All-to-key Attention for Arbitrary Style Transfer
Supplementary Material

Mingrui Zhu*
Xidian University
Xi'an, China
mrzhu@xidian.edu.cn

Xiao He*
Xidian University
Xi'an, China
xiaohe@stu.xidian.edu.cn

Nannan Wang†
Xidian University
Xi'an, China
nnwang@xidian.edu.cn

Xiaoyu Wang
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
fanghuaxue@gmail.com

Xinbo Gao
Chongqing University of Post and Telecommunications
Chongqing, China
gaoxb@cqupt.edu.cn

1. Implementation Details

1.1. Step 1 of DA

Figure 1 provides a detailed illustration of the first step in distributed attention (DA). The first step of DA is regional style aggregation. In this step, we aggregate the information of all points in a block to a point, representing the block's style information. For example, we aggregate K_1, K_2, K_3, and K_4 points in the first block of K^l to a point k. Note that all blocks perform regional style aggregation operations in parallel, so we will finally get the regional style representations corresponding to all blocks, which form new keys $\widehat{K}^l$.

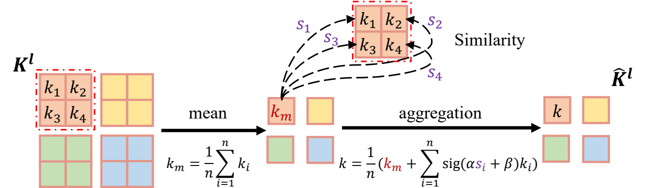


Figure 1. Detailed illustration of the step 1 in DA.

1.2. Step 1 of PA

Figure 2 gives a detailed illustration of the first step in progressive attention (PA). The first step of PA is implemented as patch-wise attention along the first axis, which takes a block region as a token instead of a specific position. In this step, we apply argmax to the attention score to obtain the indices of the most similar coarse-grained region. For example, the $4th$ block region in K^l is matched as the coarse-grained region most similar to the blue block region in Q^l, so the index of this region is set to 4. With the indices, we can reshuffle the tokens of K^l to semantically match the spatial arrangement of the tokens of Q^l. The reshuffled $\widetilde{K}^l$ is further utilized in the second step of PA.

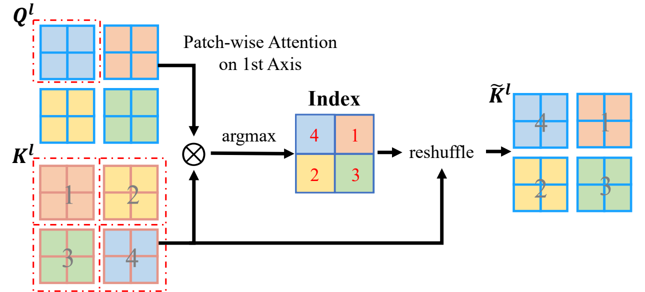


Figure 2. Detailed illustration of the step 1 in PA.

1.3. Decoder Architecture

The decoder implemented in this work follows the setting of [1], which mirrors the encoder and takes the multi-scale transferred features as input. Full decoder configuration is shown in Table 1. The decoder takes the multi-scale transferred features F^l_{cs} as input and gradually synthesizes the final image I_{cs}.

1.4. All-to-key Attention Algorithm

The PyTorch code for our proposed all-to-key attention mechanism is shown in Algorithm 1. The implementation

*Both authors contributed equally to this work.
†Corresponding author: Nannan Wang.

Table 1. Full configuration of the decoder.

Stage	Output	Architecture
F^5	$512 \times \frac{H}{8} \times \frac{W}{8}$	Input F_{cs}^5 Upsample scale 2 Add F_{cs}^4 3×3 Conv, 512, ReLU
F^4	$256 \times \frac{H}{4} \times \frac{W}{4}$	3×3 Conv, 256, ReLU Upsample scale 2
F^3	$128 \times \frac{H}{2} \times \frac{W}{2}$	Concatenate F_{cs}^3 (3×3 Conv, 256, ReLU) $\times$ 3 3×3 Conv, 128, ReLU Upsample scale 2
F^2	$64 \times H \times W$	3×3 Conv, 128, ReLU 3×3 Conv, 64, ReLU Upsample scale 2
F^1	$3 \times H \times W$	3×3 Conv, 64, ReLU 3×3 Conv, 3

Table 2. Evaluation scores of the results generated under different ratios. The best results are in **bold**.

