Autoregressive Unsupervised Image Segmentation: Supplementary Material

Yassine Ouali, Céline Hudelot and Myriam Tami

Université Paris-Saclay, CentraleSupélec, MICS, 91190, Gif-sur-Yvette, France
{yassine.ouali,celine.hudelot,myriam.tami}@centralesupelec.fr

In this supplementary material, we provide architectural details, hyperparameters settings, further discussions about the loss functions and the masked convolutions. We also provide some qualitative results and implementation details.

1 Architectural details

Tables 1 to 4 present the building blocks of the representation function $\mathcal{F}$. Specifically, we describe the architecture of the convolutional stem, the residual blocks, the decoder for AC and the separable critics used for ARL.

Convolutional Stem	
Layer	Output size
Input	$3 \times H \times W$
Conv 3×3	$64 \times H \times W$
Batch Norm - ReLU	$64 \times H \times W$
Max Pool 3×3, $s = 2$	$64 \times H/2 \times W/2$

Table 1. Convolutional Stem for an output stride of 2. For an output stride of 4, we use a Conv 3×3 with stride of 2, yielding an output of size $64 \times H/4 \times W/4$. For the fully autoregressive case, the Batch Norm and Max pool are omitted, and the Max pool is replaced with a strided masked convolution.

Decoder	
Layer	Output size
Input	$C \times H/2 \times W/2$
Conv 1×1	$K \times H \times W$
Bilinear Interpolation	$K \times H \times W$
Softmax	$K \times H \times W$

Table 2. Decoder used for a clustering objective. In this case, we have an output stride of 2 and K clusters.

2 Hyperparameters

We discuss the hyperparameters used in our experiments. We note that we have noticed that the network is very sensitive to the initialization. In our case, we

Separable Critics	
Layer	Output size
Input	$C \times H \times W$
Conv 1×1 - ReLU	$2C \times H \times W$
Conv 1×1 - ReLU	$2C \times H \times W$
Conv 1×1 applied to the input	$2C \times H \times W$
Residual Connection + Batch Norm	$2C \times H \times W$

Table 3. Separable critics used for representation learning to non-linearly project the outputs to a higher vector space.

Residual Block	
Layer	Output size
Input	$C \times H \times W$
Conv 3×3 - ReLU	$2C \times H \times W$
Conv 1×1 - ReLU	$2C \times H \times W$
Zero padding of the input to $2C$	$2C \times H \times W$
Residual Connection	$2C \times H \times W$
Conv 1×1 - ReLU	$2C \times H \times W$
Conv 1×1 - ReLU	$2C \times H \times W$
Residual Connection	$2C \times H \times W$
Conv 1×1 - ReLU	$2C \times H \times W$
Conv 1×1 - ReLU	$2C \times H \times W$
Residual Connection	$2C \times H \times W$

Table 4. Architecture of residual blocks, for residual blocks used in the autoregressive encoder g_{ar}, normal convolutions are replaced with masked ones.

initialize the parameters using Xavier initialization [2], and noticed a somehow more stable results with such an instantiation scheme. The optimizer of choice is Adam [4] with the default parameters ($\beta_1 = 0.9$ and $\beta_2 = 0.999$). The rest of the hyperparamters used are detailed in Table 5.

For the transformations applied in Paper Section 4.1, we used color jittering where we randomly change the brightness, hue, contrast and saturation of an image up to 10% for photometric transformations. For geometric transformation, we apply random horizontal flips and random rotations by multiples of 90 degrees.

3 Loss functions

In this section, we will go into more details about the loss functions introduced in Paper Section 3.2. For a given unlabeled input $\mathbf{x} \sim \mathcal{X}$, and two outputs $\mathbf{y} \sim \mathcal{F}(\mathbf{x}; o_i)$ and $\mathbf{y}' \sim \mathcal{F}(\mathbf{x}; o_j)$ with two valid orderings $(o_i, o_j) \in \mathcal{O}$, the training objective is to maximize the MI between the two encoded variables:

$$\max_{\mathcal{F}} I(\mathbf{y}; \mathbf{y}') \tag{1}$$

Parameter	COCO-stuff 3	COCO-stuff	Potsdam-3	Potsdam
LR	4.10^{-5}	6.10^{-6}	10^{-6}	4.10^{-5}
Batch size	60	60	30	30
Crop size	128×128	128×128	200×200	200×200
Rescale factor	0.33	0.33	No rescal.	No rescal.
Output stride	4	4	2	2
Num. of displacements for $\mathcal{L}_{AC}$	10	10	10	10
Attention	False	False	True	True

Table 5. Hyperparameters used for training per dataset.

