A CaSE: additional details

A.1 CaSE implementation

Standardization Empirically we have observed that standardizing the pooled representations before passing them to the MLP improves the training stability in CaSE (but not in SE). Standardization is performed by taking the pooled representation at layer l as showed in Equation (3), that is $\overline{\mathbf{h}}^{(l)} \in \mathbb{R}^C$, subtracting the mean and dividing by the standard deviation.

Activation function for the output layer Standard SE blocks usually rely on a sigmoid function in the last layer of the MLPs. This works well when the adaptive block is trained in parallel with the underlying neural network. However, in our case we use a pretrained model and learning can be speeded up considerably by enforcing the identity function as output of the MLPs. We achieve this by multiplying the output of the sigmoid by a constant scalar $c = 2$ which extends the range to $[0, 2]$, and then set to zero the weights and bias of the layer. This has the effect of enforcing the identity function at the beginning of the training. We have also used a linear activation function instead of a sigmoid, with good results. When using a linear output the identity can be enforced by setting the weights of the last layer to zero, and the bias to one. An ablation over the activation function of SE and CaSE is provided in Appendix C.4 (Table 6).

CaSE location For the choice of CaSE location in the feature extractor, we followed the same principles used in Bronskill et al. (2021) for FiLM generators. In EfficientNetB0 we place CaSE at the beginning of each hyperblock and the last layer (excluding the first layer). Differently from FiLM (placed after the BatchNorm) we place CaSE after the non-linearity (as done in standard SE) and before the Squeeze-and-Excitation block (included by default in EfficientNet):

$$\text{Conv2d} \rightarrow \text{BatchNorm2d} \rightarrow \text{SiLU} \rightarrow \text{CaSE} \rightarrow \text{SqueezeExcitation} \rightarrow \text{Conv2d} \rightarrow \text{BatchNorm2d}$$

This results in a total of 18 CaSE blocks for EfficientNetB0. Increasing the number of blocks did not provide a significant benefit. In ResNet18 we place two CaSE blocks per each basic block as:

$$\text{Conv2d} \rightarrow \text{BatchNorm2d} \rightarrow \text{ReLU} \rightarrow \text{CaSE} \rightarrow \text{Conv2d} \rightarrow \text{BatchNorm2d} \rightarrow \text{ReLU} \rightarrow \text{CaSE}$$

Similarly we place two CaSE blocks inside a bottleneck block in ResNet50. See the code for more details.

Based on the qualitative analysis reported in Section 5 we hypothesize that adaptive blocks are not needed in the initial layers of the network, since at those stages their activity is minimal. Identifying which layer needs adapters and which layer does not, can reduce even more the parameter count of adaptive blocks. Additional work is needed to fully understand this factor.

CaSE reduction The number of parameters allocated to the CaSE blocks is regulated by a divider r that is used to compute the number of hidden units in the MLPs. Given the input size C (corresponding to the number of channels in that layer) the number of hidden units is given by C/r. We also use a clipping factor r_{min} that prevents the number of units to fall under a given threshold. This prevents the allocation of a low number of units for layers with a small number of channels.

A.2 Context pooling

In this section we provide additional details about the context pooling operation performed in a CaSE adaptive block (described in Section 2).

Similarities with other methods Context pooling is a way to summarize a task with a permutation-invariant aggregation of the embeddings. A similar mechanism has been exploited in various meta-learning methods. For instance, in ProtoNets (Snell et al., 2017) a prototype for a single class is computed by taking the average over all the context embeddings associated to the inputs for that class. The embeddings are generated in the last layer of the feature extractor. In Simple-CNAPs (Bateni et al., 2020) a prototype is estimated as in ProtoNets but it is used to define a Gaussian distribution instead of a mean vector. Neural latent variable models, such as those derived from the Neural Processes family (Garnelo et al., 2018) also rely on similar permutation-invariant aggregations to define distributions over functions.

Global vs. local context-pooling Comparing CaSE with the FiLM generators of Bronskill et al. (2021) it is possible to distinguish between two types of context pooling: global and local. The FiLM generators of Bronskill et al. (2021) rely on a *global* pooling strategy, meaning that the aggregation is performed once-for-all by using a dedicated convolutional set encoder. More specifically, the encoder takes as input all the context images and produces embeddings for each one of them, followed by an average-pooling of those embeddings. The aggregated embedding is then passed to MLPs in each layer that generates a scale and shift parameter. Crucially, each MLP receives the same embedding.