λ_2	Content Loss $\downarrow$	Style Loss $\downarrow$	LPIPS $\downarrow$
0.5	0.72	**0.75**	0.57
0.75	0.63	0.82	0.55
1	0.58	0.98	0.54
1.25	0.55	1.04	0.53
1.5	0.52	1.20	0.52
2	**0.49**	1.38	**0.49**

is elegant with the usage of einsum notation.

1.5. Blocking and Unblocking Algorithm

The PyTorch code for feature blocking and unblocking operation in the all-to-key attention mechanism is shown in Algorithm 2

1.6. Loss Function

Setting of λ_1 and λ_2 We empirically set the weight of each loss term to 10 and 1.25. Since we only have two loss terms, we can fix the weight of one loss and choose the appropriate weight according to the impact of adjusting the weight of another loss. We fix λ_1 to 10 and statistically analyze the evaluation scores of the results generated under different settings of λ_2, as shown in Table 2. As the proportion of λ_2 increases, content loss, LPIPS score decreases, and style loss increases. Therefore, there is a trade-off in style and content. We finally set λ_2 to 1.25 to render consistent style texture while maintaining semantic structure.

2. More Results

2.1. All-to-key Attention VS. All-to-all Attention

To further illustrate the superiority of our proposed all-to-key attention to all-to-all attention in producing high-quality stylized images, we provide more visual comparisons in Figure 3. By replacing all-to-key attention in our full model with all-to-all attention, the visual quality of the stylized images decreases significantly, affected by distorted style patterns.

2.2. Arbitrary Style Transfer

To further demonstrate the effectiveness and robustness of our proposed StyA2K on arbitrary style transfer, we provide more stylization results of pair-wise combinations between 10 content images and 8 style images (total 80 stylized images) in Figure 4 and Figure 5. Our method can faithfully generate visually appealing results with consistent style textures. We also provide more video stylization results, as shown in Figure 6. StyleA2K shows high temporal consistency.

References

[1] Songhua Liu, Tianwei Lin, Dongliang He, Fu Li, Meiling Wang, Xin Li, Zhengxing Sun, Qian Li, and Errui Ding. Adaattn: Revisit attention mechanism in arbitrary neural style transfer. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6649–6658, 2021.