3.1 Autoregressive Clustering $\mathcal{L}_{AC}$

To see the benefits of maximizing Eq. (1) for a clustering objective, we expand the objective as the difference between two entropy terms:

$$I(\mathbf{y}; \mathbf{y}') = H(\mathbf{y}) - H(\mathbf{y}|\mathbf{y}') \tag{2}$$

By such a formulation, we can see that maximizing the MI involves maximizing the entropy and minimizing the conditional entropy. The compromise between these two terms help us avoid both degenerate and trivial solutions. For degenerate solution, where the model $\mathcal{F}$ outputs uniform distributions over all of the pixels, not assigning any cluster to any pixel, the entropy $H(\mathbf{y})$ in this case is maximized, however the second term $H(\mathbf{y}|\mathbf{y}')$ is also maximized, since the outputs are not deterministic and there is no predictability of the second output from the first. Inversely, with trivial solutions, where all of the pixel are assigned to the same cluster. the second output $\mathbf{y}'$ is totally deterministic from the first, and the conditional entropy $H(\mathbf{y}|\mathbf{y}')$ is minimized, yet, the entropy $H(\mathbf{y})$ is also minimized and we fail to maximize the MI. By balancing the maximization of the first term and the minimization of the second, we are more likely to end-up with the correct assignments, than if we only maximized the entropy.

Given that the two outputs are generated using the same input and two different orderings, there is a strong statistical dependency between them. In this case, $\mathbf{y} \sim \mathcal{F}(\mathbf{x}; o_i)$ and $\mathbf{y}' \sim \mathcal{F}(\mathbf{x}; o_j)$ are dependent and we compute the joint probability $p(\mathbf{y}, \mathbf{y}')$ as a matrix of size $K \times K$:

$$p(\mathbf{y}, \mathbf{y}') = \mathcal{F}(\mathbf{x}; o_i)^T \mathcal{F}(\mathbf{x}; o_j) \tag{3}$$

In practice we also marginalize over the batch, with an input $\mathbf{x}$ of shape $B \times 3 \times H \times W$ as a batch of B input images. Let $\mathbf{x}_i$ correspond to the i-th image in the batch $\mathbf{x}$ of B images. In this case the joint probability is computed as follows:

$$p(\mathbf{y}, \mathbf{y}') = \frac{1}{B} \sum_{i=1}^{B} \mathcal{F}(\mathbf{x}_i; o_i)^T \mathcal{F}(\mathbf{x}_i; o_j) \tag{4}$$

Additionally, following [3], we also compute the joint probability over small possible displacements $\mathbf{u} \in \Omega$. Let the input $\mathbf{x}^{(\mathbf{u})}$ correspond to shifting the input

$\mathbf{x}$ by $\mathbf{u}$ pixels (*i.e.*, zero padding and cropping). In such a case, we also need to marginalize over all possible displacements $\mathbf{u}$ as follows:

$$p(\mathbf{y}, \mathbf{y}') = \frac{1}{B} \frac{1}{|\Omega|} \sum_{i=1}^{B} \sum_{\mathbf{u} \in \Omega} \mathcal{F}(\mathbf{x}_i; o_i)^T \mathcal{F}(\mathbf{x}_i^{(\mathbf{u})}; o_j) \tag{5}$$

Finally, by summing over the rows and columns of $p(\mathbf{y}, \mathbf{y}')$, we can compute the marginals, and then the MI:

$$I(\mathbf{y}, \mathbf{y}') = D_{\mathrm{KL}}(p(\mathbf{y}, \mathbf{y}') \| p(\mathbf{y}) p(\mathbf{y}')) \tag{6}$$

