Appendix

The appendix is organized as follows:

- In Sec. A1, we provide the complete proof of Claim 4.
- In Sec. A2, we provide an additional ablation study and qualitative experimental results.
- In Sec. A3, we provide an overview of the attached code and showcase the ease of incorporating LPS into a deep-net to achieve a shift-invariance/equivariance network. Unit tests and end-to-end tests empirically validating the theory are also provided.
- In Sec. A4, we provide additional experimental details, *e.g.*, model architecture, number of trainable parameters, hyperparameters and baseline implementations.

A1 Proof of Claim 4

Claim 4. *If p_θ is shift-permutation equivariant, as defined in Eq. (11), then LPU $\circ$ LPD is shift-equivariant.*

Proof. Let $\boldsymbol{x}$ be a feature map, $\hat{\boldsymbol{x}} \triangleq \boldsymbol{T}_N \boldsymbol{x}$ its shifted version, and $k^\star = \arg\max_{k\in\{0,1\}} p_\theta(\boldsymbol{k}=k|\boldsymbol{x})$. By definition, $\boldsymbol{u} \triangleq LPU \circ LPD(\boldsymbol{x})$ can be seen as masking out the components beside $k^\star$ from $\boldsymbol{x}$:

$$\text{Poly}(\boldsymbol{u})_j = \begin{cases} \text{Poly}(\boldsymbol{x})_j, & j = k^\star \\ \boldsymbol{0}, & \text{otherwise.} \end{cases} \tag{28}$$

Let $\hat{\boldsymbol{u}} \triangleq \text{LPU} \circ \text{LPD}(\hat{\boldsymbol{x}})$ and $\hat{k}^\star = \arg\max_{k\in\{0,1\}} p_\theta(\boldsymbol{k}=k|\hat{\boldsymbol{x}})$ then

$$\text{Poly}(\hat{\boldsymbol{u}})_j = \begin{cases} \text{Poly}(\hat{\boldsymbol{x}})_j, & j = \hat{k}^\star \\ \boldsymbol{0}, & \text{otherwise.} \end{cases} \tag{29}$$

Using Lemma 1 on $\hat{\boldsymbol{x}}$, when $j=0$

$$\text{Poly}(\hat{\boldsymbol{u}})_j = \begin{cases} \text{Poly}(\boldsymbol{x})_{\pi(j)}, & \pi(j) = \hat{k}^\star \\ \boldsymbol{0}, & \text{otherwise.} \end{cases} \tag{30}$$

As p_θ is shift-permutation equivariant, therefore $\hat{k}^\star = \pi(k^\star)$. Substituting into Eq. (30),

$$\text{Poly}(\hat{\boldsymbol{u}})_j = \begin{cases} \text{Poly}(\boldsymbol{x})_{\pi(j)}, & \pi(j) = \pi(k^\star) \\ \boldsymbol{0}, & \text{otherwise.} \end{cases} \tag{31}$$

Similarly, when $j=1$ and let $M = \lfloor N/2 \rfloor$

$$\boldsymbol{T}_M\text{Poly}(\hat{\boldsymbol{u}})_j = \begin{cases} \boldsymbol{T}_M\text{Poly}(\boldsymbol{x})_{\pi(j)}, & \pi(j) = \pi(k^\star) \\ \boldsymbol{0}, & \text{otherwise.} \end{cases} \tag{32}$$

Finally, combining Eq. (31) and Eq. (32) then using Lemma 1 on $\hat{\boldsymbol{u}}$,

$$\boldsymbol{T}_N \boldsymbol{u} = \text{LPU} \circ \text{LPD}(\boldsymbol{T}_N \boldsymbol{x}), \tag{33}$$

which proves the claim. □

A2 Additional Experimental Results

Ablation Study on Gumbel Softmax. We further analyze the effect of sampling with Gumbel-Softmax during training. We compare LPS (ResNet-18) results using a 3-tap antialiasing filter (Tri-3) and trained using a Gumbel-Softmax, against its alternative version trained using a standard Softmax without sampling. The remaining training attributes, including the τ annealing schedule, remain unaltered. Tab. A1 compares both scenarios. While perfect circular shift consistency is obtained by design in both cases, the top-1 classification accuracy of the model trained without sampling (93.22%)

Table A1. Ablation study on the effect of Gumbel-softmax sampling.

Method	Anti-Alias	Sampling	Acc. ↑	C-Cons. ↑
LPS (Softmax)	Tri-3	✗	93.22 ± 0.13	$\mathbf{100 \pm 0.0}$
LPS (Gumbel-softmax)	Tri-3	✓	$\mathbf{94.8 \pm 0.14}$	$\mathbf{100 \pm 0.0}$

is significantly lower than our proposed training approach (94.8%). By incorporating stochasticity, the model performance improves.

LPS Filter Visualization. We provide visualizations of the convolution weights. Fig. A1 shows a subset of the convolutional kernels used to select polyphase components at each layer of a ResNet-50 (LPS).