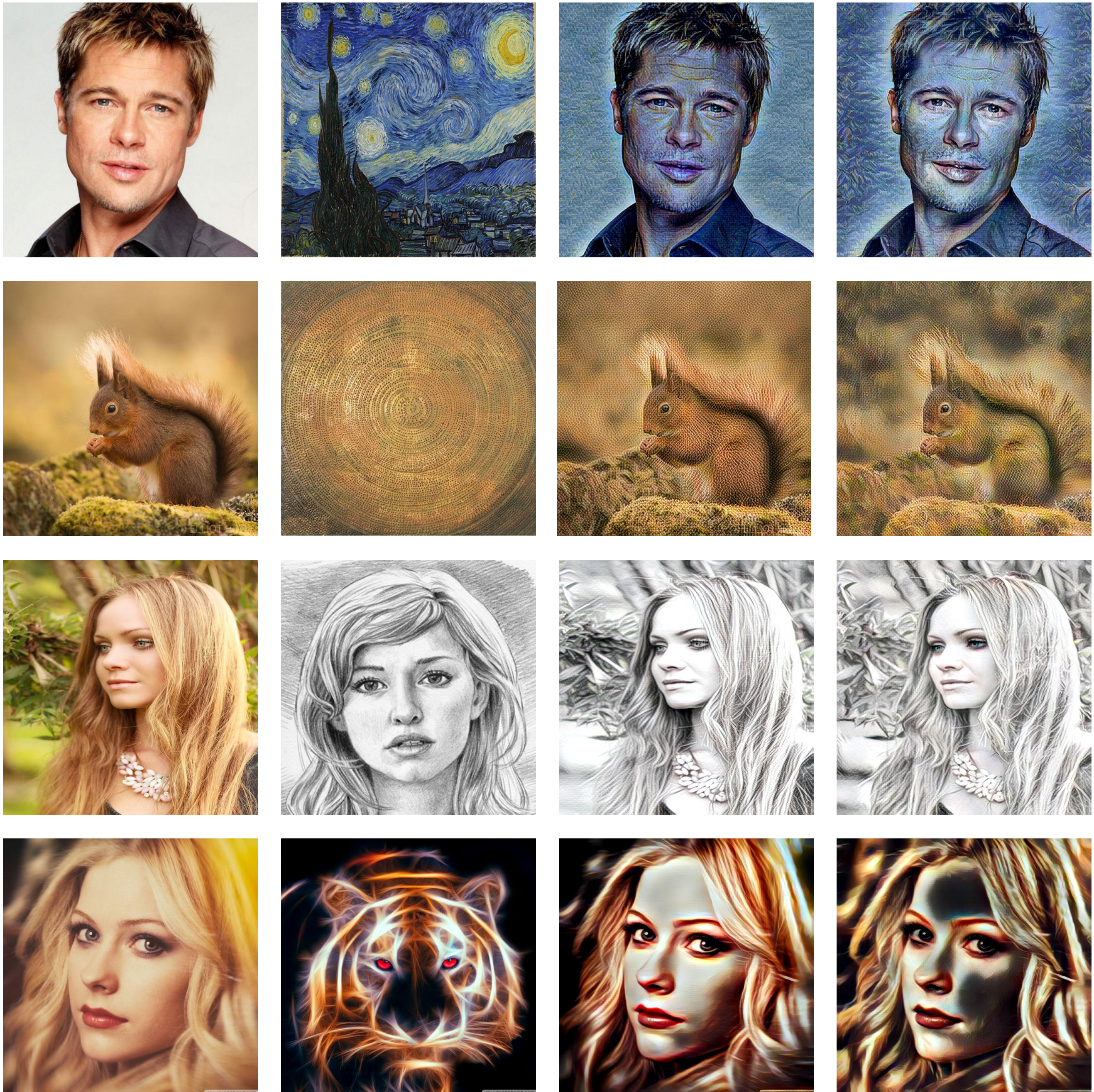


Figure 3. More visual comparisons between our proposed all-to-key attention and all-to-all attention.

Algorithm 1 Pytorch code implementing all-to-key attention.

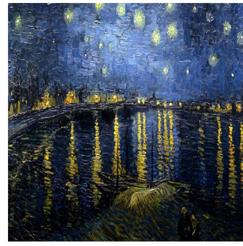
```
1  class A2K(nn.Module):
2      '''All-to-key Attention'''
3  def __init__(self, in_dim):
4      super().__init__()
5      self.Dq = nn.Conv2d(in_dim, in_dim, (1, 1))
6      self.Dk = nn.Conv2d(in_dim, in_dim, (1, 1))
7      self.Dv = nn.Conv2d(in_dim, in_dim, (1, 1))
8      self.Pq = nn.Conv2d(in_dim, in_dim, (1, 1))
9      self.Pk = nn.Conv2d(in_dim, in_dim, (1, 1))
10     self.Pv = nn.Conv2d(in_dim, in_dim, (1, 1))
11     self.sim_alpha = nn.Parameter(torch.ones(1), require_grad=True)
12     self.sim_beta = nn.Parameter(torch.zeros(1), require_grad=True)
13     self.fusion_D = nn.Conv2d(in_dim, in_dim, (1, 1))
14     self.fusion_P = nn.Conv2d(in_dim, in_dim, (1, 1))
15
16 def  forward(self, content, style):
17     # Get Q K V
18     DA_q = block(self.Dq(mean_variance_norm(content)), p_size, stride)
19     DA_k = block(self.Dk(mean_variance_norm(style)), p_size, stride)
20     DA_v = block(self.Dv((style)),p_size,stride)
21     PA_q = block(self.Pq(mean_variance_norm(content)), p_size, stride)
22     PA_k = block(self.Pk(mean_variance_norm(style)), p_size, stride)
23     PA_v = block(self.Pv(style), p_size, stride)
24     # Distributed Attention Step 1
25     DA_k_m = torch.mean(DA_k, dim=-1)
26     DA_v_m = torch.mean(DA_v, dim=-1)
27     dis = torch.einsum("bhcx,bhcxy->bhxy", DA_k_m, DA_k)
28     sim = torch.sigmoid(self.sim_beta+self.sim_alpha*dis)
29     DA_k_a = (torch.einsum("bhxy,bhcxy->bhcx", sim, DA_k) + DA_k_m) / p_size
30     DA_v_a = (torch.einsum("bhxy,bhcxy->bhcx", sim, DA_v) + DA_v_m) / p_size
31     # Distributed Attention Step 2
32     logits = torch.einsum("bhcxy,bhcz->bhyxz", DA_q, DA_k_a)
33     scores =  softmax(logits)
34     DA = torch.einsum("bhyxz,bhcz->bhcxy", scores, DA_v_a)
35     DA_unblock = unblock(DA)
36     # Progressive Attention Step 1
37     PA1_logits = torch.einsum("bhcxy,bhczy->bhxz", PA_q, PA_k)
38     index = torch.argmax(PA1_logits, dim = -1).expand_as(PA_k)
39     PA_k_reshuffle = torch.gather(PA_k, -2, index)
40     PA_V_reshuffle = torch.gather(PA_v, -2, index)
41     # Progressive Attention Step 2
42     logits2 = torch.einsum("bhcxy,bhcxz->bhxyz", PA_q, PA_k_reshuffle)
43     scores2 = softmax(logits2)
44     PA = torch.einsum("bhxyz,bhcxz->bhcxy", scores2, PA_V_reshuffle)
45     PA_unblock = unblock(O2)
46     # Feature Transformation
47     O_DA = self.fusion_D(DA_unblock)
48     O_PA = self.fusion_P(PA_unblock)
49     O = O_DA + O_PA
50     out = O + Content
51     return out
```

