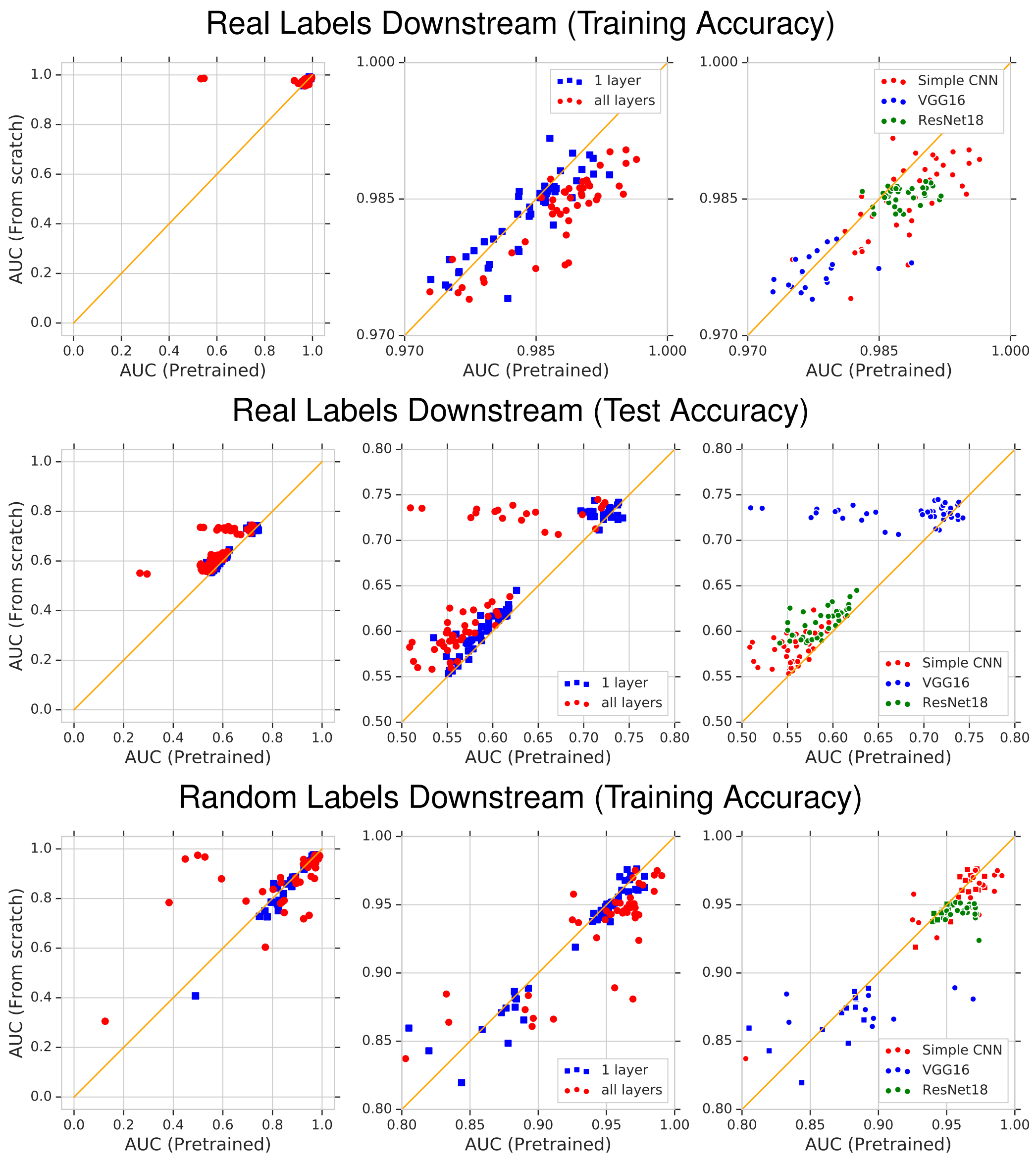


Figure 12: Scatter plots for CIFAR10 Experiments A. Models fine-tuned on real labels (TOP and CENTER) and on random labels (BOTTOM). Each dot corresponds to one *group of experiments*. Points below the orange $x = y$ line on TOP correspond to experiments where models pre-trained with random labels train faster downstream (with real labels) compared to models trained from scratch *with same hyperparameters*. CENTER AND RIGHT columns contain zoomed in versions of the plots from LEFT column.

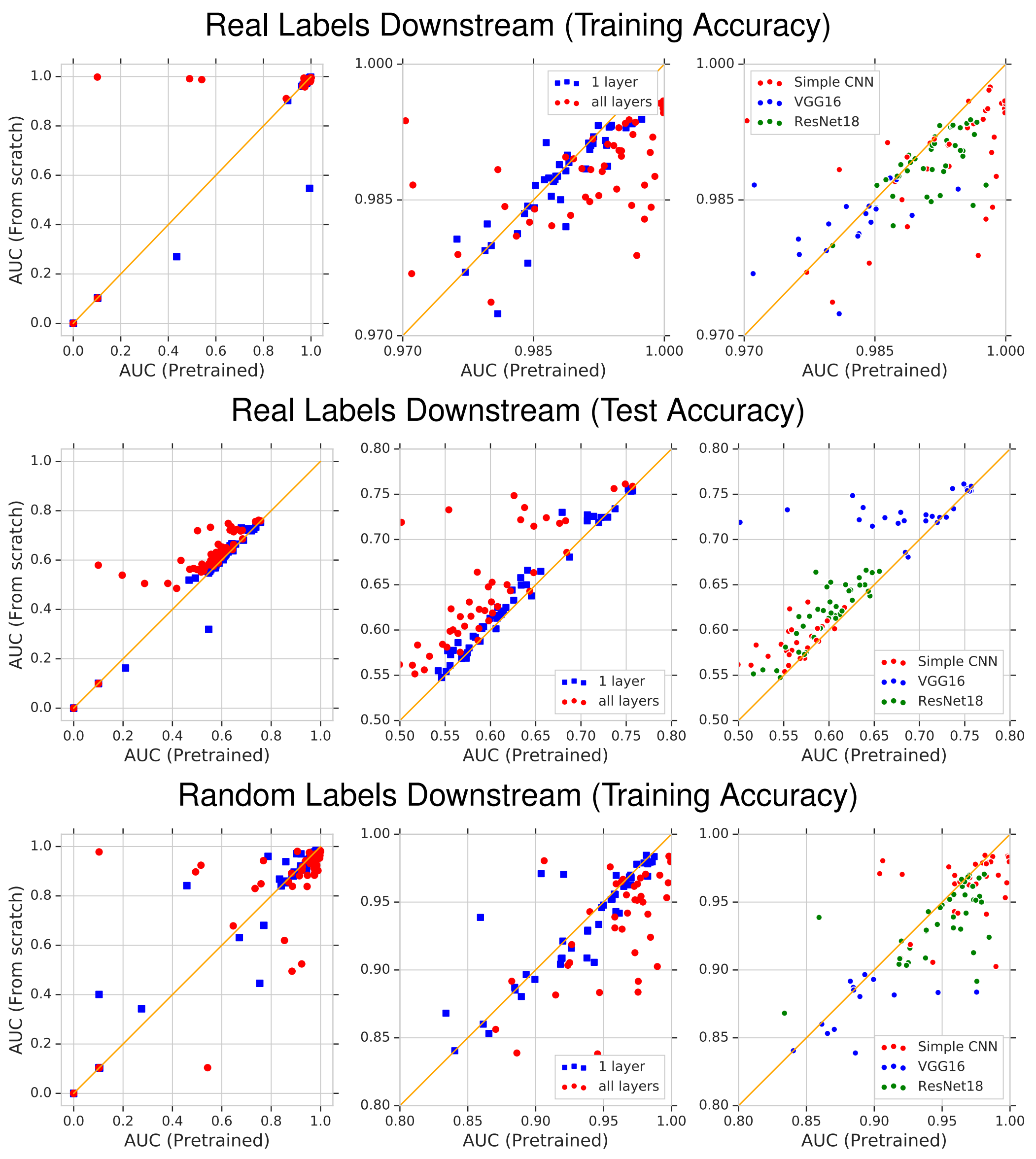


Figure 13: Scatter plots for CIFAR10 Experiments B. Models fine-tuned on real labels (TOP and CENTER) and on random labels (BOTTOM). Each dot corresponds to one *group of experiments*. Points below the orange $x = y$ line on TOP correspond to experiments where models pre-trained with random labels train faster downstream (with real labels) compared to models trained from scratch *with same hyperparameters*. CENTER AND RIGHT columns contain zoomed in versions of the plots from LEFT column.

