This supplementary material document is structured as follows: In Section 1 we provide further detail about the training data used in the paper; In section 3 we provide details on the baselines used in the paper, their implementation details and the hyperparameters used for training; In Section 4 we provide empirical evidence about our choice of point-cloud encoding architecture; In Section 5 we provide further details about the training procedure of the shape-biased image embeddings used in the paper.

1. Further Dataset Details

In this section we provide details on the composition of the datasets used in the main paper. We provide example images used to illustrate the data used for training in Figure 1. As a result of using a ray tracing-based renderer Cycles [11], the synthetic image data used for training has high realism. For all algorithms we use $224 \times 224$ RGB images as input. For point cloud-based learning we use the 3D $(x, y, z)$ coordinates 1024 randomly sampled points as input. For images we use standard geometric data augmentations e.g., flipping, cropping, slight translation and rotation, as well as color jittering, since we found these result in improved validation performance. For point clouds we use the same augmentation procedures as in [18], which include translation, jittering and dropout.

1.1. Toys4K

We provide further details on the composition of our new Toys4K dataset in Table 1. The 40 train, 10 validation, and 55 test classes split is shown in Table 5. When performing validation and testing on Toys4K, we generate low-shot episodes consisting of up to 5 shots and 10 queries.

1.2. ModelNet40-LS

The 20 train, 10 validation, 10 test classes split for ModelNet40-LS is shown in Table 4. When performing validation and testing on ModelNet40-LS, we generate low-shot episodes consisting of up to 5 shots and 15 queries.

1.3. ShapeNet55-LS

The 25 train, 10 validation, 20 test classes split for ShapeNet55-LS is shown in Table 3. When performing validation and testing on ShapeNet55-LS, we generate low-shot episodes consisting of up to 5 shots and 15 queries.

2. Further Low-Shot Analysis

In this section we provide further analysis of the low-shot performance by presenting confusion matrices and classification performance in individual low-shot episodes.

2.1. Confusion Matrices

Please refer to Figure 2 and Figure 3 for low-shot confusion matrices on ModelNet40-LS and ShapeNet55-LS. The confusion matrices are obtained by evaluation 5K low-shot episodes for each dataset (10-way for ModelNet40-LS and 20-way for ShapeNet55-LS), and counting how each sample was classified. The confusion matrices reflect the results presented in Section 4 in the main text that adding shape bias improves overall low-shot classification performance.

2.2. Per-episode Analysis

We provide a per-episode analysis of low-shot classification in Figure 4 to show qualitative evidence of low-shot learning with shape bias. We see that there are cases in which even though there are no view ambiguities, the image-only model misclassifies whereas the shape-biased model correctly classifies (e.g., in the lower left episode, confusing bicycle for sheep).

3. Baseline Algorithm Details

All algorithms in this paper are implemented using PyTorch [9]. In this section we provide further detail about the baseline implementations and hyperparameters used for training.
3.1. SimpleShot

The implementation in our codebase for SimpleShot [16] is based on the code release by the authors in [1]. The authors report a 1-shot 5-way accuracy of 49.69(0.19) and a 5-shot 5-way accuracy of 66.92(0.17) on miniImageNet [15] with the Conv4 architecture. The reimplementation of SimpleShot in our codebase with the same dataset and architecture results in 1-shot 5-way accuracy of 50.60(0.34) and a 5-shot 5-way accuracy of 68.06(0.23).

In all our experiments we train SimpleShot with SGD with an initial learning rate of 0.01 and a learning rate decay of 0.1 at epochs 300 and 360, out of a total of 400 epochs. SimpleShot employs three different feature normalization strategies, no normalization, $L_2$ normalization and $L_2$ normalization and training set mean subtraction. In experiments with SimpleShot we report the result of the best of these three normalization strategies.