CaSE exploits a *local* context-pooling at the layer level. The convolutional set encoder is discarded, and the feature maps produces by the backbone itself at each stage are used as context embeddings. Therefore, the MLPs responsible for generating the scale parameters receive a unique embedding. As showed in the experimental section (Section 5), local pooling improves performances and uses less parameters, as no convolutional encoder is needed. Additional details about the differences between CaSE and FiLM generators is also provided in the paper (Section 4).

A.3 Pytorch code for CaSE

Implementation of a CaSE adaptive block in Pytorch. The script is also available as `case.py` at `https://github.com/mpatacchiola/contextual-squeeze-and-excitation`.

```
import torch
from torch import nn

class CaSE(nn.Module):
  def __init__(self, cin, reduction=32, min_units=32,
               standardize=True, out_mul=2.0,
               device=None, dtype=None):
    """
    Initialize a CaSE adaptive block.

    Parameters:
    cin (int): number of input channels.
    reduction (int): divider for computing number of hidden units.
    min_units (int): clip hidden units to this value (if lower).
    standardize (bool): standardize the input for the MLP.
    out_mul (float): multiply the MLP output by this value.
    """
    factory_kwargs = {'device': device, 'dtype': dtype}
    super(CaSE, self).__init__()
    self.cin = cin
    self.standardize = standardize
    self.out_mul = out_mul
    hidden = max(min_units, cin // reduction)
    self.gamma_generator = nn.Sequential(
      nn.Linear(cin, hidden, bias=True, **factory_kwargs),
      nn.SiLU(),
      nn.Linear(hidden, hidden, bias=True, **factory_kwargs),
      nn.SiLU(),
      nn.Linear(hidden, cin, bias=True, **factory_kwargs),
      nn.Sigmoid() )
    self.reset_parameters()

  def reset_parameters(self):
    nn.init.zeros_(self.gamma_generator[4].weight)
    nn.init.zeros_(self.gamma_generator[4].bias)
    self.gamma = torch.tensor([1.0])

  def forward(self, x):
    if(self.training): # adaptive mode
      self.gamma = torch.mean(x, dim=[2,3]) # spatial pooling
      self.gamma = torch.mean(self.gamma, dim=[0]) # context pooling
      if(self.standardize):
        self.gamma = (self.gamma - torch.mean(self.gamma)) / \
            torch.sqrt(torch.var(self.gamma, unbiased=False) + 1e-5)
      self.gamma = self.gamma.unsqueeze(0)
      self.gamma = self.gamma_generator(self.gamma) * self.out_mul
      self.gamma = self.gamma.reshape([1,-1,1,1])
      return self.gamma * x
    else: # inference mode
      self.gamma = self.gamma.to(x.device)
      return self.gamma * x

  def extra_repr(self):
      return 'cin={}'.format(self.cin)
```

B UppereCaSE: additional details

B.1 Algorithm of UpperCaSE

Algorithm 1 UpperCaSE: training function for the few-shot classification setting.

Require: $\mathcal{D} = \{\tau_1, \ldots, \tau_D\}$ training dataset
Require: $b_{\boldsymbol{\phi}}()$ pretrained feature extractor (body) with CaSE blocks parameterized by $\boldsymbol{\phi}$.
Require: `step()`: gradient-step function; $\mathcal{L}$ loss; α, β: step-size hyperparameters for the optimizer.

Set $\boldsymbol{\phi}$ to random values ▷ optional: set $\boldsymbol{\phi}$ to enforce identity in CaSE output
while not done **do**
  Sample task $\tau = (\mathcal{C}, \mathcal{T}) \sim \mathcal{D}$
  Forward pass over context set $b_{\boldsymbol{\phi}}(\mathcal{C}^x) \rightarrow \mathbf{z}_1, \ldots, \mathbf{z}_N$ ▷ CaSE in adaptive mode
  Store context embeddings and associated labels $\mathcal{M} = \{(\mathbf{z}_n, y_n)\}_{n=1}^N$ ▷ temporary memory buffer
  Define a linear model for the head $h_{\boldsymbol{\psi}_\tau}()$ and set $\boldsymbol{\psi}_\tau$ to zero
  for `total inner-steps` **do** ▷ loop to estimate head params
    Sample (with replacement) mini-batch of training pairs $\mathcal{B} \sim \mathcal{M}$
    Update the head parameters $\boldsymbol{\psi}_\tau \leftarrow$ `step`$(\alpha, \mathcal{L}, \mathcal{B}, h_{\boldsymbol{\psi}_\tau})$
  end for
  Update the CaSE parameters $\boldsymbol{\phi} \leftarrow$ `step`$(\beta, \mathcal{L}, \mathcal{C}, \mathcal{T}, b_{\boldsymbol{\phi}}, h_{\boldsymbol{\psi}_\tau})$ ▷ CaSE in adaptive mode
end while

Algorithm 2 UpperCaSE: test function for the few-shot classification setting.