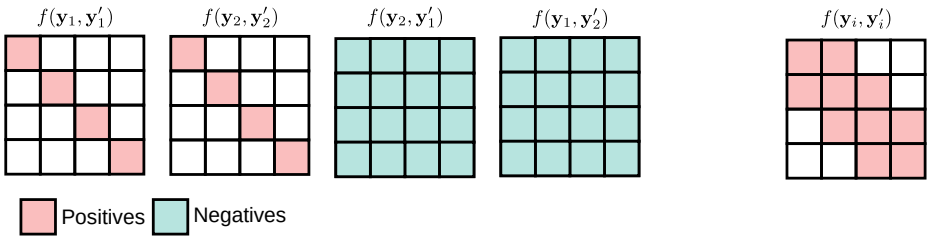

Fig. 1. *Left*: Examples of positive and negative pairs for $B = 2$ and $HW = 4$. $\mathbf{y}_i$ refers to the i-th element of the output $\mathbf{y}$ corresponding to the i-th image in the input batch. *Right*: Examples of positive pairs with possible displacements $\Omega = \{-1, 0, 1\}$.

3.2 Autoregressive Representation Learning $\mathcal{L}_{AC}$

For unsupervised representation learning objective, we maximize the infoNCE [5] as a lower bound of MI over the continuous outputs:

$$\mathcal{L}_{\mathrm{ARL}} = \log \frac{e^{f(\mathbf{y}_l, \mathbf{y}'_l)}}{\frac{1}{N} \sum_{m=1}^{N} e^{f(\mathbf{y}_l, \mathbf{y}'_m)}} \tag{7}$$

The goal of Eq. (7) is to push the network $\mathcal{F}$ to produce similar features between the two outputs $\mathbf{y}$ and $\mathbf{y}'$ at the same spatial locations, so that the critic is able to give high scores between two feature vectors $(\mathbf{y}_l, \mathbf{y}'_m)$ at the same spatial position $m = l$, and low scores for feature vectors from distinct spatial position $m \neq l$ or from two distinct images. To compute the loss in Eq. (7), we need to create a set of positive and negative pairs. With a batch of images $\mathbf{x}$ of shape $B \times 3 \times H \times W$, we generate two outputs $\mathbf{y}$ and $\mathbf{y}'$ of shape $B \times C \times H \times W$, with C-dimensional output feature maps. In this case the output of the critic $f(\mathbf{y}, \mathbf{y}') = \phi_1(\mathbf{y})^\top \phi_2(\mathbf{y}')$ is a matrix of shape $BHW \times BHW$. To construct the positive and negative pairs, we reshape the scoring matrix as B^2 matrices of shape HW, in this case the positives are the diagonals of each matrix from the same images with a given shift $\mathbf{u} \in \Omega$. The negatives are all of the possible combination across the matrices from distinct images. See Fig. 1 for an illustration for $B = 2$ and $HW = 4$. Note that we avoid using the same image to construct negative pairs, and only construct them across images, given that even with distinct spatial positions, it is very likely that two feature vectors share similar characteristics.

4 Receptive fields

To further illustrate how a given ordering o_i is constructed, we present a toy example where we plot the receptive field of a given pixel at the center of an image of size 16×16. After each application of a masked convolution with the corresponding shift, we compute the gradient of the target pixel and plot the non-zero values in blue, which correspond to the receptive field of the target pixel. The results are illustrated in Fig. 2.