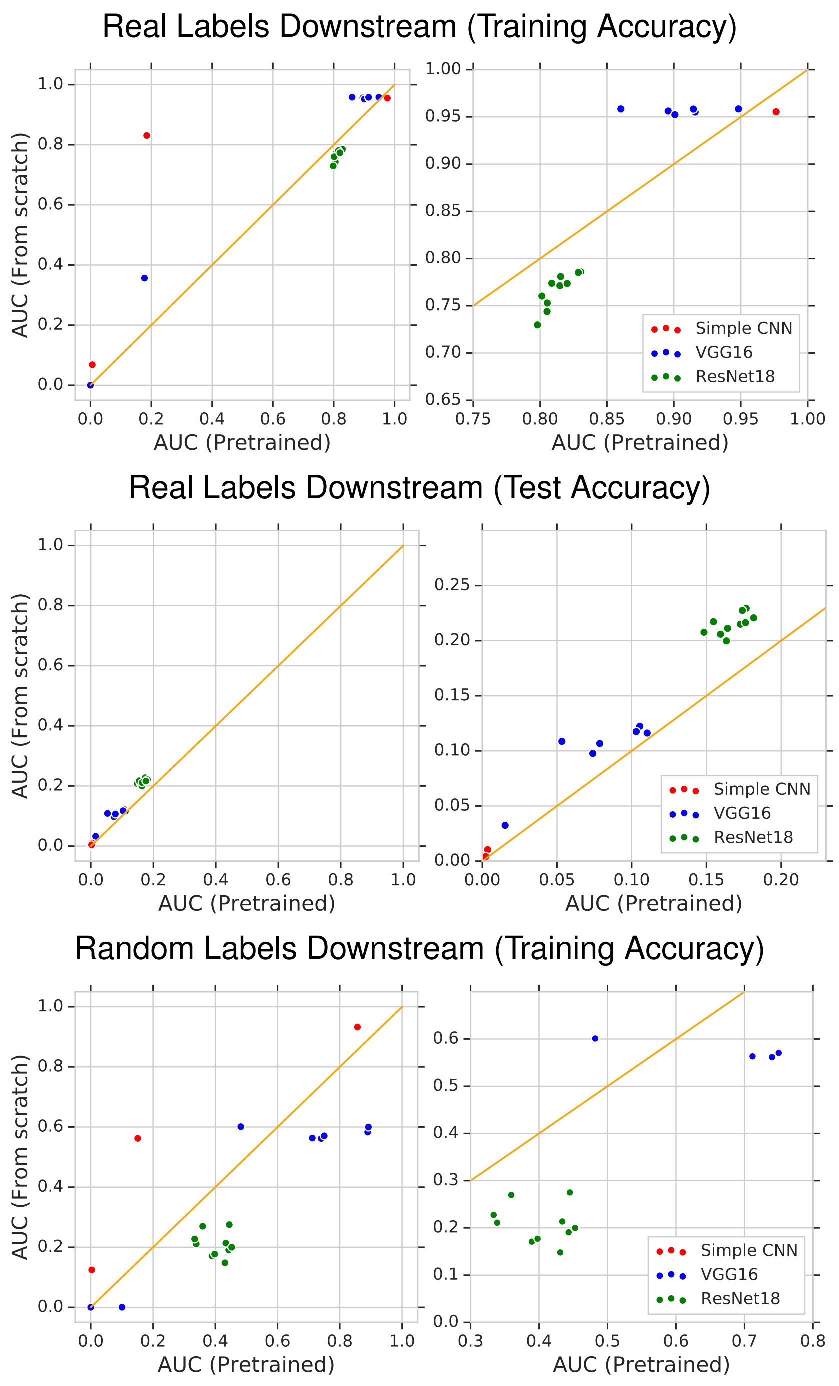


Figure 14: Scatter plots for ImageNet Experiments A. Models fine-tuned on real labels (TOP and CENTER) and on random labels (BOTTOM). Each dot corresponds to one *group of experiments*. Points below the orange $x = y$ line on TOP correspond to experiments where models pre-trained with random labels train faster downstream (with real labels) compared to models trained from scratch *with same hyperparameters*. RIGHT column contains zoomed in versions of the plots from LEFT column.

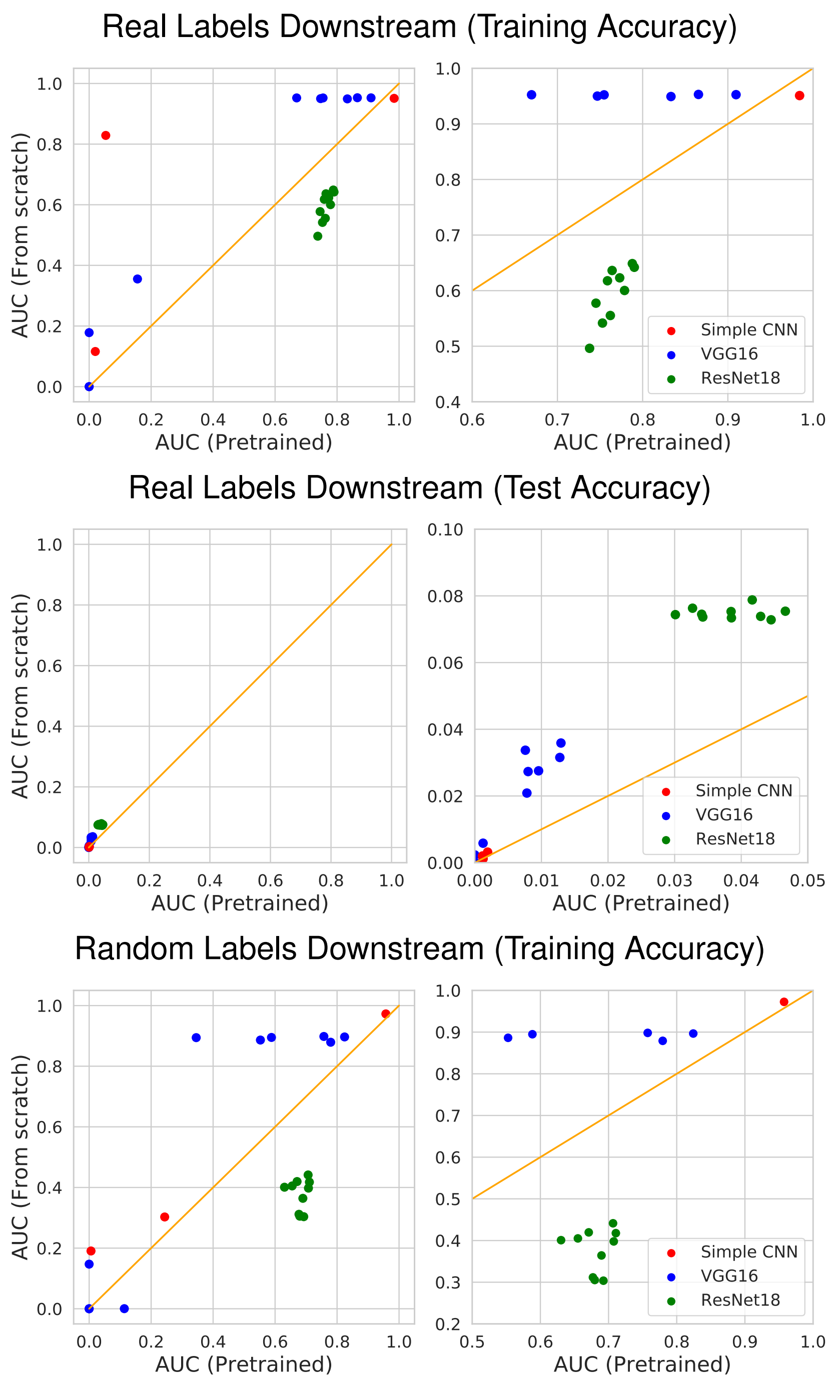


Figure 15: Scatter plots for ImageNet Experiments B. Models fine-tuned on real labels (TOP and CENTER) and on random labels (BOTTOM). Each dot corresponds to one *group of experiments*. Points below the orange $x = y$ line on TOP correspond to experiments where models pre-trained with random labels train faster downstream (with real labels) compared to models trained from scratch *with same hyperparameters*. RIGHT column contains zoomed in versions of the plots from LEFT column.

B.5 Summary of the experiments

Analysis of experiments presented in the previous four sections reveals a large number of positive transfers where the pre-trained model *trains faster* on the downstream task compared to the model trained from scratch using the *same hyperparameters*. Generally it looks like transferring more layers leads to stronger positive effects. All ResNet18 runs performed on ImageNet (and most of the ones on CIFAR10) result in positive transfer (in terms of the downstream training accuracy), which may be caused by the fact that we do not re-scale them after pre-training.

Most likely there are other important factors, yet to be discovered, playing a role in all these experiments. However, the evidence presented earlier (Figures 2 and 3, Table 1) suggests that alignment is responsible for the observed positive transfer in at least some of these cases.