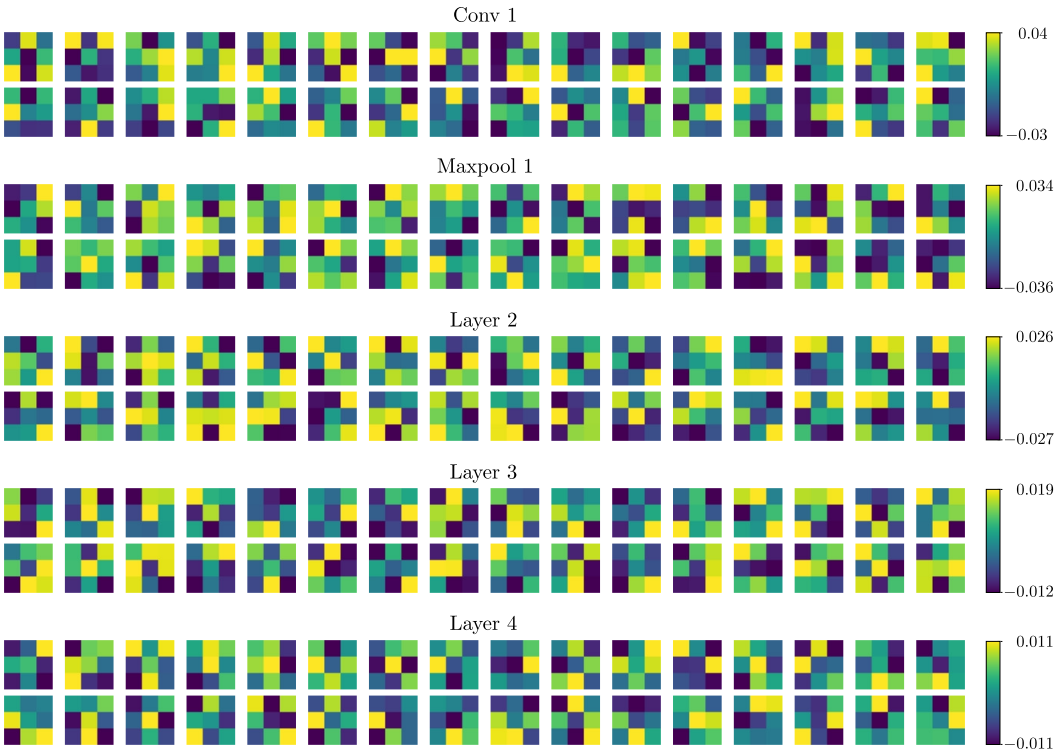

Figure A1. Convolutional weights learned by LPD layers at different ResNet-50 (ImageNet) pooling levels.

Qualitative Results for Semantic Segmentation. In Fig. A2 to Fig. A4, we provide a sampling of output masks on PASCAL VOC that were predicted by DDAC and our proposed LPS on linearly shifted inputs. Identical random shifts were applied for both DDAC and LPS. We observe that LPS predicts smoother object contours and maintains a better consistency across shifts when compared to DDAC. Regions where our shift consistency property showed significantly different results in comparison to baseline are highlighted by a red circle.

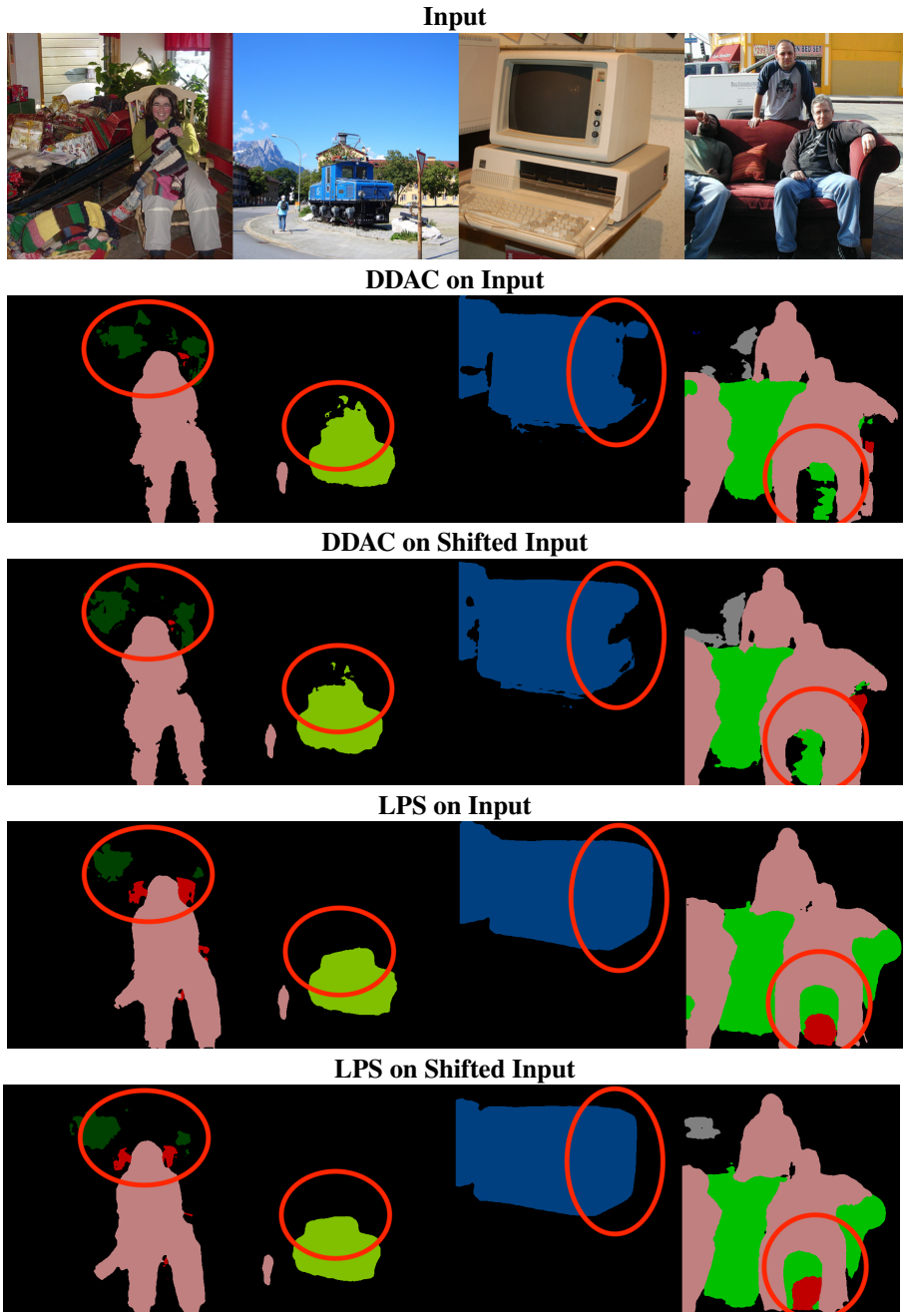

Figure A2. Qualitative comparison on PASCAL VOC ResNet-101 with DeepLabV3 architecture. Regions where our proposed network showed significant improvements under linear shifts are highlighted with a red circle.