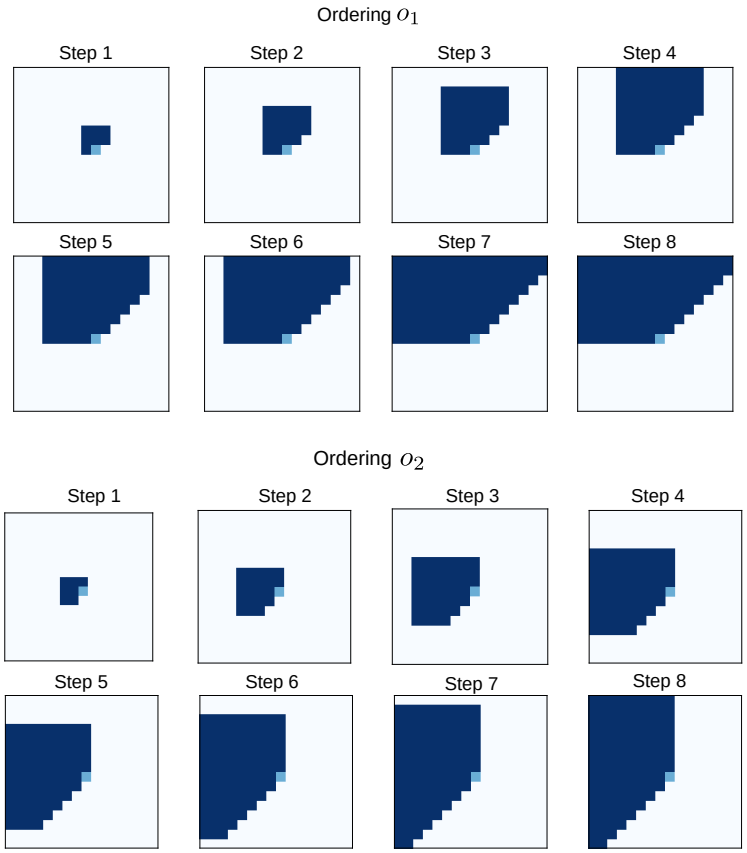

Fig. 2. Examples of the growing receptive field of pixel ■ for two orderings; o_1 and o_2, over 8 consecutive applications of masked convolutions to get the correct orderings. As expected, after enough convolutions, and with the correct shift, we can construct the desired ordering. Note that in both cases we have a significant number of pixels in the blind spots, which can be accessed using an attention block. In this case, we use Conv_A with Shift_1 and Shift_2.

Orderings. For a given pair of distinct orderings, the resulting dependencies and receptive fields of the two outputs will be different even if the applied orderings

are quite similar. It is however likely that the two outputs share some overlap in their receptive fields, but such an overlap is small and helps reduce the difficulty of the task. An illustration of the resulting receptive fields for a given pixel using raster-scan orderings is shown in Fig. 3.

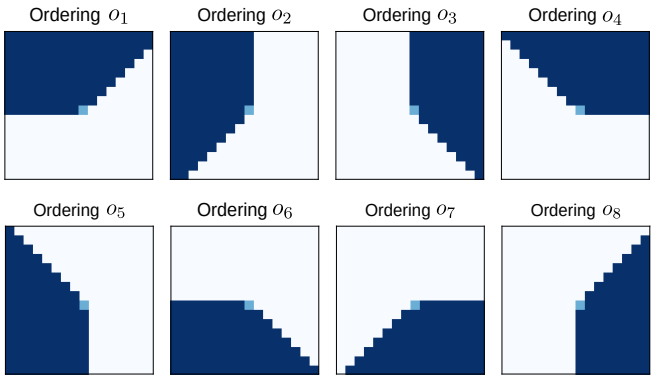

Fig. 3. The resulting receptive fields with the various raster-scan type orderings

5 Qualitative Results

Fig. 4 shows qualitative results of Autoregressive Clustering (AC) on COCO-stuff 3 *test* set, in addition to linear and non-linear evaluations, where the model trained for AC is frozen and then the corresponding layers are added on top of the decoder, that are then trained on the *train* set. Surprisingly, even if the accuracy with linear and non-linear evaluations is higher, we see that qualitatively, the fully unsupervised method gives slightly better results. This might be due to the dense nature of image segmentation, where the prediction at a given pixel is very dependent of its neighbors, and we lose this locality with linear evaluation, given that we consider each pixel as a standalone data point. This is similar to what we observed with ARL where we optimize the representations at each spatial location separately. Note that we have noticed some minor annotation errors in the ground truths that might be due to the conversion done by [3], these are very minor and can be overlooked.

We also present some examples where AC fails in Fig. 5. We observe that the model is very dependent on the appearance and colors for making the predictions. However, in special cases, like tennis courts with grass or asphalt floors, the model predicts green or sky classes, and the correct prediction is ground. This can be overcome with additional data augmentations like color jittering, or in case where a limited amount of labeled examples are available, the model can be fine-tunned to correct such mistakes. We already see some slight improvements with linear and non-linear evaluations.