B.6 Detailed results for some of the experiments

Figure 16 reports experiments with ResNet18 on ImageNet where pre-training helps. These experiments differ only in the in initialization scale and otherwise use:

```
learning_rate = 0.01
num_classes_upstream = 1000
num_examples_upstream = 500000
num_epochs_upstream = 100
num_examples_downstream = 500000
num_epochs_downstream = 100
```

Figure 17 reports experiments with VGG16 on ImageNet where pre-training both helps and hurts. It may speed up the training, or make it slower. Surprisingly, in some cases it may slightly improve the holdout test accuracy. These experiments use:

```
init_scale = 1.0
learning_rate = 0.01
num_classes_upstream = 1000
num_examples_upstream = 500000
num_epochs_upstream = 100
num_examples_downstream = 500000
num_epochs_downstream = 100
```

Figure 18 reports experiments with VGG16 on ImageNet where pre-training hurts both real and random label downstream tasks. These experiment are taken from the ImageNet Experiments A reported in Section B.3. These experiments use:

```
init_scale = 0.747
learning_rate = 0.008
num_classes_upstream = 1000
num_examples_upstream = 50000
num_epochs_upstream = 100
num_examples_downstream = 200000
num_epochs_downstream = 100
```

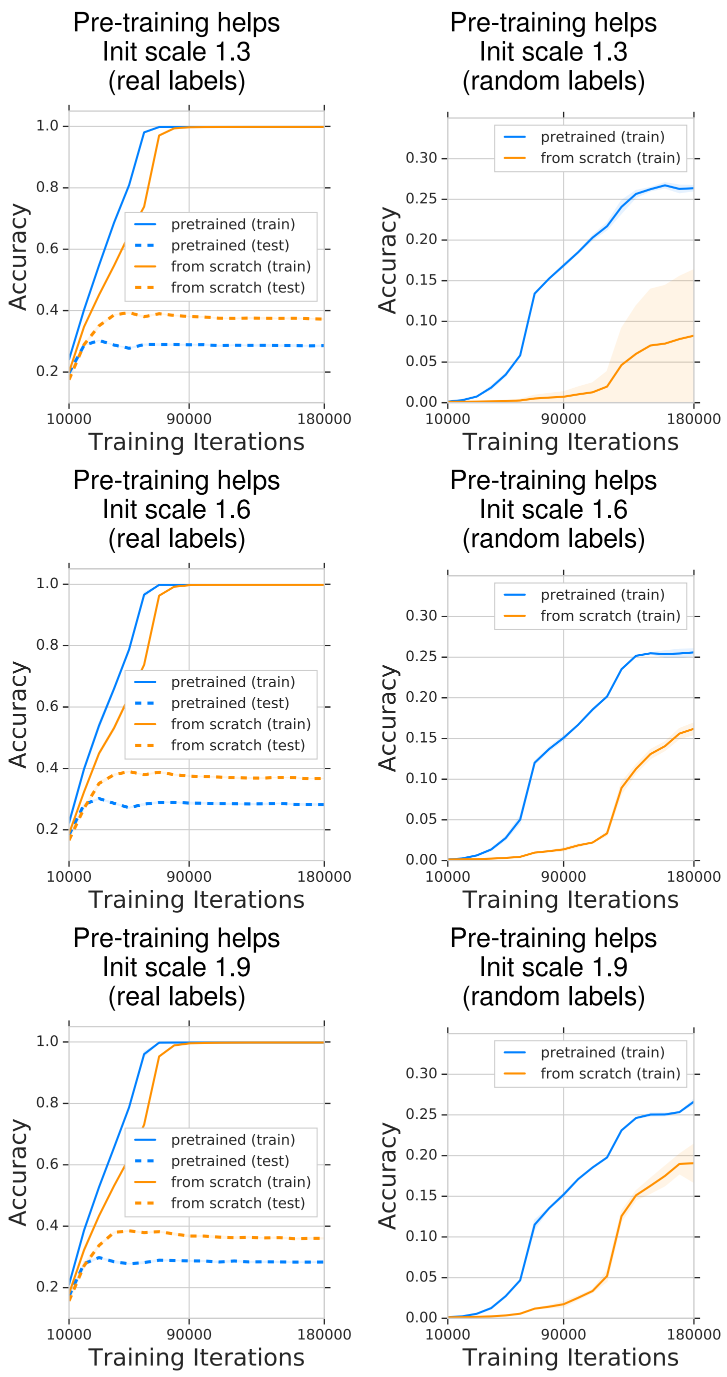


Figure 16: Pre-training on random labels often accelerates the downstream training compared to training from scratch with same hyperparameters. ResNet18 models are pre-trained on 500k ImageNet examples with 1000 random labels and subsequently fine-tuned on the fresh 500k ImageNet examples with either real labels or 1000 random labels using learning rate 0.01 and different initialization scales. Error bars correspond to ± 1 std. over 2 independent runs.

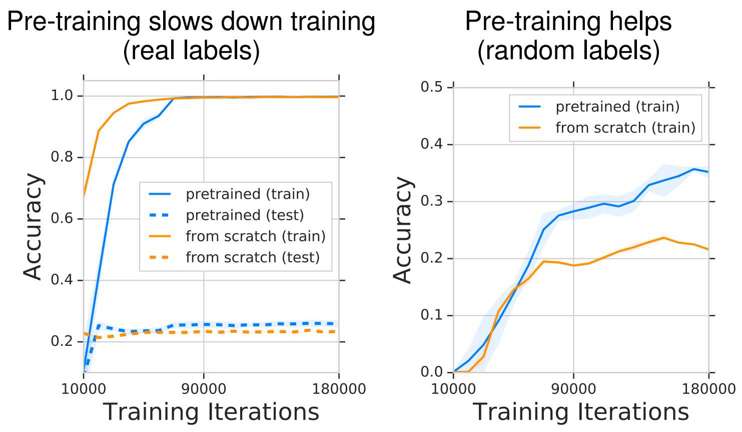


Figure 17: Pre-training on random labels may both accelerate or slow down the downstream training compared to training from scratch with same hyperparameters. VGG16 models are pre-trained on 500k ImageNet examples with 1000 random labels and subsequently fine-tuned on the fresh 500k ImageNet examples with either real labels or 1000 random labels. LEFT: Surprisingly, pre-training improves the holdout test accuracy. Error bars correspond to ± 1 std. based on 2 independent runs.

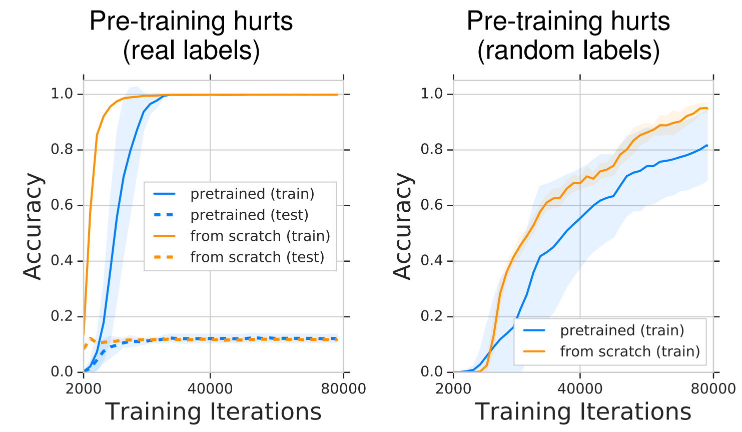


Figure 18: VGG16 models are pre-trained on 50k ImageNet examples with 1000 random labels and subsequently fine-tuned on 200k fresh ImageNet examples with real labels or 1000 random labels.

C Proof of Proposition 1

Proposition 1 makes the following assumptions:

1. The first layer is either
   - fully connected, and the input from $\mathbb{R}^d$ is normally distributed with mean $\mu_x = 0$ and covariance Σ_x, or
   - convolutional, with patches that are not overlapping and the data in each position is independent normally distributed with mean $\mu_x = 0$ and covariance Σ_x.
2. The first layer weights $w \in \mathbb{R}^d$ are initialized i.i.d. at random from some distribution on $\mathbb{R}^d$ that is invariant under the orthogonal group $O(d)$.
3. The sampled inputs are labelled randomly according to some distribution over the target set $\mathcal{Y} = \{1, 2, \dots, c\}$, independently of the input sample.

