Supplementary Material
Cross-Domain Cascaded Deep Translation

Oren Katzir[1], Dani Lischinski[2], and Daniel Cohen-Or[1]

[1] Tel-Aviv University
[2] Hebrew University of Jerusalem

1 Training details

1.1 Hyper parameters

In all our experiments, unless stated otherwise, we use Adam optimizer [18] with $\beta_1 = 0.5, \beta_2 = 0.999$. The learning rate was set to 0.0001 and the batch size to 10. During training, random crop and image mirroring is applied. Our training methodology follows WGAN-GP [10], thus for one generator update we update the discriminator four times.

1.2 Network architecture

Feature Inversion Our implementation is similar to [5]. We train an individual feature inversion network for VGG layer, where each layer has different channels (512, 512, 256, 128, 64). All layers utilize Leaky ReLU nonlinearity (0.2) and employ no normalization. The last layer utilizes Tanh. All inversion networks, first apply three non-strided convolutional layers, with N input number of channels, equal to the number of channels each deep layer has. Next, several transpose convolutional layers are applied, each doubles the resolution of the image and decreases the channel resolution (by factor of 2) until the image resolution 224 is achieved (thus, different amount of ConvTranspose layers per layer). The final layer is a non-strided convolutional layer followed by Tanh layer. Together they project the features back to the original image dimensions and range (number of output channels is 3). For the discriminator we have used Patch GAN discriminator, with four strided convolutional layers, each utilizes batch normalization (except the first one) and Leaky ReLU. For the adversarial metric, only here, we have used LS-GAN. Here we also set the batch size to 25.

Deepest layer translation The input to the deepest translation network is `conv_5_1`, thus, the input size is $14 \times 14 \times 512$ (recall the input image size is 224×224). The identity and cycle losses are multiplied by $\lambda_{idty} = \lambda_{cyc} = 100$. The architecture is reported in Table.1. The networks is relatively small and achieve good results in a few hours on a single GPU (RTX2080). We use the WGAN-GP optimization method, updating the generator once for every four discriminator updates.

Name	Input ch.	Output ch.	Kernel sz.	Stride	GN
conv	512	512	3	1	no
conv	512	256	3	2	yes
relu	-	-	-	-	-
conv	256	512	3	2	yes
relu	-	-	-	-	-
convT	512	256	3	2	yes
relu	-	-	-	-	-
convT	256	256	3	2	yes
relu	-	-	-	-	-
conv	256	512	3	1	yes
relu	-	-	-	-	-
conv	512	512	3	1	no
tanh	-	-	-	-	-

Table 1. Deepest layer translation architecture.

Coarse to fine conditional translation The coarse-to-fine generator, for generating level i, has two inputs: the current source VGG level and the previous translated VGG features $(i+1)$. An AdaIN component, acts on on the current deep features and normalizes several layers in the translator itself. We report the AdaIN component structure for generating layer four in Table.2. The architecture can be extended easily to other layers. The core components of the translator, which takes as input the previous translated layer, are reported in Table.3.

1.3 Different networks

VGG-19 fine-tuned We fine-tuned VGG-19 by fixing all layers but the fully connected layers and `conv_5_i` (`i` $= 1, 2, 3, 4$). We replaced the final fully connected layer with new fully connected layer of size 4096×2 corresponding to the two domains. We trained the classifier for 50 epochs, with batch size of 25 and learning rate of 0.01.

AlexNet We extract each of the 5 convolution of AlexNet as different layers. Each layer was normalized, in a similar manner as was described for VGG-19.

Name	Input ch.	Output ch.	Kernel sz.	Stride	GN
conv	512	512	3	2	no
lrelu	-	-	-	-	-
conv	512	512	3	2	no
lrelu	-	-	-	-	-
conv	512	512	3	2	no
lrelu	-	-	-	-	-
linear	$4 \times 4 \times 512$	1000	-	-	no
lrelu	-	-	-	-	
linear	1000	x	-	-	no

Table 2. AdaIN component for the second deepest layer. The output x is equal to the number of parameters the AdaIN normalizes. AdaIN for different VGG layer's translation are defined similarly, where we simply add more conv layer for each shallower VGG layer.

Name	Input ch.	Output ch.	Kernel sz.	Stride	AdaIN
conv	x	x	3	1	yes
lrelu	-	-	-	-	-
conv	x	x	3	1	yes
lrelu	-	-	-	-	-
convT	x	$x/2$	4	2	yes
lrelu	-	-	-	-	-
conv	$x/2$	$x/2$	3	1	no
tanh	-	-	-	-	-

Table 3. Coarse to fine translator. The input number of channels, x, varies according to the current VGG layer.

2 More comparison results

In this section we show more results, not presented in the paper, for zebra↔giraffe, zebra↔elephant and cat↔dog translations.

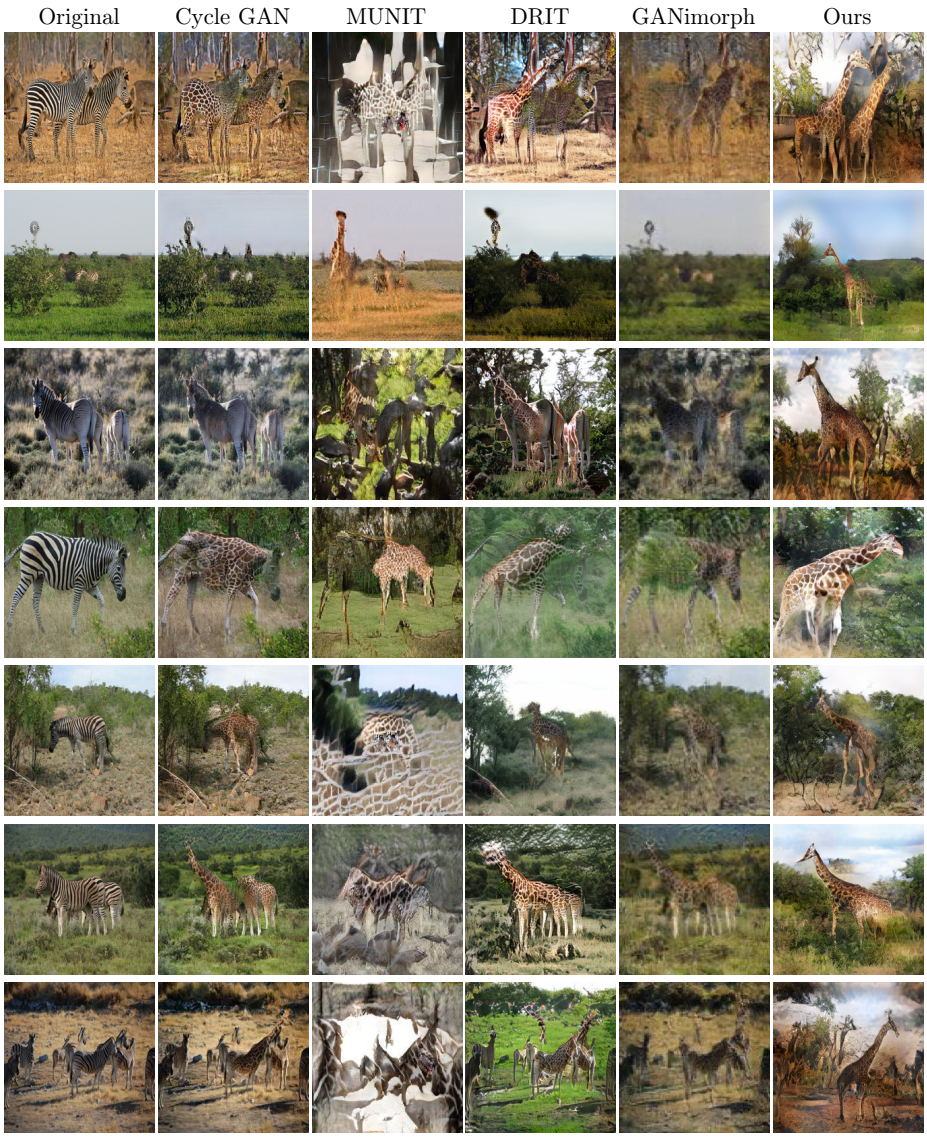

Fig. 1. Qualitative comparisons. MSCOCO zebra to giraffe.