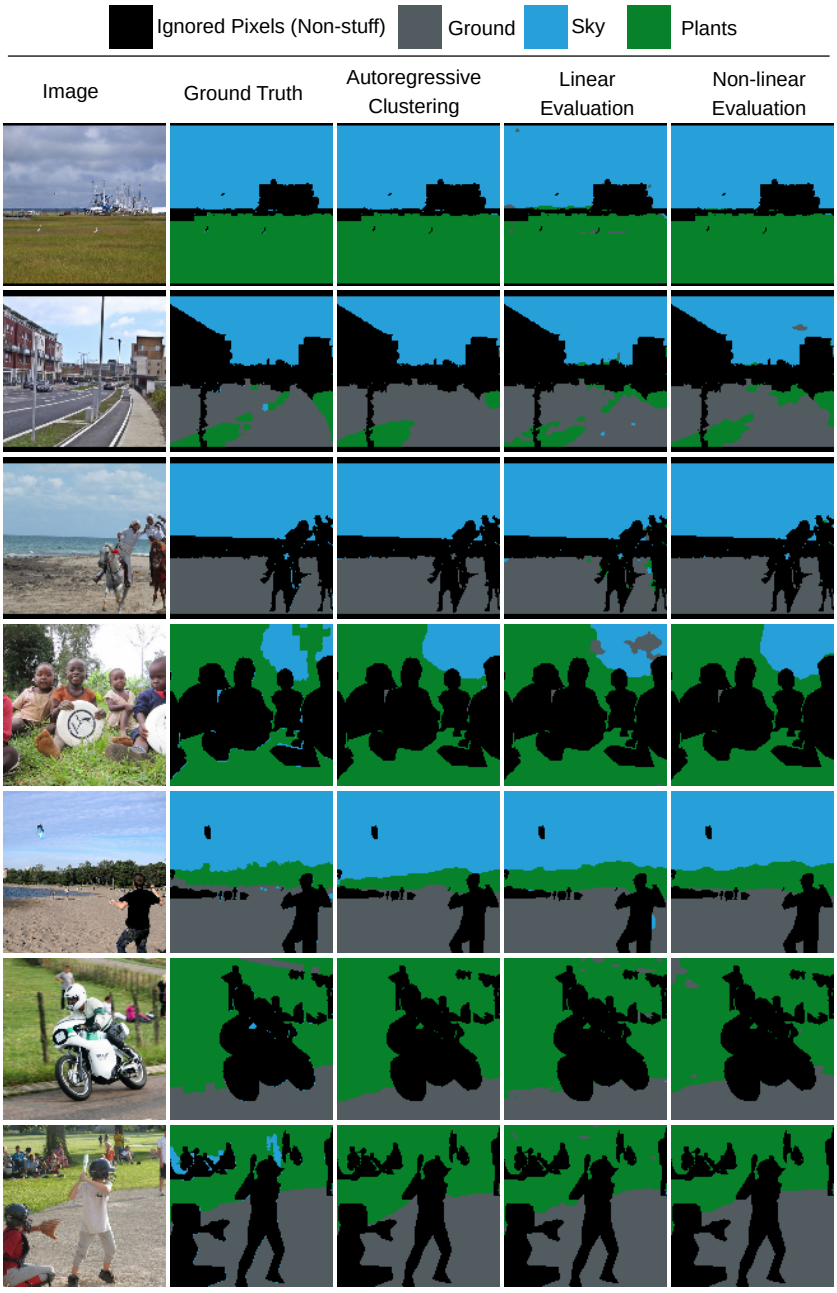
Ignored Pixels (Non-stuff)
Ground
Sky
Plants
Image
Ground Truth
Autoregressive Clustering
Linear Evaluation
Non-linear Evaluation

Fig. 4. Qualitative Results from COCO-Stuff 3 [1,3] *test* set.