Require: $\tau_* = (\mathcal{C}_*, \mathbf{x}_*)$ unseen test task with target input $\mathbf{x}_*$ an context $\mathcal{C}_*$.
Require: $b_{\boldsymbol{\phi}}()$ pretrained feature extractor (body) with meta-learned CaSE blocks parameterized by $\boldsymbol{\phi}$.
Require: `step()`: gradient-step function; $\mathcal{L}$ loss; α: step-size hyperparameter for the optimizer.

Forward pass over context set $b_{\boldsymbol{\phi}}(\mathcal{C}_*^x) \rightarrow \mathbf{z}_1, \ldots, \mathbf{z}_N$ ▷ CaSE in adaptive mode
Store context embeddings and associated labels $\mathcal{M}_* = \{(\mathbf{z}_n, y_n)\}_{n=1}^N$ ▷ temporary memory buffer
Define a linear model for the head $h_{\boldsymbol{\psi}_{\tau_*}}()$ and set $\boldsymbol{\psi}_{\tau_*}$ to zero
for `total inner-steps` **do** ▷ loop to estimate head params
  Sample (with replacement) mini-batch of training pairs $\mathcal{B}_* \sim \mathcal{M}_*$
  Update the head parameters $\boldsymbol{\psi}_{\tau_*} \leftarrow$ `step`$(\alpha, \mathcal{L}, \mathcal{B}_*, h_{\boldsymbol{\psi}_{\tau_*}})$
end for
Return Prediction $\hat{y}_* = h_{\boldsymbol{\psi}_{\tau_*}}(b_{\boldsymbol{\phi}}(\mathbf{x}_*))$ ▷ CaSE in inference mode

C Additional experimental details and results

C.1 Additional details

MACs counting MACs are proportional to the size of the task, size of the images, and number of classes. We can count MACs using synthetic tasks. In our case we used a synthetic task of 100-way, 10-shot with input images of size $224 \times 224 \times 3$ generated via Gaussian noise ($\mu = 0, \sigma = 1$), and labels generated as random integers. We used a mini-batch of size 128 and 500 update steps for UpperCaSE and BiT with an EfficientNetB0 backbone for the first and a ResNet50-S for the second. For MD-Transfer we used the same parameters reported in Dumoulin et al. (2021) with images of size $126 \times 126 \times 3$ and ResNet18 backbone. For the ORBIT experiments we counted MACs by using the code in the original repository [2] and reporting the average MACs over all test tasks for both CLE-VE and CLU-VE using a ResNet18 backbone.

VTAB+MD training We follow the protocol reported in the original papers (Triantafillou et al., 2019; Dumoulin et al., 2021) training UpperCaSE for 10K tasks on the training datasets and evaluating on the MD test set and on the VTAB datasets. At evaluation time we sample 1200 tasks from the MD test set, and report the mean and confidence intervals. On VTAB we report the results of a single run on the test data (data points are given in advance and do not change across seeds). In all experiments we used the MetaDataset-v2 (MDv2) which does not include ImageNet in the test set. We used a pretrained EfficientNetB0 from the official Torchvision repository [3], and a pretrained ResNet50-S

[2] `https://github.com/microsoft/ORBIT-Dataset`
[3] `https://pytorch.org/vision`

from the BiT repository [4]. We normalized the inputs using the values reported in the Torchvision documentation (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]), for ResNet50-S we use the BiT normalization values (mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]).

ORBIT training For the ORBIT experiments we trained UpperCaSE on MDv2 using a pretrained ResNet18 taken from the official Torchvision repository. We normalized the inputs using the values reported in the Torchvision documentation (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]). For the evaluation phase we followed the instructions reported in Massiceti et al. (2021).

C.2 CaSE vs SE

Table 4: Comparing CaSE against standard Squeeze-and-Excitation (SE) on VTAB+MD using different adaptation heads. MD: Mahalanobis distance head (Bronskill et al., 2021). Linear: linear head trained with UpperCaSE. All adaptive blocks use a reduction of 32. Best results in bold.

