

# Park *et al.*: Contrastive learning for unpaired image-to-image translation

## A tutorial

Written by:  
Domonkos VARGA

# Table of contents

- What is contrastive learning?
  - Contrastive learning of visual representations
  - SimCLR
- What is image-to-image translation? What is unpaired image-to-image translation?
- How does Park et al.'s algorithm work?

# Part 1. What is contrastive learning?

# Contrastive learning of visual representations

- Motivation

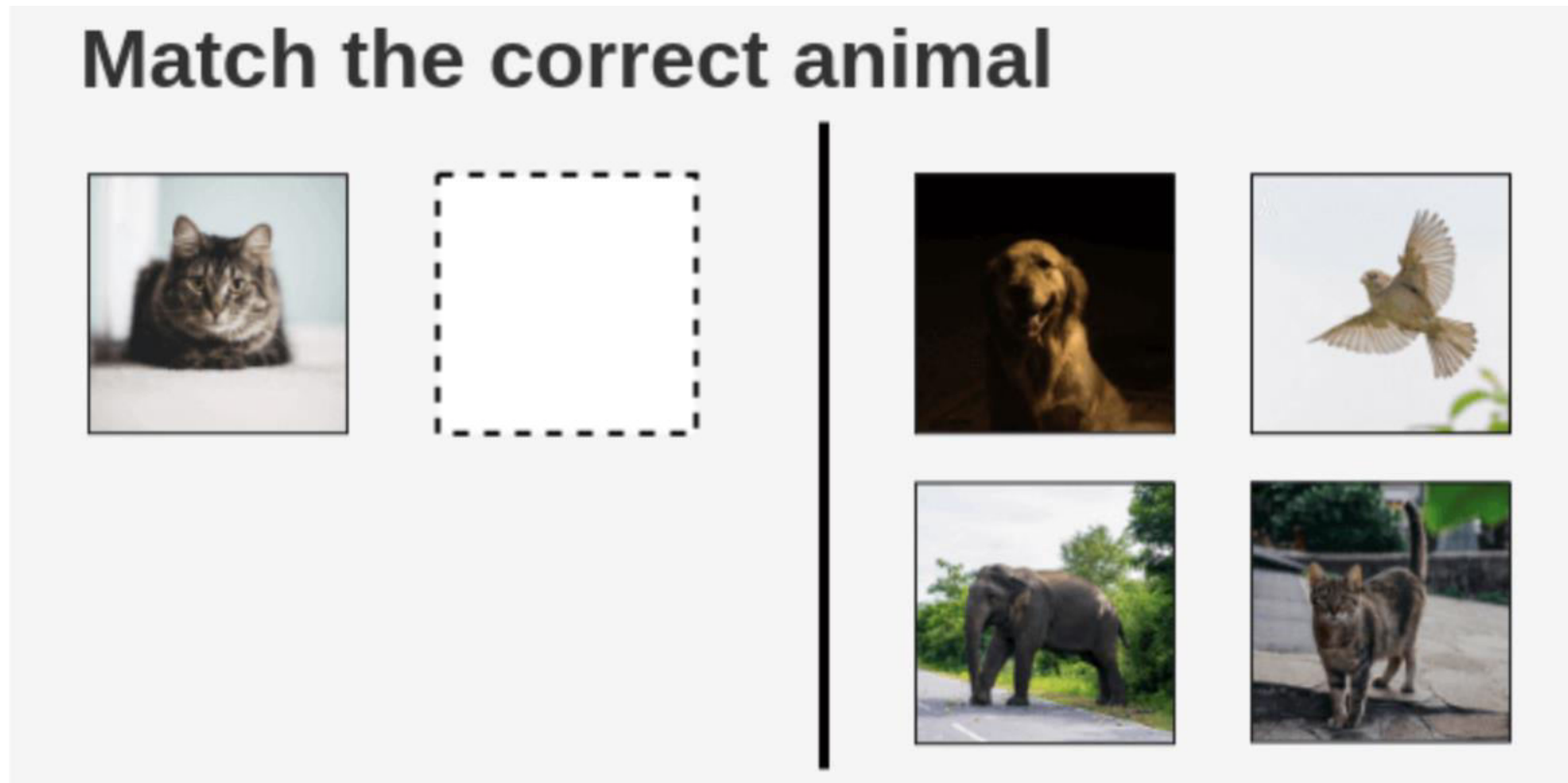
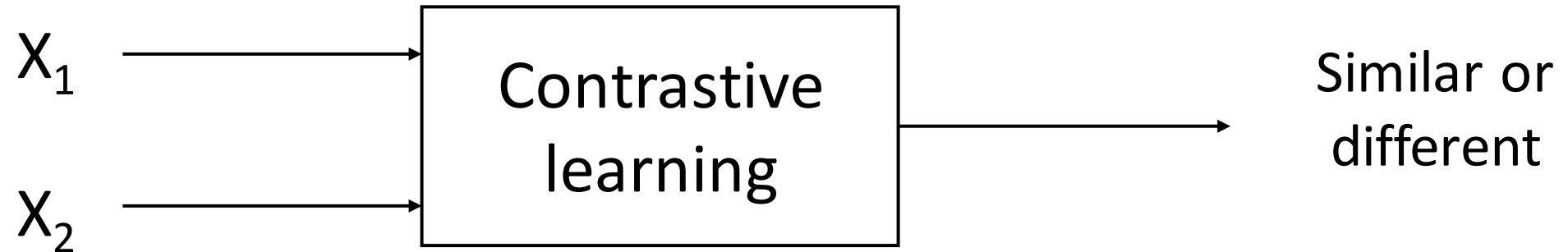


Image credit: Amit Chaudhary

# Contrastive learning of visual representations

- Contrastive learning



- Supervised learning



# Contrastive learning advantages and disadvantages

- Advantages:

1. Unsupervised, it does not need a labeled dataset.
2. Better results than supervised learning.
3. Far better results in comparison to other unsupervised techniques.
4. Promising results in other areas, such as audio and language.
5. Benefits stonger on data augmentation than supervised learning.