3.2. RFS

The implementation in our codebase for RFS [14] is based on the code release by the authors in [2]. The original codebase obtains a 1-shot 5-way accuracy of 53.73(0.81) on miniImageNet [15] with the Conv4 architecture. The reimplementation of RFS in our codebase with the same dataset and architecture results in 1-shot 5-way accuracy of 54.59(0.86). RFS requires training an embedding on the training dataset using cross-entropy. We train this embedding space with SGD using a learning rate of 0.001, momentum of 0.9 and $L_2$ weight penalty weight parameter of 0.0005. For each low-shot episode we train a logistic re-
gression classifier using Scikit-learn[10], as in the original RFS.

3.3. FEAT

The implementation for FEAT is based on the code release by the authors in [3]. The original codebase obtains a 1-shot 5-way accuracy of 54.85(0.20) and 5-shot 5-way accuracy of 71.61 on miniImageNet [15] with the Conv4 architecture. The reimplementation of FEAT in our codebase with the same dataset and architecture results in 1-shot 5-way accuracy of 54.85(0.20) 5-shot 5-way accuracy of 71.45(0.73). We train FEAT with the default hyperparameters recommended by the authors, training separate models for 5-way and 10-way classification, and separate models for 1-shot and 5-shot, as recommended by the authors. For the shape biased FEAT we do not use learning rate scheduling and momentum, since they have a negative effect on performance for shape-biased training. Removing them for image-only training does not affect performance.

3.4. Prototypical Networks

The implementation in our codebase for Prototypical Networks is based on the code release by the SimpleShot authors in [1]. In [6] the authors report that their reimplementation obtains a 5-shot 5-way accuracy of 66.68(0.68) on miniImageNet [15] with the Conv4 architecture. The reimplementation of Prototypical Networks in our codebase with the same dataset and architecture results in 5-shot 5-way accuracy of 66.94(0.71). We train separate Prototypical Networks models for 5-shot classification and 1-shot classification. As recommended by the original paper, we perform 20-way training. We use the Adam [8] optimizer, 400 low-shot iterations per epoch, 200 epochs total, and a learning rate of 0.0001 \( L_2 \) and weight penalty weight parameter of 0.00001. We perform a learning rate decay of 0.5 every 20 epochs.

3.5. Triplet Model

We implement a joint triplet model using both point cloud (DGCNN [17] and image (ResNet18 [7]) encoders, which can use both image and shape information during training. Let \( f_i \) denote the image encoder, \( f_p \) denote the point cloud encoder, and \( \phi^k_p \) and \( \phi^k_i \) denote the point cloud and image encodings of object instance \( k \) respectively. We learn a joint image/shape embedding by minimizing a standard triplet loss

\[
\mathcal{L}(\phi^k_i, \phi^k_p, \phi_p) = \max \{ d(\phi^k_i, \phi^k_p) - d(\phi^k_p, \phi_p) + \text{margin}, 0 \}
\]

\[
d(x, y) = ||x - y||_2
\]

where the anchor is an image embedding of instance \( k \), \( \phi^k_i \), the positive sample is a point cloud encoding of the same object instance, \( \phi^k_p \), and the negative sample is a \( \phi_p \) is a point cloud embedding of a different object instance \( l \). We perform \( L_2 \) normalization of the embeddings prior to computing the loss. Note that it is possible to build (anchor, positive, negative) pairs using category information, but we empirically found that this leads to worse performance.

We train the triplet model using the Adam [8] optimizer with a learning rate of 0.0001, \( L_2 \) weight penalty weight parameter of 0.0001, and margin of 0.1. We use a batch size of 72 and train for 600 epochs, each epoch consisting of 20K random samples.

4. Learning a Point Cloud Shape Embedding

In this section we describe the algorithm for learning a point-cloud based embedding space, and present an empirical study for our point cloud architecture choice.

4.1. Algorithm

The algorithm we use to train a point-cloud embedding space is based on SimpleShot [16] and is described with pseudocode in Algorithm 1. Note that the routine AccAccumulator is used denotes a function to collect the validation accuracy of each low-shot episode and compute summary statistics. The NNCLASSIFY routine takes support features and labels, and classifies each test query feature based on a nearest neighbors rule using cosine similarity. The point-cloud embedding model is trained using SGD with a learning rate of 0.01, batch size of 129, and \( L_2 \) weight penalty weight parameter of 0.0001. We perform learning rate decay by 0.1 at epochs 300 and 360. In all models we use features from the output of the pooling layer in the architecture.

