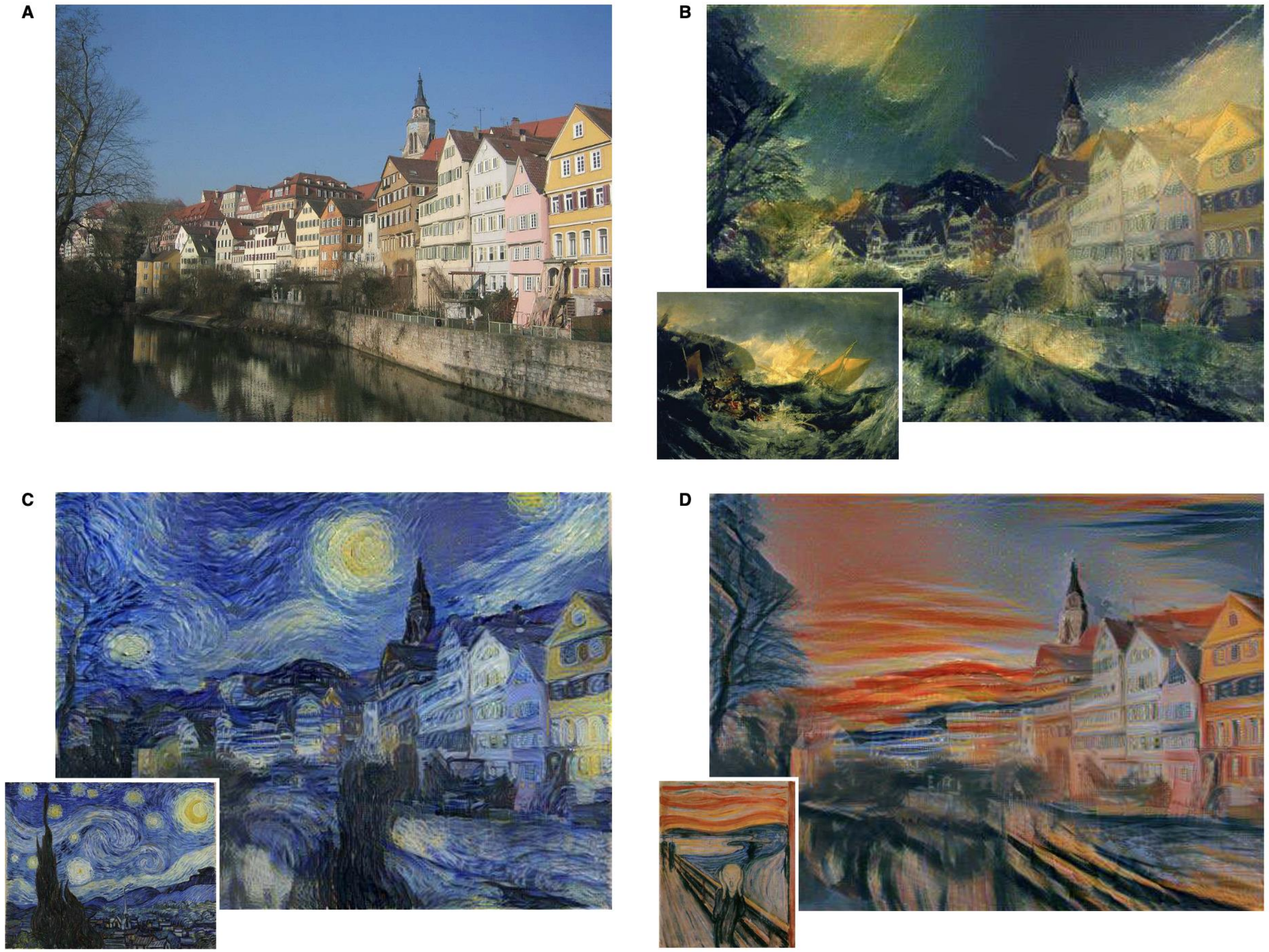
# Assignment:
# Neural Image Editing

CS4365 Applied Image Processing

2023/2024

Michael Weinmann

# Exercise 1: Style Transfer



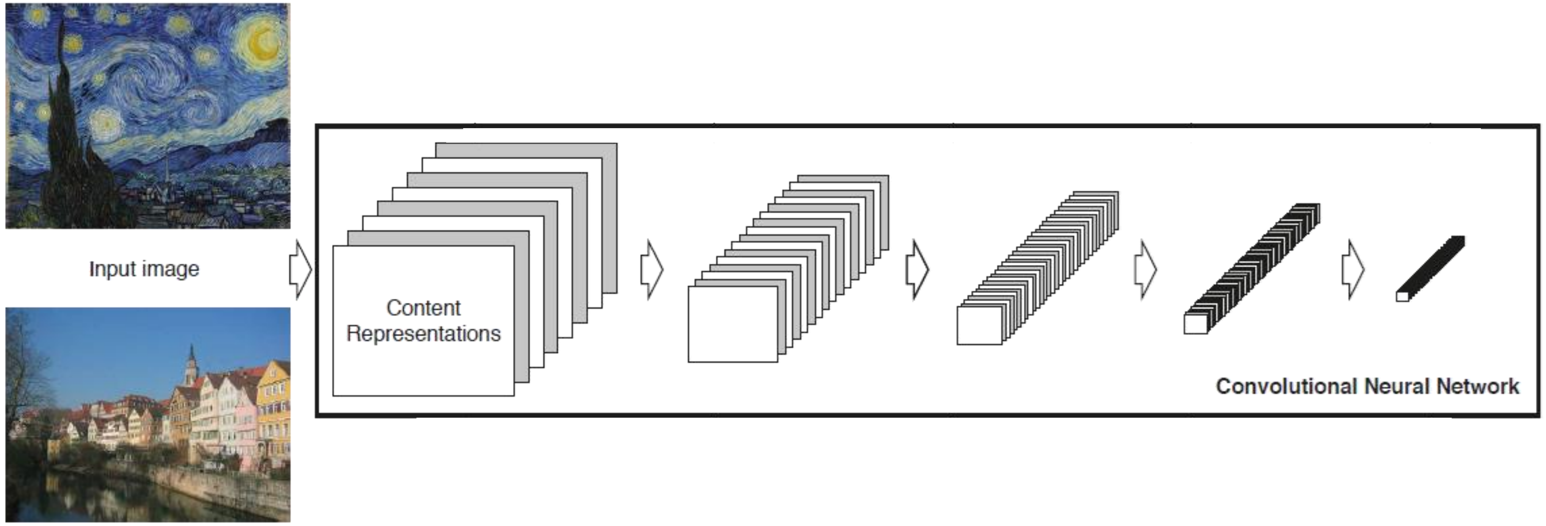

Gatys et al. (2016)

# Exercise 1: Style Transfer

- In part 1 of this assignment, we will focus on the style transfer approach by Gatys et al. (2016), „Image