# Contrastive learning advantages and disadvantages

- Disadvantages:
  1. Larger batch size is required than in case of supervised methods.
  2. More training steps are required than in supervised methods.
  3. Deeper and wider networks are required.

# Contrastive learning of visual representations

Formulation:

- 1.) Examples of similar and dissimilar images.
- 2.) Image representation: a mechanism is required that is able to obtain an image representation which is suitable for machines (on ImageNet database pretrained CNNs, such as AlexNet, VGG16, ResNet50, *etc.*)
- 3.) Defining similarity.

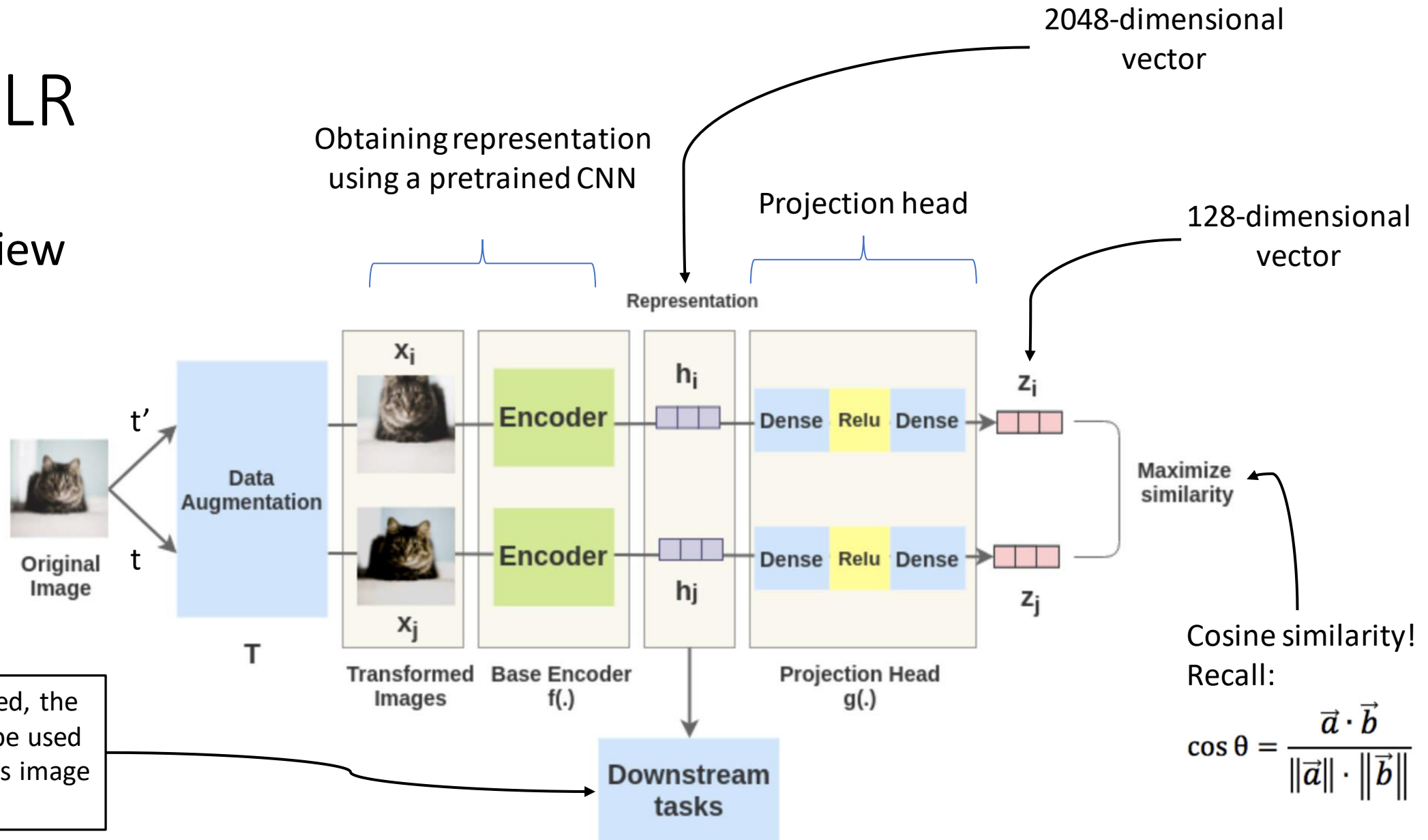


# Contrastive learning for visual representations

- Chen *et al.* in A simple framework for contrastive learning of visual representations proposed a framework – codenamed “SimCLR” – for tackling the problems enumerated in the previous slide.
- Park *et al.* in Contrastive learning for unpaired image-to-image translation heavily rely on the work of Chen *et al.* so we have to take a closer look at SimCLR.