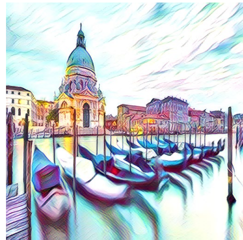
Algorithm 2 Pytorch code implementing feature blocking and unblocking.

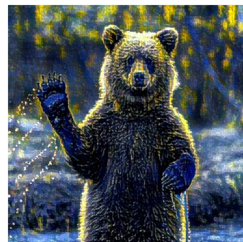
```
def block(X, patch_size, stride):
    '''feature blocking.
    Args:
        X: a tensor with shape [b, c, h, w], where b is batch size, c is the
        channel dimension, h is feature height, and w is feature width.
        patch_size: an integer for the patch (block) size
        stride: the parameter of torch.nn.functional.unfold
    Returns:
        Y: a tensor with shape [b, c, n, r], where n is patch sequence length
        and r is the patch size.
    '''
    b, c, h, w = X.shape
    r = int(patch_size**2)
    Y = torch.nn.functional.unfold(X, kernel_size=patch_size, stride=stride)
    Y = Y.view(b, c, r, -1).permute(0, 1, 3, 2)
    return Y

def unblock(X, patch_size, stride, h):
    '''feature unblocking.
    Args:
        X: a tensor with shape [b, c, n, r], where b is batch size, c is
        channel dimension, n is patch sequence length, r is the patch size.
        patch_size: an integer for the patch (block) size
        stride: the parameter of torch.nn.functional.unfold
        h: the output height of the feature
    Returns:
        Y: a tensor with shape [b, c, h, w], where h is is feature height
        and w is feature width.
    '''
    b, c, n, r = X.shape
    X = X.permute(0, 2, 1, 3)
    X = X.contiguous().view(b, n, -1).permute(0, 2, 1)
    Y = torch.nn.functional.unfold(X, h, kernel_size=patch_size, stride=stride)
    return Y
```

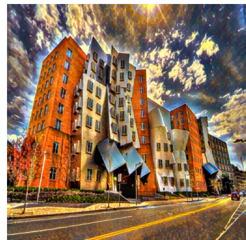
Style

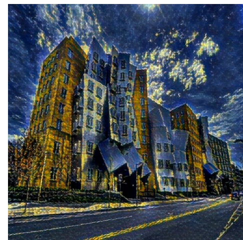
Content

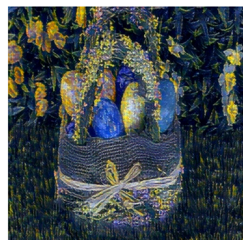












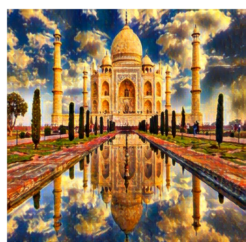
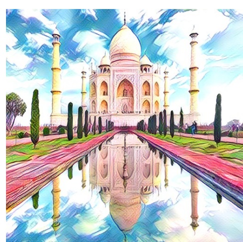
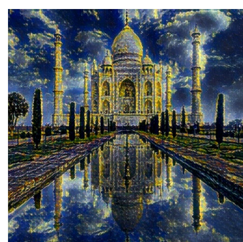
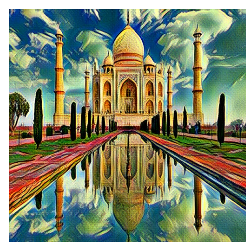





Figure 4. More image stylization results.

Figure 5. More image stylization results.

Figure 6. Video stylization results of AdaAttN (second column) and our StyA2K (third column). Adobe Acrobat is recommended to view the videos.