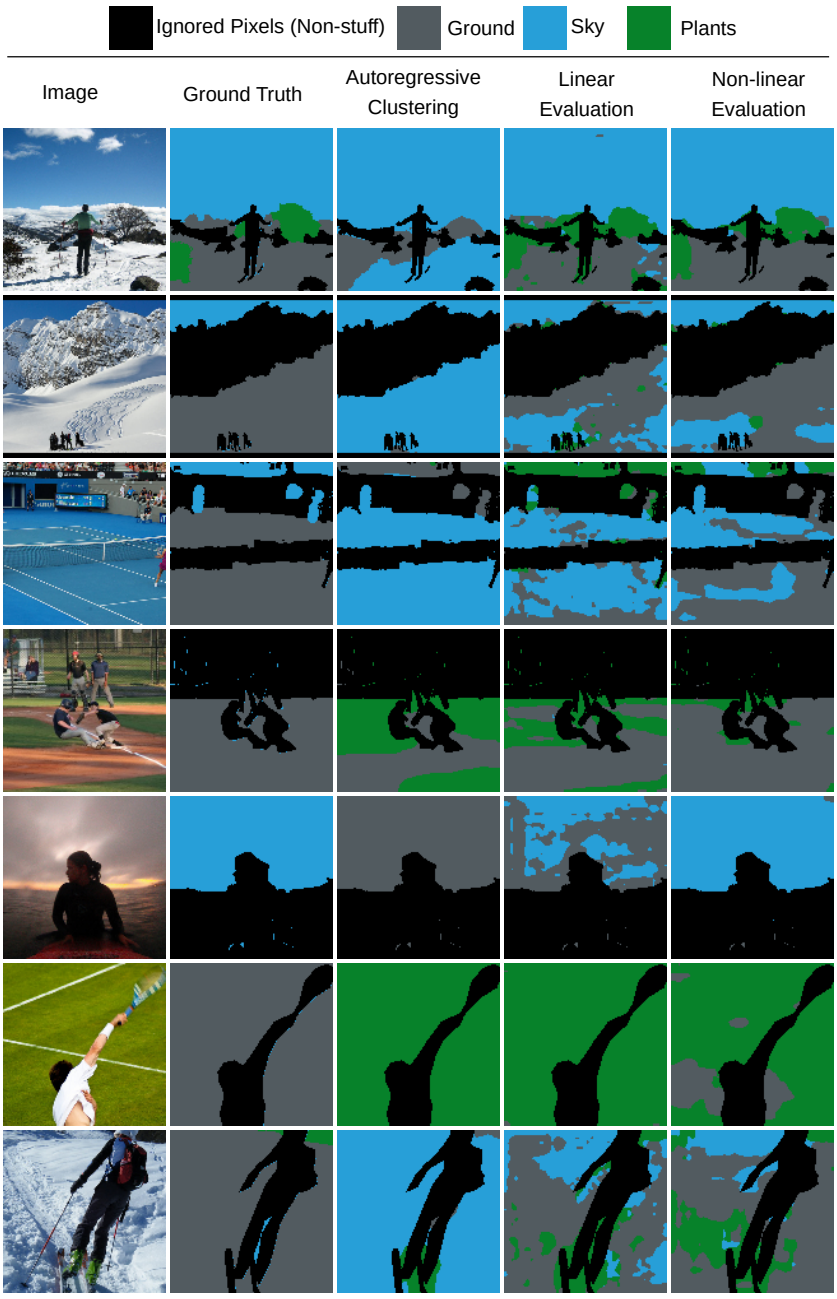

Fig. 5. Failure Cases for Autoregressive Clustering from COCO-Stuff 3 [1,3] *test* set.