Fig. 2. Qualitative comparisons. MSCOCO zebra to giraffe.

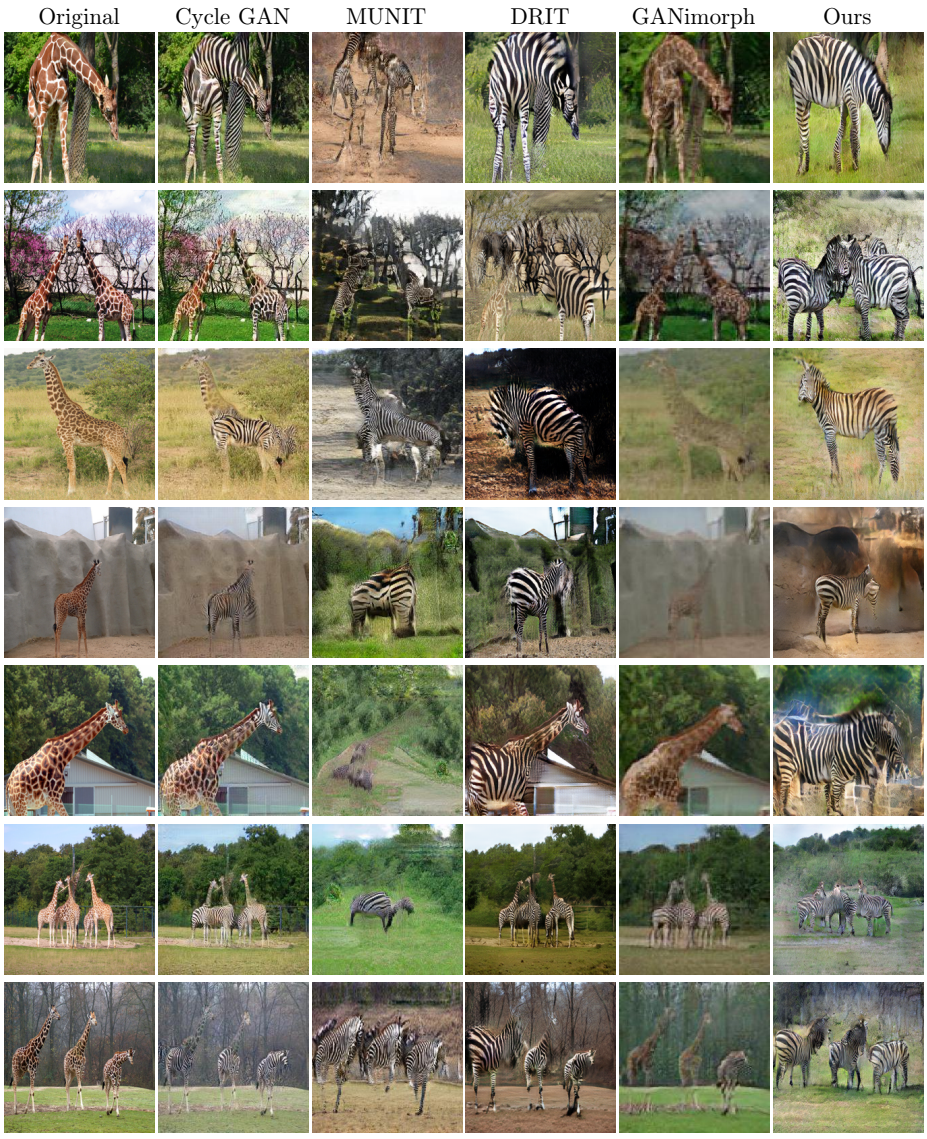

Fig. 3. Qualitative comparisons. MSCOCO giraffe to zebra.

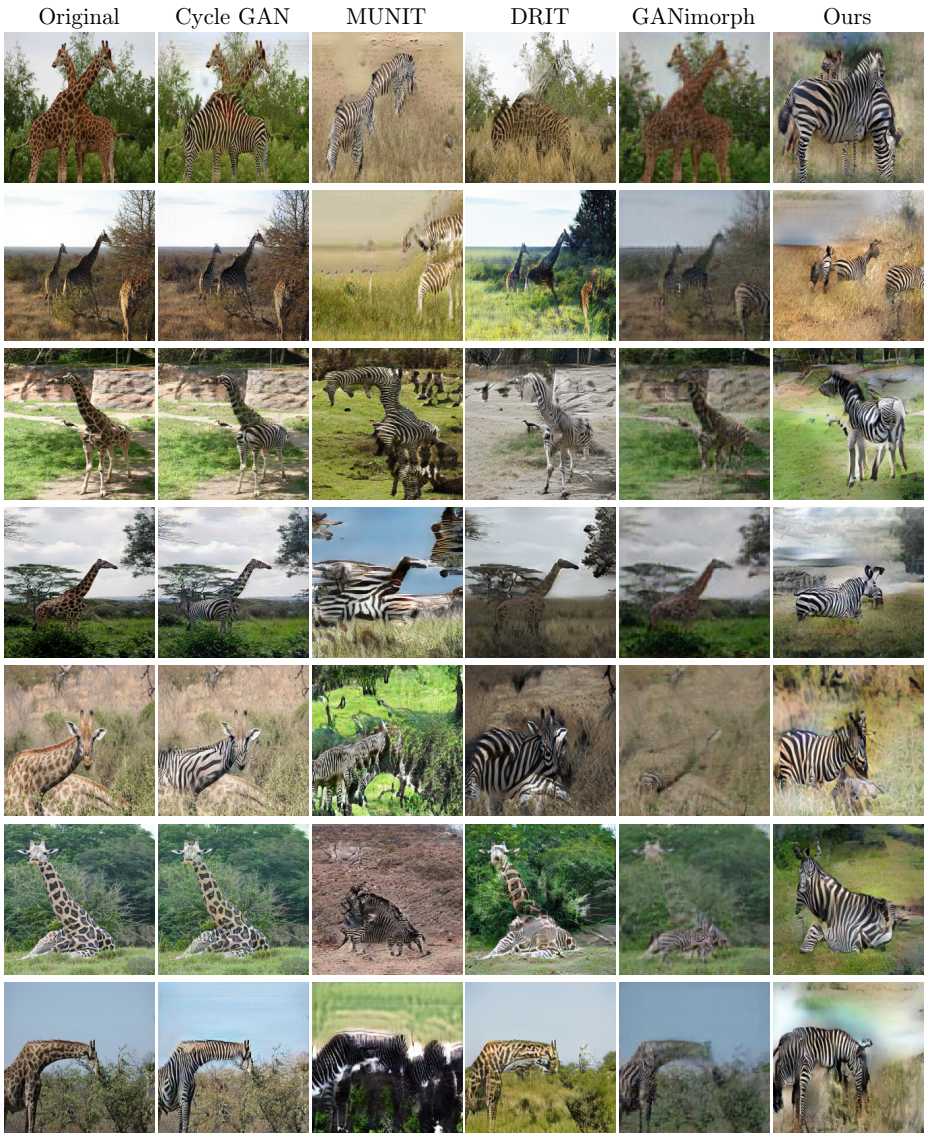

Fig. 4. Qualitative comparisons. MSCOCO giraffe to zebra.

