A APPENDIX

A.1 IMPLEMENTATION DETAIL

Our experiments are conducted on ModelNet 3D Wharehouse, S3DIS, and SemanticKITTI dataset. For the ModelNet 3D Wharehouse dataset, we train all models on the train set and evaluate on the validation set. For the S3DIS, we train all models on area 1, 2, 3, 4, 6 and evaluate on area 5. For the SemanticKITTI dataset, we train all models on splits 00-10 except 08 which is used for evaluation. For each of the dataset, all ResNet-series models use the same training scheme, and all experiments are implemented with Pytorch.

When training on the *ModelNet 3D Wharehouse dataset*, coordinates of point-cloud are randomly scaled, translated, and jittered. We use SGD optimizer with momentum 0.9, weight-decay 10^{-4}, and initial learning rate 0.1 with cosine learning rate scheduler. Each mini batch is set to 32, and models are trained for 300 epochs. For both training and inference phase, we only utilize x, y, z coordinates without other features and set voxel size to 0.05. The experiments for ModelNet 3D Wharehouse are all conducted on a Titan RTX GPU.

When training on the *S3DIS dataset*, we concatenate all subparts of an indoor scene to train and validate on. Along x, y directions, scenes are randomly applied horizontal flip. RGB features are randomly jittered, translated, and auto contrasted. Finally, we normalize and clip point-clouds. We set voxel size to 0.05. We use SGD optimizer with momentum 0.9, weight-decay 10^{-4}, and initial learning rate 0.1 with polynomial learning rate scheduler. Each mini batch is set to 3, and models are trained for 400 epochs on 2 Titan RTX GPUs.

When training on the *SemanticKITTI dataset*, coordinates of each point-cloud are randomly scaled and rotated. We use SGD optimizer with momentum 0.9, weight-decay 10^{-4}, and initial learning rate 0.24 with cosine warmup learning rate scheduler. Each mini batch is set to 2, and models are trained for 15 epochs on 4 Titan RTX GPUs. For both training and inference phase, we utilize x, y, z coordinates as well as intensity feature and set voxel size to 0.05.

Most of our pretrained models come from open-sources, [1] [2] [3] [4] [5] [6] , so we do not need to take time and computational resources for pretraining. We use torchsparse [7] to produce sparse 3D convolutions.

Details of Section 4.1. In this section, we take the ResNet architecture, inflate the pretrained models of different image dataset, and add linear input and output layers as shown in Section A.5. The ResNet50 pretrained on ImageNet1K is directly taken from Pytorch. We use the same training recipe provided by Pytorch to train the ResNet50 on Tiny-ImageNet. The pretrained ResNet50 on ImageNet21K comes from (Ridnik et al., 2021).

Details of Section 4.2, Section 4.3 and 4.4. In this section, the ResNet pretrained models are taken from the same sources as illustrated above.

For PointNet++ pretraining on ImageNet1K, we utilize the PointNet++ SSG version (Qi et al., 2017). We break the images into pixels, and regard the pixels as a point-cloud, coordinates of which are the x, y position in original image, with appending $z = 1$ to all pixels. Then we set center sampling number to 1024 and 256 for first and second stage, the radius are set into 8 and 64 for them respectively. For each center point, we will query 64 neighbouring points. The training recipe is also provided by Pytorch.

For ViT models, we directly take the pretrained weights from Dosovitskiy et al. (2020). To apply it on ModelNet 3D Wharehouse, we sample 256 centers and group 64 nearby points, regarding these as "point-cloud patches". Then we use a linear embedding to project the point-cloud patches into a sequence, and the ViT process them as same as the images. Except the linear embedding and the final

[1] https://pytorch.org/vision/stable/models.html
[2] https://github.com/Alibaba-MIIL/ImageNet21K
[3] https://github.com/hirokatsukataoka16/FractalDB-Pretrained-ResNet-PyTorch
[4] https://github.com/HRNet/HRNet-Semantic-Segmentation/tree/pytorch-v1.1
[5] https://github.com/rgeirhos/Stylized-ImageNet
[6] https://github.com/wielandbrendel/bag-of-local-features-models
[7] https://github.com/mit-han-lab/torchsparse

Table 6: ResNet50 results (evaluated on ModelNet 3D Wharehouse) of fintuning the mean and variance in batch normalization layers on different datasets. IO indicates input and output layer, IOms indicate input, output, mean and variance, IOmsWb indicates input, output and the whole BN.