Style Transfer Using Convolutional Neural Networks”.
- The major subtasks will be …
  - … the computation of the content loss
  - … the computation of the style loss
  - … the computation of the total variation loss
  - … respective experiments

# Image Representation in CNN

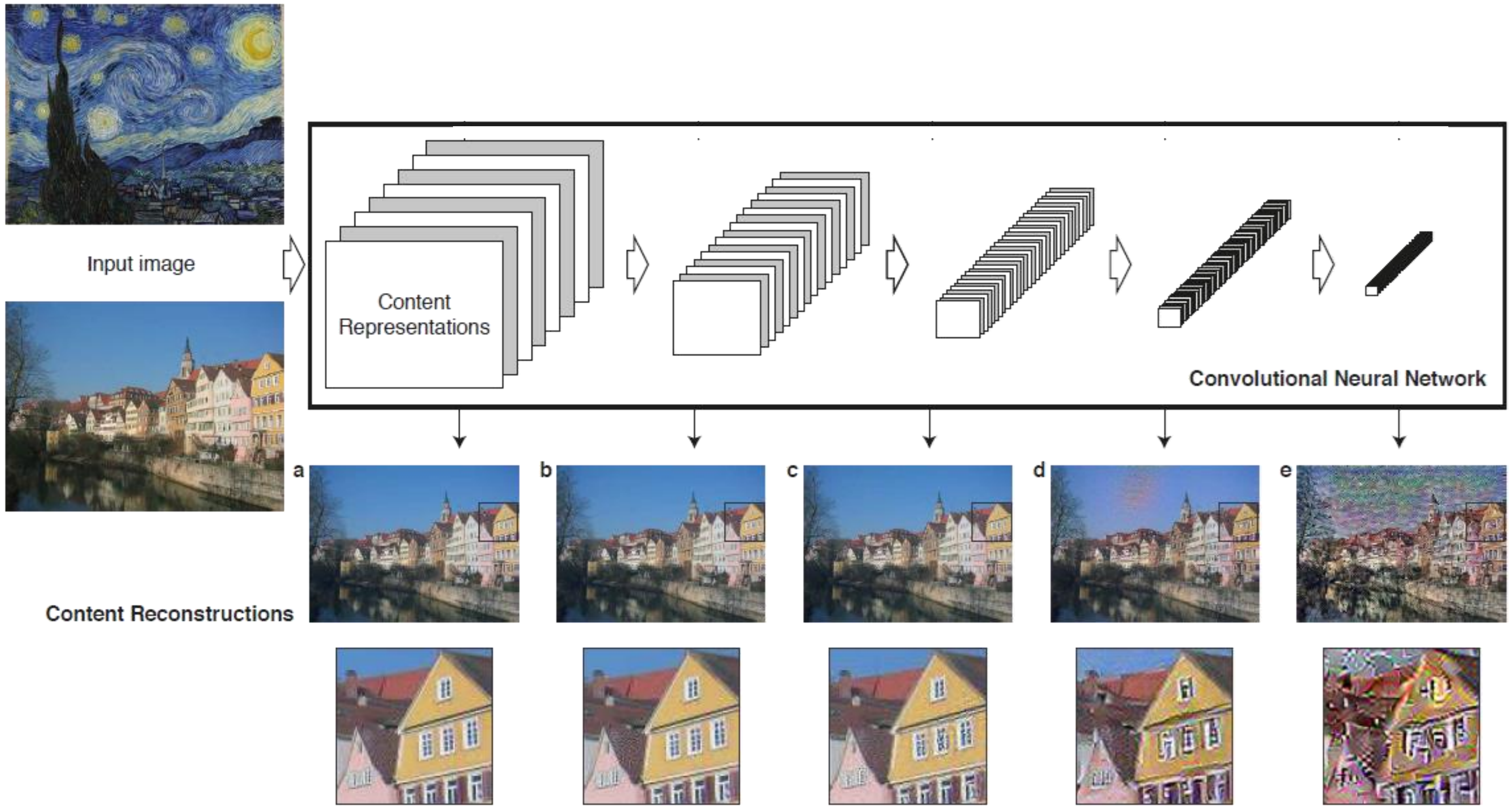
- How to represent content and style?