Let $\mathcal{G} := \{G \in O(d) \mid G^T \Sigma_x G = \Sigma_x\}$ be the group of orthogonal matrices that leave the distribution of the data x_i invariant. Proposition 1 follows from these two claims:

Claim 1. *A probability distribution $\mathcal{D}_w$ on $\mathbb{R}^d$ with mean μ_w and covariance matrix Σ_w that is invariant under $\mathcal{G}$ has $\mu_w = 0$ and Σ_w aligned with Σ_x.*

Claim 2. *The probability distribution of the weights w in the first layer after t iterations of training is invariant under $\mathcal{G}$ (for any $t = 0, 1, ...$).*

Proof of Claim 1. Since $-I \in \mathcal{G}$, we have $\mathbb{E}[w] = \mathbb{E}[-w]$, so we must have $\mathbb{E}[w] = 0$. For the second part, assume

$$\mathbb{R}^d = V_1 \oplus V_2 \oplus ... \oplus V_r$$

is the orthogonal decomposition of $\mathbb{R}^d$ into eigenspaces of Σ_x and $d_i := \dim(V_i)$ are the dimensions of its parts. Then

$$\mathcal{G} = O(d_1) \times O(d_2) \times ... \times O(d_r)$$

where each $O(d_i)$ operates in the canonical way on V_i and leaves the other parts invariant.

By assumption $\mathcal{G}$ leaves $\mathcal{D}_w$ invariant, so we must have $G^T \Sigma_w G = \Sigma_w$ for all $G \in \mathcal{G}$. Then each $G \in \mathcal{G}$ must also leave eigenspaces of Σ_w invariant since for an eigenvector u with $\Sigma_w u = \lambda u$, we have

$$\Sigma_w(Gu) = (\Sigma_w G)u = (G\Sigma_w)u = \lambda \cdot Gu.$$

We have to show that each V_i is contained in an eigenspace of Σ_w. Since the eigenvectors of Σ_w span $\mathbb{R}^d$, there must be an eigenvector u in an eigenspace U of Σ_w that is not orthogonal to V_i. We will show that $V_i \subseteq U$: Let $M_i \in \mathcal{G}$ be the matrix that is I on V_i and $-I$ on all other V_j, $j \neq i$, and let $pr_i : \mathbb{R}^d \to V_i$ be the orthogonal projection onto V_i. Then by assumption $pr_i(u) \neq 0$, but since $pr_i(u) = (u + M_i u)/2$, we must also have $pr_i(u) \in U$. Since $O(d_i)$ operates transitively on the set of all vectors $v \in V_i$ of a given length, and $\{I\} \times ... \times O(d_i) \times ... \times \{I\} \subseteq \mathcal{G}$, all vectors of length $|pr_i(u)|$ in V_i must as well be in U. Since U is closed under scalar multiplication and $pr_i(u) \neq 0$, this means that indeed $V_i \subseteq U$. □

Proof of Claim 2. Each run of the given network is determined by sampled initial data:

- The initial weights $w_1, ..., w_M \in W = \mathbb{R}^d$ in the first layer,
- the inputs $x_1, ..., x_N \in V = \mathbb{R}^d$,
- the initial weights in the later layers, and biases in all layers,
  we will write them as one large vector $w' \in W' = \mathbb{R}^{d'}$,
- the targets $y_1, ..., y_N \in \mathcal{Y}$.

We will consider the targets as fixed and write the other initial data as

$$(w_i, x_j, w') \in \mathcal{V} := W^M \times V^N \times W'.$$

The group $\mathcal{G} \subseteq O(d)$ operates on $V = \mathbb{R}^d$ and $W = \mathbb{R}^d$, we also define its operation on $\mathcal{V}$ by

$$G : \mathcal{V} \to \mathcal{V}, (w_i, x_j, w') \mapsto (Gw_i, Gx_j, w') \tag{3}$$

By our assumptions and the definition of $\mathcal{G}$, this operation leaves the distribution of initial data invariant for any $G \in \mathcal{G}$. This proves the "$t = 0$" case in Claim 2, it remains to show that this invariance is kept when we do one step of Gradient Descent, this will follow if we can show that the update commutes with the operation of $\mathcal{G}$.

At each step of the training the current state is given by a point in the vector space $\mathcal{V}$, the loss is a function $L : \mathcal{V} \to \mathbb{R}$ and one step of Gradient Descent is given by

$$w_i \mapsto w_i - \epsilon \cdot \nabla_{w_i} L(w_i, x_j, w'), \qquad w' \mapsto w''$$

for some $w'' \in W'$. We have to show that this update commutes with the operation of $\mathcal{G}$, i.e. for all $G \in \mathcal{G}$

$$Gw_i \mapsto Gw_i - \epsilon \cdot G\nabla_{w_i} L(w_i, x_j, w'), \qquad w' \mapsto w'' \tag{4}$$

for the same w''.

Since the first layer only gets information about the w_i and x_j via their scalar products $\langle w_i, x_j \rangle$, and for any $G \in O(d)$ we have $\langle Gw_i, Gx_j \rangle = \langle w_i, x_j \rangle$, the operation (3) of $G \in \mathcal{G}$ on $\mathcal{V}$ leaves the loss function $L : \mathcal{V} \to \mathbb{R}$ invariant. For the same reason also the updates of the biases and later weights w' are unaffected by the operation of $\mathcal{G}$. So we only have to prove the update equations for Gw_i, i.e. show that

$$\nabla_{w_i} L(Gw_i, Gx_j, w') = G\nabla_{w_i} L(w_i, x_j, w')$$

The gradients $\nabla_{w_i} L$ in the w_i–directions are part of the full gradient ∇L (which is a vector in $\mathcal{V}$, so it also contains the derivatives with respect to the points x_j and the weights w' of the later layers). Hence it is sufficient (or even stronger) if we can show

$$\nabla L(Gp) = G\nabla L(p) \qquad \text{for each point } p \in \mathcal{V} \text{ and } G \in \mathcal{G}. \tag{5}$$

Since L is invariant under G, also the differential form dL is invariant: $G^* dL = dL$.

The standard Euclidean metric on $\mathcal{V} = \mathbb{R}^{d\times M + d\times N + d''}$ provides a translation between differential forms and vector fields that determines ∇L from dL and vice versa by the condition

$$\langle \nabla L(p), v \rangle = dL|_p(v) \qquad \text{for all } \ v \in \mathcal{V}.$$

Since the Euclidean metric on $\mathcal{V}$ is invariant under $G \in \mathcal{G}$, we have for each vector $v \in \mathcal{V}$

$$\langle \nabla L(Gp), Gv \rangle = dL|_{Gp}(Gv) = dL|_p(v) = \langle \nabla L(p), v \rangle = \langle G\nabla L(p), Gv \rangle$$

Since $G \in \mathcal{G}$ is invertible, Gv can be any vector in $\mathcal{V}$, so (5) follows. □

Note that the same proof argument holds for many other optimization techniques, not only for gradient descent. For example, it does not matter whether we use Momentum, AdaGrad, Adam, Nesterov, weight decay, etc. in the gradient descent. However, the optimization needs to respect symmetries from $O(d)$, i.e. it cannot make use of the special coordinate system for the input. This excludes for example Exponentiated Gradient [28].

Similarly, the proof is also independent of the loss function used, if that loss only involves the output of the network: Since the proof only makes use of symmetries $\langle Gx, Gw \rangle = \langle x, w \rangle$ in the first layer, it is not affected by what we do in the later layers or at the output level. However, if the loss function includes regularization terms that involve the weights in the first layer, these terms also need to be invariant under the orthogonal group. This means the proof still applies if we use $L2$ regularization, but $L1$ regularization does make use of the special coordinate system and is not invariant under rotations, so the proof arguments would not hold anymore if we used it in the first layer.

D Measuring Alignment

D.1 How well can we measure eigenvectors?

We do not observe the covariance matrices directly, but estimate them from samples of the corresponding distributions. This estimate has a variance that results in a variance of the computed eigenvectors. As a first intuitive experiment we compare the eigenvectors obtained from two disjoint samples:

Data We compare the covariance matrices Σ_x and Σ'_x obtained from 5×5 patches of the first and second 30 000 images in CIFAR10. The first eigenvectors are extremely well aligned:

i	σ_i^2	$\lvert\langle e_i, e'_i\rangle\rvert$	i	σ_i^2	$\lvert\langle e_i, e'_i\rangle\rvert$
1	12.3	0.9999996	7	0.28	0.99985
2	1.45	0.99995	8	0.23	0.99988
3	1.18	0.99992	9	0.128	0.99010
4	1.06	0.99994	10	0.126	0.99030
5	0.37	0.99969	11	0.115	0.99969
6	0.32	0.99970	12	0.096	0.99999

Also the other eigenvectors are very well aligned, all scalar products obtained were above 0.975. So 30,000 images are sufficient to estimate the eigenvectors of Σ_x to a high accuracy.

Weights We use the same ResNet experiment as for Figure 3, described in Section A.7. We use two disjoint groups of 30 randomly initialized networks, trained in the same way on CIFAR10 with random labels.

i	σ_i^2	$\lvert\langle e_i, e'_i\rangle\rvert$	i	σ_i^2	$\lvert\langle e_i, e'_i\rangle\rvert$
1	0.281	0.974	7	0.085	0.906
2	0.202	0.958	8	0.078	0.946
3	0.154	0.916	9	0.077	0.924
4	0.141	0.934	10	0.072	0.811
5	0.107	0.861	11	0.063	0.598
6	0.104	0.861	12	0.059	0.645

So we can only measure the most important eigenvectors with reasonable accuracy. Of course it is not unexpected that we can determine the covariance of image patches much better than the covariance of weights: In the above example, we used $30000 \times 30 \times 30$ = 27 million image patches, but only $30 \times 64 = 1920$ weight vectors (filters). However, apart from the number of examples, also the type of information we want to extract from the covariance matrix determines the accuracy of our measurement.