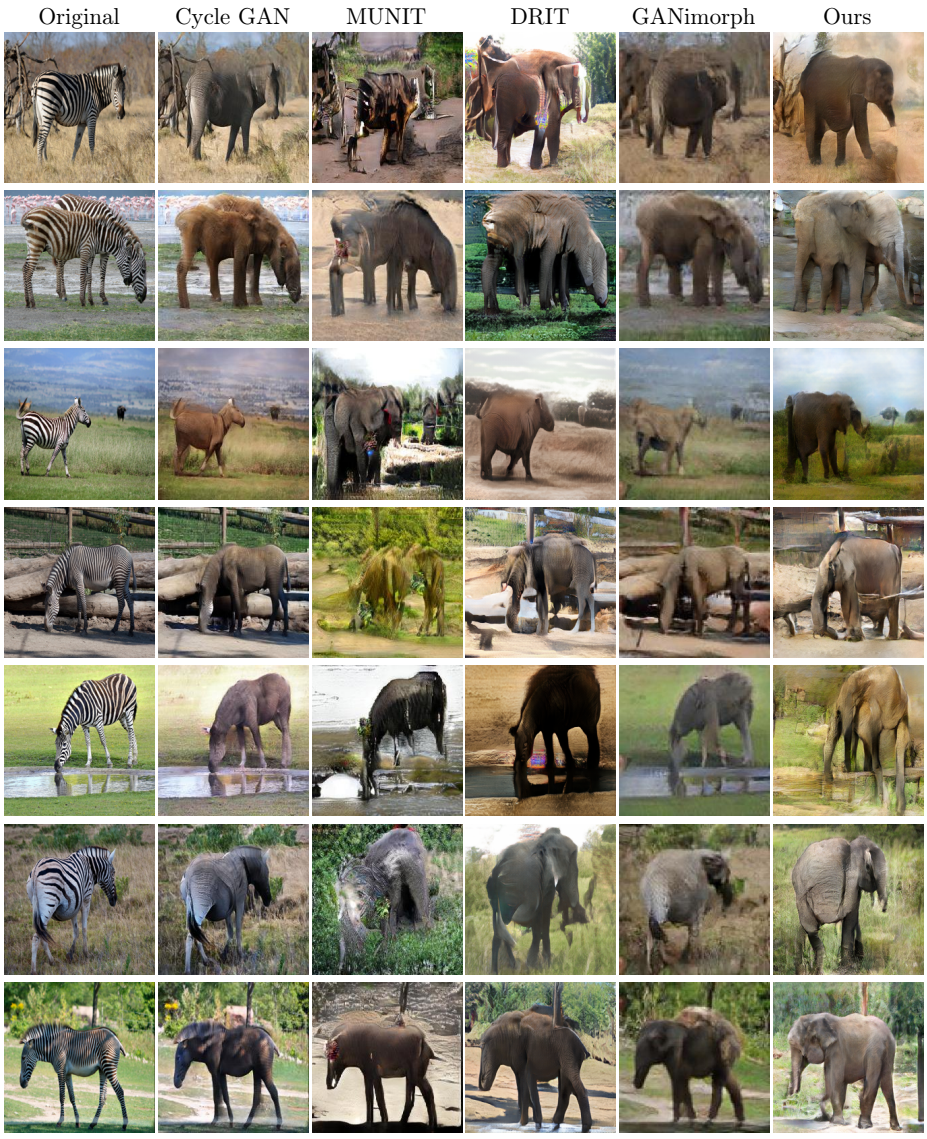

Fig. 5. Qualitative comparisons. zebra to elephant.

Fig. 6. Qualitative comparisons. zebra to elephant.

Fig. 7. Qualitative comparisons. MSCOCO elephant to zebra.

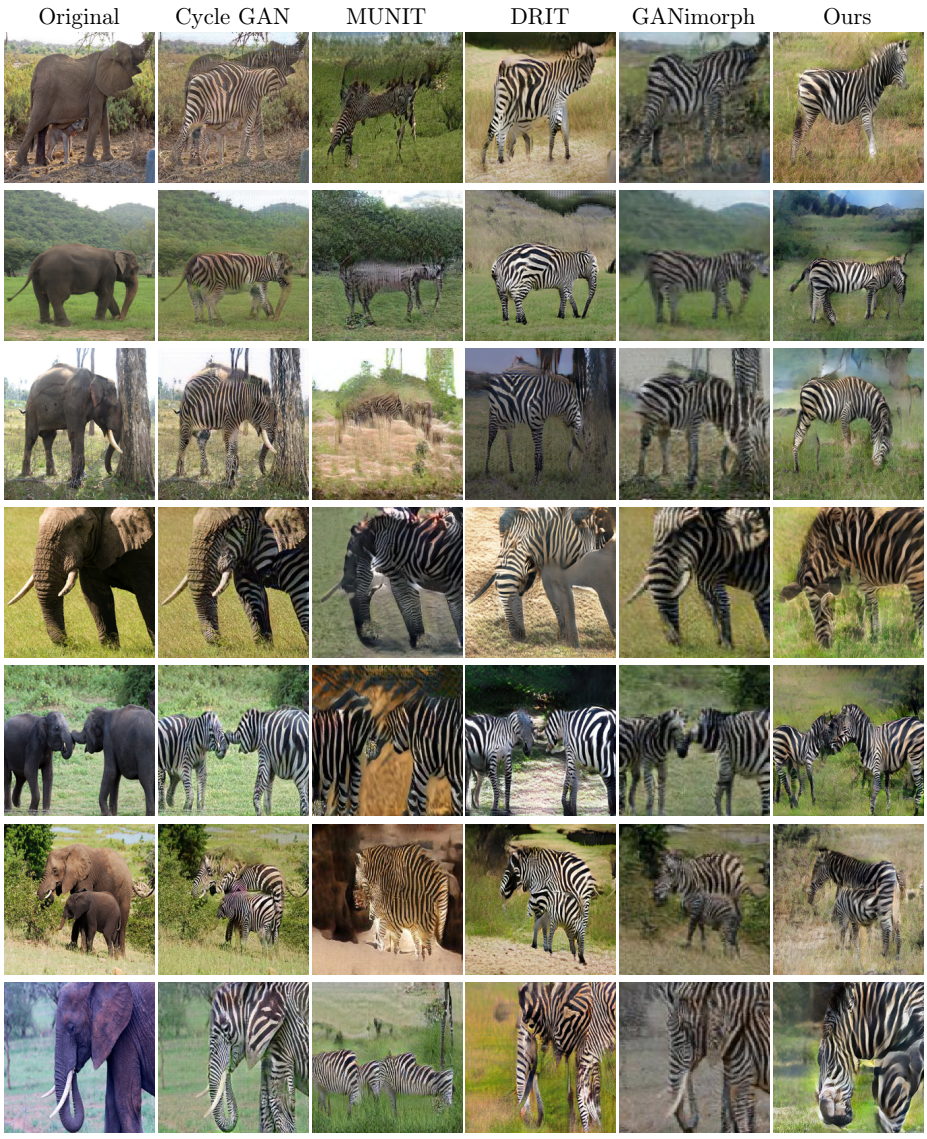

Fig. 8. Qualitative comparisons. MSCOCO elephant to zebra.

