A. Code base

In this section we describe our implementation and highlight the technical details that allow its generality for use in any architecture. We used TensorFlow version 1.13.1 to conduct all experiments, and adhered to its interface. All code can be found at https://github.com/neuroailab/neural-alignment.

A.1. Layers

The essential idea of our code base is that by implementing custom layers that match the TensorFlow API, but use custom operations for matrix multiplication (`matmul`) and two-dimensional convolutions (`conv2d`), then we can efficiently implement arbitrary feedforward networks using any credit assignment strategies with untied forward and backward weights. Our custom `matmul` and `conv2d` operations take in a forward and backward kernel. They use the forward kernel for the forward pass, but use the backward kernel when propagating the gradient. To implement this, we leverage the `@tf.custom_gradient` decorator, which allows us to explicitly define the forward and backward passes for that op. Our `Layer` objects implement custom dense and convolutional layers which use the custom operations described above. Both layers take in the same arguments as the native TensorFlow layers and an additional argument for the learning rule.

A.2. Alignments

A learning rule is defined by the form of the layer-wise regularization $\mathcal{R}$ added to the model at each layer. The custom layers take an instance of an alignment class which when called will define its alignment specific regularization and add it to the computational graph.

The learning rule are specializations of a parent `Alignment` object which implements a `__call__` method that creates the regularization function. The regularization function uses tensors that prevent the gradients from flowing to previous layers via `tf.stop_gradient`, keeping the alignment loss localized to a single layer. Implementation of the `__call__` method is delegated to subclasses, such that they can define their alignment specific regularization as a weighted sum of primitives, each of which is defined as a function.

A.3. Optimizers

The total network loss is defined as the sum of the global cost function $\mathcal{J}$ and the local alignment regularization $\mathcal{R}$. The optimizer class provides a framework for specifying how to optimize each part of the total network loss as a function of the global step.

In the `Optimizers` folder you will find two important files:

- `rate_scheduler.py` defines a scheduler which is a function of the global step, that allows you to adapt the components of the alignment weighting based on where it is in training. If you do not pass in a scheduling function, it will by default return a constant rate.

- `optimizers.py` provides a class which takes in a list of optimizers, as well as a list of losses to optimize. Each loss element is optimized with the corresponding optimizer at each step in training, allowing you to have potentially different learning rate schedules for different components of the loss. Minibatching is also supported.

B. Experimental Details

In what follows we describe the metaparameters we used to run each of the experiments reported above. For conciseness, we will only provide information on metaparameters that deviate from the defaults. Please refer to our source code for metaparameter defaults. Any defaults from TensorFlow correspond to those in version 1.13.1.

B.1. TPE search spaces

We detail the search spaces for each of the searches performed in §4. For each search, we trained approximately 60 distinct settings at a time using the HyperOpt package (Bergstra et al., 2011) using the ResNet-18 architecture and L2 weight decay of $\lambda = 10^{-4}$ (He et al., 2016) for 45 epochs, corresponding to the point in training midway between the first and second learning rate drops. Each model was trained on its own Tensor Processing Unit (TPUv2-8 and TPUv3-8).

We employed a form of Bayesian optimization, a Tree-structured Parzen Estimator (TPE), to search the space of continuous and categorical metaparameters (Bergstra et al., 2011). This algorithm constructs a generative model of $P[score \mid configuration]$ by updating a prior from a maintained history H of metaparameter configuration-loss pairs. The fitness function that is optimized over models is the expected improvement, where a given configuration c is meant to optimize $EI(c) = \int_{x<t} P[x \mid c, H]$. This choice of Bayesian optimization algorithm models $P[c \mid x]$ via a Gaussian mixture, and restricts us to tree-structured configuration spaces.

B.1.1. $\mathcal{R}^{\text{TPE}}_{\text{WM}}$ SEARCH SPACE

Below is a description of the metaparameters and their ranges for the search that gave rise to $\mathcal{R}^{\text{TPE}}_{\text{WM}}$ in Table 3.