6 Implementation Details

For implementing the masked convolution, we tested different approaches, such as implementing a custom convolution layer using PyTorch's `autograd.Function`, or by first folding the inputs, multiplying by the convolution's weights, and then masking the corresponding positions in the outputs before summing. However, both of these approaches are very expensive in terms of memory and computation. To this end, we choose a very simple method, consisting of using two convolutions, a `main` and a `dummy` convolution, and at a given training iteration, we copy the weights from the `main` convolution to the `dummy` convolution, and then mask the corresponding weights of `dummy` convolution and apply it to input. In the backward pass, we copy the gradients of `dummy` convolution to the `main` convolution to the update its weights. This way, we keep the current state of the weights in the `main`, and only update the unmasked weights and the masked ones remain unchanged. During inference, we directly apply the `main` convolution. A pseudo-code in PyTorch is presented bellow.
Note that in this case, to apply the desired shift for a given filter of size $F \times F$, we simply adjust the normal padding values from $(\lfloor \frac{F}{2} \rfloor, \lfloor \frac{F}{2} \rfloor, \lfloor \frac{F}{2} \rfloor, \lfloor \frac{F}{2} \rfloor)$ corresponding to (left, right, top, bottom), to the correct values after the shift. For example, for Shift_1 where we want to shift the input down, the padding values will be $(\lfloor \frac{F}{2} \rfloor, \lfloor \frac{F}{2} \rfloor, F-1, 0)$

```
class MaskedConv2d(nn.Module):
    def __init__(self, inplanes, outplanes, kernel_size=3, stride=1,
                       pad_mode="constant", bias=True, dilation=1):
        super(MaskedConv2d, self).__init__()
        assert(pad_mode in ['constant', 'reflect'])

        self.pad_mode = pad_mode

        # Defining the two convolutions
        self.main_conv = nn.Conv2d(inplanes, outplanes, kernel_size,
                          stride=stride, padding=0, bias=bias, dilation=dilation)
        self.dummy_conv = nn.Conv2d(inplanes, outplanes, kernel_size,
                          stride=stride, padding=0, bias=bias, dilation=dilation)

        # Weight-tying the two biases
        self.dummy_conv.bias = self.main_conv.bias

        # Kernel size with dilation to compute the correct paddings
        kernel_size = kernel_size + (kernel_size - 1) * (dilation - 1)

        # The normal padding values
        self.num_to_pad = kernel_size // 2

        # The new padding values after shift (left, right, top, bottom)
        self.padding = {
            "pad_1": (self.num_to_pad, self.num_to_pad, kernel_size - 1, 0),
            "pad_2": (kernel_size - 1, 0, self.num_to_pad, self.num_to_pad),
            "pad_3": (self.num_to_pad, self.num_to_pad, 0, kernel_size - 1),
            "pad_4": (0, kernel_size - 1, self.num_to_pad, self.num_to_pad)
        }

    def apply_mask(self, weight, conv_type):
        '''
        Masks the corresponding weights of a given convolutions, the weights
        can also be the gradients in the backward pass
        '''
```

```
        if conv_type == "convA":
            weight.data[:, :, -1, self.num_to_pad+1:].zero_()
        elif conv_type == "convB":
            weight.data[:, :, -1, :self.num_to_pad].zero_()
        elif conv_type == "convC":
            weight.data[:, :, 0, self.num_to_pad+1:].zero_()
        elif conv_type == "convD":
            weight.data[:, :, 0, :self.num_to_pad].zero_()
        else:
            raise ValueError
        return weight

    def forward(self, x):
        # During training
        if self.training:

            # Get the corresponding conv and shift of the current orderings
            conv_type, pad_type = Orderings._current_ordering

            # Pad the input to get the desired shift
            x = F.pad(x, self.padding[pad_type], mode=self.pad_mode)

            # Copy the weight of the main conv to the dummy conv (not inplace)
            self.dummy_conv.weight.data = self.main_conv.weight.data.clone()

            # Mask the corresponding weight to get the desired ordering
            self.dummy_conv.weight = self.apply_mask(self.dummy_conv.weight, conv_type)

            # Apply the masked conv
            x = self.dummy_conv(x)
            return x

        # In inference, we fall back to the normal case
        # Normal padding
        x = F.pad(x, (self.num_to_pad, self.num_to_pad, self.num_to_pad,
                      self.num_to_pad), mode=self.pad_mode)
        # Apply the unmasked conv
        x = self.main_conv(x)
        return x

    def switch_gradients(self):
        # Called after loss.backward() and before optimizer.step()

        # Orderings._last_ordering contains the two orderings that were applied during
        # the forward pass
        for conv_type in Orderings._last_ordering:
            # Masking the gradients of the masked weights
            self.dummy_conv.weight.grad = self.apply_mask(self.dummy_conv.weight.grad, conv_type)

        # Copy the gradients
        self.main_conv.weight.grad = self.dummy_conv.weight.grad.clone()
        # Remove the gradients of the dummy conv (not necessary)
        self.dummy_conv.weight.grad = None
```