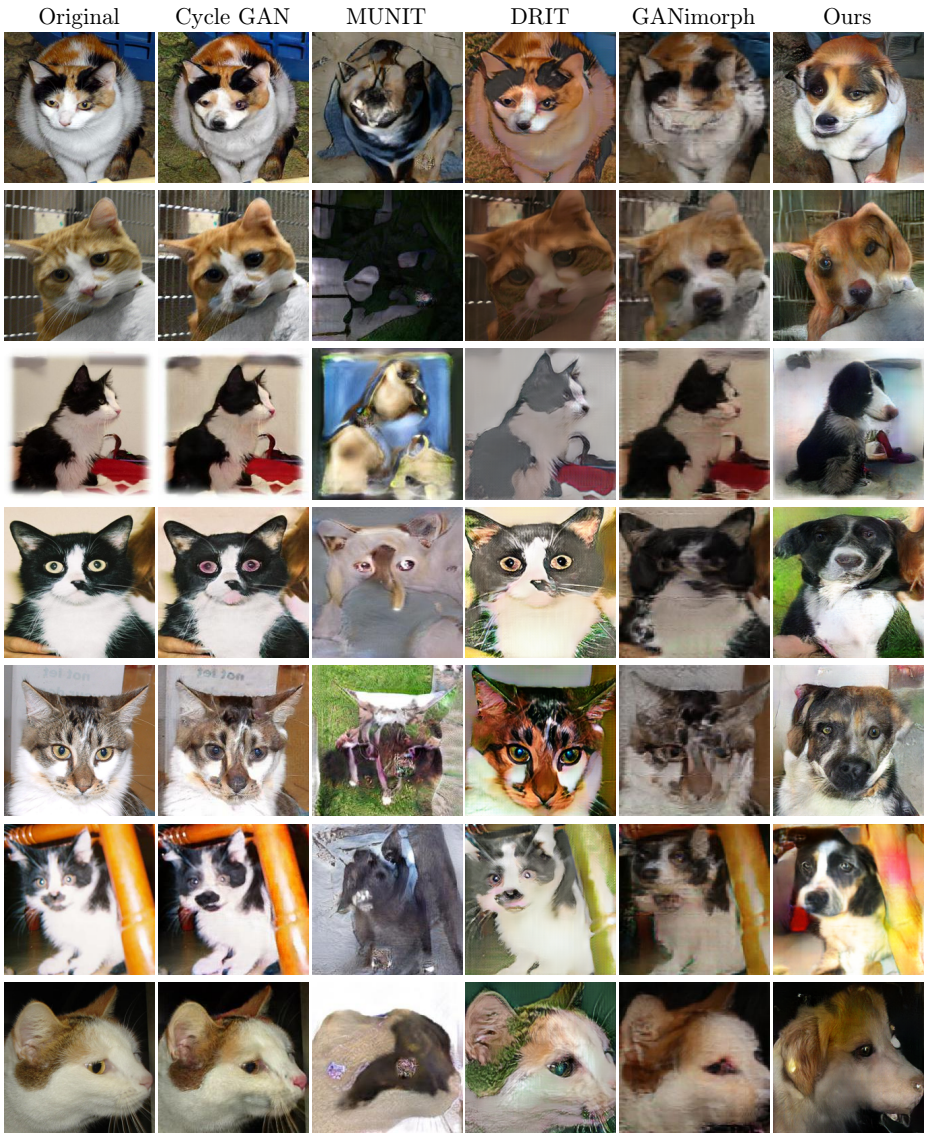

Fig. 9. Qualitative comparisons. Kaggle cat to dog.

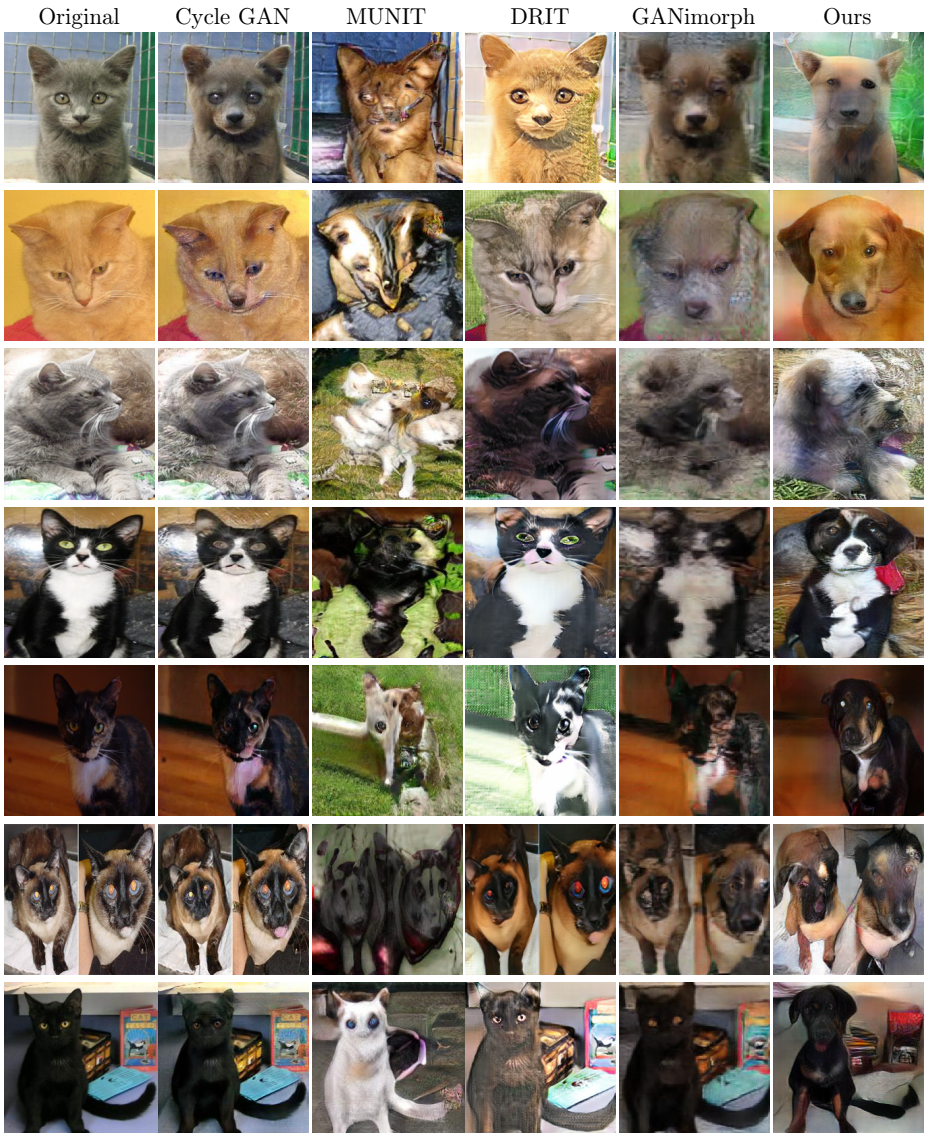

Fig. 10. Qualitative comparisons. Kaggle cat to dog.