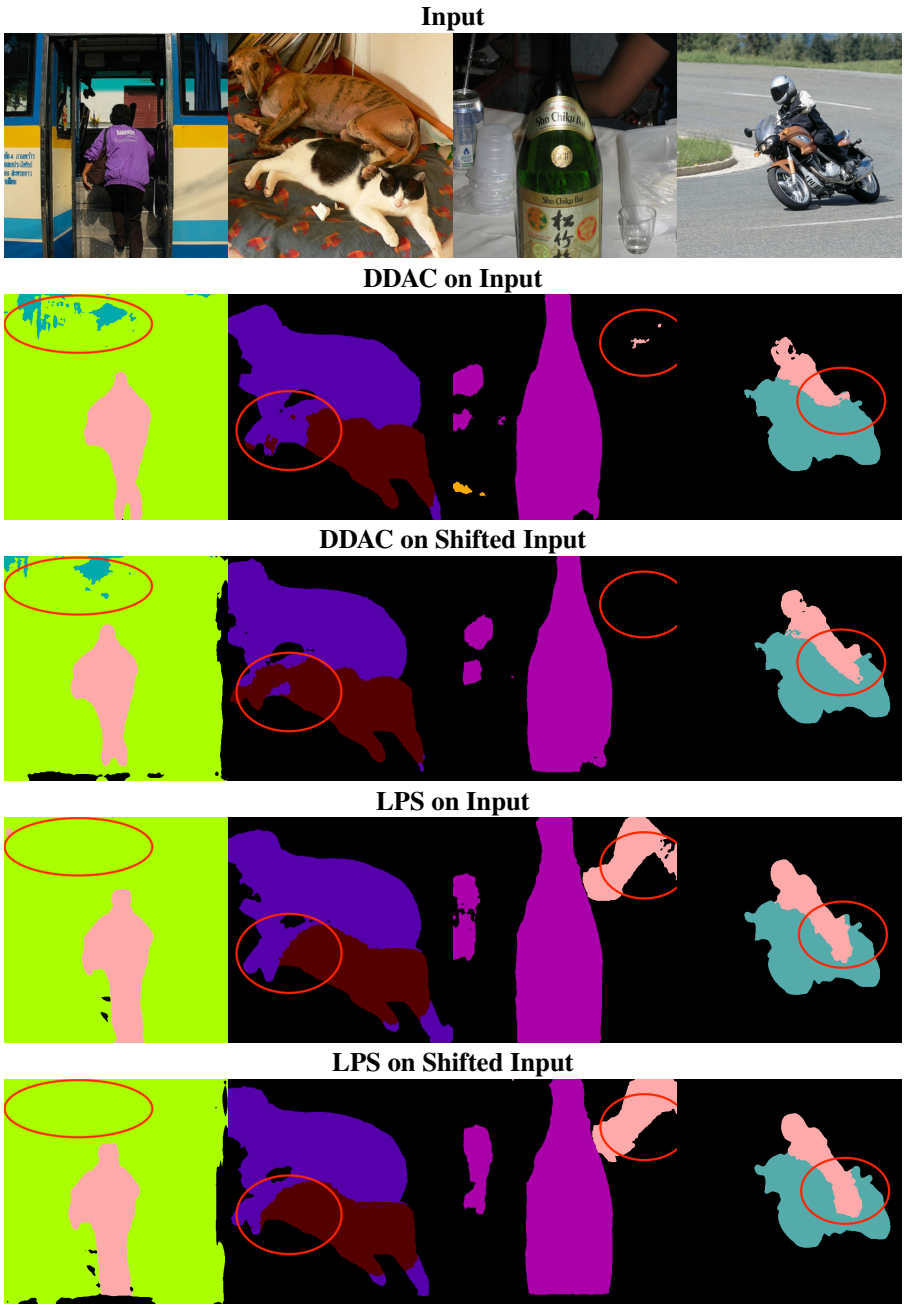

Figure A3. Qualitative comparison on PASCAL VOC ResNet-101 with DeepLabV3 architecture. Regions where our proposed network showed significant improvements under linear shifts are highlighted with a red circle.