Gatys et al. „*Image Style Transfer Using Convolutional Neural Networks*”, 2016

# Image Representation in CNN

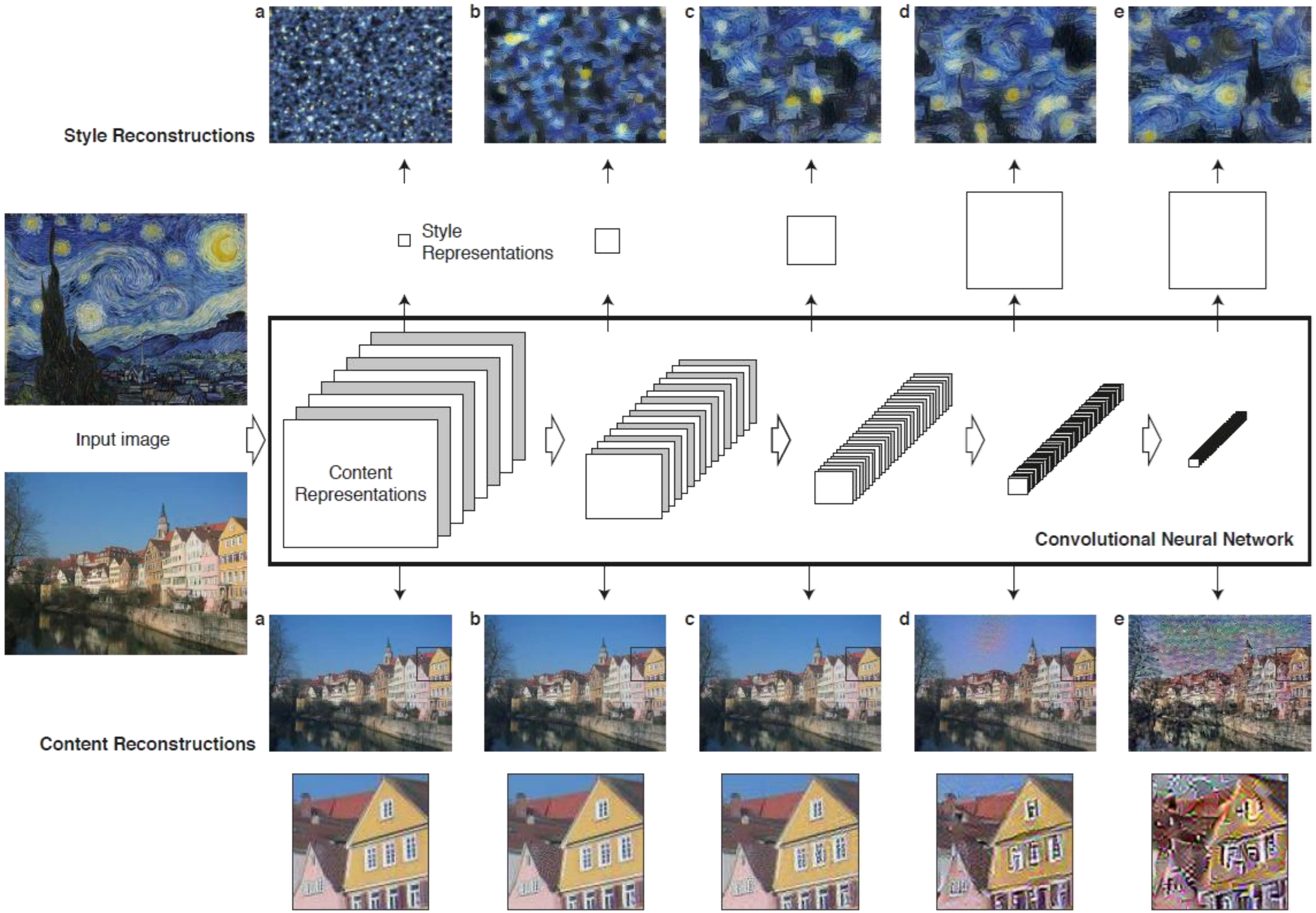
- **Content** representation:
  - Feature responses in **higher layers**
    - Capture the **high-level content** in terms of objects
      and their arrangement in the image
    - Do not constrain the exact **pixel values** of the reconstruction