Model	SE	CaSE	SE	CaSE
Contextual pooling	No	Yes	No	Yes
Adaptation head	MD	MD	Linear	Linear
Image size	84	84	224	224
MetaDataset (all)	67.8	**69.6**	74.6	**76.2**
VTAB (all)	43.6	**45.3**	56.6	**58.2**
VTAB (natural)	47.5	**50.2**	65.3	**68.1**
VTAB (specialized)	63.6	**64.9**	**79.8**	79.6
VTAB (structured)	30.6	**31.8**	38.6	**40.1**

C.3 CaSE vs other adapters

Table 5: Comparing CaSE adaptive blocks (with reduction 64, 32, 16) on VTAB+MD against the FiLM generators used in Bronskill et al. (2021), and a baseline with no body adaptation. CaSE blocks are more efficient in terms of adaptive and amortization parameters while providing higher classification accuracy. All models have been trained and tested on 84×84 images, using a Mahalanobis distance head. Best results in bold.

Adaptation type	None	FiLM	CaSE64	CaSE32	CaSE16
Adaptive Params (M)	n/a	0.02	**0.01**	**0.01**	**0.01**
Amortiz. Params (M)	n/a	1.7	**0.4**	0.8	1.6
MetaDataset (all)	53.4	68.4	69.8	69.6	**70.4**
VTAB (all)	43.5	44.7	46.2	45.3	**46.4**
VTAB (natural)	45.4	49.5	52.1	50.2	**52.6**
VTAB (specialized)	**69.4**	63.8	66.3	64.9	65.5
VTAB (structured)	29.1	31.7	31.8	31.8	**32.1**

C.4 Ablation studies

In this section we provide additional experimental results focusing on ablation studies of the CaSE adaptive block. The results can be summarized as follows:

- Ablation of the activation function for the output layer for both CaSE and SE. We have tested three activation funcitons: linear, sigmoid, sigmoid with multiplier. The sigmoid with multiplier uses a constant value set to 2 to center the sigmoid at 1 (this enforces the identity function). The empirical results reported in Table 6 show that the sigmoid with multiplier and the linear layer provide the best results.

[4] `https://github.com/google-research/big_transfer`

- Ablation of the number of hidden units in the hidden layers of CaSE. The number of hidden units is controlled by the reduction and min-units parameters in the code and it depends on the number of inputs. See the paper for more details. The results reported in Table 8 show that blocks with more units provide marginal gains or no gains at all. This is probably due to overfitting issues affecting the models with more units.

- Ablation of the number of hidden layers of CaSE. The results reported in Table 7 show that the best performance is obtained with 1 and 2 layers. The performance worsen when there are 3 or more layers which is likely due to overfitting issues affecting the models with more parameters.

- Ablation of the activation function for the hidden layers. Results reported in Table 9 show that CaSE is quite robust against this factor when activations like ReLU and SiLU are used but the performance worsen with Tanh. We have chosen SiLU for the experiments as this is the same activation typically used in Squeeze-and-Excitation layers (e.g. in EfficientNet backbones).

Table 6: Performance on VTAB+MD for various activation functions used in the last layer of SE and CaSE. Sigmoid-2 indicates that the output of a standard Sigmoid is multiplied by 2. Both SE and CaSE use a reduction factor of 32 with min-clipping of 32. All model have been trained using an EfficientNetB0 backbone with a linear head on images of size 224×224. Results for SE with linear activation have not been reported because the training was unstable (loss rapidly diverging at the first iterations). Best results in bold.

Adaptive block Activation (output)	SE Sigmoid	SE Sigmoid-2	CaSE Linear	CaSE Sigmoid	CaSE Sigmoid-2
MetaDataset (all)	74.2	74.6	75.8	74.9	**76.2**
VTAB (all)	56.8	56.6	**58.4**	56.8	58.2
VTAB (natural)	67.0	65.3	**68.3**	67.1	68.1
VTAB (specialized)	**81.1**	79.8	79.5	80.8	79.6
VTAB (structured)	36.9	38.6	**40.3**	37.1	40.1

Table 7: Comparing CaSE adaptive blocks with different number of hidden layers on VTAB+MD. All models have been trained and tested on 224×224 images, using CaSE with reduction 64 and clip factor (min-units) 16, using UpperCaSE and EfficientNetB0 backbone. Best results in bold.