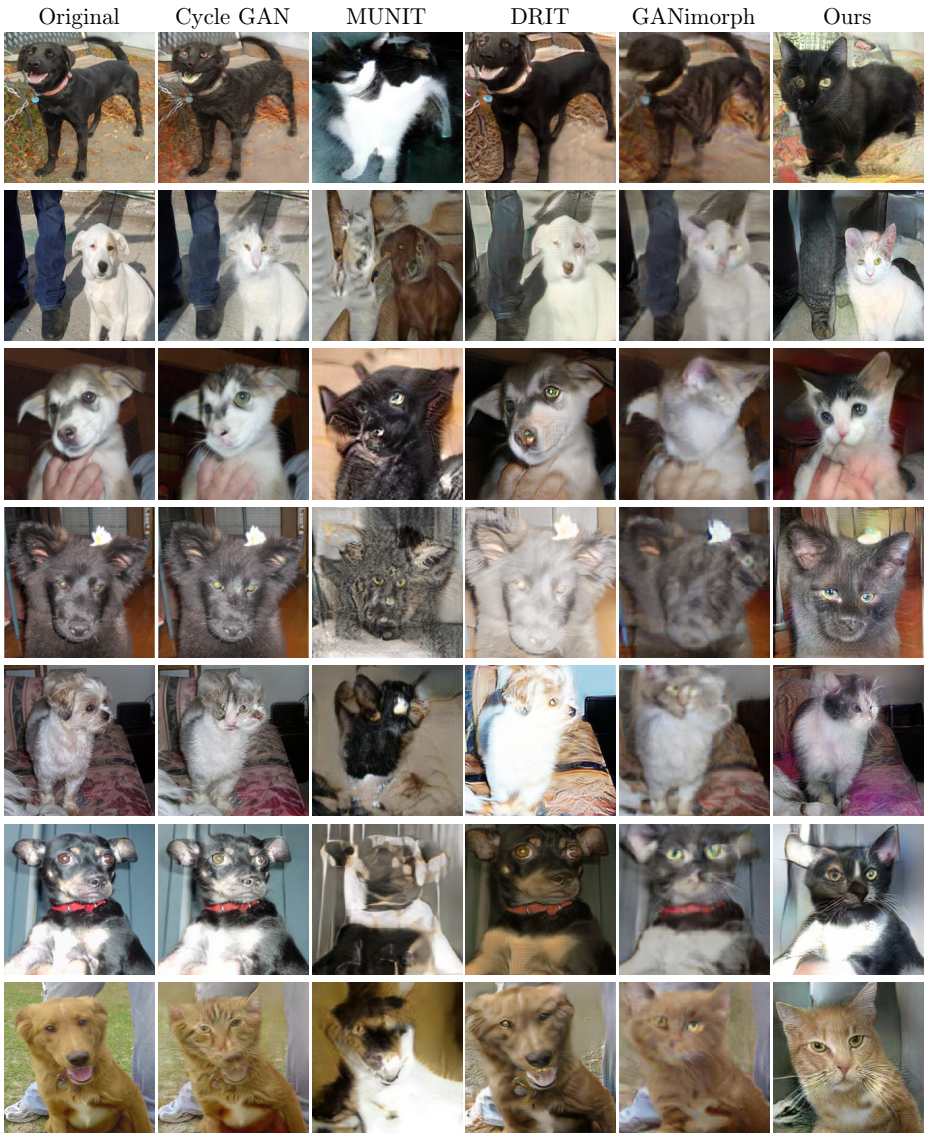

Fig. 11. Qualitative comparisons. Kaggle dog to cat.

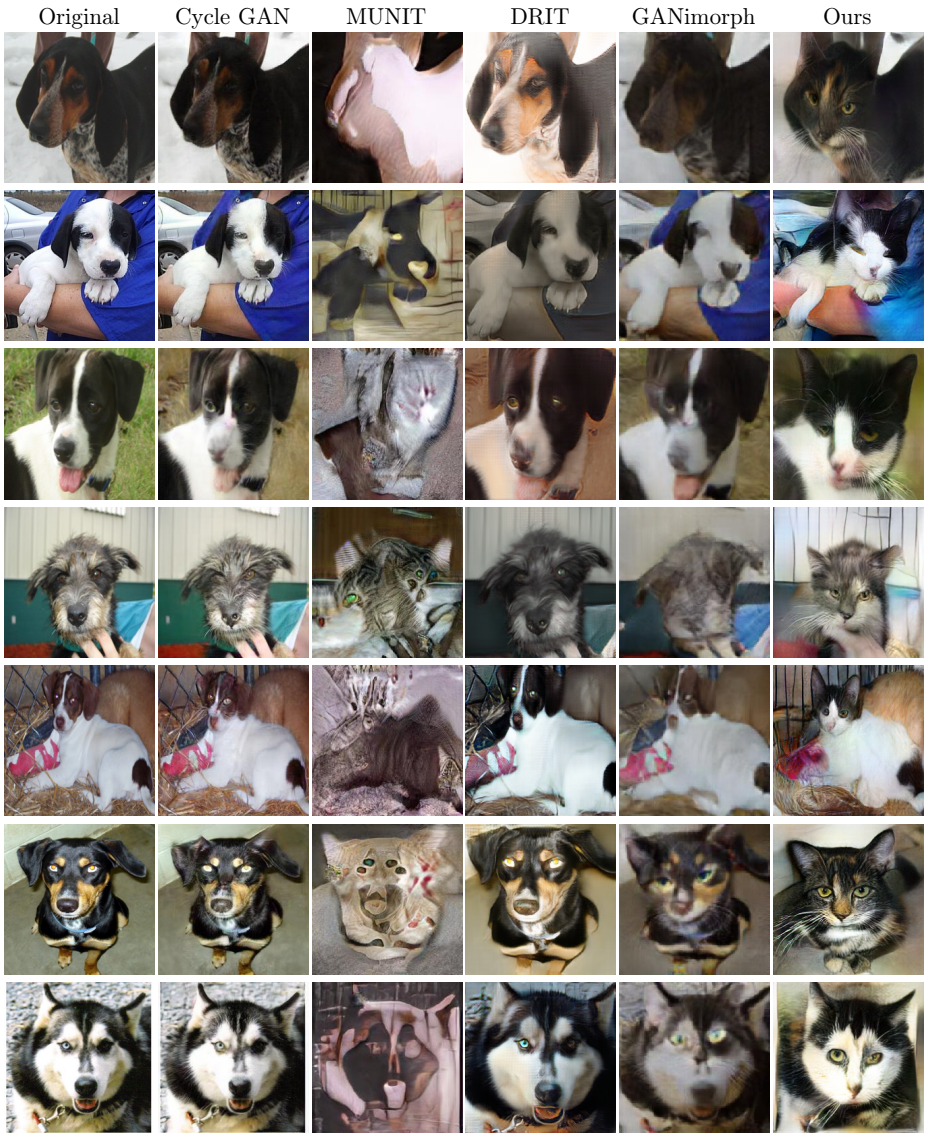

Fig. 12. Qualitative comparisons. Kaggle dog to cat.