Gatys et al. „*Image Style Transfer Using Convolutional Neural Networks*", 2016

# Image Representation in CNN

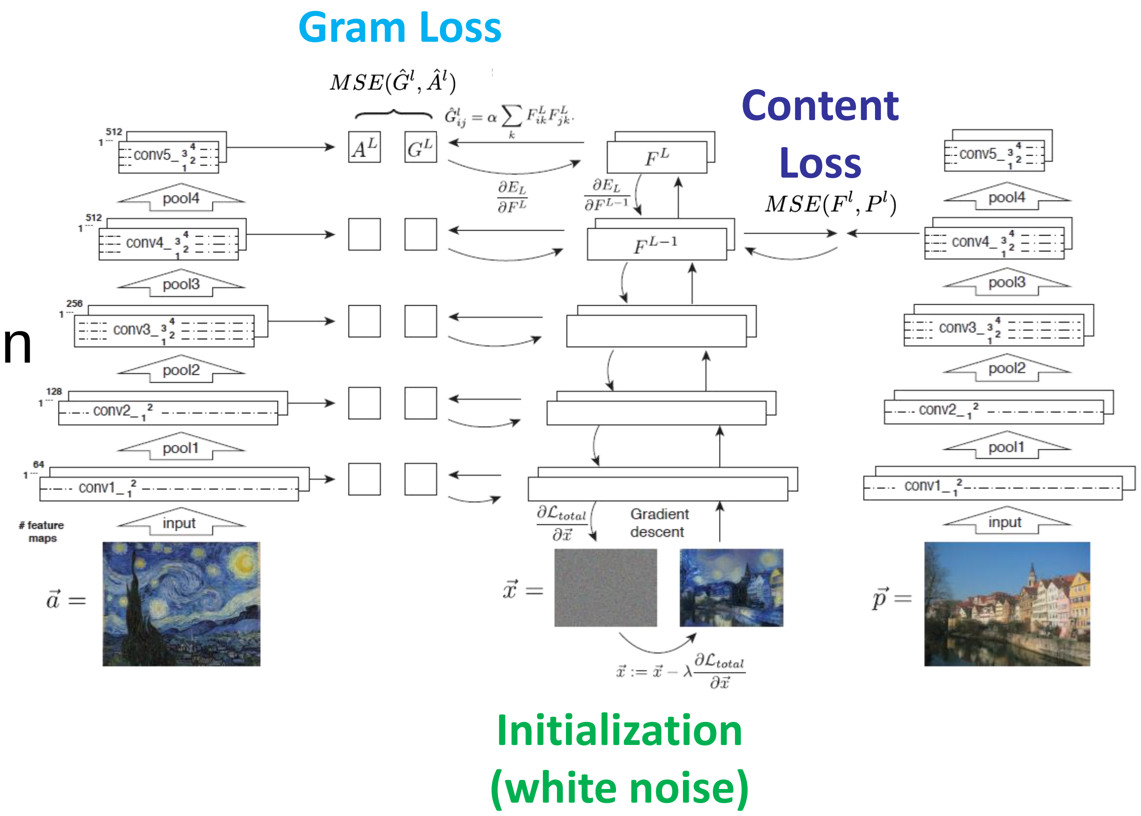
- **Style** representation:
    - **Feature correlations** (**Gram matrices**) on multiple layers (similar to texture synthesis)
        - Multi-scale style representation
        - Captures **the texture information**, but **not the global arrangement**
            - leads to a smoother and more continuous stylization