# Hidden layers Amortiz. Params (M)	1 **0.420**	2 0.426	3 0.432	4 0.438
MetaDataset (all)	76.0	**76.1**	75.5	75.2
VTAB (all)	58.2	**58.4**	58.2	58.0
VTAB (natural)	68.3	**69.1**	68.0	67.4
VTAB (specialized)	79.7	80.3	**80.5**	80.3
VTAB (structured)	**40.0**	39.4	39.7	39.7

Table 8: Comparing CaSE adaptive blocks with different number of hidden units on VTAB+MD. The number of hidden units depends on the input size and is defined by the reduction and the clip factor (min-units). All models have been trained and tested on 224×224 images, using UpperCaSE and EfficientNetB0 backbone. Best results in bold.

Reduction factor	64	32	16	8
Clip factor	16	32	48	64
Amortiz. Params (M)	**0.4**	0.8	1.6	3.0
MetaDataset (all)	76.1	**76.2**	75.8	**76.2**
VTAB (all)	58.4	58.2	57.9	**58.5**
VTAB (natural)	**69.1**	68.1	67.9	68.3
VTAB (specialized)	**80.3**	79.6	79.4	79.0
VTAB (structured)	39.4	40.1	39.7	**40.9**

Table 9: Comparing CaSE adaptive blocks with different activation functions for the hidden layers on VTAB+MD. All models are based on a reduction factor of 64 and a clip factor of 16 (0.4M amortization parameters) and they have been trained and tested on 224×224 images, using UpperCaSE and EfficientNetB0 backbone. Best results in bold.

Activation (hidden)	SiLU	ReLU	Tanh
MetaDataset (all)	**76.1**	75.8	74.8
VTAB (all)	**58.4**	57.8	48.2
VTAB (natural)	69.1	**69.8**	67.0
VTAB (specialized)	**80.3**	79.7	80.8
VTAB (structured)	**39.4**	**39.4**	36.4