- Gaussian input noise standard deviation $\sigma \in [10^{-10}, 1]$ used in the backward pass, sampled uniformly. $\mathcal{R}^{\text{TPE}}_{\text{WM}}$ **ended up setting** $\sigma = \mathbf{0.690465059717068}$**.**

- Ratio between the weighting of $\mathcal{P}^{\text{amp}}$ and $\mathcal{P}^{\text{decay}}$ given by $\alpha/\beta \in [0.1, 200]$, sampled uniformly. $\mathcal{R}^{\text{TPE}}_{\text{WM}}$ **ended up setting** $\alpha/\beta = \mathbf{15.6606919586}$**.**

- The weighting of $\mathcal{P}^{\text{decay}}$ given by $\beta \in [10^{-11}, 10^{7}]$, sampled log-uniformly. $\mathcal{R}^{\text{TPE}}_{\text{WM}}$ **ended up setting** $\beta = \mathbf{0.028327320537228435}$**.**

We fix all other metaparameters as prescribed by Akrout et al. (2019), namely batch centering the backward path inputs and forward path outputs in the backward pass, as well as applying a ReLU activation function and bias to the forward path but not to the backward path in the backward pass. To keep the learning rule fully local, we do not allow for any transport during the mirroring phase of the batch normalization mean and standard deviation as Akrout et al. (2019) allow.

B.1.2. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$ SEARCH SPACE

Below is a description of the metaparameters and their ranges for the search that gave rise to $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$ in Table 3.

- Training batch size $|\mathcal{B}| \in \{256, 1024, 2048, 4096\}$. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$ **ended up setting** $|\mathcal{B}| = \mathbf{256}$**.** This choice also determines the forward path Nesterov momentum learning rate on the *pseudogradient* of the categorization objective $\mathcal{J}$, as it is set to be $|\mathcal{B}|/2048$ (which is 0.125 in the case of $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$), and linearly warm it up to this value for 6 epochs followed by 90% decay at 30, 60, and 80 epochs, training for 100 epochs total, as prescribed by Buchlovsky et al. (2019).

- Alignment learning rate $\eta \in [10^{-6}, 10^{-2}]$, sampled log-uniformly. This parameter sets the adaptive learning rate on the Adam optimizer applied to the *gradient* of the alignment loss $\mathcal{R}$, and which will be dropped synchronously by 90% decay at 30, 60, and 80 epochs along with the Nesterov momentum learning rate on the *pseudogradient* of the categorization objective $\mathcal{J}$. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$ **ended up setting** $\eta = \mathbf{0.005268466140227959}$.

- Number of delay epochs $de \in \{0, 1, 2\}$ for which we delay optimization of the categorization objective $\mathcal{J}$ and solely optimize the alignment loss $\mathcal{R}$. If $de > 0$, we use the alignment learning rate η during this delay period and the learning rate drops are shifted by de epochs; otherwise, if $de = 0$, we linearly warmup η for 6 epochs as well. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$ **ended up setting** $\mathbf{de = 0}$ **epochs.**

- Whether or not to alternate optimization of $\mathcal{J}$ and $\mathcal{R}$ each step, or not do this and simultaneously optimize these objectives in a single training step. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$ **ended up performing this alternation per step.**

The remaining metaparameters and their ranges were the same as those from Appendix B.1.1:

- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$ ended up setting the backward pass Gaussian input noise standard deviation $\sigma = \mathbf{0.949983201505862}$.

- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$ ended up setting the ratio between the weighting of $\mathcal{P}^{\text{amp}}$ and $\mathcal{P}^{\text{decay}}$ given by $\alpha/\beta = \mathbf{13.9035614049}$.

- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD}}$ ended up setting the weighting of $\mathcal{P}^{\text{decay}}$ given by $\beta = \mathbf{2.8109006032182933 \times 10^{-8}}$.

We fix the layer-wise operations as prescribed by Akrout et al. (2019), namely batch centering the backward path and forward path outputs in the backward pass, as well as applying a ReLU activation function and bias to the forward path but not to the backward path in the backward pass.

B.1.3. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ SEARCH SPACE

Below is a description of the metaparameters and their ranges for the search that gave rise to $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ in Table 3. In this search, we expand the search space described in Appendix B.1.2 to include boolean choices over layer-wise operations performed in the *backward pass*, involving either the inputs, the forward path f_l (involving only the forward weights W_l), or the backward path b_l (involving only the backward weights B_l):

Use of biases in the forward and backward paths:

- Whether or not to use biases in the forward path. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **uses biases in the forward path.**
- Whether or not to use biases in the backward path. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **does *not* use biases in the backward path.**

Use of nonlinearities in the forward and backward paths:

- Whether or not to apply a ReLU to the forward path output. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **applies a ReLU to the forward path output.**

- Whether or not to apply a ReLU to the backward path output. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **applies a ReLU to the backward path output.**

Centering and normalization operations in the forward and backward paths:

- Whether or not to mean center (across the *batch* dimension) the forward path output $f_l = f_l - \bar{f}_l$. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **applies batch-wise mean centering to the forward path output.**
- Whether or not to mean center (across the *batch* dimension) the backward path *input*. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **applies batch-wise mean centering to the backward path input.**
- Whether or not to mean center (across the *feature* dimension) the forward path output $f_l = f_l - \hat{f}_l$. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **applies feature-wise mean centering to the forward path output.**
- Whether or not to mean center (across the *feature* dimension) the backward path output $b_l = b_l - \hat{b}_l$. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **does *not* apply feature-wise mean centering to the backward path output.**
- Whether or not to L2 normalize (across the feature dimension) the forward path output $f_l = (f_l - \hat{f}_l)/||f_l - \hat{f}_l||_2$. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **does *not* apply feature-wise L2 normalization to the forward path output.**
- Whether or not to L2 normalize (across the feature dimension) the backward path output $b_l = (b_l - \hat{b}_l)/||b_l - \hat{b}_l||_2$. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **applies feature-wise L2 normalization to the backward path output.**

Centering and normalization operations applied to the inputs to the backward pass:

- Whether or not to mean center (across the feature dimension) the backward pass input $x_l = x_l - \hat{x}_l$. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **does *not* apply feature-wise mean centering to the backward pass input.**
- Whether or not to L2 normalize (across the feature dimension) the backward pass input $x_l = (x_l - \hat{x}_l)/||x_l - \hat{x}_l||_2$. $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ **does *not* apply feature-wise L2 normalization to the backward pass input.**

The remaining metaparameters and their ranges were the same as those from Appendix B.1.2:

- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ ended up setting the training batch size $|\mathcal{B}| = \mathbf{256}$. Therefore, as explained in Appendix B.1.2, the forward path Nesterov momentum learning rate is 0.125.
- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ ended up setting the alignment Adam learning rate to $\eta = \mathbf{0.0024814785849458275}$.
- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ ended up setting the number of delay epochs to be $\mathbf{de} = \mathbf{0}$ epochs.
- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ ended up performing alternating minimization of $\mathcal{J}$ and $\mathcal{R}$ per training step.
- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ ended up setting the backward pass Gaussian input noise standard deviation $\sigma = \mathbf{0.6401550883361206}$.
- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ ended up setting the ratio between the weighting of $\mathcal{P}^{\text{amp}}$ and $\mathcal{P}^{\text{decay}}$ given by $\alpha/\beta = \mathbf{0.13444256251}$.
- $\mathcal{R}^{\text{TPE}}_{\text{WM+AD+OPS}}$ ended up setting the weighting of $\mathcal{P}^{\text{decay}}$ given by $\beta = \mathbf{232.78563021186207}$.

B.1.4. $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ SEARCH SPACE

Below is a description of the metaparameters and their ranges for the search that gave rise to $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ in Table 3. In this search, we expand the search space described in Appendix B.1.3, to now include the additional $\mathcal{P}^{\text{null}}$ primitive.

- The weighting of $\mathcal{P}^{\text{null}}$ given by $\gamma \in [10^{-11}, 10^{7}]$, sampled log-uniformly. $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ **ended up setting** $\gamma = \mathbf{3.1610192645608835 \times 10^{-6}}$**.**

The remaining metaparameters and their ranges were the same as those from Appendix B.1.3:

- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ uses biases in the forward path.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ uses biases in the backward path.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ applies a ReLU to the forward path output.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ does *not* apply a ReLU to the backward path output.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ applies batch-wise mean centering to the forward path output.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ does *not* apply batch-wise mean centering to the backward path input.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ applies feature-wise mean centering to the forward path output.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ applies feature-wise mean centering to the backward path output.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ does *not* apply feature-wise L2 normalization to the forward path output.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ applies feature-wise L2 normalization to the backward path output.
- $\mathcal{R}^{\text{TPE}}_{\text{IA}}$ does *not* apply feature-wise mean centering to the backward pass input.