Gatys et al. „*Image Style Transfer Using Convolutional Neural Networks*", 2016

# IO-based Style Transfer based on Summary Statistics [Gatys et al. 2016]

- Algorithm:
  1. Pass content image $\vec{p}$ through the CNN → Compute and store content representation $P^l$ for one layer $l$ in the network
  2. Pass style image $\vec{a}$ through the CNN → Compute and store style representation $A^l$ for each layer $l$ in the network
  3. Initialise the output image $\vec{x}$ with white noise
  4. Pass $\vec{x}$ through CNN; compute style features $G^l$, content features $A^l$ and the total loss $\mathcal{L}_{total}$
  5. Perform backpropagation to image
  6. Take a gradient descent step
  7. Goto 4



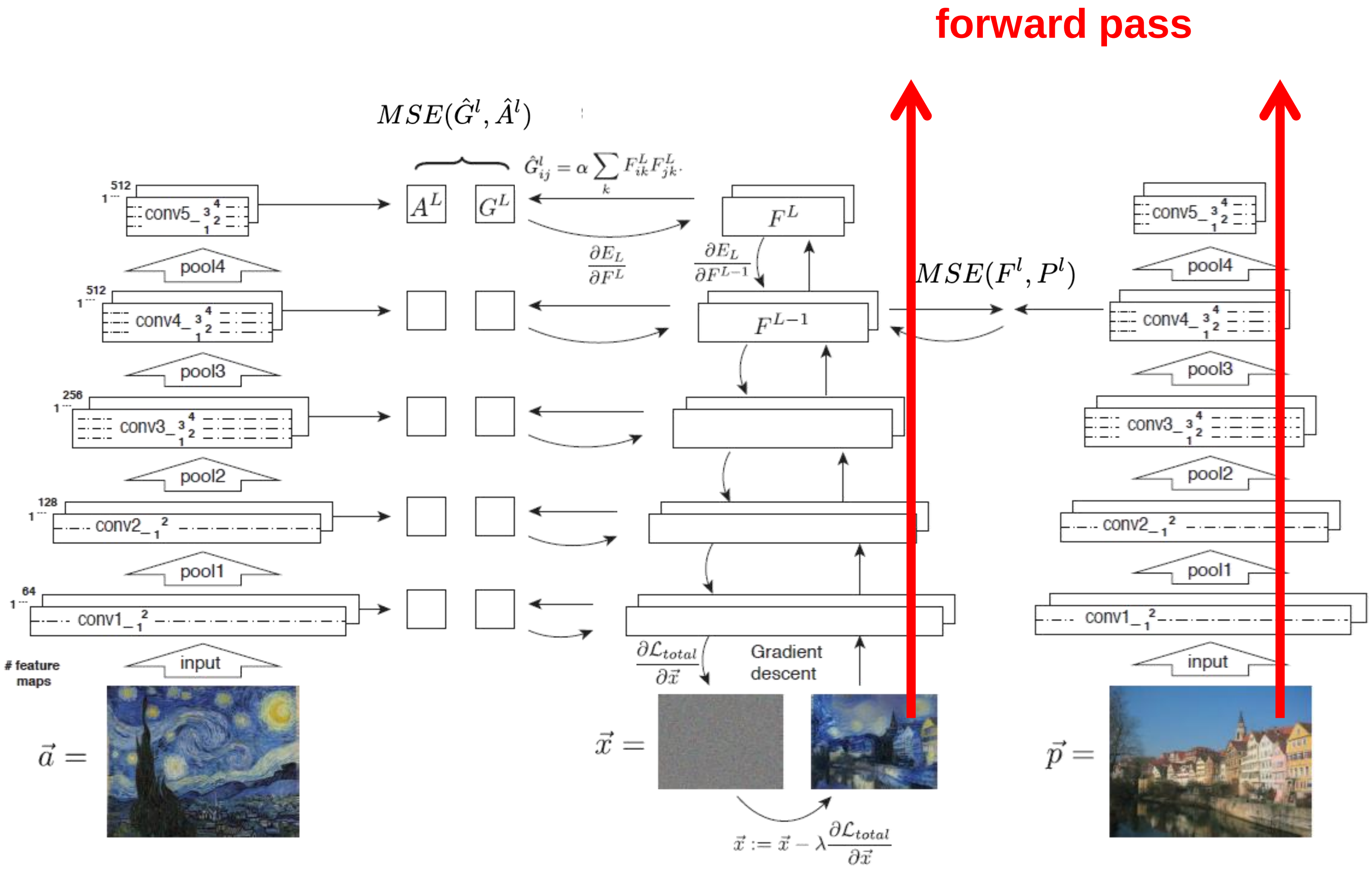
# Exercise 1: Style Transfer

- Task 1:
  - Compute a function to normalize tensors:
    - Do channel-wise z-score normalization

```python
def normalize(img, mean, std):
    """ Normalizes an image tensor.

    # Parameters:
        @img, torch.tensor of size (b, c, h, w)
        @mean, torch.tensor of size (c)
        @std, torch.tensor of size (c)

    # Returns the normalized image
    """
    # TODO: 1. Implement normalization doing channel-wise z-score normalization.

    return img
```