Layers	Tiny-ImageNet	ImageNet1K	ImageNet21K	FractalDB1K	FractalDB10K
IO	67.666	81.199	73.744	83.347	80.105
IOms	83.793	82.942	84.076	72.326	79.66
IOmsWb	89.992	89.87	90.721	89.263	89.344
From scratch	90.316	90.316	90.316	90.316	90.316

Table 7: ResNet18, 50, 152 results (evaluated on ModelNet 3D Wharehouse) of finetuing the mean and variance in batch normalization layers.

Layers	ResNet18	ResNet50	ResNet152
IO	71.029	81.199	64.627
IOmv	81.888	82.942	82.658
IOmvWb	88.574	89.87	90.438
From Scratch	90.397	90.316	90.276

output classifier, all the models are kept same as the origin version. For the experiments on S3DIS and SemanticKITTI, the architecture detail of ResNet18 is shown in A.5 listing 2.

For SimpleView model, all the experiment settings are as same as Goyal et al. (2021a). The only difference is to use or not use the pretrained ResNet18. For HRNetV2-W48, we directly use the ImageNet1K pretrained model and Cityscape pretrained model from (Sun et al., 2019).

We conduct three trials on the few-shot experiments. For each trial, we change the random seed but keep all the others are same. To plot the training speed curve, we directly use the training log without any other change like smoothing.

A.2 Finetuning the mean and variance of batch normalization.

For the first group of experiments, ResNet50 FIP either has IO or IO+BN finetuned. In addition to these two experimental setting, we also investigate fintuning input, output layers, and mean, variance of normalization layers, while fixing the convolution layer weights, normalization layer weights and bias. The full experiments result with this extra setting is reported in Table 6 and 7. We can observe that compared with only finetuning input and output layers, updating mean and variance can also largely improve the performance of point-cloud recognition. As suggested in Section 5.2 in the main paper, updating mean and variance is also to push the finetuned pretrained image model to generate point-cloud representation as similar as image representation.

A.3 More visualization of Network dissection.

We present more visualization results based on the technique of network dissection, as shown in Figure 5. We can observe that for most cases, although there is no obvious visual concept transferring, the pretrained filters are prone to cluster similar objects on ModelNet 3D Wharehouse dataset.

A.4 Details of First-Wasserstein Distance on ResNet18.

We list all the results in each layer of ResNet18, as shown in Table 8. We can observe that for each layer, the first-wasserstein distance is largely reduced when finetuning more parameters. Interestingly, we also find that the distance between training-from-scratch on ModelNet 3D Wharehouse and image models are not too far, which may indicate the image and point-cloud could be represented similar even no transferring.

Table 8: First-wasserstein distance between distance between different features of all 16 layers in ResNet18

Image Model	FIP/IO	FIP/IO-BN	FIP/ALL	Training from scratch
Average	2.1×10^2	0.27	0.093	0.15
Layer 1	1.9	0.2	0.094	0.12
Layer 2	1.1	0.064	0.11	0.048
Layer 3	1.3	0.24	0.068	0.095
Layer 4	2.6	0.14	0.051	0.077
Layer 5	1.1	0.23	0.039	0.082
Layer 6	3	0.21	0.025	0.063
Layer 7	5.4	0.29	0.05	0.15
Layer 8	13	0.26	0.051	0.097
Layer 9	12	0.28	0.04	0.095
Layer 10	26	0.2	0.077	0.13
Layer 11	54	0.27	0.032	0.13
Layer 12	2.2×10^2	0.25	0.076	0.094
Layer 13	1.9×10^2	0.29	0.028	0.11
Layer 14	7.7×10^2	0.16	0.075	0.041
Layer 15	9.5×10^2	0.29	0.081	0.088
Layer 16	1.2×10^3	0.89	0.59	1