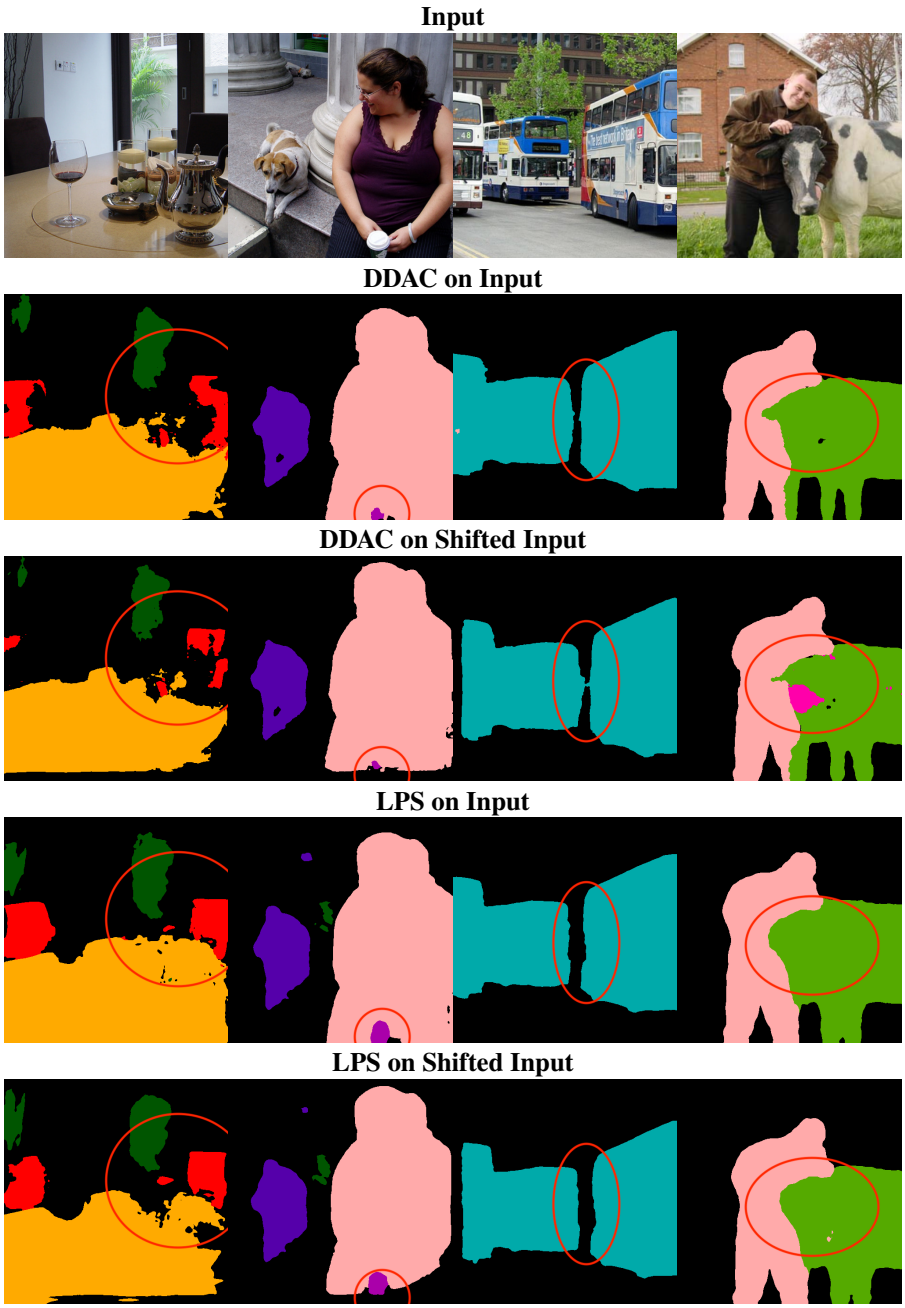

Figure A4. Qualitative comparison on PASCAL VOC ResNet-101 with DeepLabV3 architecture. Regions where our proposed network showed significant improvements under linear shifts are highlighted with a red circle.

A3 Additional Implementation Details

We have attached code to reproduce the reported experiments. We will publicly release the code and the model checkpoints upon acceptance.

A3.1 LPS Implementation Overview

To effectively conduct experiments, we utilize the research framework PytorchLighting [14] to avoid boilerplate and code redundancies. For documenting our experiments, we use Neptune [33] to log experimental details and outputs to ensure reproducibility. We implement the described LPS layers in Pytorch [34]. For these layers, we have written unit tests and end-to-end tests verifying the

shift-invariant/equivariant properties numerically. Overall, the attached code-base is organized as follows:

```
learn_poly_sampling
├── demo  # Ipython notebook illustrating layer usage.
├── learn_poly_sampling
│   ├── callbacks
│   ├── clargs
│   ├── configs
│   ├── data
│   ├── eval.py
│   ├── eval_segmentation.py
│   ├── layers    # LPS layers' implementation
│   ├── Makefile  # Runs tests
│   ├── models    # Classifier and DeepLabV3+ implementation
│   ├── README.md
│   ├── requirements.txt
│   ├── tests    # Test cases for the models
│   ├── train.py
│   ├── train_segmentation.py
│   └── utils
├── Makefile
└── README.md
```

Please refer to `README.md` for installation and experimentation instructions.

A3.2 LPD Usage Illustration

We illustrate how to incorporate the learnable polyphase downsampling (LPD) layer into a simple classifier. The network architecture consists of a single convolution layer, followed by LPD, a global pooling and finally a fully connected layer. We numerically verify that this model is shift-invariant. As our downsampling layer is implemented as a `nn.Module`, it can be easily incorporated into any existing deep-net implemented in Pytorch.

```python
# Define Model
class SimpleClassifier(nn.Module):
    def __init__(self, num_classes=3,padding_mode='circular'):
        # Conv. Layer
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1,
                               padding_mode=padding_mode)
        # Learnable Polyphase Downsampling Layer
        self.lpd = set_pool(partial(
            PolyphaseInvariantDown2D,
            component_selection=LPS,
            get_logits=get_logits_model('LPSLogitLayers'),
            pass_extras=False
            ),p_ch=32,h_ch=32)
        # Global Pooling + Classifier
        self.avgpool=nn.AdaptiveAvgPool2d((1,1))
        self.fc=nn.Linear(32, num_classes)
    def forward(self,x):
        x = self.conv1(x)
        x = self.lpd(x)  # Just like any layer.
        x = torch.flatten(self.avgpool(x),1)
        return self.fc(x)

# Construct Model
torch.manual_seed(0)
model = SimpleClassifier().cuda().eval().double()
```

```
# Load Image
img = torch.from_numpy(np.array(Image.open('butterfly.png'))).permute(2,0,1)
img = img.unsqueeze(0).cuda().double()
# Check is circular shift invariant
y_orig = model(img).detach().cpu()
img_roll = torch.roll(img,shifts=(1, 1), dims=(-1, -2))
y_roll = model(img_roll).detach().cpu()
print("y_orig : %s" % y_orig)
print("y_roll : %s" % y_roll)
assert(torch.allclose(y_orig,y_roll)) # Check shift invariant
print("Norm(y_orig-y_roll): %e" % torch.norm(y_orig-y_roll))
```