# Exercise 1: Style Transfer

- Task 2:
  - Compute content loss:
    - Forward passes:
      - Run forward propagation for content image $\vec{p}$
      - Run forward propagation for image $\vec{x}$

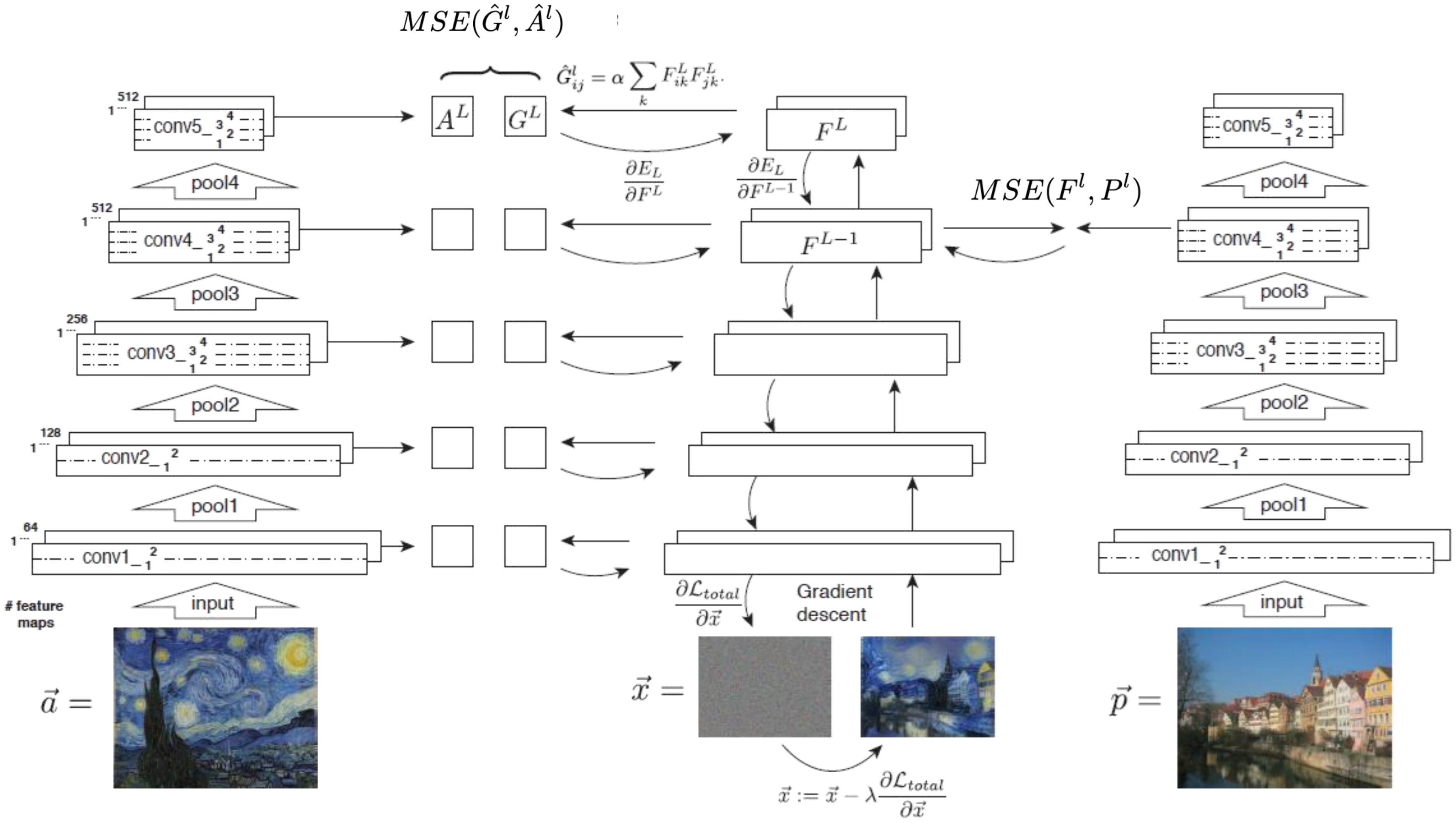



$$\ell_{total} = w_1 \ell_{content} + w_2 \ell_{style} + w_3 \ell_{tv}$$

# Exercise 1: Style Transfer

- Task 2:
  - Compute content loss:
    - Compute **MSE** based on activations $F^l$ , $P^l$ for the desired layers