A.5 Details of used architectures

```
Class 3DRes_cls(nn.Module):
    def __init__(self, res_block):
        super().__init__() # res_block means the residual block as same as the conventional ResNet.
        self.input_layer = nn.Sequential(
            sparse_conv3d(input_dim, layer1_Idim, k=3, s=1),
            sparse_bn(layer1_Idim))

        self.layer1 = inflated_resnet_layer1(res_block, layer1_Idim, layer1_Odim)
        self.layer2 = inflated_resnet_layer2(res_block, layer2_Idim, layer2_Odim)
        self.layer3 = inflated_resnet_layer3(res_block, layer3_Idim, layer3_Odim)
        self.layer4 = inflated_resnet_layer4(res_block, layer4_Idim, layer4_Odim)

        self.output_layer = nn.Sequential(
            global_average_pooling,
            nn.Linear(layer4_Odim, class_num),
            nn.bn(class_num))

    def forward(self, x):
        x = self.input_layer(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
```

```
        return self.output_layer(x)
```

Listing 1: Pseudo code of inflated ResNet with linear input and output for classification

```
Class 3DRes_seg(nn.Module):
    def __init__(self, res_block):
        super().__init__() # res_block means the residual block as same as the conventional ResNet.
        self.input_layer = nn.Sequential(
            sparse_conv3d(input_dim, layer1_Idim, k=3, s=1),
            sparse_bn(layer1_Idim),
            sparse_ReLU(True),
            sparse_conv3d(layer1_Idim, layer1_Idim, k=3, s=1),
            sparse_bn(layer1_Idim),
            sparse_ReLU(True),
            sparse_conv3d(layer1_Idim, layer1_Idim, k=3, s=2),
            sparse_bn(layer1_Idim),
            sparse_ReLU(True))

        self.layer1 = inflated_resnet_layer1(res_block, layer1_Idim, layer1_Odim)
        self.layer2 = inflated_resnet_layer2(res_block, layer2_Idim, layer2_Odim)
        self.layer3 = inflated_resnet_layer3(res_block, layer3_Idim, layer3_Odim)
        self.layer4 = inflated_resnet_layer4(res_block, layer4_Idim, layer4_Odim)

        self.up1 = sparse_deconv(layer4_Odim, layer4_Odim, k=2, s=2),
        self.decode1 = self.Sequential(
            res_block(layer4_Odim+layer3_Odim, layer3_Odim),
            res_block(layer3_Odim, layer3_Odim))

        self.up2 = sparse_deconv(layer3_Odim, layer3_Odim, k=2, s=2)
        self.decode2 = self.Sequential(
            res_block(layer3_Odim+layer2_Odim, layer2_Odim),
            res_block(layer2_Odim, layer2_Odim))

        self.up3 = sparse_deconv(layer2_Odim, layer2_Odim, k=2, s=2)
        self.decode3 = self.Sequential(
            res_block(layer2_Odim+layer1_Odim, layer1_Odim),
            res_block(layer1_Odim, layer1_Odim))

        self.up4 = sparse_deconv(layer1_Odim, layer1_Odim, k=2, s=2)
        self.decode4 = self.Sequential(
            res_block(layer1_Odim+layer1_Odim, layer1_Odim),
            res_block(layer1_Odim, layer1_Odim))

        self.output_layer = nn.Sequential(
            nn.Linear(layer1_Odim, class_num))

    def forward(self, x):
        x_i = self.input_layer(x)
        x1 = self.layer1(x_i)
        x2 = self.layer2(x1)
        x3 = self.layer3(x2)
        x4 = self.layer4(x3)

        x3_ = self.decoder1(cat(x3, self.up1(x4)))
        x2_ = self.decoder2(cat(x2, self.up2(x3_)))
        x1_ = self.decoder3(cat(x1, self.up3(x2_)))
        xi_ = self.decoder4(cat(x_i, self.up4(x1_)))
        return self.output_layer(xi_)
```