C.5 Role of CaSE blocks

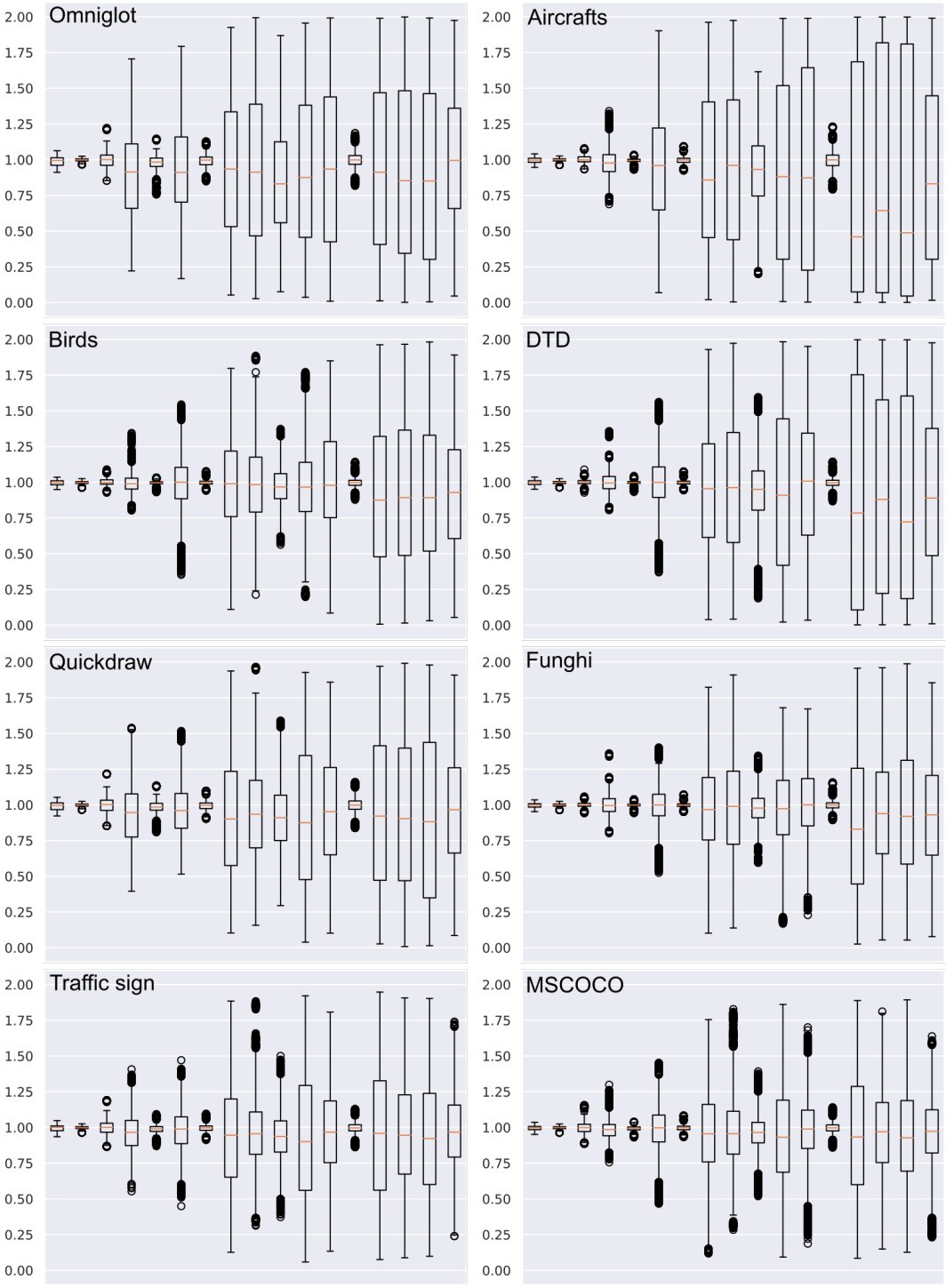

Figure 4: Boxplots for all the MDv2 test datasets (100 tasks per dataset) reporting the CaSE activation (vertical axis) at different stages of an EfficientNetB0 (horizontal axis, with early stages on the left). The box encloses first to third quartile, with the median represented by the orange line. The whiskers extend from the box by 1.5 the inter-quartile range. Outlier (point past the end of the whiskers) are represented with black circles.