4.2. Architecture Study

We perform an empirical study on the point cloud architectures to determine which is capable of the best low-shot generalization performance. Our PointNet [12] implementation is based on [4], our PointNet++ [13] is based on [18] and our DGCNN [17] implementation is based on [5]. We use a DGCNN architecture with a reduced embedding dimension (size after the pooling operation) of 512 rather than the original 1024, to match the dimensionality of the ResNet18 embeddings. We find no decrease in performance by this reduction. We present the results of this study on ModelNet in Table 2. The DGCNN [17] architecture outperforms other point cloud architectures at low-shot generalization to novel categories. We find that randomly rotating the input point cloud about the origin during training (random rotation about all axes of rotation, indicated by SO3 in the table) results in a performance improvement. We use this SO3 strategy for all shape-embedding space learning experiments.
Figure 2. Confusion matrices over 5K low-shot episodes of SimpleShot for Image Only, Shape-Biased without access to point clouds (w/o pc) at test time and Shape-Biased with (w/ pc) access to point clouds at test time on the ModelNet-LS dataset. Even without access to point clouds (w/o pc) for building class prototypes, the shape-biased image embedding leads to improvements. Adding point cloud support information (w/ pc) improves performance further. See Table 3 in the main text for aggregate results.

Figure 3. Confusion matrices over 5K low-shot episodes of SimpleShot for Image Only, Shape-Biased without access to point clouds (w/o pc) at test time and Shape-Biased with (w/ pc) access to point clouds at test time on the ShapeNet-LS dataset. As in ModelNet40-LS, without access to point clouds (w/o pc) for building class prototypes, the shape-biased image embedding leads to improvements. Adding point cloud support information (w/ pc) improves performance further. See Table 5 in the main text for aggregate results. Best viewed with zoom.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>1-shot 5-way accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointNet [12]</td>
<td>66.13</td>
</tr>
<tr>
<td>PointNet++ [13]</td>
<td>67.49</td>
</tr>
<tr>
<td>DGCNN [17]</td>
<td>75.2</td>
</tr>
<tr>
<td>DGCNN (SO3)</td>
<td>77.5</td>
</tr>
</tbody>
</table>

Table 2. Empirical study for choosing the best point cloud architecture. Reported is 1-shot 5-way classification accuracy on the ModelNet40-LS validation set. We find that DGCNN performs the best, and that randomly rotating each input point cloud during training (indicated with SO3) results in a improvement in low-shot generalization performance as well.

5. Details for Learning a Shape Biased Image Embedding

The algorithm we use to train a shape-biased image embedding is described with pseudocode in Algorithm 2. We use the Adam optimizer with a batch size of 256, an initial learning rate of 0.001 and a $L_2$ weight penalty weight parameter of 0.0001. The model is trained for 400 epochs, with a learning rate decay of 0.1 at epochs 300 and 360.

5.1. SimpleShot with Shape Bias

The SimpleShot [16] approach does not require any learning (parameter updates) during the low-shot phase.
Classification is done using nearest centroid classification in the embedding space. The image embedding function $f_i$ is trained as described in Algorithm 2, and low-shot testing is done following the same procedure as described in L8-16 in Algorithm 1 but using nearest centroid rather than nearest neighbor classification.