$$\ell_{content} = \frac{1}{|L|}\sum_{l}^{|L|} MSE(F^l, P^l)$$




$$\ell_{total} = w_1 \ell_{content} + w_2 \ell_{style} + w_3 \ell_{tv}$$

TU Delft

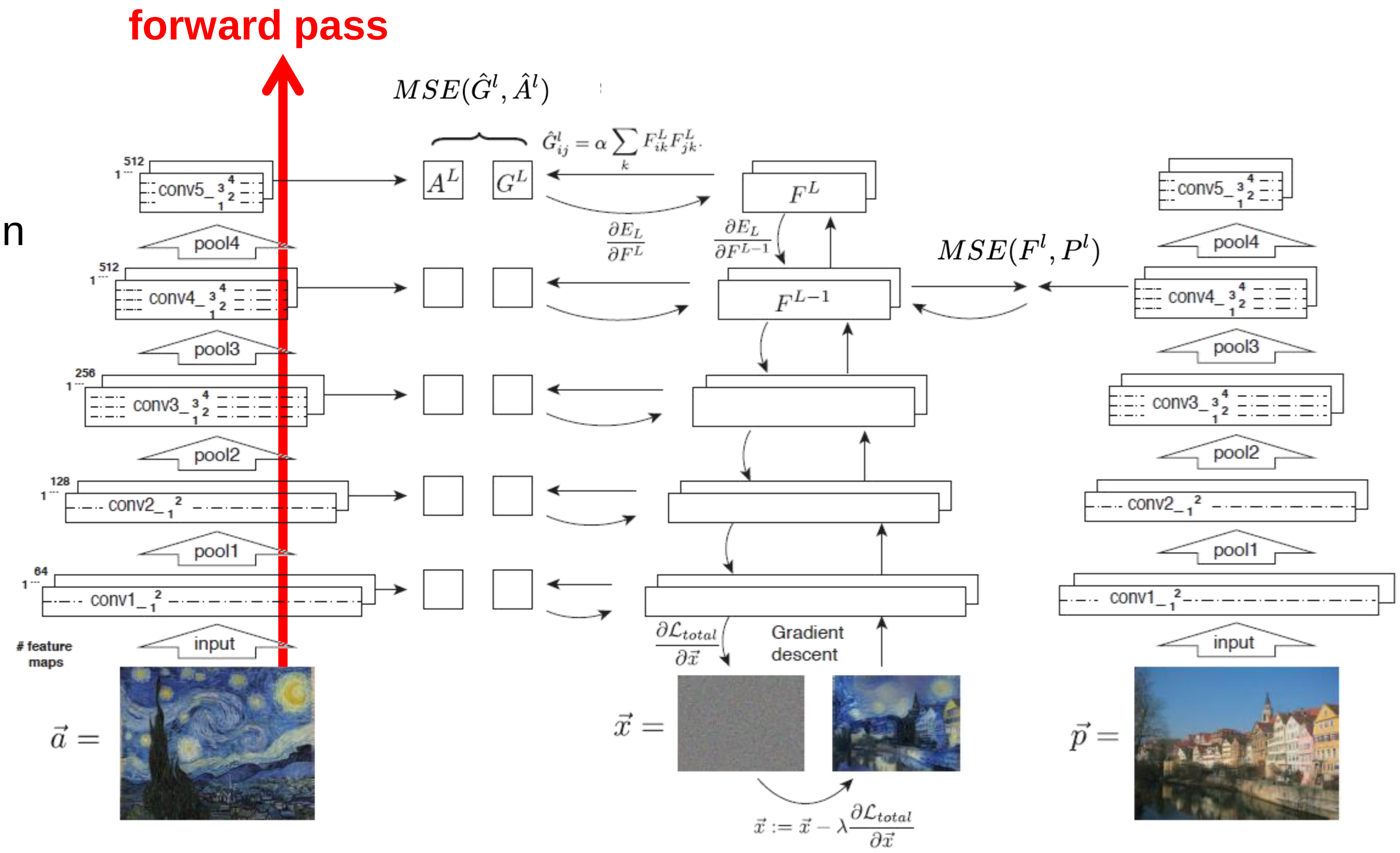
# Exercise 1: Style Transfer

- Task 2:
  - Compute content loss:

```
def content_loss(input_features, content_features, content_layers):
""" Calculates the content loss as in Gatys et al. 2016.

# Parameters:
        @input_features, VGG features of the image to be optimized. It is a
                dictionary containing the layer names as keys and the corresponding
                features volumes as values.
        @content_features, VGG features of the content image. It is a dictionary
                containing the layer names as keys and the corresponding features
                volumes as values.
        @content_layers, a list containing which layers to consider for calculating
                the content loss.
# Returns the content loss, a torch.tensor of size (1)
"""
# TODO: 2. Implement the content loss given the input feature volume and the
# content feature volume. Note that:
# - Only the layers given in content_layers should be used for calculating this loss.
# - Normalize the loss by the number of layers.


return torch.rand((1), requires_grad=True) # Initialize placeholder such that the code runs
```