- $\mathcal{R}_{\text{IA}}^{\text{TPE}}$ does *not* apply feature-wise L2 normalization to the backward pass input.
- $\mathcal{R}_{\text{IA}}^{\text{TPE}}$ ended up setting the training batch size $|\mathcal{B}| = \mathbf{256}$. Therefore, as explained in Appendix B.1.2, the forward path Nesterov momentum learning rate is 0.125.
- $\mathcal{R}_{\text{IA}}^{\text{TPE}}$ ended up setting the alignment Adam learning rate to $\eta = \mathbf{0.009785729149984566}$.
- $\mathcal{R}_{\text{IA}}^{\text{TPE}}$ ended up setting the number of delay epochs to be $\mathbf{de} = \mathbf{1}$ epochs.
- $\mathcal{R}_{\text{IA}}^{\text{TPE}}$ ended up performing alternating minimization of $\mathcal{J}$ and $\mathcal{R}$ per training step.
- $\mathcal{R}_{\text{IA}}^{\text{TPE}}$ ended up setting the backward pass Gaussian input noise standard deviation $\sigma = \mathbf{0.8176668980121832}$.
- $\mathcal{R}_{\text{IA}}^{\text{TPE}}$ ended up setting the ratio between the weighting of $\mathcal{P}^{\text{amp}}$ and $\mathcal{P}^{\text{decay}}$ given by $\alpha/\beta = \mathbf{129.112317983}$.
- $\mathcal{R}_{\text{IA}}^{\text{TPE}}$ ended up setting the weighting of $\mathcal{P}^{\text{decay}}$ given by $\beta = \mathbf{7.999046944679209}$.

B.2. Symmetric and Activation Alignment metaparameters

We now describe the metaparameters used to generate Table 4. We used a batch size of 256, forward path Nesterov with Momentum of 0.9 and a learning rate of 0.1 applied to the categorization objective $\mathcal{J}$, warmed up linearly for 5 epochs, with learning rate drops at 30, 60, and 80 epochs, trained for a total of 90 epochs, as prescribed by He et al. (2016).

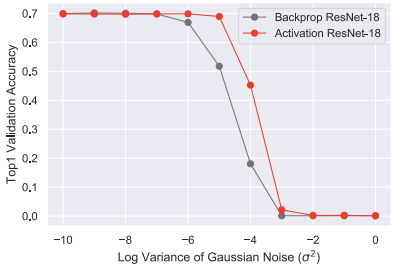
For Symmetric and Activation Alignment ($\mathcal{R}_{\text{SA}}$ and $\mathcal{R}_{\text{AA}}$), we used Adam on the alignment loss $\mathcal{R}$ with a learning rate of 0.001, along with the following weightings for their primitives:

- Symmetric Alignment: $\alpha = 10^{-3}, \beta = 2 \times 10^{-3}$
- Activation Alignment: $\alpha = 10^{-3}, \beta = 2 \times 10^{-3}$

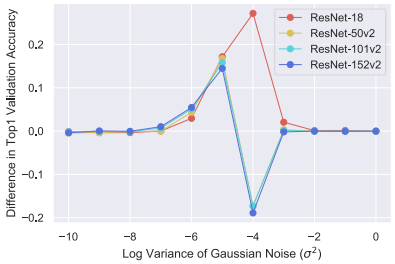
We use biases in both the forward and backward paths of the backward pass, but do *not* employ a ReLU nonlinearity to either path.

B.3. Noisy updates

We describe the metaparameters used in §5 to generate Fig. 5.



(a) ResNet-18



(b) Deeper Models

Figure S1. **Noisy updates.** Activation Alignment is *more* robust to noisy updates than backpropagation for ResNet-18 as shown in (a). For deeper models the difference in performance between Activation Alignment and backpropagation vanishes as shown in (b).

For backpropagation we used a momentum optimizer with an initial learning rate of 0.1, standard batch size of 256, and learning rate drops at 30 and 60 epochs.

For Symmetric and Activation Alignment we used the same metaparameters as backpropagation for the categorization objective $\mathcal{J}$ and an Adam optimizer with an initial learning rate of 0.001 and learning rate drops at 30 and 60 epochs for the alignment loss $\mathcal{R}$. All other metaparameters were the same as described in Appendix B.2.

In all experiments we added the noise to the update given by the respective optimizers and scaled by the current learning rate, that way at learning rate drops the noise scaled appropriately. To account for the fact that the initial learning rate for the backpopagation experiments was 0.1, while for symmetric and activation experiments it was 0.001, we shifted the latter two curves by 10^4 to account for the effective difference in variance.