In particular, when two eigenvalues are close together (like σ_5, σ_6 above), it may be difficult to determine the exact eigenvector. However, the 2-dim space spanned by both eigenvectors is relatively stable – in the above example the expansion of e_5, e_6 in terms of the basis e'_i is

$$\begin{aligned} e_5 &= 0.86e'_5 - 0.36e'_6 + \text{...(smaller terms)...} \\ e_6 &= 0.36e'_5 + 0.86e'_6 + \text{...(smaller terms)...} \end{aligned}$$

This is a general problem - if two eigenvalues are close to each other, the direction of eigenvectors is not measurable in practice, or if they are even the same, also theoretically individual eigenvectors are not defined, only the higher dimensional eigenspace is. We will investigate this problem of statistical sampling uncertainty closer in D.3.

Apart from the statistical sampling uncertainty we may also consider noise in the images themselves. However, given that the two computations of the eigenvectors agreed so well for the data covariance, we can expect that this error is small compared to the sampling error for the weight covariance.

D.2 Motivational definition of misalignment

Any attempt of defining "misalignment measures" naively by using the direction of eigenvectors (e.g. "misalignment = sum of angles between corresponding eigenvectors") runs into the problem mentioned above: It would be undefined when we have eigenspaces of dimension > 1 or at least discontinuous / not measurable / not informative in practice when two of the eigenvalues are close. So instead we will construct a "distance to an aligned matrix" that takes into account what can actually be measured.

In [22] a Riemannian metric is defined on the space of color perceptions. In this metric, the length of a path in color space is given by the minimal number of colors on this path connecting start and end such that each color is indistinguishable (by a human) from the next one.

We can use a similar construction for covariance matrices: Fix a (large) n, then we call two matrices indistinguishable, if after sampling n samples from $\mathcal{N}(0, \Sigma_k)$ we most of the time cannot reject the hypothesis that the correct covariance matrix was Σ_{k+1}. The minimal number of "indistinguishable" matrices connecting start to end of a curve is (in good approximation) proportional to $\sqrt{n}$, to get a path length independent of n, we divide the number by $\sqrt{n}$ and take the limit $n \to \infty$. The misalignment could now be defined as the minimal path length of a path between B and a matrix Σ that is aligned to A. (The exact definition would need to specify "indistinguishable", i.e. the fraction of times we can reject the hypothesis, the confidence level used in this rejection. The path lengths corresponding to different definitions would differ by a multiplicative constant.)

The Riemannian metric defined in this way is given locally by the Fisher Information, which can be easily computed in this case. However, it seems there is not a simple known formula for the resulting (global) distance between two points. To simplify computations and proofs, will use instead the upper bound to the square length given by the symmetrized Kullback–Leibler divergence. Like the square length of the shortest path between A and B it can be expressed as an integral over the Fisher Information, but the path is not the usual shortest path (geodesic) that is traversed with constant speed, but the straight line $(1-t) \cdot A + t \cdot B$. For small distances this is a good approximation, but it will give a larger value in general.

D.3 Fisher information and Riemannian manifold of symmetric positive definite matrices

As a concrete toy example for the problem of close eigenvalues observed above, we take the 2-dimensional normal distribution with $\mu = 0$ and Σ the diagonal matrix with entries $(1, \lambda)$. We generate $n = 100$ i.i.d. samples $\vec{w}_1, ..., \vec{w}_n$ of this distribution, measure their empirical covariance $\hat{\Sigma}_w$, and plot the direction of their eigenspaces. Repeating this 50 times, we can visualize the distribution of these directions (Figure 19, first row).

While we get a reasonable approximation to the direction of the eigenspaces of Σ (i.e. the coordinate axes) for large λ, the eigenvector directions of $\hat{\Sigma}$ become less well-aligned as $\lambda = \sigma_2^2$ gets close to $1 = \sigma_1^2$. So if we were to measure "whether the diagonal matrix $diag(1, \lambda)$ is aligned with itself" by computing two empirical covariance matrices (from 2 independent sets of sampled data), their eigenvectors, and angles between the two empirical results, we would think that these two empirical covariances are not aligned for $\lambda \approx 1$, although they actually come from the same distribution.

Mathematically, the problem is that any "angles between eigenvectors" measure would be undefined for matrices with an eigenspace of dimension > 1. We could fix that e.g. by using the choice of eigenvectors that gives the smallest possible result, but then the resulting function would not be continuous around such matrices, i.e. we may need estimates with infinite precision to get the alignment measure with a fixed precision.

On the other hand, for any fixed $\sigma_1^2 \neq \sigma_2^2$ we can in theory make n large enough to get the sampling error as small as we want, e.g. in the above example we can go from $n = 100$ to $n = 100\,000$ (Figure 19).

We can say that for $\lambda_1 \approx \lambda_2$ we have less information per sample from the normal distribution $\mathcal{N}(0, \Sigma)$ about the rotation angle α than for $\lambda_1 \gg \lambda_2$. This is formalized by the Fisher information, we now summarize some of its properties for the case we are interested in.

We denote by M the manifold of symmetric positive definite $d \times d$ matrices, it is an open cone in the $D := \frac{d(d+1)}{2}$ dimensional vector space of symmetric matrices. We identify the points $\Sigma \in M$ also with their corresponding normal probability distributions $\mathcal{N}(0, \Sigma)$. Given n samples $x_1, ..., x_n \in \mathbb{R}^d$, the empirical covariance matrix $\frac{1}{n} \sum_{i=1}^{n} x \cdot x^T \in M$ is the maximum likelihood estimator. Given some local coordinates $\theta_1, ..., \theta_D$ around a matrix Σ, the Fisher information of θ_k at Σ is defined as

$$I(\theta_k) := \mathbb{E}_{x \sim \mathcal{N}(0,\Sigma)} \left[\left(\frac{\partial \log p(x|\theta_k)}{\partial \theta_k} \right)^2 \right] \qquad (6)$$

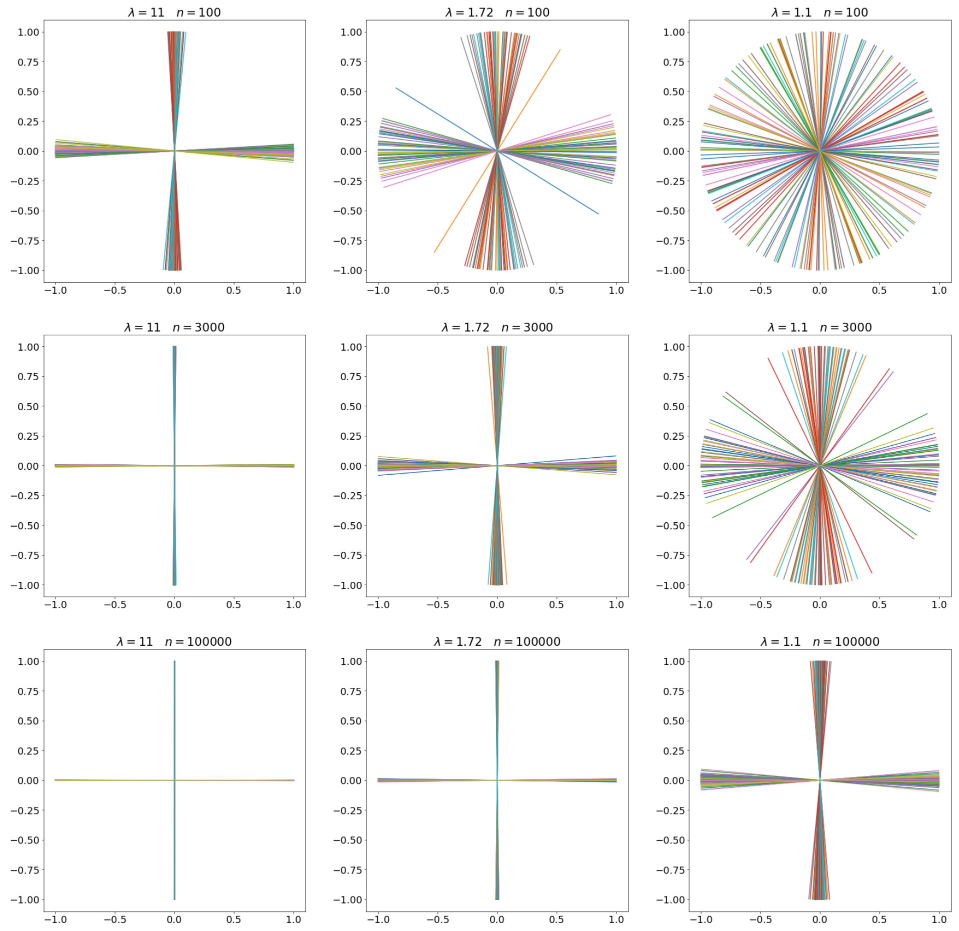


Figure 19: Visualization of the variation of estimating the directions of the eigenspaces for 2-dimensional normal distribution with $\mu = 0$ and Σ the diagonal matrix with entries $(1, \lambda)$ for three different values of λ and varying number of samples n.