# SimCLR

- Overview

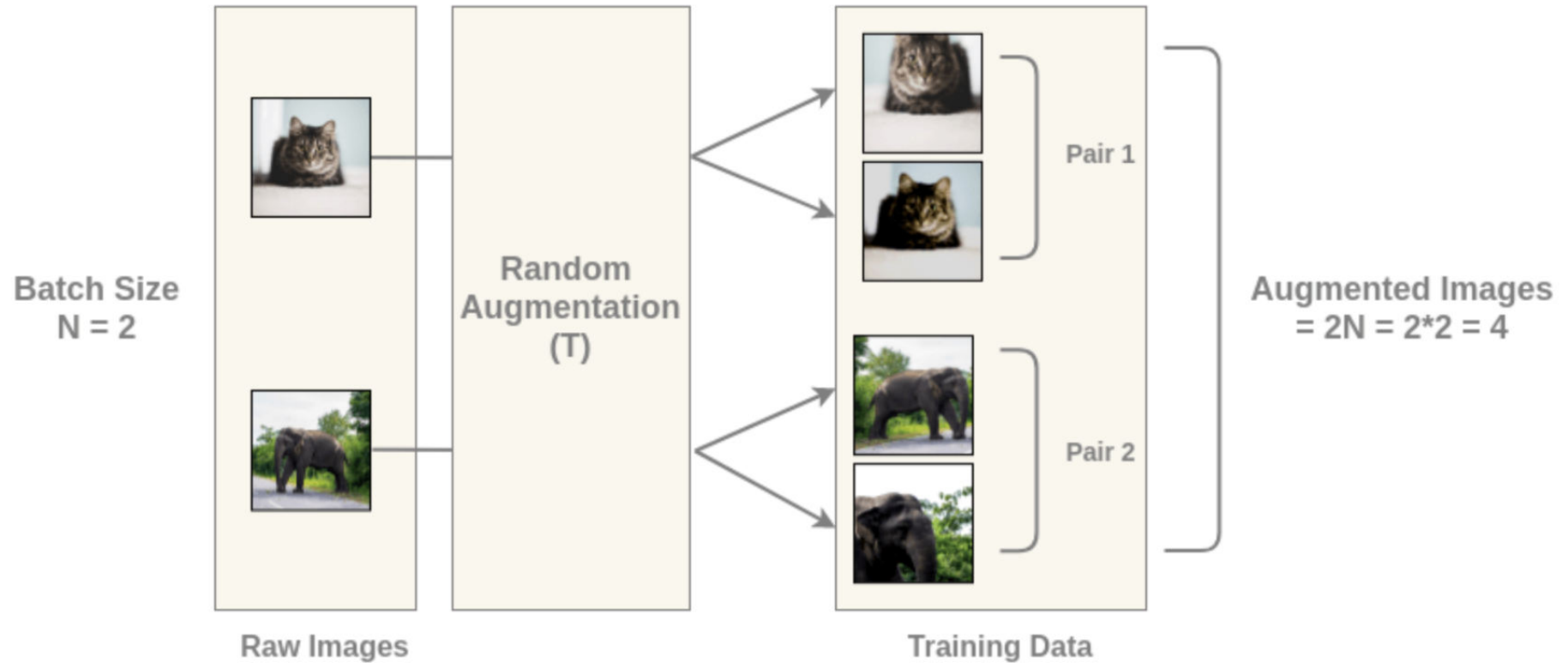


# SimCLR

- Step by Step: **Data augmentation**
- SimCLR selects transformation functions – denoted by  $T(.)$  – that applies a random combination of cropping, flipping, color jitter, and RGB to grayscale conversion.
- Parameter study in the paper justifies these operations.

# SimCLR

- Step by step: **Data augmentation**



# SimCLR

- Step by step: **Image representation**
- In the past decade, CNNs have gained a lot of attention. It have been shown that features extracted from on ImageNet database pretrained CNNs are able to provide powerful feature representations for a wide variety of tasks.
- In SimCLR, each augmented image is run through a ResNet-50 body which carries out all its defined operations. The image representation is given as 2048-dimensional vectors.

# SimCLR

- Step by step: **Projection head**
- The image representations are passed through three layers (Dense - ReLU – Dense) to project them into  $z_i$  and  $z_j$ . These three layers are called projection head and denoted by function  $g(.)$  in the original paper.

# SimCLR


- Step by step: **Similarity**
- The similarity of  $z_i$  and  $z_j$  are determined by *cosine similarity* which is derived from the formula of Euclidean dot product.

$$\text{Similarity} = \mathbf{z}_i^T \mathbf{z}_j / (\|\mathbf{z}_i\| \|\mathbf{z}_j\|)$$

- Cosine similarity is divided by a temperature hyperparameter. It is left out for simplicity reasons.

# SimCLR

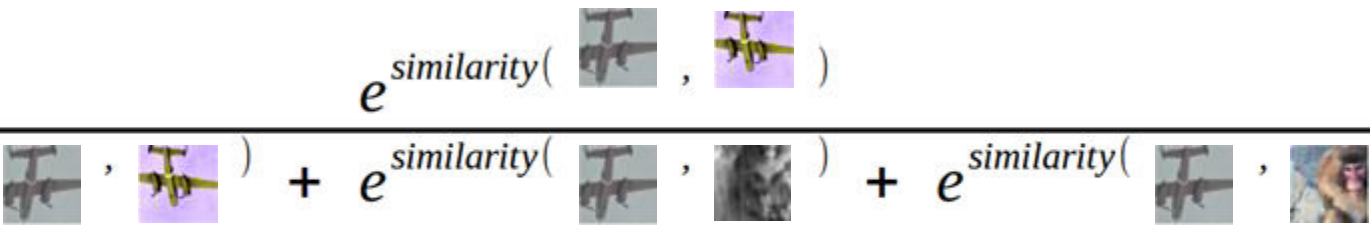
- Step by step: **loss function**
- **SimCLR** proposed the “NT-Xent loss” – Normalized Temperature-scaled Cross-entropy Loss
- **First**, the augmented pairs in the batch are taken one by one. Specifically, *softmax function* is applied to obtain the probability of two images being similar.

$$\text{Softmax} = \frac{e^{\text{similarity}(\text{img}_1, \text{img}_2)}}{e^{\text{similarity}(\text{img}_1, \text{img}_2)} + e^{\text{similarity}(\text{img}_1, \text{img}_3)} + e^{\text{similarity}(\text{img}_1, \text{img}_4)}}$$




# SimCLR

- Step by step: **loss function**

$$Softmax = \frac{e^{\text{similarity}(\text{img}_1, \text{img}_2)}}{e^{\text{similarity}(\text{img}_1, \text{img}_2)} + e^{\text{similarity}(\text{img}_1, \text{img}_3)} + e^{\text{similarity}(\text{img}_1, \text{img}_4)}}$$


- The loss for a pair is determined by taking the negative log of the above calculation.

$$l(i, j) = -\log \frac{\exp(s_{i,j})}{\sum_{k=1, k \neq i}^{2N} \exp(s_{i,k})}$$

# SimCLR

- Step by step: **loss function**
- Another loss for the same pair is also calculated.

$$l(i, j) = -\log \frac{\exp(s_{i,j})}{\sum_{k=1, k \neq i}^{2N} \exp(s_{i,k})}$$

$$l(j, i) = -\log \frac{\exp(s_{j,i})}{\sum_{k=1, k \neq j}^{2N} \exp(s_{j,k})}$$

Interchange images.