B.4. Metaparameter importance quantification

We include here the set of discrete metaparameters that mattered the most across hundreds of models in our large-scale search, sorted by most to least important, plotted in Fig. S2. Specifically, these amount to choices of activation, layer-wise normalization, input normalization, and Gaussian noise in the forward and backward paths of the backward

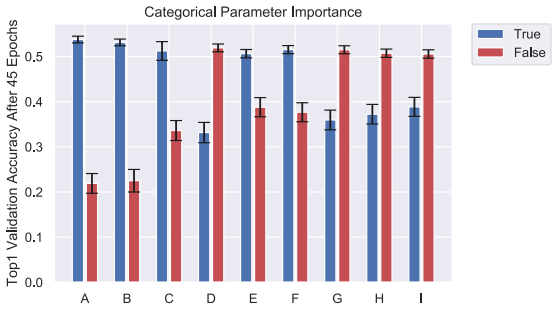


Figure S2. **Analysis of important categorical metaparameters of top performing local rule $\mathcal{R}_{\text{IA}}^{\text{TPE}}$.** Mean across models, and the error bars indicate SEM across models.

pass. The detailed labeling is given as follows: **A:** Whether or not to L2 normalize (across the feature dimension) the backward path outputs in the backward pass. **B:** Whether to use Gaussian noise in the backward pass inputs. **C:** Whether to solely optimize the alignment loss in the first 1-2 epochs of training. **D, E:** Whether or not to apply a non-linearity in the backward or forward path outputs in the backward pass, respectively. **F:** Whether or not to apply a bias in the forward path outputs (pre-nonlinearity). **G, H:** Whether or not to mean center or L2 normalize (across the feature dimension) the inputs to the backward pass. **I:** Same as **A**, but instead applied to the forward path outputs in the backward pass.

B.5. Neural Fitting Procedure

We fit trained model features to multi-unit array responses from (Majaj et al., 2015). Briefly, we fit to 256 recorded sites from two monkeys. These came from three multi-unit arrays per monkey: one implanted in V4, one in posterior IT, and one in central and anterior IT. Each image was presented approximately 50 times, using rapid visual stimulus presentation (RSVP). Each stimulus was presented for 100 ms, followed by a mean gray background interleaved between images. Each trial lasted 250 ms. The image set consisted of 5120 images based on 64 object categories. Each image consisted of a 2D projection of a 3D model added to a random background. The pose, size, and x- and y-position of the object was varied across the image set, whereby 2 levels of variation were used (corresponding to medium and high variation from (Majaj et al., 2015).) Multi-unit responses to these images were binned in 10ms windows, averaged across trials of the same image, and normalized to the average response to a blank image. They were then averaged 70-170 ms post-stimulus onset, producing a set of (5120 images x 256 units) responses, which were the targets for our model features to predict. The 5120 images were split 75-25 within each object category into a training set and a held-out testing set.

C. Visualizations

In this section we present some visualizations which deepen the understanding of the weight dynamics and stability during training, as presented in §4 and §5. By looking at the weights of the network at each validation point, we are able to compare corresponding forward and backward weights (see Fig. S3) as well as to measure the angle between the vectorized forward and backward weight matrices to quantify their degree of alignment (see Fig. S4). Their similarity in terms of scale can also be evaluated by looking at the ratio of the Frobenius norm of the backward weight matrix to the forward weight matrix, $\|B_l\|_F / \|W_l\|_F$. Separately plotting these metrics in terms of model depth sheds some insight into how different layers behave.