Out:

```
y_orig : tensor([[-22.0681, -36.2678,  20.5928]],
dtype=torch.float64)
y_roll : tensor([[-22.0681, -36.2678,  20.5928]],
dtype=torch.float64)
Norm(y_orig-y_roll): 0.000000e+00
```

A3.3 LPU Usage Illustration

We now illustrate how to incorporate the learnable polyphase upsampling (LPU) layer into a simple encoder-decoder architecture. The network archticture consists of a convolution layer, followed by LPD, another convolution layer, followed by LPU. We numerically verify that this architecture is circular shift-equivariant.

```
class SimpleUNet(nn.Module):
    def __init__(self, num_classes=3,padding_mode='circular'):
        # Conv. Layer
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1,
                               padding_mode=padding_mode)
        # Learnable Polyphase Downsampling Layer
        self.lpd = set_pool(partial(
            PolyphaseInvariantDown2D,
            component_selection=LPS,
            get_logits=get_logits_model('LPSLogitLayers'),
            pass_extras=False
            ),p_ch=32,h_ch=32)
        # Conv. Layer
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1,
                               padding_mode=padding_mode)
        # Learnable Polyphase Upsampling Layer
        antialias_layer = get_antialias(antialias_mode='LowPassFilter',
                                        antialias_size=3,
                                        antialias_padding='same',
                                        antialias_padding_mode='circular',
                                        antialias_group=1)
        self.lpu = set_unpool(partial(
            PolyphaseInvariantUp2D,
            component_selection=LPS_u,
            antialias_layer=antialias_layer), p_ch=32)
    def forward(self,x):
        x = self.conv1(x)
        x, prob = self.lpd(x,ret_prob=True)  # Just like any layer.
        x = self.conv2(x)
```

```
        x = self.lpu(x,prob=prob) # Just like any layer.
        return x
# Construct Model
torch.manual_seed(0)
model = SimpleUNet().cuda().eval().double()
# Load Image
img = torch.from_numpy(np.array(Image.open('butterfly.png'))).permute(2,0,1)
img = img.unsqueeze(0).cuda().double()
# Check is circular shift equivariant
y_orig = model(img).detach().cpu()
img_roll = torch.roll(img,shifts=(1, 1), dims=(-1, -2))
y_roll = model(img_roll).detach().cpu()
# Roll back to check equality
y_roll_s = torch.roll(y_roll, shifts=(-1,-1), dims=(-1, -2))
print("Norm(y_orig-y_roll_s): %e" % torch.norm(y_orig-y_roll_s))
assert torch.allclose(y_orig, y_roll_s)
```

Out:

```
Norm(y_orig-y_roll_s): 0.000000e+00
```

A4 Additional Experimental Details

A4.1 LPS Architecture

As described in Sec. 4.2, LPS selects optimal polyphase components via a mapping f_θ that is shift-permutation equivariant. Given a feature map $\boldsymbol{x} \in \mathbb{R}^{C\times N_1\times N_2}$, let its polyphase decomposition of order 2 be denoted as $\{\boldsymbol{x}_k\}_{k=0}^3, \boldsymbol{x}_k \in \mathbb{R}^{C\times\lfloor N_1/2\rfloor\times\lfloor N_2/2\rfloor}$. f_θ is then parameterized as a two-layer CNN followed by global average pooling. See Tab. A2 and Tab. A3 for the LPD layer (LPD) and the logits model architecture details.

Table A2. Learnable polyphase downsampling (LPD) model. After computing the polyphase components and their logits, the component selection step keeps the phase with the largest logit value.

	Layer	Kernel Size	Bias	Stride	Pad	Input Size	Output Size	Input Channels	Output Channels
1	Polyphase decomposition	–	–	2	–	$N_1 \times N_2$	$4 \times \lfloor N_1/2\rfloor \times \lfloor N_2/2\rfloor$	C	C
2	Logits model	–	–	–	–	$4 \times \lfloor N_1/2\rfloor \times \lfloor N_2/2\rfloor$	4	C	1
3	Component selection	–	–	–	–	$4 \times \lfloor N_1/2\rfloor \times \lfloor N_2/2\rfloor, 4$	$\lfloor N_1/2\rfloor \times \lfloor N_2/2\rfloor$	C	C

Table A3. Polyphase logits model for a single polyphase component $f_\theta : \mathbb{R}^{C\times\lfloor N_1/2\rfloor\times\lfloor N_2/2\rfloor} \mapsto \mathbb{R}$.