We are interested in the variance of the maximum likelihood estimate $\hat{\theta}_k$ when we estimate the parameter θ_k from n samples $x_1, .., x_n$, it can be approximated as

$$Var(\hat{\theta}_k) \approx \frac{1}{n \cdot I(\theta_k)},$$

and this approximation becomes exact as $n \to \infty$ (see e.g. [49], chapter 9.7. for a precise statement). In our toy example we are interested in the rotation angle α, and local coordinates would be $\alpha, \lambda_1, \lambda_2$ (we assume for the moment that $\lambda_1 \neq \lambda_2$ at Σ). In this case one can compute the Fisher information about α as

$$I(\alpha) = \frac{(\lambda_1 - \lambda_2)^2}{\lambda_1 \cdot \lambda_2}$$

(see e.g. [12]).

In the Figure 19, the images on the diagonal look similar, and indeed we have chosen λ such that the Fisher Information about α of n samples is approximately the same: In each case $n \cdot (\lambda - 1)^2/(\lambda \cdot 1)$ is between 904.1 and 909.1, which gives a standard deviation of about 1/30 (corresponding to about 2 degrees) for the rotation angle α, and indeed generating 100 directions from a normal distribution of α with a standard deviation of 1/30 matches the above plots on the diagonal.

This suggests that

$$\delta = \alpha \cdot \sqrt{I(\alpha)}$$

is an appropriate distance measure between matrices that differ by a rotation of angle α: When two matrices A, B have "distance" δ, this means that for large n we can put about $k = \lfloor \delta \cdot \sqrt{n} \rfloor$

points $\Sigma_1, \Sigma_2, ..., \Sigma_k$ between $A = \Sigma_0$ and $B = \Sigma_{k+1}$ such that each pair Σ_i and Σ_{i+1} are within measurement error of each other when our measurement consists of estimating the covariance from a sample of n points.

So far, we have defined the Fisher information in a particular coordinate system on the manifold M. However, the definition (6) depends only on the tangent vector $\partial/\partial\theta_k$. So the Fisher information is actually assigned to tangent vectors of the manifold, is independent of the coordinate system, and it turns out to satisfy the properties of a Riemannian metric, so it gives for any tangent vector $v \in T_\Sigma(M)$ a "length" $\sqrt{I(v)}$. This is a special case of the general theory of Information Geometry, see e.g. [2, 5].

D.4 Defining misalignment measures

While the Fisher information gives a local distance measure (i.e. a length of tangent vectors), we rather need a global distance measure $D(\Sigma_1, \Sigma_2)$ between points. For that we integrate the Fisher Information along the straight line

$$\gamma : t \mapsto (1-t) \cdot \Sigma_1 + t \cdot \Sigma_2 \qquad \text{for } t \in [0,1]. \tag{7}$$

If instead of (7) we had used the real shortest path (geodesic) of the Riemannian metric, which goes in constant speed from $t = 0$ to $t = 1$, this would give the square length of the shortest path. Since in general (7) is not the shortest path, and the parameterization does not have constant derivative with respect to the standard connection, this can give slightly larger results. Of course, for small distances it still gives the same result in first order.

In terms of information geometry, the straight line is the e–geodesic, and the integration of the Fisher information against the e–geodesic gives the same result as integrating along the m–geodesic: The symmetrized Kullback–Leibler divergence between p and q (see e.g. Theorem 3.2. of [2], also compare section 4.4.2. in [5]). The symmetrized Kullback–Leibler divergence exists in two normalizations: With or without the factor $1/2$. We are using the version with $1/2$

$$D(\Sigma_1, \Sigma_2) := \frac{D_{KL}\Big(\mathcal{N}(0,\Sigma_1) \,||\, \mathcal{N}(0,\Sigma_2)\Big) \;+\; D_{KL}\Big(\mathcal{N}(0,\Sigma_2) \,||\, \mathcal{N}(0,\Sigma_1)\Big)}{2}$$

which is equal to half of the integral of the Fisher information:

$$D(\Sigma_1, \Sigma_2) = \frac{1}{2}\int_0^1 I\left(\frac{\partial\gamma}{\partial t}(t)\right) dt$$

So we will use this distance measure as a basis for our (mis)alignment measure. It can be expressed analytically as

$$D(\Sigma_1, \Sigma_2) = \frac{\mathbf{tr}(\Sigma_1^{-1}\Sigma_2 + \Sigma_2^{-1}\Sigma_1)}{2} - d$$

While it would be possible to use different distance measures that also have the Fisher information as the infinitesimal version (e.g. the usual, asymmetric Kullback–Leibler divergence), this definition has the additional benefit that it is invariant under scaling: $D(\lambda\Sigma_x, \lambda\Sigma_w) = D(\Sigma_x, \Sigma_w)$, and this allows the simple definition of the alignment measure given in Section 2 (for more general distance measures one would need to restrict the Σ to a subset e.g. by requiring $\mathbf{tr}(\Sigma) = 1$ and write $\inf_{\Sigma,\lambda>0} D(\Sigma, \lambda \cdot \Sigma_w)$ to get a result > 0 when the matrices are not aligned).

So we define the "misalignment" score for two positive definite symmetric matrices A, B as

$$M(A,B) := \inf_{\Sigma \succ 0 \text{ aligned with } A} D(\Sigma, B) = \inf_{\Sigma \succ 0 \text{ aligned with } A} \left\{ \frac{\mathbf{tr}(\Sigma^{-1}B + B^{-1}\Sigma)}{2} - d \right\},$$

which was our definition in Section 2.2.

Proposition 2. *This misalignment measure has the following properties:*
For all positive definite symmetric matrices A, B

1. $M(A,B) \geq 0$

2. *$M(A, B) = 0$ if and only if B is aligned with A.*

3. *$M(A, B)$ is continuous in B.*

4. *Equivariance under orthogonal group: $M(UAU^T, UBU^T) = M(A, B)$ for $U \in O(d)$*

5. *Invariance under scalar multiples of B: $M(A, \lambda B) = M(A, B)$ for $\lambda > 0$*

6. *$M(A, B)$ only depends on the eigenspaces of A.*

7. *$M(A, B) + d = \sum_{i=1}^{r} \sqrt{\textbf{\textit{tr}}(B|_{V_i}) \cdot \textbf{\textit{tr}}(B^{-1}|_{V_i})}$, where $V_1 \oplus ... \oplus V_r$ is the orthogonal decomposition of $\mathbb{R}^d$ into eigenspaces of A, and $B|_{V_i}$ is the linear map $V_i \to V_i, \boldsymbol{v} \mapsto pr_i(B(\boldsymbol{v}))$ with pr_i the orthogonal projection $\mathbb{R}^d \to V_i$.*

The function $M(A, B)$ is not continuous in A, and there cannot be a function $M(A, B)$ that is continuous in both arguments and still satisfies condition 2.

Proof. 1, 4, 5, 6, and the "if" part of 2 follow directly from the definitions.
3 follows from 7. We will see in the proof of 7 that the infimum is obtained for a particular matrix Σ, and from that also the "only if" part of 2 follows.
Formula 7: We use the orthogonal decomposition $V_1 \oplus ... \oplus V_r$ of $\mathbb{R}^d$ into eigenspaces of A. A positive definite symmetric matrix Σ aligned with A is given by its eigenvalues λ_i on the subspaces V_i. For a linear map $f : \mathbb{R}^d \to \mathbb{R}^d$ we have

$$\textbf{tr}(f) = \sum_{i=1}^{r} \textbf{tr}(f|_{V_i})$$

So the definition of $M(A, B)$ can be rewritten

$$\begin{aligned} M(A,B) + d &= \frac{1}{2} \cdot \inf_{\lambda_1,\ldots,\lambda_r>0} \sum_{i=1}^{r} \textbf{tr}\left(\lambda_i^{-1}B + \lambda_i B^{-1}\right)|_{V_i} \\ &= \frac{1}{2} \cdot \sum_{i=1}^{r} \inf_{\lambda>0} \lambda^{-1}\textbf{tr}(B|_{V_i}) + \lambda\textbf{tr}(B^{-1}|_{V_i}) \end{aligned}$$

Since B is positive definite, B^{-1} is positive definite as well. When B is positive definite, then also $B|_V$ is positive definite, so we have $\textbf{tr}(B|_{V_i}) > 0$ and $\textbf{tr}(B^{-1}|_{V_i}) > 0$. Thus for $b := \textbf{tr}(B|_{V_i})$ and $c := \textbf{tr}(B^{-1}|_{V_i})$ the function $\lambda^{-1}b + \lambda c$ has a minimum for a finite positive λ, and it is

$$\arg\min_{\lambda>0} \lambda^{-1}b + \lambda c = \sqrt{b/c}$$

as is seen e.g. by comparing the derivative with zero, and hence

$$\inf_{\lambda>0} \lambda^{-1}b + \lambda c = \min_{\lambda>0} \lambda^{-1}b + \lambda c = 2 \cdot \sqrt{b \cdot c},$$

which gives the above formula 7.