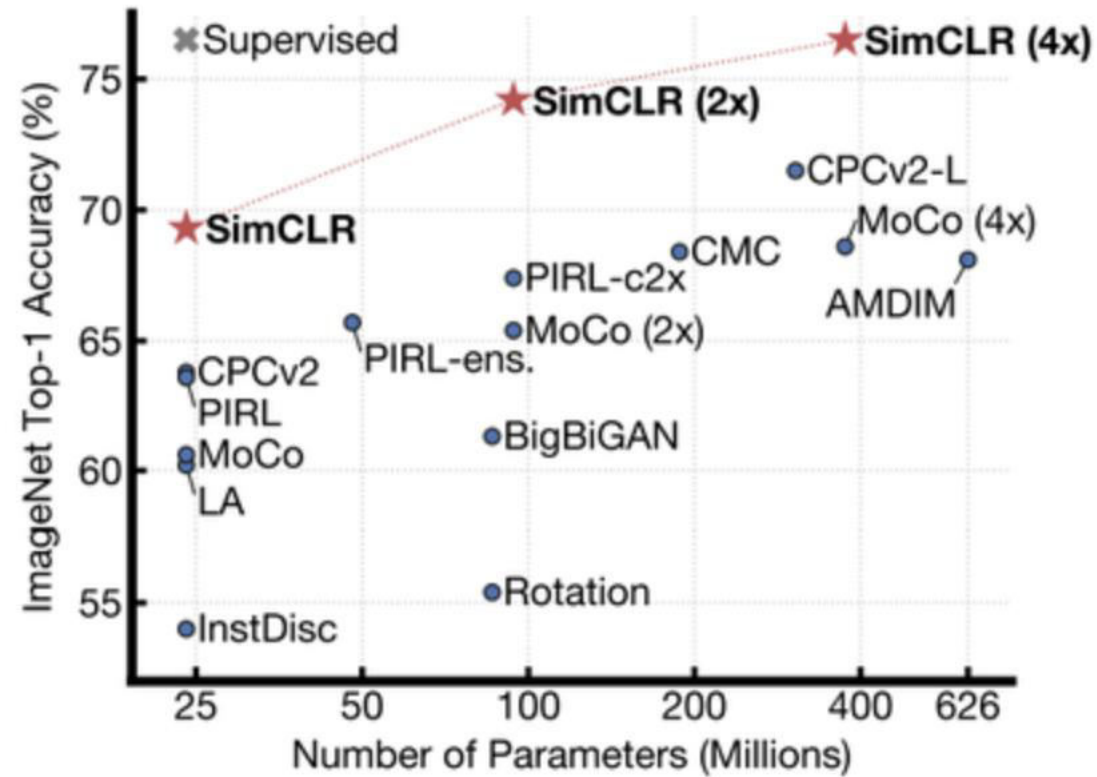
# SimCLR

- Step by step: **loss function**
- The overall loss is computed by determining the loss for all pairs in the batch and taking the average