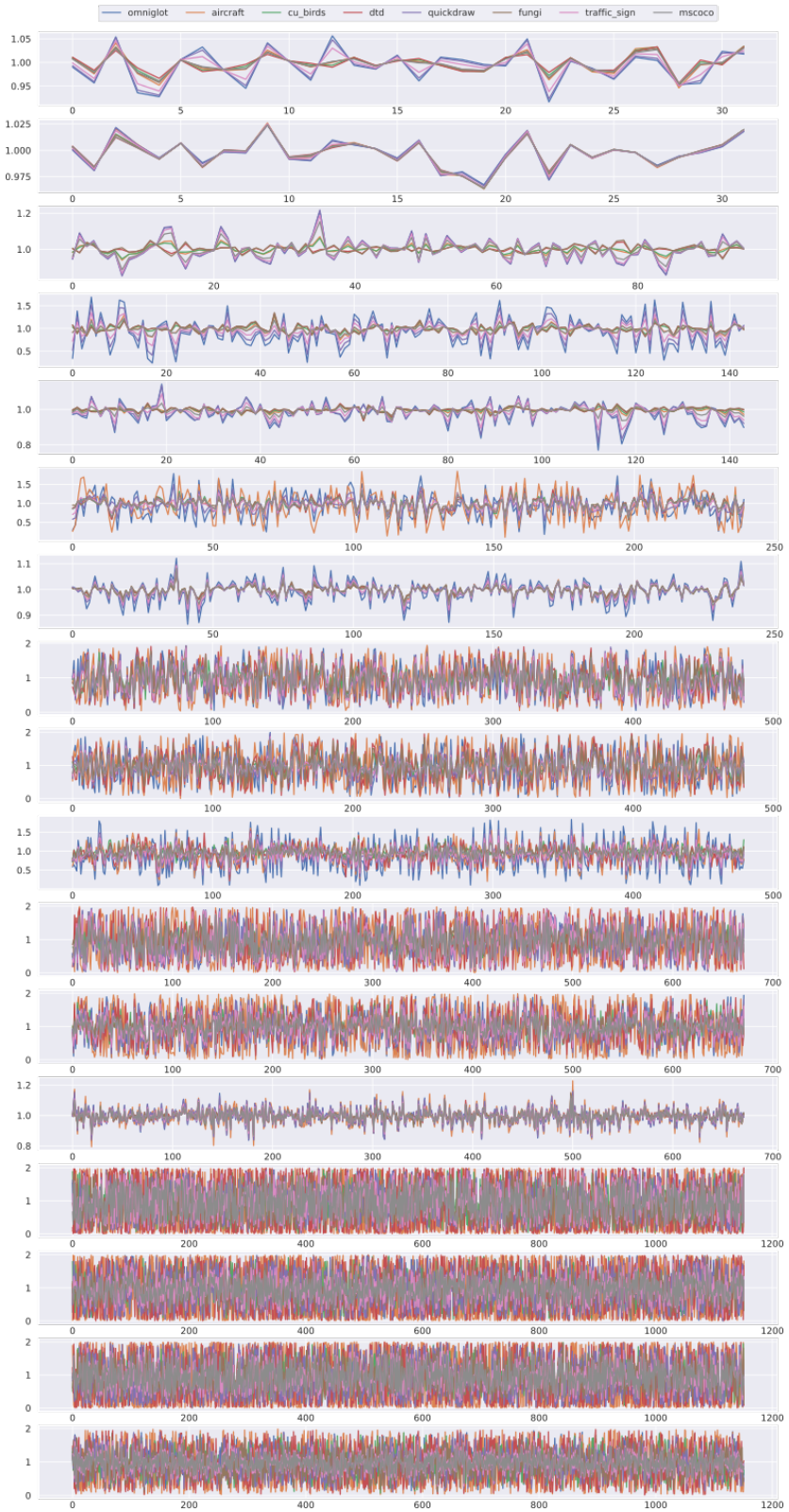

Figure 5: CaSE activation values (vertical axis) for all channels (horizontal axis) at different stages (top plots are early stages) in EfficientNetB0 for the MDv2 test dataset (one task per dataset). Values are similar and closer to one in the first stages but diverge in the latest. The magnitude tends to increase with depth.

C.6 UpperCaSE: results on VTAB+MD

In this section we provide a full breakdown of the results for UpperCaSE vs. other methods on the VTAB+MD benchmark. Results for other methods are taken from Bronskill et al. (2021) and Dumoulin et al. (2021). UpperCaSE uses CaSE with reduction 64 (min-clip 16) for EfficientNetB0 and reduction 32 (min-clip 32) for ResNet50-S. Results for UpperCaSE on MD are the average over 1200 test tasks.

In Table 10 we report the results for UpperCaSE against fine-tuning methods (BiT, MD-Trasnfer, SUR) and in Table 11 the results for UpperCaSE against meta-learning and hybrid methods (ProtoNet, ProtoMAML, Cross Transformer CTX, LITE). Overall UpperCaSE performs well on MD and the natural split of VTAB, this may be due to the fact that transfer learning is more beneficial on those datasets as they are more similar to those used during meta-training. The largest difference in performance between UpperCaSE and fine-tuning methods is on the structured split of VTAB, which includes tasks that require counting and pose estimation. This is likely due to the difference w.r.t. the meta-training set. In this case, fine-tuning the entire network is more effective than body adaptation as the knowledge gap is wider and it requires more adjustments to the parameters.

Table 10: Comparing UpperCaSE against fine-tuning methods. Best result in bold.

Model	BiT	MD-Transfer	SUR	UpperCaSE	UpperCaSE
Image Size	224	126	224	224	224
Network	RN50-S	RN18	RN50×7	ENB0	RN50-S
Params (M)	23.5	11.2	164.5	4.0	23.5
Omniglot	68.0±4.5	82.0±1.3	**92.8±0.5**	90.7±0.4	89.1±0.5
Aircraft	77.4±3.5	76.8±1.2	84.4±0.6	**89.4±0.4**	87.5±0.4
Birds	**90.8±1.5**	61.2±1.3	75.8±1.0	**90.4±0.4**	**89.6±0.4**
DTD	**85.0±2.5**	66.0±1.1	74.3±0.7	**83.4±0.4**	**84.8±0.5**
QuickDraw	66.6±3.7	61.3±1.1	70.3±0.7	**76.8±0.5**	73.7±0.6
Fungi	59.4±4.2	35.5±1.1	**81.7±0.6**	59.3±0.8	56.8±0.8
Traffic Sign	73.5±4.7	**84.7±0.9**	50.0±1.1	68.5±0.8	70.6±0.8
MSCOCO	**65.7±2.7**	39.6±1.0	49.4±1.1	50.8±0.7	46.7±0.8
Caltech101	87.2	70.6	82.3	**88.3**	86.2
CIFAR100	**54.4**	31.3	33.7	52.7	47.0
Flowers102	83.3	66.1	55.7	**85.3**	83.0
Pets	87.9	49.1	76.3	89.9	89.3
Sun397	33.3	13.9	27.5	**35.8**	32.5
SVHN	**70.4**	83.2	18.7	62.7	59.8
EuroSAT	**94.4**	88.7	78.9	92.2	91.6
Resics45	**76.1**	63.7	62.4	75.5	74.4
Patch Camelyon	**83.1**	81.5	75.6	79.3	80.9
Retinopathy	70.2	57.6	27.9	**74.3**	73.7
CLEVR-count	**74.0**	40.3	30.0	40.3	42.0
CLEVR-dist	51.5	**52.9**	37.1	38.9	37.3
dSprites-loc	82.7	**85.9**	30.0	45.3	38.1
dSprites-ori	**55.1**	46.4	19.8	42.5	41.4
SmallNORB-azi	17.8	**36.5**	12.9	15.7	15.1
SmallNORB-elev	**32.1**	31.2	18.1	22.7	21.0
DMLab	**43.2**	37.9	33.3	38.7	36.1
KITTI-dist	**79.9**	58.7	52.3	71.0	69.6
MetaDataset (all)	73.3	63.4	71.0	**76.1**	74.9
VTAB (all)	**65.4**	55.6	42.9	58.4	56.6
VTAB (natural)	**69.4**	52.4	49.0	69.1	66.3
VTAB (specialized)	**81.0**	72.9	61.2	80.3	80.1
VTAB (structured)	**54.5**	49.4	29.2	39.4	37.6