$M(A, B)$ cannot be continuous in A if condition 2 should hold: Take B the diagonal matrix with entries 1 and 2, and for A consider the diagonal matrices with entries 1 and λ. Then for λ = 1 the matrix B is not aligned with A, but it is for all $\lambda \neq 1$. Therefore we must have $M(diag(1, 1), B) > 0$, but $M(diag(1, \lambda), B) = 0$ for all $\lambda \neq 0$, so A would not be continuous with respect to A. $\square$

E The shape of $f(\sigma)$

E.1 Gaussian centered input

In the "ideal" case of Gaussian centered input, we experimentally observe curves that "look continuous", see Figure 20. For this figure, we used different settings to obtain curves that go only down (left), rise and then fall (middle) or only go up (right). We always used mean 0 and a diagonal matrix for the covariance (which is no restriction of generality, since applying an orthogonal matrix to the input does not change the dynamic of the neural networks, and any symmetric positive definite matrix can be written as $O^T LO$ with diagonal matrix L and orthogonal matrix O).

The σ that we used were: $\{1, 1.1, 1.2, ...1.9\}$ for the plot on the left (which used input dimension 10), and $\{0.1, 0.2, ..., 3.0\}$ for the plots in the middle and on the right (which used input dimension 30). For the images on the left and middle the network had two fully connected layers with 2048 and 256 units, and was trained on 10000 inputs, for the right image we used a network with two layers with 256 units each and it was trained on 2000 inputs. The targets were always random labels of 10 classes.

We can give a heuristic argument for why the curves should "look continuous": If two eigenvalues of the data covariance Σ_x are close, exchanging them gives an input distribution that is close to the original distribution. Because this exchange is an orthogonal transformation, we will get trained networks in which the weights also only differ by the same orthogonal transformation. This means that the effect of exchanging the sigmas will also exchange the taus, and if exchanging the sigmas had a small effect on the input distribution, we may expect also a small effect on the weights distribution, which means we expect the corresponding taus also to be close.

For the other experimental observation, that the curves first rise and then fall again (where one of these two parts can also be missing, as above), we already sketched a heuristic argument in 2.3.

E.2 Convolutional networks on natural images

While the $f(\sigma)$ curves in the ideal centered Gaussian case look smooth, the real-world $f(\sigma)$ curves seem to contain perturbations, although they still roughly have a rising/falling shape as well. To investigate what the largest contributor to this perturbations is, we will go from the ideal case to the real case in a series of steps.

As a first step, we crop the images of CIFAR10 to 27×27 pixels and approximate their distribution by the normal distribution with mean 0 and the covariance of all 3×3 patches. When we apply a convolutional network with 3×3 convolutions and stride 3, we are in the "ideal" situation of E.1 and expect a $f(\sigma)$ curve that goes up and then down. This is indeed the case, as shown by the green curve in Figure 21. Using the same distribution / covariance for all patch positions is of course a simplification: For example, for the patches at the upper boundary it is more likely to see light blue (from the sky) than at the lower boundary. To get closer to reality, we can replace the one global covariance by the covariances corresponding to the possible patch positions, yielding the red curve in Figure 21.

The next step is to abandon Gaussian approximation, and take the real patches. In the 27×27 pixel images we have 9×9 patches of 3×3 pixels each. To destroy the correlations between neighboring patches, we permute for each of the 9×9 positions the patches in that position of all images. So for

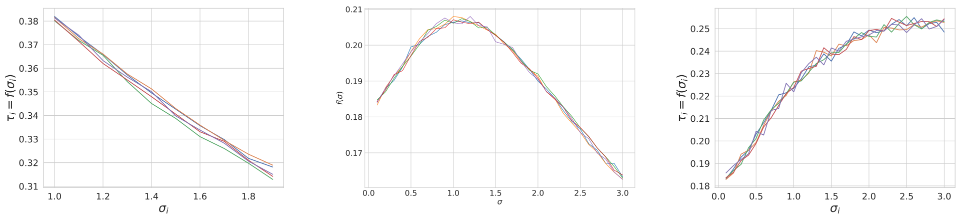


Figure 20: $f(\sigma)$–curves for fully connected networks with two hidden layers and Gaussian centered inputs. See text for details.

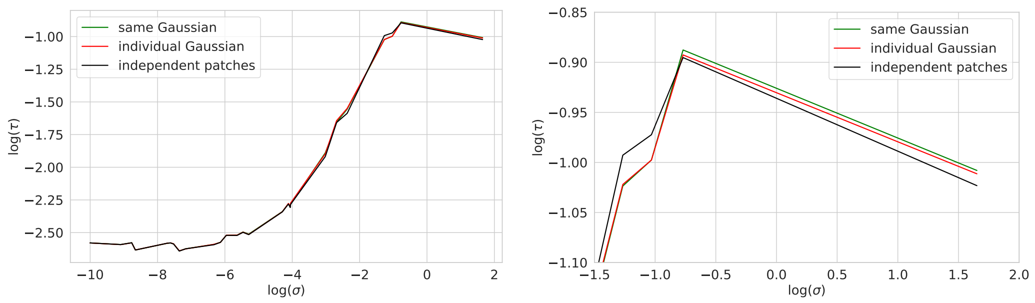
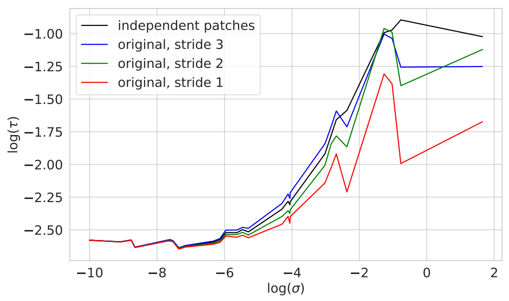


Figure 21: $f(\sigma)$ curve for training on random labels with different approximations of CIFAR10 images, cropped to 27×27 pixels. Two 3×3-convolutional layers with stride 3, one fully connected layer. RIGHT: Zoomed in to the upper right corner to show the small differences.

Figure 22: $f(\sigma)$ curve for training on random labels with different approximations of CIFAR10 images. Two 3×3-convolutional layers, one fully connected layer. Correlations between neighboring patches create deviation from the curve seen for the "ideal" case of independent Gaussian approximations. Smaller strides lead to stronger correlations and stronger deviation.



each position we still have the same set of 3×3 patches that can appear in an image, but the patches appearing in one new image no longer fit together since they (almost always) came from different original images. This leads to images that are stitched together from random patches and also means that previously (potentially similar) patches from the same original image can now occur with two different random labels. This setting yields the black curve in Figure 21. So far it seems we are still very close to the ideal situation.

The next step is to take the original images; now the correlations between neighboring patches do create a more significant change in the $f(\sigma)$ curve, see the blue curve in Figure 22. In particular, we see the first significant deviation from "up and then down". This effect becomes stronger when we go from stride 3 to overlapping inputs of the convolution (stride 2 and stride 1, green and red curves in Figure 22).

E.3 Deeper Layers

We use the following simple CNN on the CIFAR10 data set here (as in Section 2.5, Table 1):

```
conv 3x3, 16 filters
conv 3x3, 16 filters
conv 3x3, 16 filters
maxpool 3x3, stride 2
dense layer, 512 units
dense layer, num_outputs units (classifier head)
```

Training is done using 30k images for pre-training on random labels (for comparison run), 20k images for training on real labels, 10k images for determining the test error, and using a learning rate of 0.002.

Sampling from covariance vs. picking eigenvectors In Figure 5 we compared a pre-trained convolutional layer with a layer consisting of filters randomly sampled from the same covariance matrix and found that this preserved the performance benefit exactly. In the experiment for Table 1 we instead used the most significant eigenvectors $e_1, ..., e_{16}$ with factors $\tau_1, ..., \tau_{16}$ directly as filters. This is a (more stable) approximation to sampling from the covariance matrix that has τ_i^2 as eigenvalue for

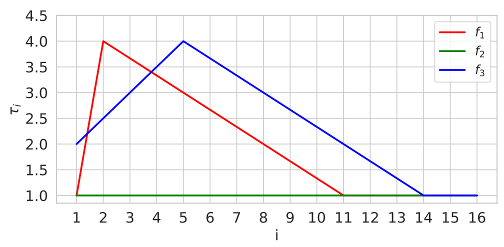


Figure 23: Sequence of $\tau_1, ..., \tau_{16}$ used to weight the 16 most significant eigenvectors in our experiments. The sequence f_1 was used for Table 1 in section 2.5, f_2 and f_3 were used for Table 2 and Table 3.