$$L = \frac{1}{2N} \sum_{k=1}^N [l(2k - 1, 2k) + l(2k, 2k - 1)]$$

# SimCLR

- Results

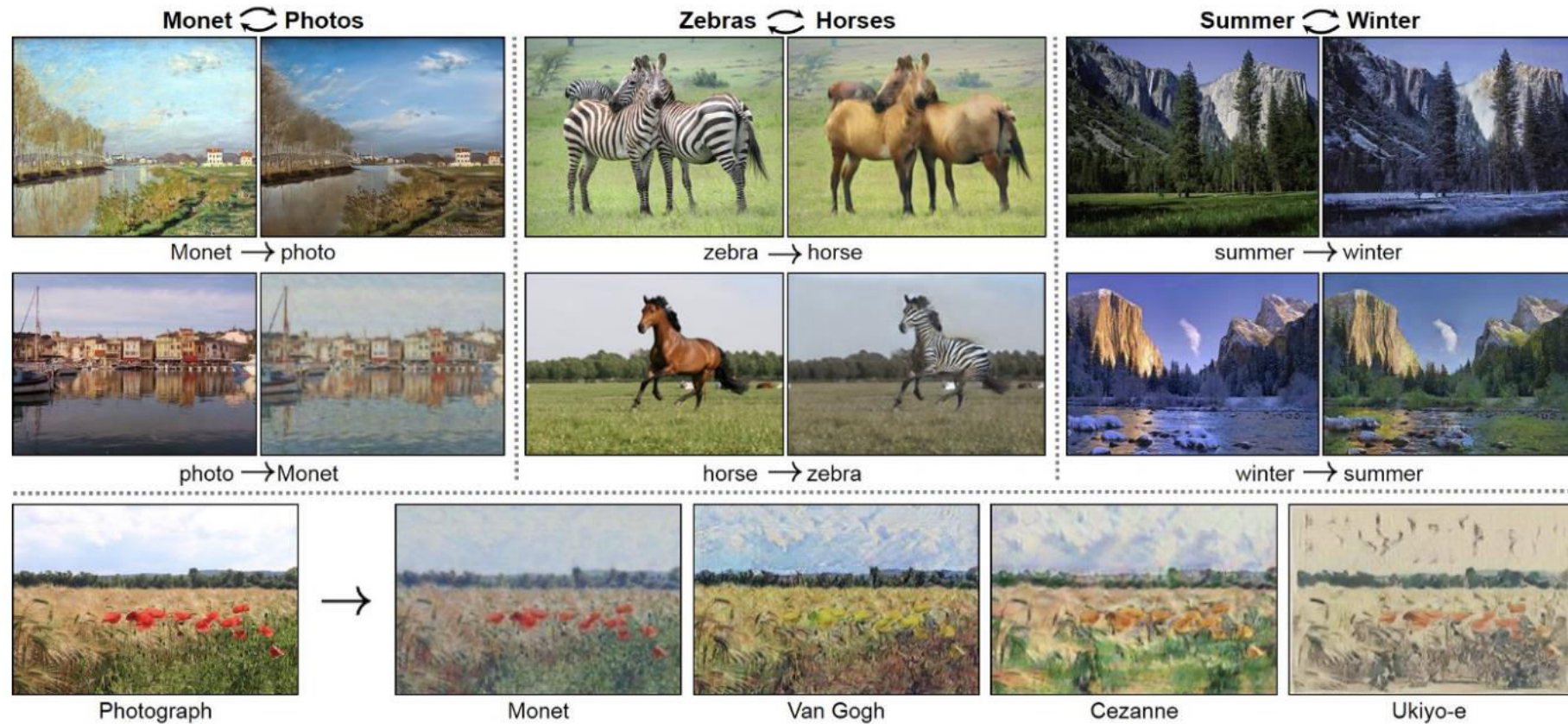


Part 2. What is (unpaired) image-to-image translation?

# Image-to-image translation

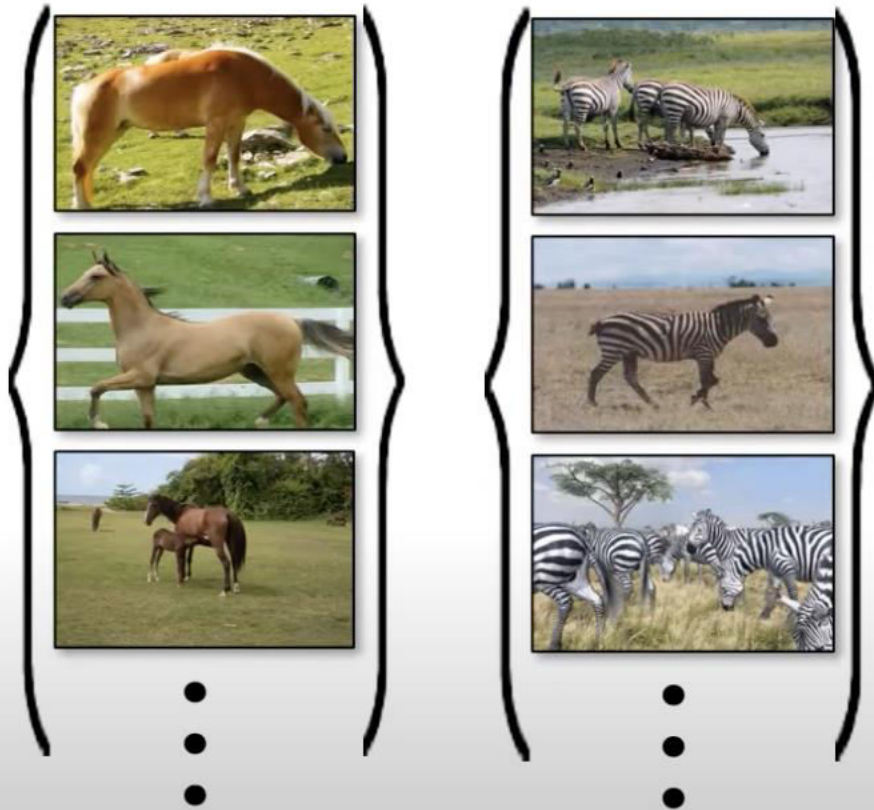
- **Image-to-image translation** is a computer vision and graphics problem where the goal is to transform an input image into another image given certain conditions.
- **Paired:** Full information is provided about the input and output image.
- **Unpaired:** Full information is provided only about the input image. High-level information is given about the desired output image.

# Unpaired image-to-image translation

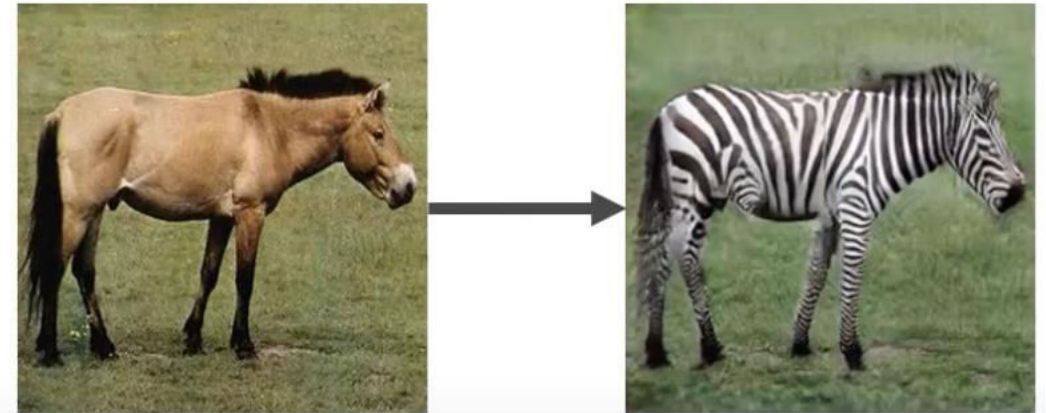


# Unpaired image-to-image translation

**Training database**



**Test-time behaviour**





Part 3. How does Park et al.'s algorithm work?