	Layer	Kernel Size	Bias	Stride	Pad	Input Size	Output Size	Input Channels	Output Channels
1	Conv2d + ReLU	3×3	✓	1	1	$\lfloor N_1/2\rfloor \times \lfloor N_2/2\rfloor$	$\lfloor N_1/2\rfloor \times \lfloor N_2/2\rfloor$	C	C_{hid}
2	Conv2d	3×3	✓	1	1	$\lfloor N_1/2\rfloor \times \lfloor N_2/2\rfloor$	$\lfloor N_1/2\rfloor \times \lfloor N_2/2\rfloor$	C_{hid}	C_{hid}
3a	Flatten	–	–	–	–	$\lfloor N_1/2\rfloor \times \lfloor N_2/2\rfloor$	$C_{\text{hid}}\lfloor N_1/2\rfloor\lfloor N_2/2\rfloor$	C_{hid}	1
3b	Global average pooling	–	–	–	–	$C_{\text{hid}}\lfloor N_1/2\rfloor\lfloor N_2/2\rfloor$	1	1	1

Hidden Layer. In practice, we have the freedom of choosing the number of channels in the hidden layer, denoted as C_{hid}. For our classification results, C_{hid} is equivalent to the number of channels of the input tensor. The only exception of this rule corresponds to the LPS ResNet-101 (adaptive antialias filter) case reported in Tab. 3. A top-1 classification accuracy of 78.8% plus a standard shift consistency of 92.4% is achieved by reducing the number of hidden channels w.r.t. the input at each pooling layer. Please see Tab. A4 for the number of input and hidden channels for each LPS layer used in our ResNet-101 experiments.

Table A4. ResNet-101: Number of channels at each LPS-D layer.

Layer	Conv 1	Maxpool	Layer 2 Downsample	Layer 3 Downsample	Layer 4 Downsample
Input Channels C	64	64	128	256	1024
Hidden Channels C_{hid}	8	8	16	32	72

Table A5. LPS-based ResNet-50 architecture for ImageNet.

	Layer	Kernel Size	Bias	Stride	Pad	Input Size	Output Size	Hidden Channels	Input Channels	Output Channels
1a	Conv2d + BN + ReLU	7×7	✗	1	3	224×224	224×224	–	3	64
1b	LPD	–	–	2	–	224×224	112×112	–	64	64
2a	Zero pad	–	–	–	–	112×112	113×113	–	64	64
2b	Max filter	2×2	–	1	0	113×113	112×112	–	64	64
2c	LPD	–	–	–	–	112×112	56×56	64	64	64
	Block 1									
3a	Res. layer	–	–	1	–	56×56	56×56	64	64	256
3b	Res. layer	–	–	1	–	56×56	56×56	64	256	256
3c	Res. layer	–	–	1	–	56×56	56×56	64	256	256
	Block 2									
4a	Res. layer (LPD)	–	–	2	–	56×56	28×28	128	256	512
4b	Res. layer	–	–	1	–	28×28	28×28	128	512	512
4c	Res. layer	–	–	1	–	28×28	28×28	128	512	512
4d	Res. layer	–	–	1	–	28×28	28×28	128	512	512
	Block 3									
5a	Res. layer (LPD)	–	–	2	–	28×28	14×14	256	512	1024
5b	Res. layer	–	–	1	–	14×14	14×14	256	1024	1024
5c	Res. layer	–	–	1	–	14×14	14×14	256	1024	1024
5d	Res. layer	–	–	1	–	14×14	14×14	256	1024	1024
5e	Res. layer	–	–	1	–	14×14	14×14	256	1024	1024
5f	Res. layer	–	–	1	–	14×14	14×14	256	1024	1024
	Block 4									
6a	Res. layer (LPD)	–	–	2	–	14×14	7×7	512	1024	2048
6b	Res. layer	–	–	1	–	7×7	7×7	512	2048	2048
6c	Res. layer	–	–	1	–	7×7	7×7	512	2048	2048
7	Global average pool	–	–	–	–	7×7	1×1	–	2048	2048
8	Flatten	–	–	–	–	1×1	2048	–	2048	1
9	Fully connected	–	✓	–	–	2048	1000	–	1	1

Input Dimensionality. Under a mini-batch training setting, the polyphase decomposition of an input corresponds to a five-dimensional tensor. Let this be denoted as $\mathbf{X} \in \mathbb{R}^{B \times P \times C \times \lfloor \frac{N_1}{2} \rfloor \times \lfloor \frac{N_2}{2} \rfloor}$, where B corresponds to the total number of feature maps in the batch and $P = 4$ to the number of polyphase components (assuming a downscaling factor of 2). To efficiently obtain a logit for each component, independently of its relative position in the tensor, we reshape it by combining the batch and polyphase component dimensions.

This alternative representation corresponds to $\hat{\mathbf{X}} \in \mathbb{R}^{4B \times C \times \lfloor \frac{N_1}{2} \rfloor \times \lfloor \frac{N_2}{2} \rfloor}$, and allows for each polyphase component to be processed independently of the rest. In practice, our CNN-based logits model receives $\hat{\mathbf{X}}$ as input and generates a set of $4B$ logits, one for each polyphase component, in a single forward pass.

Overall Architecture. Tab. A5 and Tab. A6 provide a general description of the ResNet-based architecture that incorporates LPD as pooling layer and its custom residual layer, respectively. In contrast to the original ResNet model, each pooling or downsampling step is replaced by our learn-based layer. For illustration purposes, we focus on the ResNet-50 model.

Table A6. Example of an LPS-based residual layer. Architecture corresponds to the first residual layer of ResNet-50 block 2 (4a in Tab. A5). The main and shortcut branches receive the input feature map of dimensions $N_1 \times N_2$. The LPD layer in the shortcut branch also receives the logits precomputed on the main branch of dimensions 4 to consistently select the same component.