3 Non-shape deformation translation

Our method is also suited to none-shape deformation tasks, as in the case of dataset(1) [20].

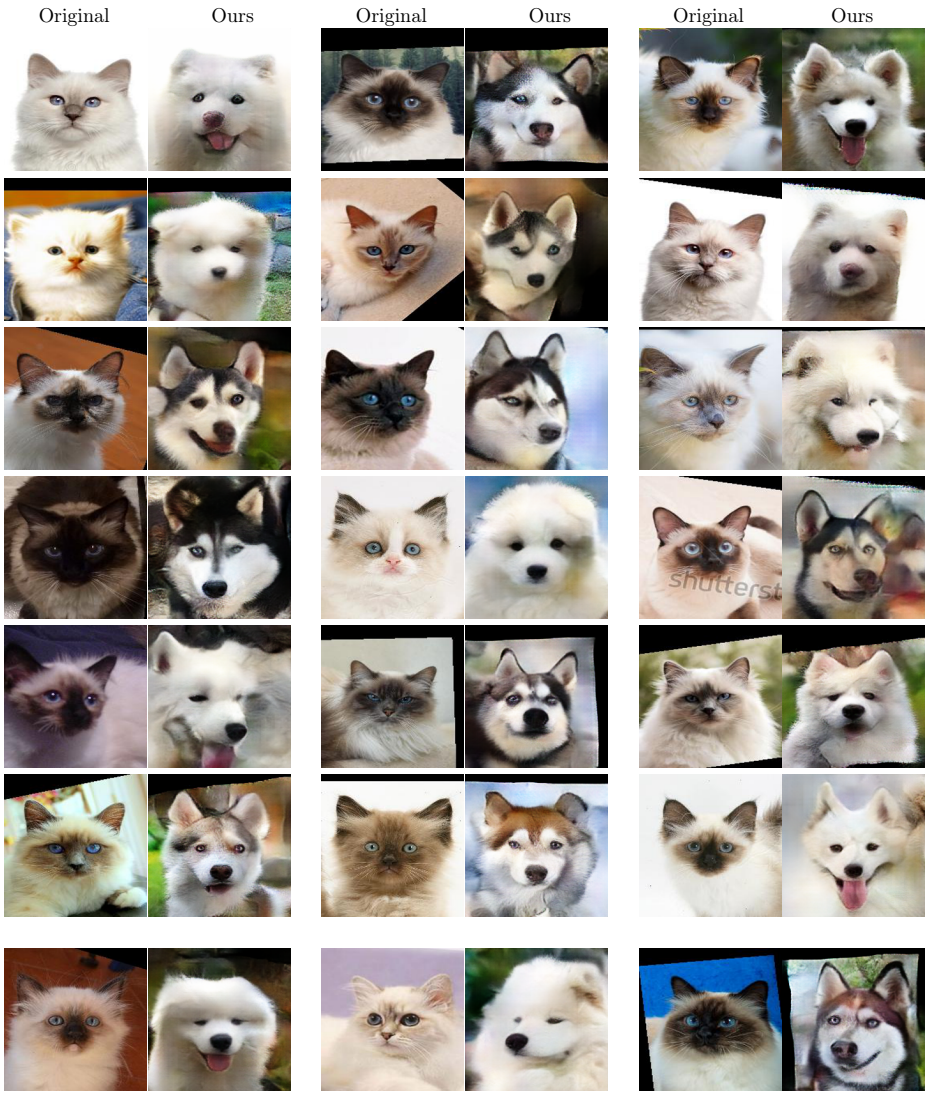

Fig. 13. Translation results from cats to dogs (faces).

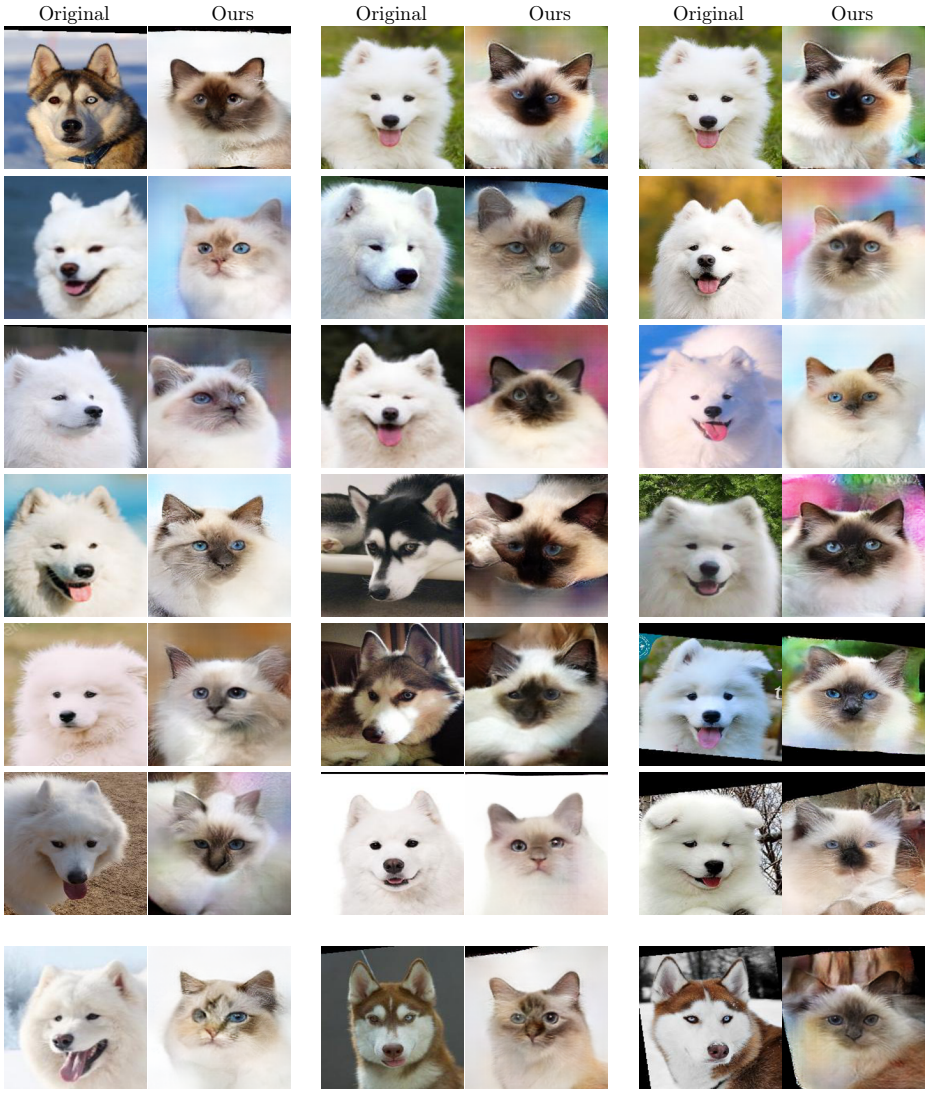

Fig. 14. Translation results from dogs to cats (faces).