Table 11: Comparing UpperCaSE against meta-learning and hybrid methods. Best result in bold.

Model	ProtoNet	ProtoMAML	CTX	LITE	UpperCaSE	UpperCaSE
Image Size	224	126	224	224	224	224
Network	ENB0	RN18	RN34	ENB0	ENB0	RN50-S
Params (M)	4.0	11.2	21.3	4.0	4.0	23.5
Omniglot	88.3±0.8	**90.2±0.7**	84.6±0.9	86.5±0.8	**90.7±0.4**	89.1±0.5
Aircraft	85.0±0.7	82.1±0.6	85.3±0.8	83.6±0.7	**89.4±0.4**	87.5±0.4
Birds	**90.2±0.5**	73.4±0.9	72.9±1.1	88.6±0.7	**90.4±0.4**	89.6±0.4
DTD	81.4±0.6	66.3±0.8	77.3±0.7	**84.1±0.7**	83.4±0.4	**84.8±0.5**
QuickDraw	**76.0±0.7**	66.4±1.0	73.3±0.8	**75.7±0.8**	59.3±0.8	56.8±0.8
Fungi	57.4±1.1	46.3±1.1	48.0±1.2	56.9±1.2	**59.3±0.8**	56.8±0.8
Traffic Sign	53.5±1.1	50.3±1.1	**80.1±1.0**	65.8±1.1	68.5±0.8	70.6±0.8
MSCOCO	**49.8±1.1**	39.0±1.0	**51.4±1.1**	**50.0±1.0**	**50.8±0.7**	46.7±0.8
Caltech101	87.4	73.1	84.2	87.7	**88.3**	86.2
CIFAR100	43.1	29.7	37.5	48.8	**52.7**	47.0
Flowers102	78.2	60.2	81.8	83.5	**85.3**	83.0
Pets	88.6	56.6	70.9	89.3	**89.9**	89.3
Sun397	32.9	8.1	24.8	30.9	**35.8**	32.5
SVHN	35.2	46.8	**67.2**	51.0	62.7	59.8
EuroSAT	83.3	80.1	86.4	89.3	**92.2**	91.6
Resics45	68.8	53.5	67.7	**76.4**	75.5	74.4
Patch Camelyon	73.3	75.9	79.8	**81.4**	79.3	80.9
Retinopathy	31.3	73.2	35.5	40.3	**74.3**	73.7
CLEVR-count	27.2	32.7	27.9	31.4	40.3	**42.0**
CLEVR-dist	28.5	35.4	29.6	32.8	**38.9**	37.3
dSprites-loc	13.4	42.0	23.2	12.3	**45.3**	38.1
dSprites-ori	19.6	23.0	**46.9**	31.1	42.5	41.4
SmallNORB-azi	9.4	13.4	**37.0**	14.5	15.7	15.1
SmallNORB-elev	17.0	18.8	21.6	21.0	**22.7**	21.0
DMLab	35.8	32.5	31.9	**39.4**	38.7	36.1
KITTI-dist	56.5	54.4	54.3	63.9	**71.0**	69.6
MetaDataset (all)	72.7	64.2	71.6	73.9	**76.1**	74.9
VTAB (all)	46.1	45.0	50.5	51.4	**58.4**	56.6
VTAB (natural)	60.9	45.7	61.1	65.2	**69.1**	66.3
VTAB (specialized)	64.2	70.7	67.3	71.9	**80.3**	80.1
VTAB (structured)	25.9	31.5	34.1	30.8	**39.4**	37.6