# Patchwise contrastive learning

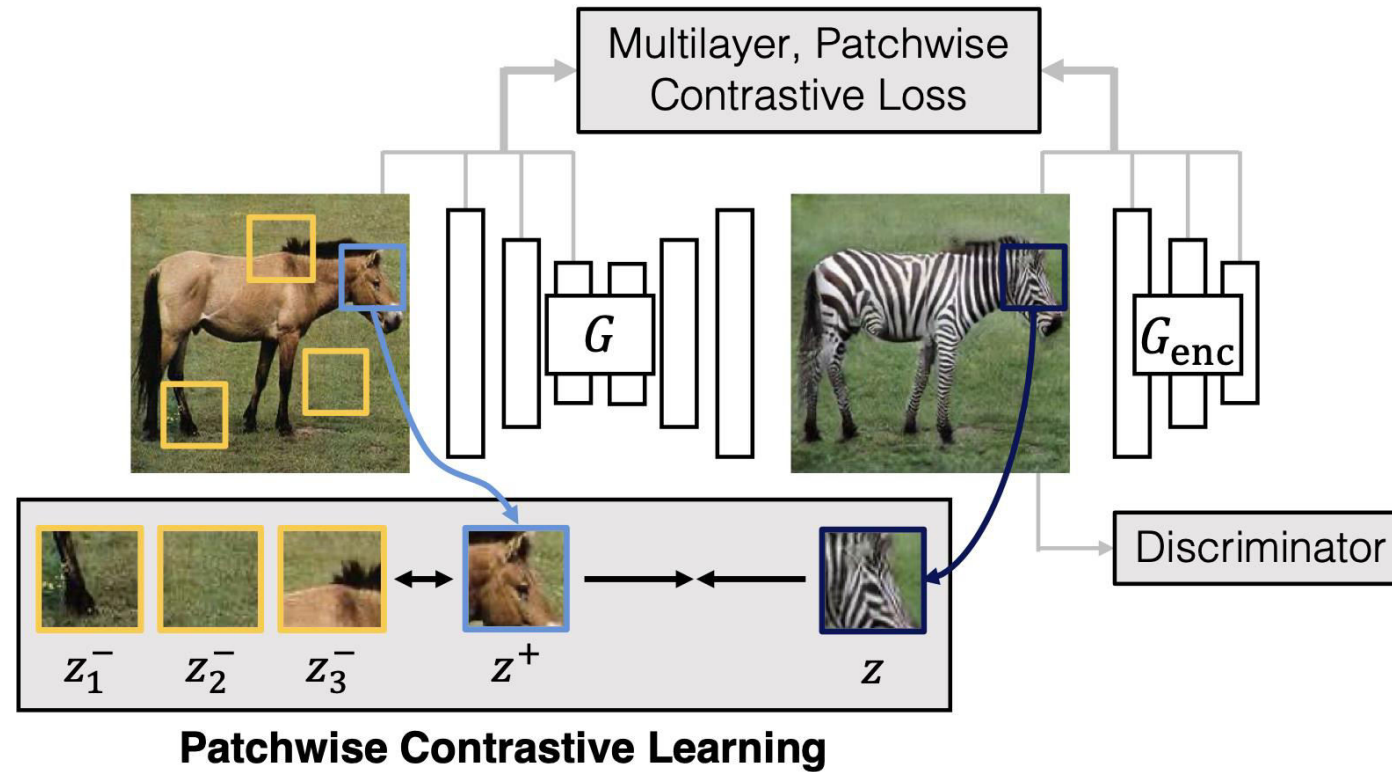
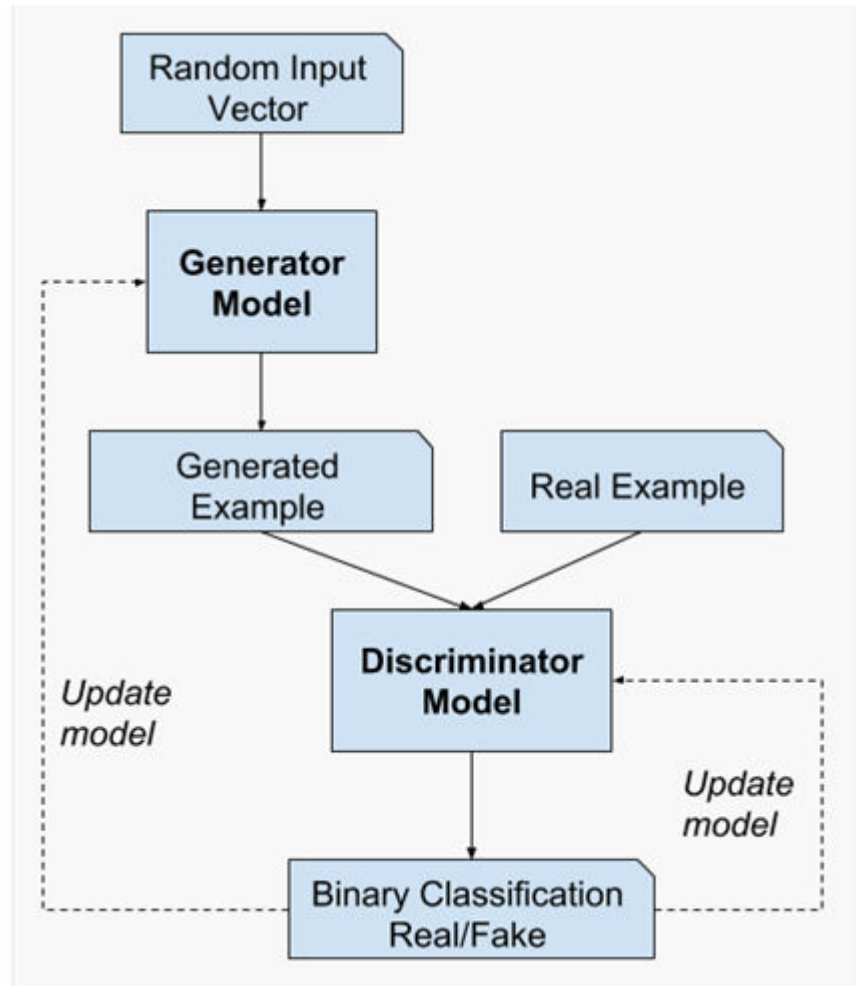


Fig. 1: **Patchwise Contrastive Learning for one-sided translation.** A generated **output patch** should appear closer to its **corresponding input patch**, in comparison to other **random patches**. We use a multilayer, patchwise contrastive loss, which maximizes *mutual information* between corresponding input and output patches. This enables one-sided translation in the unpaired setting.