4 Coarse to fine translation

We here present the translation of each layer. The translation of each shallower layer is conditioned on the translation result of the previous layer, and learns to add fine scale and appearance, such as texture. At every layer, in order to visualize the generated deep features, we use a network pre-trained for inverting the deep features of VGG-19, following the method in [5].

Fig. 15. Coarse to fine translation of zebra to giraffe. Two different examples are shown in each row. The original image (left) is translated by the deepest translator (second left) and then in coarse to fine manner, shallower layers are translated (second right and most right).

Fig. 16. Coarse to fine translation of giraffe to zebra. Two different examples are shown in each row. The original image (left) is translated by the deepest translator (second left) and then in coarse to fine manner, shallower layers are translated (second right and most right).

5 Nearest neighbor comparison

In this section we show side by side, source images, our translation and the three nearest neighbors in the target domain. We use the LPIPS metric, presented in "The unreasonable effectiveness of deep features as a perceptual metric" by Zhang et al. This metric is based on ℓ_2 distance of deep features extracted from pre-trained network. In our case we use the default settings proposed by Zhang et al. (i.e. alex net). As we show, the closest image in the target dataset vary in pose, scale and content (i.e. different parts of the objects).

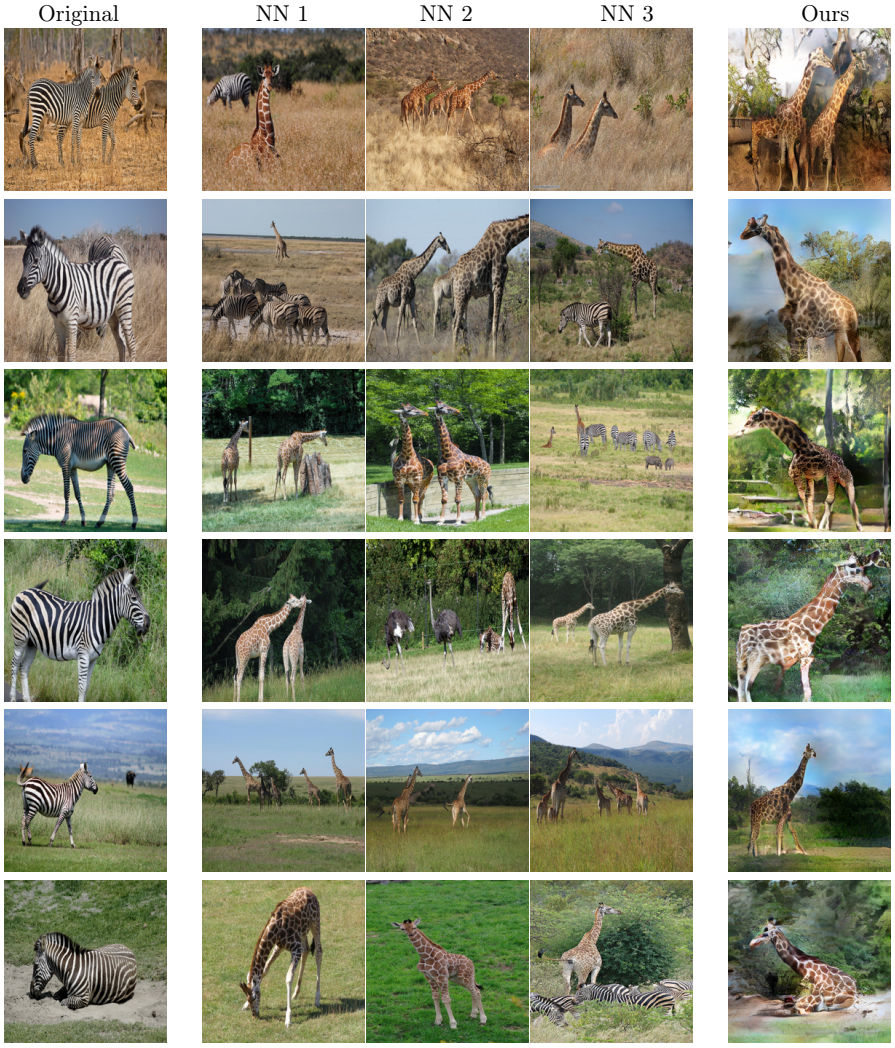

Fig. 17. Nearest neighbor comparison to our result for zebra to giraffe translation. The NNs were found by exhaustive search on all the giraffe dataset using perceptual metric (LPIPS). The closest giraffe to the source zebra vary in scale, position and content

Fig. 18. Nearest neighbor comparison to our result for giraffe to zebra translation. The NNs were found by exhaustive search on all the giraffe dataset using perceptual metric (LPIPS). The closest zebra to the source giraffe vary in scale, position and content.

6 deepVAE

Here we explain the exact architecture and training procedure of our unconditional synthesis module, deepVAE. We feed-forward all images through a pre-trained (on ImageNet) VGG-19, extracting `conv_i_1` (`i=1,...5`). We start by training the deepest synthesis module, G_5, synthesizing `conv_5_1`. We chose a simple variational auto encoder (VAE) architecture for this synthesis module, as shown in Fig. 19, as it easier to train and the blurry results, VAE are notoriously known for, will be refined by the following synthesis modules. Thus, for the deepest layer our loss function is

$$\mathcal{L}_{VAE}\left(G_5, E_5, x\right) = \underset{q_{E_5}(z_5|x)}{\mathbb{E}}\left[\log p_{G_5}(x|z_5)\right] - D_{KL}\left(q_{E_5}(z_5|x)|p(z_5)\right), \quad (1)$$

where, the prior distribution for z_5 was chosen to be Gaussian and the reconstruction error was achieved by ℓ_2 loss. Given the synthesized `conv_5_1`, we synthesize the shallower layer `conv_4_1` conditioned on it by using adversarial training. We preform this operation, in a cascaded manner, synthesizing `conv_(i)_1` given `conv_(i+1)_1`. We achieve such synthesis by training a generator G_i with input noise z_i which is sampled from a Gaussian distribution. The discriminator D_i is tasked with discriminating between real `conv_i_1` and generated ones. The loss function is

$$\begin{aligned}\mathcal{L}_{adv}\left(G_i, D_i|G_{i+1}\right) = &\underset{x\sim\mathbb{P}_{\texttt{conv\_i+1\_1}}, z_i\sim\mathcal{N}(\mathbf{0},\mathbf{1})}{\mathbb{E}}\left[D_i\left(G_i\left(x,z\right)\right)\right] - \underset{y\sim\mathbb{P}_{\texttt{conv\_i\_1}}}{\mathbb{E}}\left[D_i\left(y\right)\right] \\ &+\lambda_{gp}\underset{\hat{y}\sim\mathbb{P}_{\hat{Y}}}{\mathbb{E}}\left[\left(\|\nabla D_i\left(\hat{y}\right)\| - 1\right)^2\right], \end{aligned} \quad (2)$$

where $\hat{Y}$ is a set of linear combinations between fake and real data samples and λ_{gp} was set to 10. Conceptually, we proceed in this manner until we reach the image space synthesis module, i.e. G_0. However, we have found G_2 to add little information to the generated images, thus we directly generate the images from `conv_3_1`. The entire training procedure and inference synthesis are shown in Fig. 19.