	Layer	Kernel Size	Bias	Stride	Pad	Input Size	Output Size	Input Channels	Output Channels
				Main branch					
1a	Conv2d + BN + ReLU	1×1	✗	1	0	$N_1 \times N_2$	$N_1 \times N_2$	256	128
1b	Conv2d + BN + ReLU	3×3	✗	1	1	$N_1 \times N_2$	$N_1 \times N_2$	128	128
1c	LPD	–	–	2	–	$N_1 \times N_2$	$\lfloor N_1/2 \rfloor \times \lfloor N_2/2 \rfloor, 4$	128	128
1d	Conv2d + BN	1×1	✗	1	0	$\lfloor N_1/2 \rfloor \times \lfloor N_2/2 \rfloor$	$\lfloor N_1/2 \rfloor \times \lfloor N_2/2 \rfloor$	128	512
				Shortcut branch					
2a	LPD (pre-computed)	–	–	2	–	$N_1 \times N_2, 4$	$\lfloor N_1/2 \rfloor \times \lfloor N_2/2 \rfloor$	256	256
2b	Conv2d + BN	1×1	✗	1	0	$\lfloor N_1/2 \rfloor \times \lfloor N_2/2 \rfloor$	$\lfloor N_1/2 \rfloor \times \lfloor N_2/2 \rfloor$	256	512
3	Sum + ReLU	–	–	–	–	$\lfloor N_1/2 \rfloor \times \lfloor N_2/2 \rfloor, \lfloor N_1/2 \rfloor \times \lfloor N_2/2 \rfloor$	$\lfloor N_1/2 \rfloor \times \lfloor N_2/2 \rfloor$	512	512

A4.2 Baseline Implementations

A4.2.1 Image Classification

In this section, we provide additional implementation details and differences from the three main classification baselines, Lowpass Filtering (LPF) [56], Adaptive Polyphase Sampling (APS) [5] and Adaptive Lowpass Filtering (DDAC) [57].

Lowpass Filtering (LPF). LPF classification accuracy and shift consistency values included in Tab. 1, Tab. 2 and Tab. 3 correspond to those reported by LPF and APS manuscripts. Experimental results for standard shift consistency were taken from the LPF official repository, while experimental results analyzing circular shift consistency correspond to those reported by APS. It is important to note that, while LPF training setup for standard shift consistency uses *rescaled random cropping* as part of its preprocessing, our experiments on circular shift consistency follow APS settings and discard it. Please refer to Sec. A4.4 for details regarding data preprocessing.

Adaptive Polyphase Sampling (APS). As with LPF, APS's accuracy and consistency values via no antialiasing or lowpass filtering included in Sec. 5 are obtained from their official manuscript. APS results via adaptive filtering, denoted in Tab. 1 and Tab. 2 were obtained by replacing our learnable polyphase selection criteria by APS ℓ_2 energy-based selection and incorporated the adaptive filtering to it.

Adaptive Lowpass Filtering (DDAC). We compare the accuracy and shift consistency of our proposed pooling method against that of DDAC under both circular and standard shifts. For the circular shift case, reported for CIFAR-10 and ImageNet in Tab. 1 and Tab. 2, respectively, we replace the LPS selection criteria by keeping always the even polyphase components ($k^\star = 0$), followed by a learnable low-pass filter with the exact same specifications as the one provided in the official DDAC code base. For consistency purposes, DDAC experiments analyzing circular shift consistency follow the same data preprocessing used in APS experiments. More precisely, no rescaled random cropping is applied for data augmentation.

For the standard shift case, we compare against the best performing DDAC model, ResNet-101, without any changes. Tab. 3 includes its reported top-1 classification accuracy and shift consistency. For the sake of clarity, we also include the results obtained by training the model from scratch using their official code base and hyperparameters.

A4.2.2 Semantic Segmentation

In the paper, we directly compared to the reported values in DDAC [57]. Unfortunately, the authors did not released their evaluation code and did not provide a clear description of the metric. We were not able to exactly reproduce their reported mASSC. For a fair comparison, we evaluate DDAC's released checkpoint using our mASSC implementation. We will make our implementation publicly available.

A4.3 Comparison against Classifiers with More Trainable Parameters.

To show the performance improvement attained by our LPD approach is not simply an effect of introducing more trainable parameters, we compared our ResNet-101 + LPD model ($44,751,034$ parameters) against the larger ResNet-152 model ($60,192,808$ parameters). Both models were trained on ImageNet using the same augmentation and optimizer configuration. Under such settings, ResNet-152 achieves 78.3% top-1 classification accuracy and 90.9% shift consistency, while our ResNet-101 + LPD model obtains 78.8% top-1 accuracy and 92.4% shift-consistency. Despite having 25% less trainable parameters, our model attains 0.5% higher accuracy and 1.5% higher shift-consistency.

A4.4 Hyperparameters and Tuning Procedure

A4.4.1 Image Classification

Learn Rate and Optimization Parameters. Following the standard ImageNet setup, the initial learn rate value corresponds to 0.1 and follows a multi-step schedule, decaying every 30 epochs by a factor of 0.1. Our models are trained via stochastic gradient descent with 0.9 momentum. A weight decay of factor 10^{-4} is imposed to all model trainable weights except those of the LPS layers. Empirically, this has shown a substantial consistency improvement, avoiding cases where polyphase logits have very similar values and other numerical precision issues. Additionally, for our experiments on ResNet-101 and ResNet-50 with adaptive antialiasing filters, following DDAC settings, a learning rate warmup of five epochs is applied.

Data Preprocessing and Split. Tab. A7 describes the data preprocessing used in our CIFAR-10 experiments. No shifts or resizing augmentations are applied to highlight the fact that perfect circular shift invariance/equivariance is achieved by design and not induced during training.

Table A7. CIFAR-10 data preprocessing.