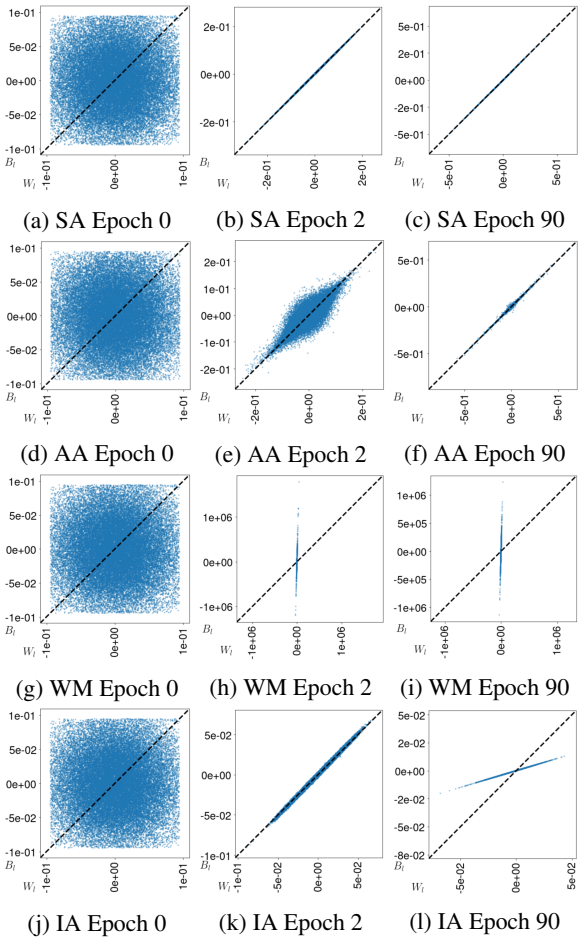
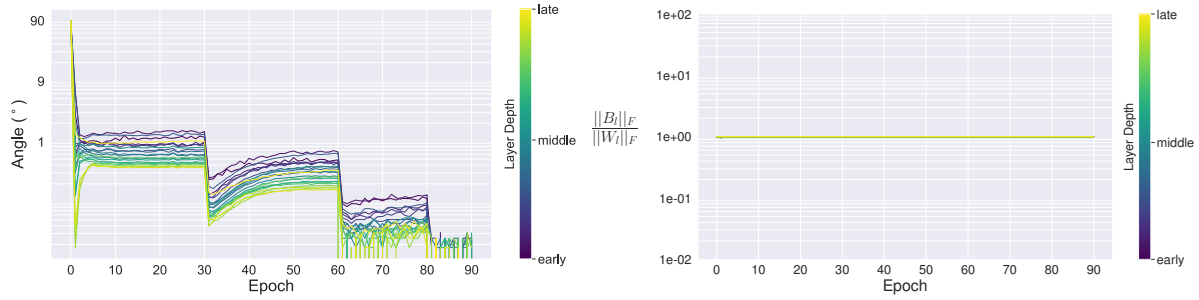
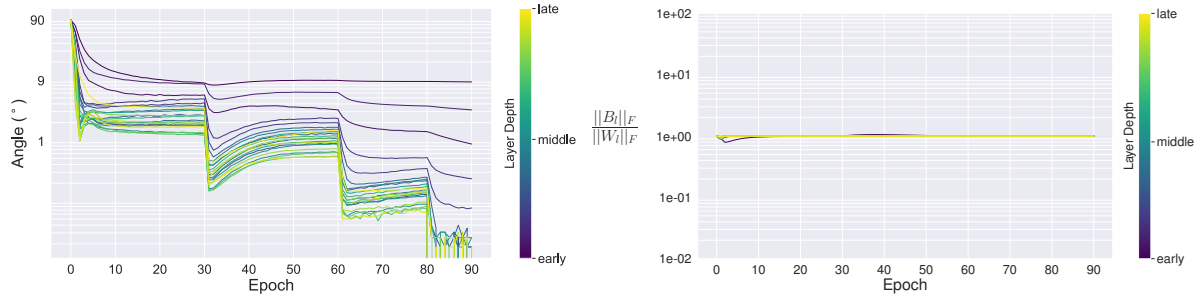


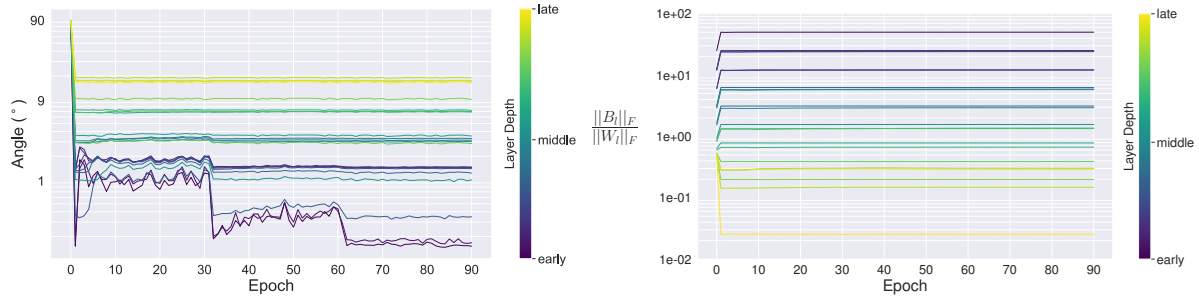
Figure S3. **Learning symmetry.** Weight values of the third convolutional layer in ResNet-18 throughout training with various learning rules. Each dot represents an element in layer l's weight matrix and its (x, y) location corresponds to its forward and backward weight values, $(W_l^{(i,j)}, B_l^{(j,i)})$. The dotted diagonal line shows perfect weight symmetry, as is the case in backpropagation.