In Table 4 and Table 5 we elaborate the architecture used for the encoder and decoder, respectively, synthesizing `conv_5_1`.

Table 4. E_5, deepVAE, architecture. nz is the noise size, and was set to 512

Name	Input ch.	Output ch.	Kernel sz.	Stride	GN	Activation
conv	512	512	3	2	yes	lrelu
conv	512	512	3	1	yes	lrelu
conv	512	512	3	1	yes	lrelu
conv	512	512	3	1	yes	lrelu
conv	512	512	3	1	yes	lrelu
linear	$7 \times 7 \times 512$	$2 * nz$	-	-	none	none

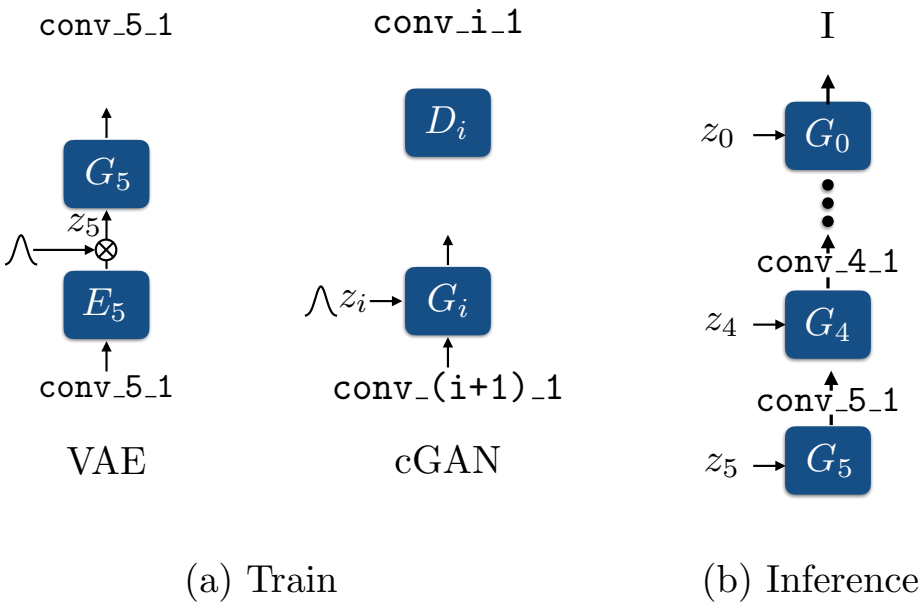

Fig. 19. DeepVAE training and inference phases. The training process (a) is divided to unconditional synthesis of the deepest layer via VAE, and conditional synthesis using conditional WGAN-GP. The inference stage (b) is achieved by applying all generators in a cascade manner.

Table 5. G_5, deepVAE, architecture.

Name	Input ch.	Output ch.	Kernel sz.	Stride	GN	Activation	Upsampled
linear	nz	$7 \times 7 \times 512$	-	-	none	none	-
conv	512	512	3	1	yes	lrelu	yes
conv	512	512	3	1	yes	lrelu	no
conv	512	512	3	1	yes	lrelu	no
conv	512	512	3	1	yes	lrelu	no
conv	512	512	3	1	yes	lrelu	no

Additional results of synthesized images are shown in Fig. 20, Fig. 21, and Fig. 22.

Fig. 20. DeepVAE synthesis of zebras.

Fig. 21. DeepVAE synthesis of elephants.

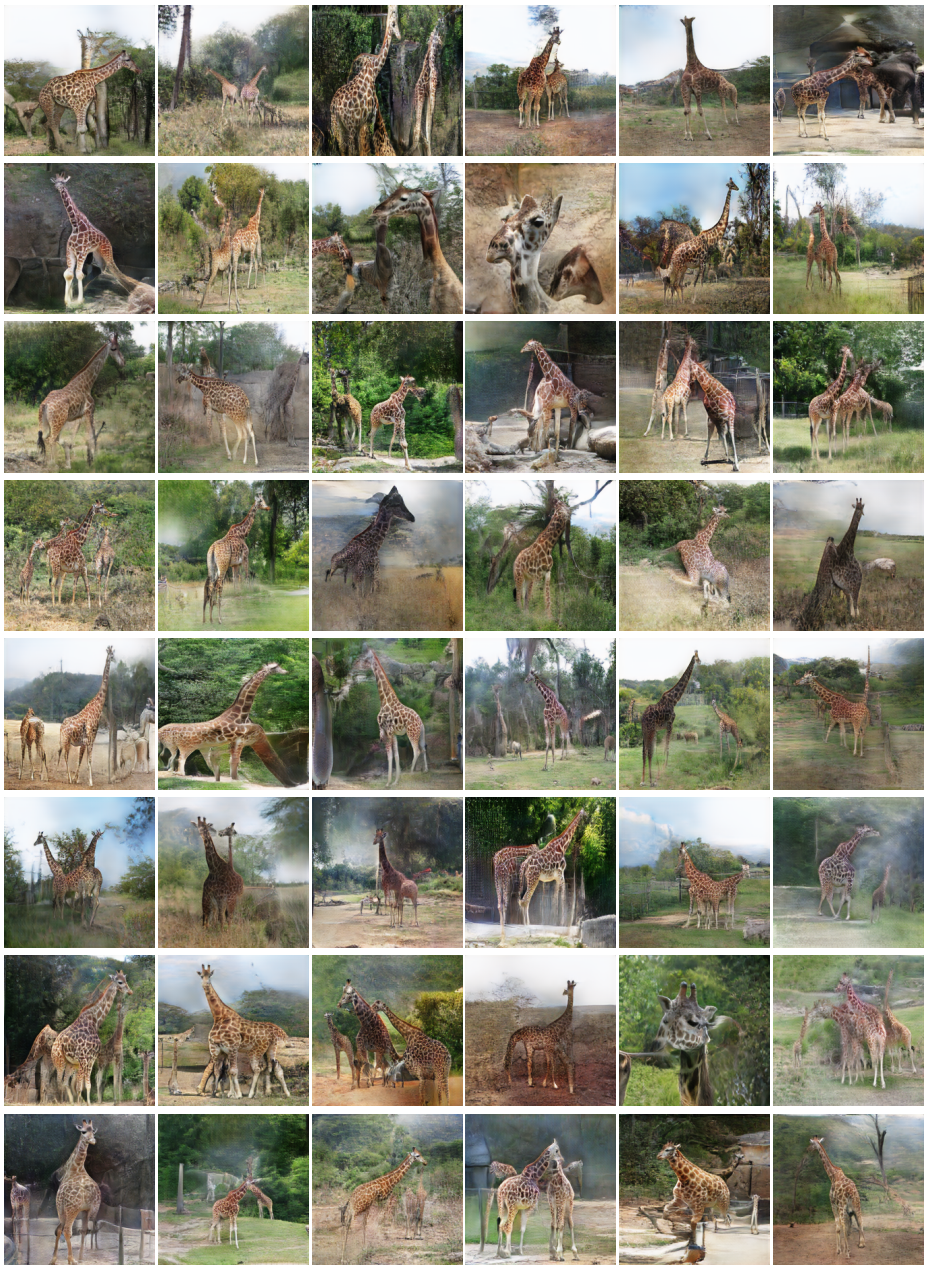

Fig. 22. DeepVAE synthesis of giraffes.