# Generative adversarial network

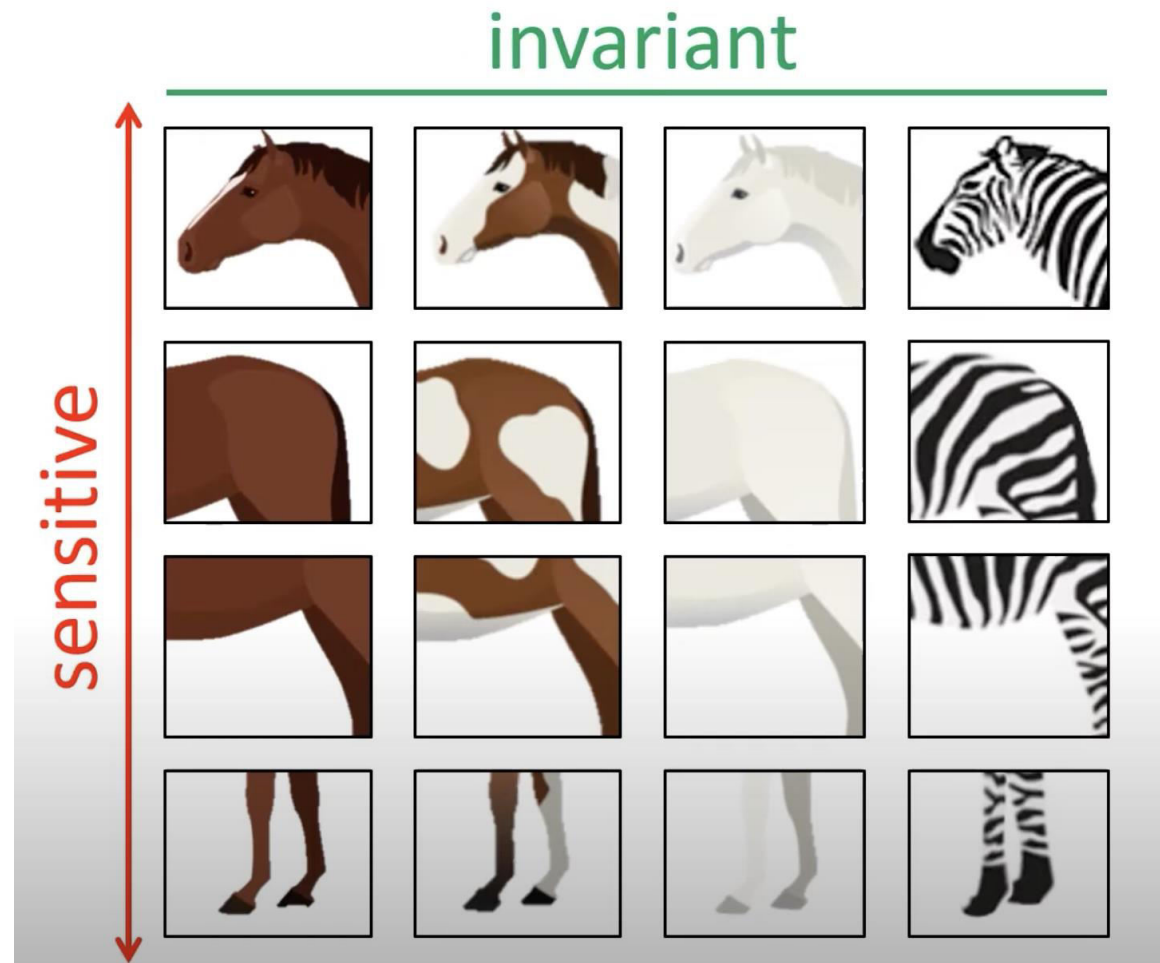


the problem is treated as supervised learning with two sub-models:

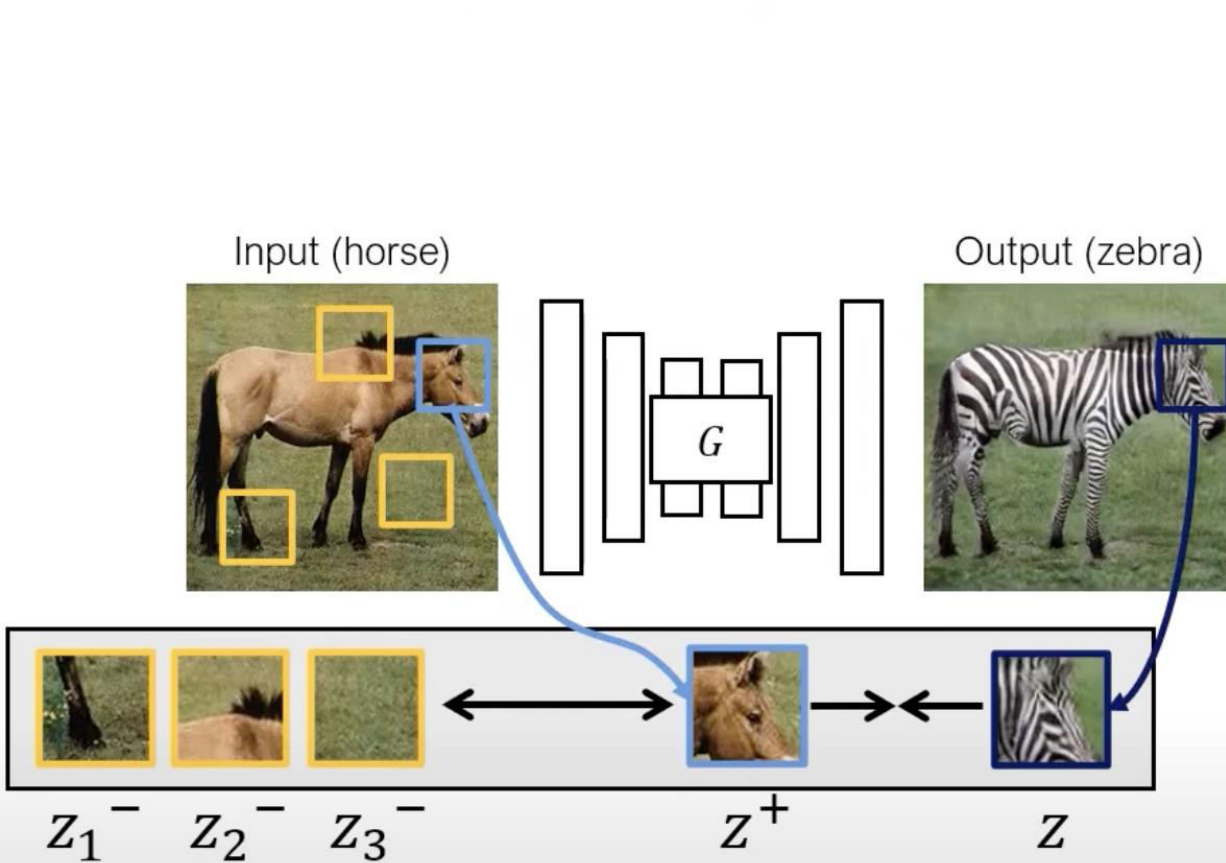
the **generator** model that we train to generate new examples,  
and  
the **discriminator** model that tries to classify examples  
as either real (from your dataset) or fake (generated).