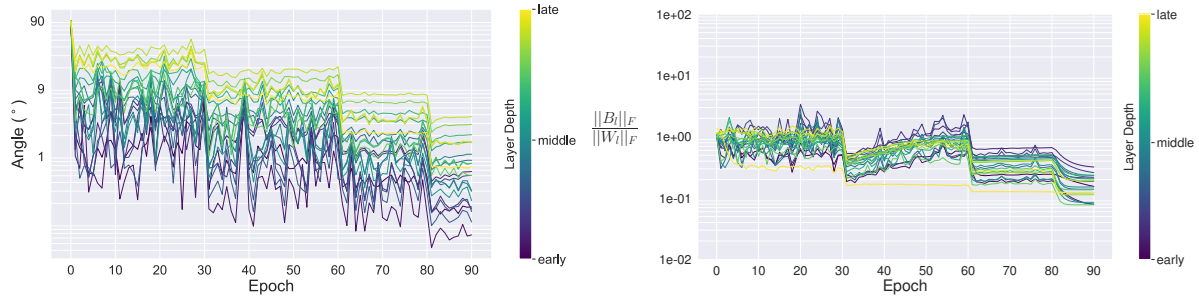
(a) Symmetric Alignment



(b) Activation Alignment



(c) Weight Mirror



(d) Information Alignment

Figure S4. **Weight metrics during training.** Figures on the left column show the angle between the forward and the backward weights at each layer, depicting their degree of alignment. Figures on the right column show the ratio of the Frobenius norm of the backward weights to the forward weights during training. For Symmetric Alignment (a) we can clearly see how the weights align very early during training, with the learning rate drops allowing them to further decrease. Additionally, the sizes of forward and backwards weight also remain at the same scale during training. Activation Alignment (b) shows similar behavior to activation, though some of the earlier layers fail to align as closely as the Symmetric Alignment case. Weight Mirror (c) shows alignment happening within the first few epochs, though some of the later layers don't align as closely. Looking at the size of the weights during training, we can observe the unstable dynamics explained in §4 with exploding and collapsing weight values (Fig. 2) within the first few epochs of training. Information Alignment (d) shows a similar ordering in alignment as weight mirror, but overall alignment does improve throughout training, with all layers aligning within 5 degrees. Compared to weight mirror, the norms of the weights are more stable, with the backward weights becoming smaller than their forward counterparts towards the end of training.

D. Further Analysis

D.1. Instability of Weight Mirror

As explained in §4, the instability of weight mirror can be understood by considering the dynamical system given by the symmetrized gradient flow on $\mathcal{R}_{\text{SA}}$, $\mathcal{R}_{\text{AA}}$, and $\mathcal{R}_{\text{WM}}$ at a given layer l. By symmetrized gradient flow we imply the gradient dynamics on the loss $\mathcal{R}$ modified such that it is symmetric in both the forward and backward weights. We ignore biases and non-linearities and set $\alpha = \beta$ for all three losses.

When the weights, w_l and b_l, and input, x_l, are all scalar values, the gradient flow for all three losses gives rise to the dynamical system,

$$\frac{\partial}{\partial t}\begin{bmatrix} w_l \\ b_l \end{bmatrix} = -A\begin{bmatrix} w_l \\ b_l \end{bmatrix},$$

For Symmetric Alignment and Activation Alignment, A is respectively the positive semidefinite matrix

$$A_{\text{SA}} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \quad \text{and} \quad A_{\text{AA}} = \begin{bmatrix} x_l^2 & -x_l^2 \\ -x_l^2 & x_l^2 \end{bmatrix}.$$

For weight mirror, A is the symmetric indefinite matrix

$$A_{\text{WM}} = \begin{bmatrix} \lambda_{\text{WM}} & -x_l^2 \\ -x_l^2 & \lambda_{\text{WM}} \end{bmatrix}.$$

In all three cases A can be diagonally decomposed by the eigenbasis

$$\{u, v\} = \left\{\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \end{bmatrix}\right\},$$

where u spans the symmetric component and v spans the skew-symmetric component of any realization of the weight vector $\begin{bmatrix} w_l & b_l \end{bmatrix}^\intercal$.

As explained in §4, under this basis, the dynamical system decouples into a system of ODEs governed by the eigenvalues λ_u and λ_v associated with u and v. For all three learning rules, $\lambda_v > 0$ (λ_v is respectively 1, x^2, and $\lambda_{\text{WM}} + x_l^2$ for SA, AA, and weight mirror). For SA and AA, $\lambda_u = 0$, while for weight mirror $\lambda_u = \lambda_{\text{WM}} - x_l^2$.