$e_1, ..., e_{16}$ and $\tau = 0$ for the other eigenvectors. Experimentally, we can see that this does not seem to make a significant difference: Compare Table 2 with Table 4 for the case that $\tau_1 = ... = \tau_{16} = 1$.

Choice of eigenvectors $e_1, ..., e_{16}$ and $\tau_1, ..., \tau_{16}$ In our previous experiments we observed curves for $f(\sigma)$ that were "rising and then falling", favoring the most significant eigenvalues, but somewhat downweighting the most significant one(s) (e.g. the center and right plot in Figure 4 – this was a different network on the same data). So we picked the 16 largest σ to be the ones with $f(\sigma) > 0$ (i.e. used the most significant eigenvectors), and used a made-up curve (f_1 in Figure 23) for the $\tau_1, ..., \tau_{16}$ which downweights the largest eigenvalue. We can see that the choice of this curve does not make a big difference: We can also just set $\tau_1 = ... = \tau_{16} = 1$ (f_2, resulting in Table 2), or choose another curve which downweights the largest eigenvalue less (f_3, resulting in Table 3). The stronger downweighting of the most significant eigenvalue seems to give a slight advantage when only initializing the first layer, but when initializing two or three layers, all curves seem to lead to essentially the same performance. However, compared to adjusting some values $f(\sigma)$ between 1 and 4, the much more important choice is which $f(\sigma)$ we set to 0, i.e. which eigenvectors we use. Tables 5 and 6 show the effect of choosing other eigenvectors, which leads to a significant drop in performance (for the choices in these tables even below the standard initialization).

Table 2: Using $f_2 = 1$: The 16 eigenvectors with largest eigenvalues are equally weighted. This corresponds to a $f(\sigma)$ which is 1 for the 16 largest σs and 0 for the other (11 or 128) σs. We achieve significant gains compared to the standard initialization.

		Convolutional layers sampled			
Iterations	Data	$\{\}$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$
100	Train	0.31	0.29	0.37	0.44
	Test	0.31	0.29	0.37	0.43
1000	Train	0.58	0.59	0.65	0.68
	Test	0.53	0.55	0.56	0.59

Table 3: Using f_3: Changing the $f(\sigma)$ between 1 and 4 does not affect the performance significantly, as long as we keep the same set of σ with $f(\sigma) > 0$.

		Convolutional layers sampled			
Iterations	Data	$\{\}$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$
100	Train	0.31	0.30	0.37	0.44
	Test	0.31	0.29	0.37	0.42
1000	Train	0.58	0.61	0.64	0.72
	Test	0.53	0.55	0.58	0.56

Table 4: Sampling from a covariance matrix using $f(\sigma) = 1$ for the highest 16 eigenvalues and $f(\sigma) = 0$ else. This gives essentially the same performance as using the 16 most significant eigenvectors directly as filters.

		Convolutional layers sampled			
Iterations	Data	$\{\}$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$
100	Train	0.31	0.29	0.37	0.41
	Test	0.31	0.29	0.36	0.40
1000	Train	0.58	0.57	0.62	0.69
	Test	0.53	0.54	0.54	0.56

Table 5: Using the 16 eigenvectors with the smallest eigenvalues gives significantly worse performance than random initialization. The 16 chosen eigenvectors were given the same weight.

		Convolutional layers sampled			
Iterations	Data	$\{\}$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$
100	Train	0.31	0.15	0.14	0.15
	Test	0.31	0.15	0.14	0.15
1000	Train	0.58	0.50	0.48	0.47
	Test	0.53	0.46	0.47	0.44

Table 6: Using eigenvectors $e_4, ...e_{19}$ (out of 27) in the first layer, $e_{10}, ..., e_{25}$ (out of 144) in the second and third layer. The 16 chosen eigenvectors were given the same weight.

		Convolutional layers sampled			
Iterations	Data	$\{\}$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$
100	Train	0.31	0.17	0.20	0.27
	Test	0.31	0.18	0.20	0.26
1000	Train	0.58	0.56	0.60	0.62
	Test	0.53	0.53	0.56	0.59

Table 7: Base line: These are the accuracies we obtain by pre-training on random labels. In particular when we use two or three convolutional layers, the results are very similar to what we get with our method that only uses the data, no training. (Compare e.g. with Table 2)

		Convolutional layers sampled			
Iterations	Data	$\{\}$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$
100	Train	0.31	0.37	0.39	0.41
	Test	0.31	0.34	0.37	0.39
1000	Train	0.58	0.70	0.75	0.78
	Test	0.53	0.51	0.53	0.57

F Specialization at the later layers

Earlier, we claimed that neural activations at the outer layer can drop abruptly and permanently after switching to the downstream task. Figure 24 illustrates this. In this figure, the x-axis corresponds to the number of training iterations, where the transfer to the downstream tasks happens at the middle. Note that in both network architectures shown in the figure, neural activation in the upper layers drops abruptly and permanently after switching to the downstream tasks. We interpret this to effectively reduce the available capacity for the downstream task, which masks the positive transfer happening from the alignment effect at the lower layers. This effect can be mitigated by increasing the width, as discussed in the context of Figure 8 in the main text.

Regarding specialization, Figures 10 and 11 are the extended versions of Figure 8. They include activation plots for *all intermediate layers* of VGG16 models (i) at initialization, (ii) in the end of the pre-training, (iii) in the end of fine-tuning on real labels, (iv) in the end of fine-tuning on 10 random labels. Figures 10 and 11 illustrate the negative and positive transfer examples respectively. See Section A.11 for details about the parameters used when generating those figures. As shown in those figures, neurons at the upper layers tend to *specialize*, i.e. become activated by fewer images. This is evident if we compare the fraction of examples activating neurons when pretrained for 1 training iteration (TOP-LEFT) vs. pre-trained for 6240 training iterations (TOP-RIGHT).

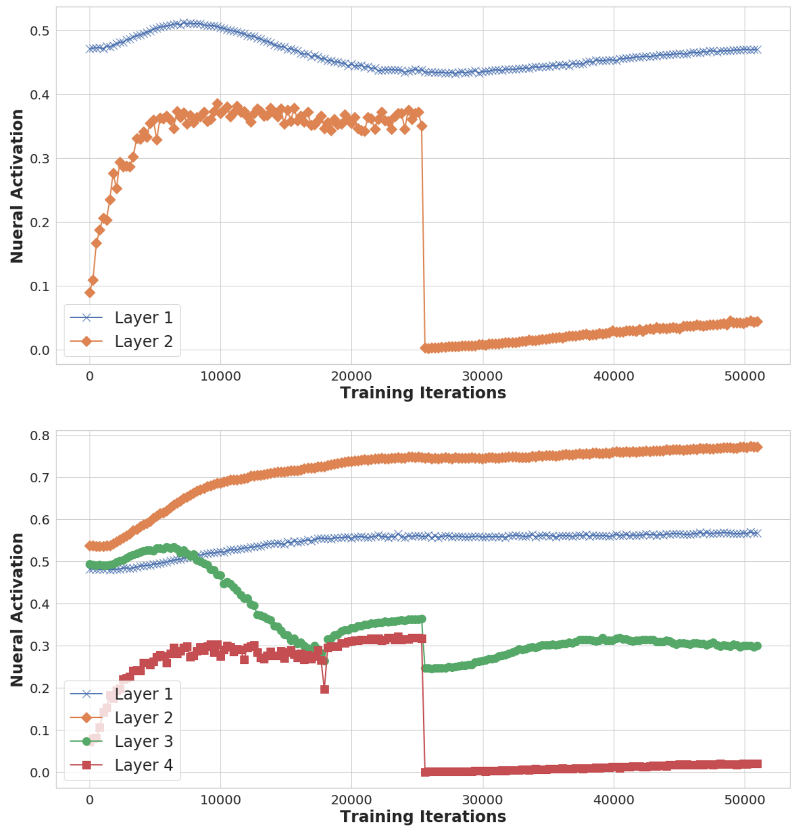


Figure 24: TOP: Neural activation is plotted against the number of training iterations in a two-layer CNN (256 3×3 filters followed by a dense layer of width 64). The y-axis is the frequency of the output of a hidden activation function being non-zero measured over a hold-out dataset. The abrupt drop in neural activation coincides with the switch to the downstream task. BOTTOM: A similar plot for a deeper neural network with three convolutional layers (256 3×3 filters) followed by a single dense layer of width 64.