Split	Train Set	Test Set
Preprocessing	(i) Random horizontal flipping (ii) Normalization	(i) Normalization

For ImageNet experiments evaluating circular shift consistency, we follow APS's preprocessing settings. Tab. A8 describes its data preprocessing. For ImageNet experiments evaluating standard shift consistency, we follow DDAC settings. Tab. A9 describes its data preprocessing.

Table A8. ImageNet data preprocessing for **circular shift consistency** evaluation.

Split	Train Set	Test Set
Preprocessing	(i) Resizing to 256×256 (ii) Center cropping to 224×224 (iii) Random horizontal flipping (iv) Normalization	(i) Resizing to 256×256 (ii) Center cropping to 224×224 (iii) Normalization

Table A9. ImageNet data preprocessing for **standard shift consistency** evaluation.

Split	Train Set	Test Set
Preprocessing	(i) Resizing to 256×256 (ii) Resized random cropping to 224×224 (iii) Random horizontal flipping (iv) Normalization	(i) Resizing to 256×256 (ii) Center cropping to 224×224 (iii) Normalization

Computational Settings. Classification experiments on CIFAR-10 are trained on a single NVIDIA Titan V using a batch size of 256 for 250 epochs. Classification experiments on ImageNet are trained in distributed data parallel mode on four NVIDIA A6000 GPUs using a batch size of 64 for 90 epochs.

Polyphase Selection for Circular Shift Consistency. For circular shift consistency on both CIFAR-10 and ImageNet, the polyphase component selection depends on the model state. During training, each LPS layer samples from a Gumbel-softmax distribution. This leads to a convex combination of polyphase components that improves the backpropagation process. Following the original Gumbel-Softmax formulation, we use an annealing factor to slowly converge to a one-hot vector along epochs. Considering 250 and 90 training epochs for CIFAR-10 and ImageNet experiments, respectively, the annealing factor $\tau \in \mathbb{R}_+$ gradually decays to improve the gradient flow during the error backpropagation step. A step decay approach is used for CIFAR-10 experiments, while a multistep linear decay is used for ImageNet experiments.

During testing, optimal polyphase components correspond to those with the largest logit values (hard-selection), which leads to a classifier with perfect circular shift consistency by design.

Polyphase Selection for Standard Shift Consistency. For standard shift consistency evaluation on ImageNet, we adopt a fine-tuning procedure to balance the shift consistency and classification accuracy obtained by our models. Recall that our model guarantees perfect consistency under circular shifts, which leaves open the posibility of applying more refined training to improve the performance under standard shifts.

With this in mind, instead of switching to a hard-selection, we relax the annealing factor during the last 28 training epochs, replacing the gumbel-softmax sampling by a standard softmax (soft-association) for both training and testing.

Tab. A10 includes the details of the Gumbel-softmax annealing schedule for both circular and standard shift consistency experiments.

Table A10. Gumbel-softmax annealing schedule for image classification.

Consistency Evaluation	CIFAR-10	ImageNet	
	Circular Shift	Circular Shift	Standard Shift
Method	Step Decay	Multirate Linear Decay	Multirate Linear Decay
τ Schedule	Initial value: τ=1 Decay step: 10 epochs Decay factor: 0.85 Minumum value: $\tau = 0.025$	Epoch $1/90$: $\tau = 1$ Epoch $62/90$: $\tau = 0.5$ Epoch $82/90$: $\tau = 0.05$ Epoch $90/90$: $\tau = 0.01$	Epoch $1/90$: $\tau = 1$ Epoch $62/90$: $\tau = 0.5$ Epoch $90/90$: $\tau = 0.25$

A4.4.2 Semantic Segmentation

Unpooling Component Selection. During training, for both standard and circular shift consistency evaluation, feature maps obtained from the backbone are unpooled (upsampled and shifted) by LPU layers. The shift applied to each feature map, intended to place them back into their original indices, depends on the logit probabilities computed by the backbone. Since these are obtained from a Gumbel-softmax distribution during training, instead of placing the upsampled feature map at a single position, LPU layers generate an upscaled representation composed by all four possible positions (assuming an upscaling factor of 2), each weighted by its corresponding probability. In other words, LPU *soft*-unpools the input feature map to all four possible positions, weights them by their probabilities and adds them together to obtain the output feature map.

For our ImageNet experiments we modify the annealing schedule used for image classification and tailor it to the segmentation model and its 125 training epochs. First, we linearly increase the annealing factor from the last value used during the backbone training process (recall that the backbone was trained using its own Gumbel-softmax annealing schedule) to 0.5. Then, we gradually decay it to improve the gradient flow during backpropagation.

During testing, LPU layers receive logits from the backbone. Then, feature maps are unpooled according to the polyphase component with the largest logit value (hard-selection), allowing the segmentation model to become shift-equivariant by design.

Tab. A11 includes the details of the Gumbel-softmax annealing schedule used in our semantic segmentation experiments for both circular and standard shift consistency evaluation.

Table A11. Gumbel-softmax annealing schedule for image segmentation on ImageNet. * Initial τ_{backbone} values correspond to the final values used during our ResNet-18 and ResNet-101 backbone training.

Consistency Evaluation	ResNet-18 Circular Shift	ResNet-101 Circular and Standard Shift
Method	Multirate Linear Decay	Multirate Linear Decay
τ Schedule	Epoch $1/125$: $\tau = \tau_{\text{backbone}}$ * Epoch $60/125$: $\tau = 0.5$ Epoch $82/125$: $\tau = 0.15$ Epoch $90/125$: $\tau = 0.01$	Epoch $1/125$: $\tau = \tau_{\text{backbone}}$ * Epoch $10/125$: $\tau = 0.5$ Epoch $80/125$: $\tau = 0.3$ Epoch $100/125$: $\tau = 0.125$ Epoch $125/125$: $\tau = 0.01$