5.2. FEAT with Shape Bias

The algorithm we use to train a shape-biased FEAT [19] architecture is described in Algorithm 3. Note that the $f_i$ used in this algorithm is being fine tuned from a mapping already trained with Algorithm 2 while the FEAT set-to-set function $E$ is trained from scratch. For this experiment we use the default hyperparameters recommended by the FEAT authors. Low shot testing is done following the same procedure as described in L13-22 in Algorithm 3 but using the test set. The procedure we refer to as FEATCLASSIFY is described in Eq. 4 on pg. 4 of the FEAT paper [19]. In the pseudocode FEATCLASSIFY performs classification and directly outputs the per-episode classification accuracy.
Algorithm 1: Training Shape Embedding $f_p$

**Input:** Randomly initialized point-cloud classifier architecture $f_p$ with embedding function $f_p^E$
Total number of epochs $N_e$
Total number of mini-batches per epoch $N_b$
Total number of low-shot iterations for validation $N_{it}$

**Data:** (point cloud, label) pair datasets $D^{train}$, $D^{val}$

**Define:** $\ell :$ cross-entropy loss

1. **foreach** epoch in $1, 2, \ldots, N_e$ do
   2. **foreach** mini-batch $(o_p, y) \sim D^{train}$ of $N_b$ do
      3. Predict $\hat{y} = f_p(o_p)$
      4. Compute $\ell(y, \hat{y})$
      5. Compute $\nabla \ell$ with respect to $f_p$
      6. Update $f_p$ with SGD
   end
   A = ACCUMULATOR

3. **foreach** validation episode in $1, 2, \ldots, N_e$ do
   4. Sample 5-way 1-shot $(o_p^{train}, y^{train}, o_p^{test}) \sim D^{val}$
   5. Predict $\hat{o}_p^{train} = f_p^E(o_p^{train})$
   6. Predict $\hat{o}_p^{test} = f_p^E(o_p^{test})$
   7. $\text{acc} = \text{NNCLASSIFY}((\hat{o}_p^{train}, y^{train}, \hat{o}_p^{test}))$
   8. A(acc)
end

10. val accuracy = A.average()
11. if val accuracy $>$ best accuracy then
12.   best accuracy $\leftarrow$ val accuracy
13.   $f_p^{best} \leftarrow f_p$
end
result: Trained $f_p^{best}$

Algorithm 2: Training Shape-Biased Image Embedding Function $f_i$

**Input:** Randomly initialized image embedding architecture $f_i$
Point-cloud embedding function $f_p$ (1)
Total number of epochs $N_e$
Total number of mini-batches per epoch $N_b$
Total number of low-shot iterations for validation $N_{it}$

**Data:** (image, point cloud, label) pair datasets $D^{train}$, $D^{val}$

**Define:** $\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2$ (see main text for def.)

1. **foreach** epoch in $1, 2, \ldots, N_e$ do
2. **foreach** mini-batch $(o_i, o_p, y) \sim D^{train}$ of $N_b$ do
3. Predict shape embedding $\phi_p = f_p(o_p)$
4. Predict image embedding $\phi_i = f_i(o_i)$
5. Compute $\mathcal{L}$ using $\phi_p$ and $\phi_i$
6. Compute $\nabla \mathcal{L}$ with respect to $f_i$
7. Update $f_i$ with Adam
end
A = ACCUMULATOR

2. **foreach** validation episode in $1, 2, \ldots, N_e$ do
3. Sample 5-way 1-shot $(o_i^{train}, o_p^{train}, y^{train}, o_i^{test}) \sim D^{val}$
4. Predict $\hat{\phi}_i^{train} = f_i(o_i^{train})$
5. Predict $\hat{\phi}_i^{test} = f_i(o_i^{test})$
6. Predict $\hat{\phi}_i^{test} = f_i(o_i^{test})$
7. $\text{acc} = \text{NNCLASSIFY}(\hat{\phi}_i^{train}, y^{train}, \hat{\phi}_i^{test})$
8. A(acc)
end

10. val accuracy = A.average()
11. if val accuracy $>$ best accuracy then
12.   best accuracy $\leftarrow$ val accuracy
13.   $f_i^{best} \leftarrow f_i$
end
result: Trained $f_i^{best}$
Algorithm 3: Training FEAT with Shape Bias

**Input:** Shape-biased image encoder $f_i$ (2)
- Point-cloud embedding function $f_p$ (1)
- Randomly initialized FEAT [19] set-to-set function $E$—see p3 in [19].
- Total number of epochs $N_e$
- Total number of low-shot iterations per training epoch $N_{it}$
- Total number of low-shot iterations for validation $N_{v-it}$