As for updating and fetching the current ordering at each forward pass, we opted to use a class method containing the current ordering, with the corresponding convolution type and shift as a class attribute. At each forward pass, we change the current ordering from the main model.

```
# The 8 possible orderings and the corresponding conv and shift
ORDERING_POSSIBILITIES = [
    ("convA", "pad_1"),
    ("convA", "pad_2"),
    ("convB", "pad_4"),
    ("convB", "pad_1"),
    ("convC", "pad_3"),
```

```
    ("convC", "pad_2"),
    ("convD", "pad_3"),
    ("convD", "pad_4")
]

def get_zigzag_ordering(ordering, H, W):
    """
    Converts a given raster-scan ordering to a zigzag ordering
    """
    indices = ordering.flip(1).flip(0).flip(1)
    coords = []
    for i, j in enumerate(range(-(H-1), W)):
        x = indices.diag(j)
        if i % 2 == 1:
            x = x.flip(0)
        coords.extend(x)
    coords = torch.stack(coords)
    return coords


def get_ordering(conv_type, pad_type, H, W):
    """
    Generates all possible raster scan orderings O1 ... O8
    """
    if conv_type == "convA" and pad_type == "pad_1":
        return torch.arange(H*W).reshape(H, W)
    if conv_type == "convA" and pad_type == "pad_2":
        return torch.arange(H*W).reshape(H, W).T
    if conv_type == "convB" and pad_type == "pad_4":
        return torch.arange(H*W).reshape(H, W).flip(0).T
    if conv_type == "convB" and pad_type == "pad_1":
        return torch.arange(H*W).reshape(H, W).flip(1)
    if conv_type == "convC" and pad_type == "pad_3":
        return torch.arange(H*W).reshape(H, W).flip(0)
    if conv_type == "convC" and pad_type == "pad_2":
        return torch.arange(H*W).reshape(H, W).flip(1).T
    if conv_type == "convD" and pad_type == "pad_3":
        return torch.arange(start=H*W, end=0, step=-1).reshape(H, W) - 1
    if conv_type == "convD" and pad_type == "pad_4":
        return (torch.arange(start=H*W, end=0, step=-1).reshape(H, W) - 1).T

from functools import lru_cache

@lru_cache(maxsize=64)
def from_order_to_att_mask(H, W, ordering, conv_type, pad_type):
    """
    Computes the attention mask for a given ordering
    """
    if ordering.ndim == 2:
        ordering = ordering.reshape(-1)
    normal_mask = torch.triu(torch.ones(H*W, H*W), diagonal=0).T
    xv, yv = torch.meshgrid([ordering, ordering])
    new_att_mask = torch.zeros_like(normal_mask)
    new_att_mask[xv, yv] = normal_mask

    if (conv_type == "convB" and pad_type == "pad_4") or \
            (conv_type == "convC" and pad_type == "pad_2"):
        return new_att_mask.T.int()
    return new_att_mask.int()


class Orderings(object):
    _current_ordering = None
    # A list of the two chosen orderings to be used to mask the gradinets
    _last_ordering = []
    # In case of attention, we alternate between masked and zigzag masks
    _mask_types = [None]
    # In case of attention, the current mask type
```

```
_current_mask_type = None

@classmethod
def _change_mask(cls):
    """
    In case we're using both zigzag and raster-scan masks with attention
    """
    cls._current_mask_type = np.random.choice(cls._mask_types)

@classmethod
def change_order(cls):
    # Random sampling precedure of the orderings
    idx = np.random.choice(range(len(ORDERING_POSSIBILITIES)))
    cls._current_ordering = ORDERING_POSSIBILITIES[idx]

    # Keeping track of the chosen orderings to mask the gradients afterwards
    if len(cls._last_ordering) == 2:
        cls._last_ordering = []
    cls._last_ordering.append(cls._current_ordering[0])

    # Change mask, only used with attention
    cls._change_mask()

@classmethod
def get_mask(cls, H, W):
    """
    Called from the attention block to get
    the corresponding mask Mi for orderings Oi
    """
    conv_type, pad_type = cls._current_ordering
    ordering = get_ordering(conv_type, pad_type, H, W)
    if cls._current_mask_type == "zigzag":
        ordering = get_zigzag_ordering(ordering, H, W)
    att_mask = from_order_to_att_mask(H, W, ordering, conv_type, pad_type)
    return att_mask.unsqueeze(0)
```

References

1. Caesar, H., Uijlings, J., Ferrari, V.: Coco-stuff: Thing and stuff classes in context. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 1209–1218 (2018) 8, 9
2. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the thirteenth international conference on artificial intelligence and statistics. pp. 249–256 (2010) 2
3. Ji, X., Henriques, J.F., Vedaldi, A.: Invariant information clustering for unsupervised image classification and segmentation. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 9865–9874 (2019) 3, 6, 8, 9
4. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014) 2
5. Oord, A.v.d., Li, Y., Vinyals, O.: Representation learning with contrastive predictive coding. arXiv preprint arXiv:1807.03748 (2018) 4