D.2. Beyond Feedback Alignment

An underlying assumption of our work is that certain forms of layer-wise regularization, such as the regularization introduced by Symmetric Alignment, can actually improve the performance of feedback alignment by introducing dynamics on the backward weights. To understand these improvements, we build off of prior analyses of backpropagation (Saxe et al., 2013) and feedback alignment (Baldi et al., 2018).

Consider the simplest nontrivial architecture: a two layer scalar linear network with forward weights w_1, w_2, and backward weight b. The network is trained with scalar data $\{x_i, y_i\}_{i=1}^n$ on the mean squared error cost function

$$\mathcal{J} = \sum_{i=1}^{n} \frac{1}{2n}(y_i - w_2 w_1 x_i)^2.$$

The gradient flow of this network gives the coupled system of differential equations on (w_1, w_2, b)

$$\dot{w_1} = b(\alpha - w_2 w_1 \beta) \tag{2}$$

$$\dot{w_2} = w_1(\alpha - w_2 w_1 \beta) \tag{3}$$

where $\alpha = \sum_{i=1}^n \frac{y_i x_i}{n}$ and $\beta = \sum_{i=1}^n \frac{x_i^2}{n}$. For backpropagation the dynamics are constrained to the hyperplane $b = w_2$, while for feedback alignment the dynamics are contained on the hyperplane $b = b(0)$ given by the initialization. For Symmetric Alignment, an additional differential equation

$$\dot{b} = w_2 - b, \tag{4}$$

attracts all trajectories to the backpropagation hyperplane $b = w_2$.

To understand the properties of these alignment strategies, we explore the fixed points of their flow. From equation (2) and (3) we see that both equations are zero on the hyperbola

$$w_2 w_1 = \frac{\alpha}{\beta},$$

which is the set of minima of $\mathcal{J}$. From equation (4) we see that all fixed points of Symmetric Alignment satisfy $b = w_2$. Thus, all three alignment strategies have fixed points on the hyperbola of minima intersected with either the hyperplane $b = b(0)$ in the case of feedback alignment or $b = w_2$ in the case of backpropagation and Symmetric Alignment.

In addition to these non-zero fixed points, equation (2) and (3) are zero if b and w_1 are zero respectively. For backpropagation and Symmetric Alignment this also implies $w_2 = 0$, however for feedback alignment w_2 is free to be any value. Thus, all three alignment strategies have rank-deficient fixed points at the origin $(0, 0, 0)$ and in the case of feedback alignment more generally on the hyperplane $b = w_1 = 0$.

To understand the stability of these fixed points we consider the local linearization of the vector field by computing the Jacobian matrix[4]

$$J = \begin{bmatrix} \partial_{w_1}\dot{w_1} & \partial_{w_1}\dot{w_2} & \partial_{w_1}\dot{b} \\ \partial_{w_2}\dot{w_1} & \partial_{w_2}\dot{w_2} & \partial_{w_2}\dot{b} \\ \partial_b\dot{w_1} & \partial_b\dot{w_2} & \partial_b\dot{b} \end{bmatrix}.$$

A source of the gradient flow is characterized by non-positive eigenvalues of J, a sink by non-negative eigenvalues of J, and a saddle by both positive and negative eigenvalues of J.

[4] In the case that the vector field is the negative gradient of a loss, as in backpropagation, then this is the negative Hessian of the loss.

On the hyperbola $w_2 w_1 = \frac{\alpha}{\beta}$ the Jacobian matrix for the three alignment strategies have the corresponding eigenvalues:

	λ_1	λ_2	λ_3
Backprop.	$-\left(w_1^2 + w_2^2\right) x^2$	0	
Feedback	$-\left(w_1^2 + b w_2\right) x^2$	0	0
Symmetric	$-\left(w_1^2 + w_2^2\right) x^2$	0	-1

Thus, for backpropagation and Symmetric Alignment, all minima of the the cost function $\mathcal{J}$ are sinks, while for feedback alignment the stability of the minima depends on the sign of $w_1^2 + b w_2$.

From this simple example there are two major takeaways:

1. All minima of the cost function $\mathcal{J}$ are sinks of the flow given by backpropagation and Symmetric Alignment, but only some minima are sinks of the flow given by feedback alignment.

2. Backpropagation and Symmetric Alignment have the exact same critical points, but feedback alignment has a much larger space of rank-deficient critical points.

Thus, even in this simple example it is clear that certain dynamics on the backward weights can have a stabilizing effect on feedback alignment.