**Data:** (image, point cloud, label) pair datasets $\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{val}}$

Define: $\mathcal{L}_{\text{FEAT}}$ – Eq. 7 in [19]

```plaintext
foreach epoch in 1, 2, ..., $N_e$ do
    foreach training episode in of 1, 2, ..., $N_{it}$ do
        Sample $m$-way $n$-shot
        $\langle \text{o}^{\text{train}}_p, \text{o}^{\text{train}}_i, \text{o}^\text{query}_p, \text{y}^{\text{train}}, \text{y}^\text{query} \rangle \sim \mathcal{D}_{\text{train}}$
        Predict ptcl. support $\phi^\text{train}_p = f_p(\text{o}^{\text{train}}_p)$
        Predict image support $\phi^\text{train}_i = f_i(\text{o}^{\text{train}}_i)$
        Predict image queries $\phi^\text{query}_i = f_i(\text{o}^{\text{query}}_i)$
        $\phi^\text{train} \leftarrow \text{AVERAGE}(\phi^\text{train}_p, \phi^\text{train}_i)$
        $\phi^\text{train}, \phi^\text{query} \leftarrow E(\phi^\text{train}, \phi^\text{query})$
        Compute $\mathcal{L}$ using $\hat{\phi}^\text{train}, \hat{\phi}^\text{query}$ and
        $\text{y}^{\text{train}}, \text{y}^\text{query}$
        Compute $\nabla \mathcal{L}$ with respect to $f_i$ and $E$
        Update $f_i, E$ with SGD
    end
    $A = \text{ACCUMULATOR}$
    foreach validation episode in 1, 2, ..., $N_{v-it}$ do
        Sample 5-way 1-shot
        $\langle \text{o}^{\text{train}}_i, \text{o}^{\text{train}}_i, \text{y}^{\text{train}}, \text{y}^\text{test} \rangle \sim \mathcal{D}_{\text{val}}$
        Predict ptcl. support $\phi^\text{train}_p = f_p(\text{o}^{\text{train}}_p)$
        Predict image support $\phi^\text{train}_i = f_i(\text{o}^{\text{train}}_i)$
        Predict image queries $\phi^\text{test}_i = f_i(\text{o}^{\text{test}}_i)$
        $\phi^\text{train} \leftarrow \text{AVERAGE}(\phi^\text{train}_p, \phi^\text{train}_i)$
        acc = FEATCLASSIFY($\phi^\text{train}, \text{y}^{\text{train}}, \phi^\text{test}_i$)
        $A(\text{acc})$
    end
    val accuracy = $A$.average()
    if val accuracy > best accuracy then
        best accuracy $\leftarrow$ val accuracy
        $f_{i_{\text{best}}} \leftarrow f_i$
        $E_{\text{best}} \leftarrow E$
    end
end
```