Listing 2: Pseudo code of inflated ResNet for segmentation

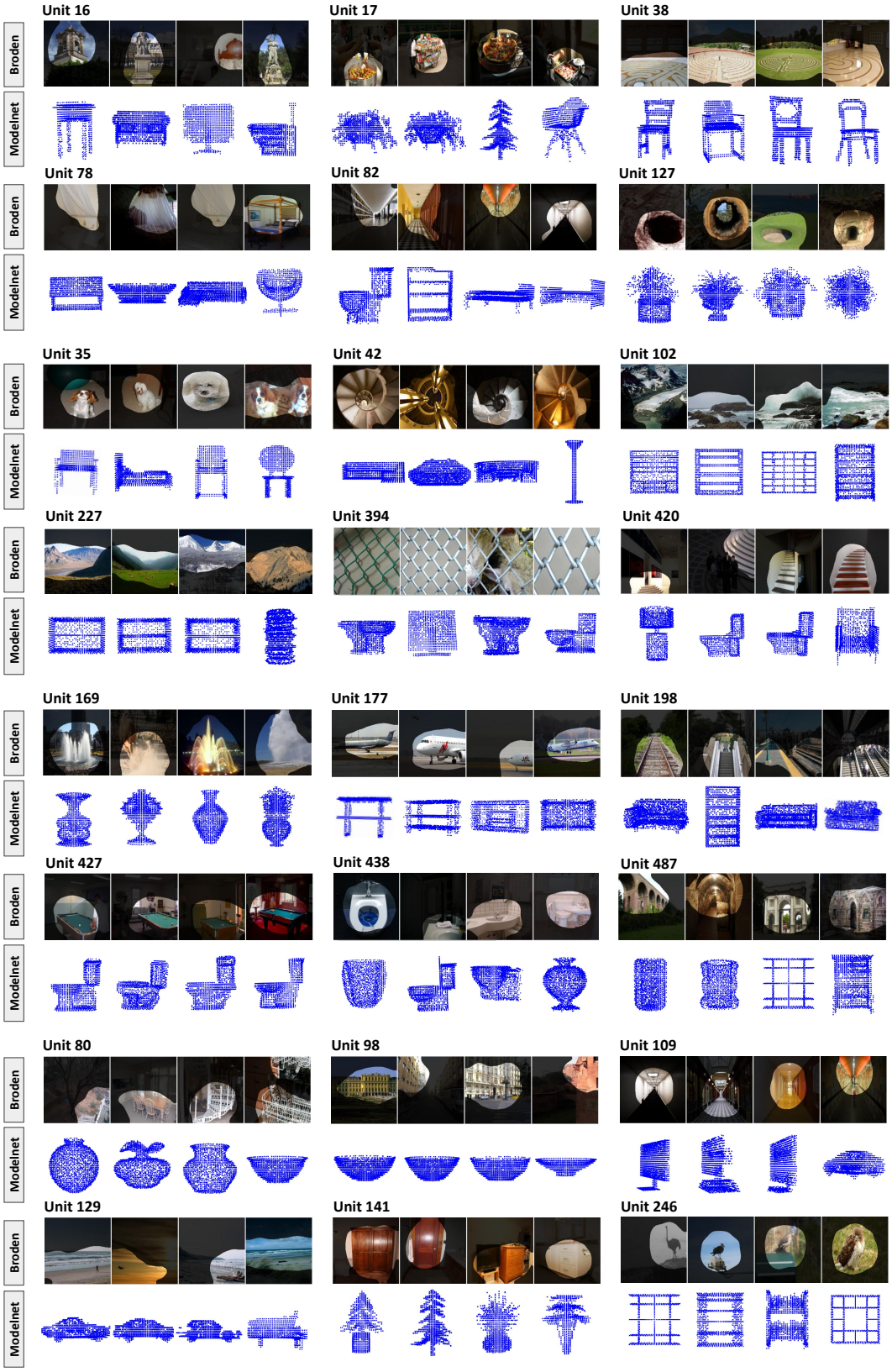

Figure 5: More network dissection results.