the generator and discriminator are trained interconnected using  
gradient descent techniques

# Patch-based contrastive loss



# Patch-based contrastive loss



Corresponding patches should have high similarity!

softmax

$$\begin{pmatrix} \uparrow z \cdot z^+ / \tau \\ \downarrow z \cdot z_1^- / \tau \\ \downarrow z \cdot z_2^- / \tau \\ \vdots \\ \downarrow z \cdot z_N^- / \tau \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

softmax (cosine similarities  $/\tau$ )  
 $\tau=0.07$

# Sample images of own runnings


The screenshot displays a Jupyter Notebook environment with the following code and outputs:

```
File Edit View Insert Cell Kernel Widgets Help
(26): Conv2d(64, 3, kernel_size=(7, 7), stride=(1, 1))
(27): Tanh()

In [5]: 1 from PIL import Image
        2 from torchvision import transforms

In [6]: 1 preprocess = transforms.Compose([transforms.Resize(256),
        2 transforms.ToTensor()])

In [77]: 1 img = Image.open("../data/plch2/buffalo2.jpeg")
        2 img


Out[77]: 
```

The output of In [77] is a small image of a brown bison in a grassy field. An arrow points from this output to a larger version of the same image on the right side of the screen.

```
In [78]: 1 img_t = preprocess(img)
        2 batch_t = torch.unsqueeze(img_t, 0)

In [79]: 1 batch_out = netG(batch_t)

In [80]: 1 out_t = (batch_out.data.squeeze() + 1.0) / 2.0
        2 out_img = transforms.ToPILImage()(out_t)
        3 # out_img.save("../data/plch2/zebra.jpg")
        4 out_img

Out[80]: 
```

The output of In [80] is a small image of a zebra-striped bison in a grassy field. An arrow points from this output to a larger version of the same image on the right side of the screen.



# Sample images of own runnings

A screenshot of a Jupyter Notebook interface showing a series of code cells and their outputs. The notebook is titled "3\_cyclegan" and shows the process of applying a GAN model to generate zebra-like patterns on a dog image.


The code cells include:

```
(23): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
(24): ReLU(inplace=True)
(25): ReflectionPad2d((3, 3, 3, 3))
(26): Conv2d(64, 3, kernel_size=(7, 7), stride=(1, 1))
(27): Tanh()
```

```
In [5]: 1 from PIL import Image
        2 from torchvision import transforms

In [6]: 1 preprocess = transforms.Compose([transforms.Resize(256),
        2 transforms.ToTensor()])


In [81]: 1 img = Image.open("../data/p1ch2/dog2.jpeg")
         2 img

Out[81]: 
```

```
In [82]: 1 img_t = preprocess(img)
         2 batch_t = torch.unsqueeze(img_t, 0)

In [83]: 1 batch_out = netG(batch_t)

In [84]: 1 out_t = (batch_out.data.squeeze() + 1.0) / 2.0
         2 out_img = transforms.ToPILImage()(out_t)
         3 # out_img.save("../data/p1ch2/zebra.jpg")
         4 out_img

Out[84]: 
```

Two sample images of the generated results are shown on the right side of the notebook:

- A brown dog standing in a grassy field.
- A brown dog with zebra stripes standing in a grassy field.

Arrows point from the generated images in the notebook output to the sample images on the right.

# Sample images of own runnings

A screenshot of a Jupyter Notebook interface showing a series of code cells and their outputs. The notebook is titled "3\_cyclegan" and shows the process of applying a GAN model to generate zebra-like patterns on donkey images.


The code cells and their outputs are as follows:

- In [5]:** Imports PIL and torchvision.

```
1 from PIL import Image
2 from torchvision import transforms
```
- In [6]:** Defines a preprocessing transform.

```
1 preprocess = transforms.Compose([transforms.Resize(256),
2                                 transforms.ToTensor()])
```
- In [86]:** Loads an image from the file system.


```
1 img = Image.open("../data/p1ch2/donkey2.jpeg")
2 img
```

**Out[86]:** 
- In [87]:** Applies the preprocessing transform to the image.

```
1 img_t = preprocess(img)
2 batch_t = torch.unsqueeze(img_t, 0)
```
- In [88]:** Passes the batched image to a network.

```
1 batch_out = netG(batch_t)
```
- In [89]:** Converts the output back to a PIL image and saves it.

```
1 out_t = (batch_out.data.squeeze() + 1.0) / 2.0
2 out_img = transforms.ToPILImage()(out_t)
3 # out_img.save("../data/p1ch2/zebra.jpg")
4 out_img
```

**Out[89]:** 

Two large images on the right side of the notebook illustrate the input and output of the GAN model:

- The top image shows two brown donkeys standing on a gravel path, representing the input image.
- The bottom image shows two donkeys with zebra-like black and white stripes, representing the output image generated by the GAN model.

Arrows point from the generated images in the notebook output to the larger images on the right.