**Result:** Trained $f_{i_{\text{best}}}, E_{\text{best}}$
<table>
<thead>
<tr>
<th>Training # samples</th>
<th>Validation # samples</th>
<th>Testing # samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>vessel 873</td>
<td>train 389</td>
<td>mug 214</td>
</tr>
<tr>
<td>car 530</td>
<td>bed 233</td>
<td>tower 133</td>
</tr>
<tr>
<td>sofa 500</td>
<td>stove 218</td>
<td>motorcycle 337</td>
</tr>
<tr>
<td>lamp 500</td>
<td>bowl 186</td>
<td>cap 56</td>
</tr>
<tr>
<td>cell</td>
<td>pillow 96</td>
<td>pistol 307</td>
</tr>
<tr>
<td>faucet 500</td>
<td>mailbox 94</td>
<td>earphone 73</td>
</tr>
<tr>
<td>pot 500</td>
<td>rocket 85</td>
<td>skateboard 152</td>
</tr>
<tr>
<td>guitar 500</td>
<td>birdhouse 73</td>
<td>camera 113</td>
</tr>
<tr>
<td>airplane 500</td>
<td>microphone 67</td>
<td>piano 239</td>
</tr>
<tr>
<td>bus 500</td>
<td>keyboard 65</td>
<td>printer 166</td>
</tr>
<tr>
<td>chair 500</td>
<td>bag 83</td>
<td>trashcan 343</td>
</tr>
<tr>
<td>rifle 500</td>
<td>file 298</td>
<td>dishwasher 93</td>
</tr>
<tr>
<td>cabinet 499</td>
<td>bathtub 499</td>
<td>microwave 152</td>
</tr>
<tr>
<td>telephone 495</td>
<td>jar 499</td>
<td>remote 66</td>
</tr>
<tr>
<td>bottle 498</td>
<td>display 496</td>
<td>basket 113</td>
</tr>
<tr>
<td>clock 496</td>
<td>loudspeaker 496</td>
<td>can 108</td>
</tr>
<tr>
<td>table 495</td>
<td>laptop 460</td>
<td>keyboard 63</td>
</tr>
<tr>
<td>bookshelf 452</td>
<td>knife 423</td>
<td>Total 25 classes 12716 10 classes 1506 20 classes 3377</td>
</tr>
</tbody>
</table>

Table 3. Split composition of ShapeNet55-LS

<table>
<thead>
<tr>
<th>Training # samples</th>
<th>Validation # samples</th>
<th>Testing # samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>candy 56</td>
<td>airplane 35</td>
<td>boat 38</td>
</tr>
<tr>
<td>flower 54</td>
<td>shark 30</td>
<td>lion 17</td>
</tr>
<tr>
<td>dragon 43</td>
<td>truck 34</td>
<td>whale 41</td>
</tr>
<tr>
<td>apple 54</td>
<td>phone 23</td>
<td>cupcake 28</td>
</tr>
<tr>
<td>guitar 55</td>
<td>giraffe 15</td>
<td>train 22</td>
</tr>
<tr>
<td>tree 57</td>
<td>horse 37</td>
<td>pizza 26</td>
</tr>
<tr>
<td>glass 63</td>
<td>fish 37</td>
<td>marker 19</td>
</tr>
<tr>
<td>cup 60</td>
<td>fan 31</td>
<td>cookie 28</td>
</tr>
<tr>
<td>pig 41</td>
<td>shoe 41</td>
<td>sandwich 15</td>
</tr>
<tr>
<td>cat 79</td>
<td>snake 32</td>
<td>octopus 31</td>
</tr>
</tbody>
</table>

Table 4. Split composition of ModelNet40-LS

<table>
<thead>
<tr>
<th>Training # samples</th>
<th>Validation # samples</th>
<th>Testing # samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>bed 615</td>
<td>cup 99</td>
<td>range hood 213</td>
</tr>
<tr>
<td>car 297</td>
<td>xbox 123</td>
<td>bowl 84</td>
</tr>
<tr>
<td>guitar 255</td>
<td>bathtub 156</td>
<td>stool 110</td>
</tr>
<tr>
<td>bottle 435</td>
<td>cone 187</td>
<td>radio 124</td>
</tr>
<tr>
<td>desk 286</td>
<td>curtain 158</td>
<td>stairs 144</td>
</tr>
<tr>
<td>night stand 286</td>
<td>door 129</td>
<td>lamp 144</td>
</tr>
<tr>
<td>glass box 271</td>
<td>flower pot 169</td>
<td>tent 183</td>
</tr>
<tr>
<td>sofa 780</td>
<td>person 108</td>
<td>sink 148</td>
</tr>
<tr>
<td>piano 331</td>
<td>wardrobe 107</td>
<td>bench 193</td>
</tr>
<tr>
<td>toilet 444</td>
<td>keyboard 165</td>
<td>laptop 169</td>
</tr>
<tr>
<td>total 565</td>
<td>table 492</td>
<td>Total 40 classes 2506 10 classes 315 55 classes 1358</td>
</tr>
</tbody>
</table>

Table 5. Split composition of Toys4K
Appendix References