# Exercise 1: Style Transfer

- Task 3:
  - Compute style loss:
    - Forward pass:
      - Run forward propagation for style image $\vec{a}$

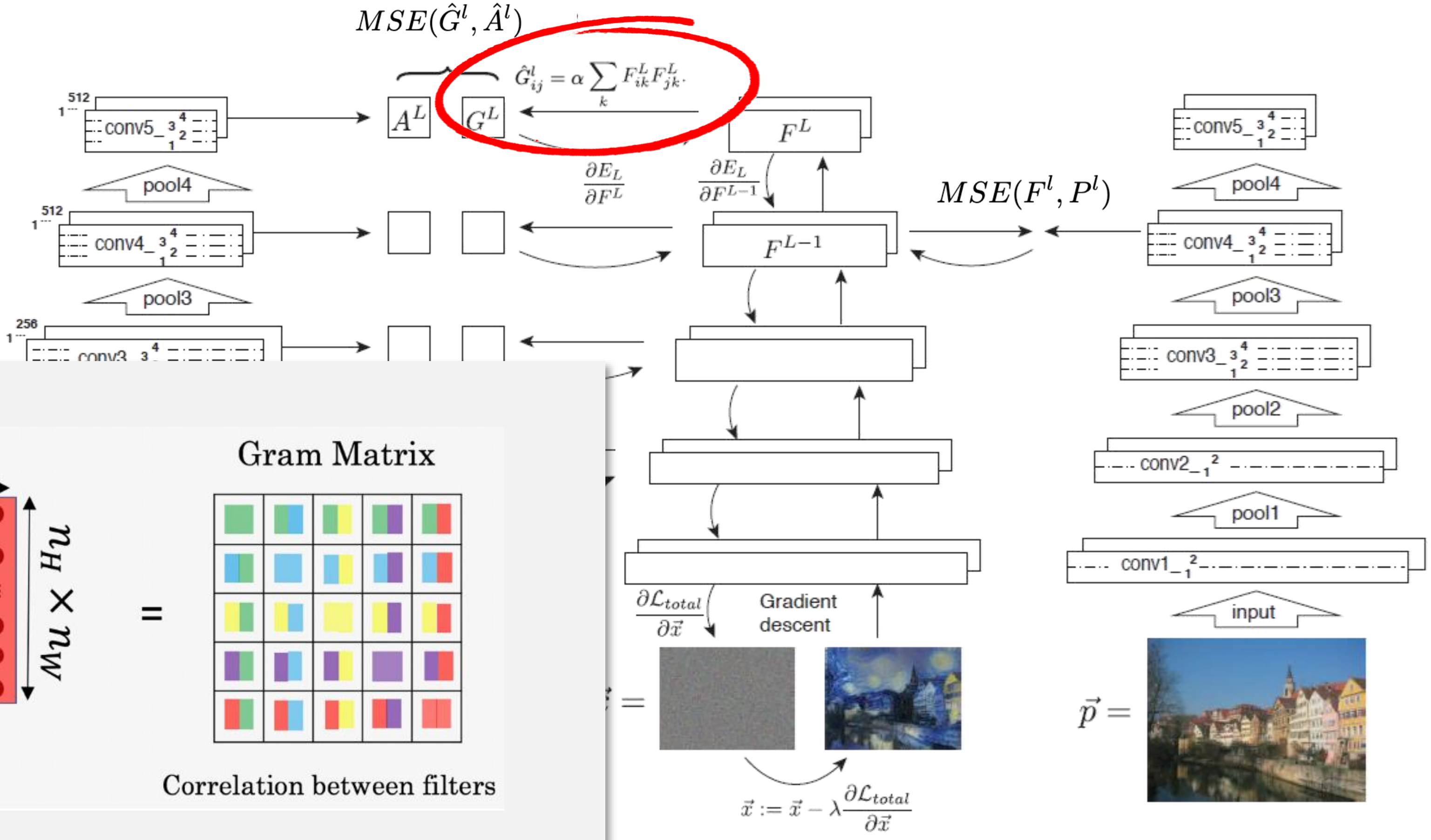


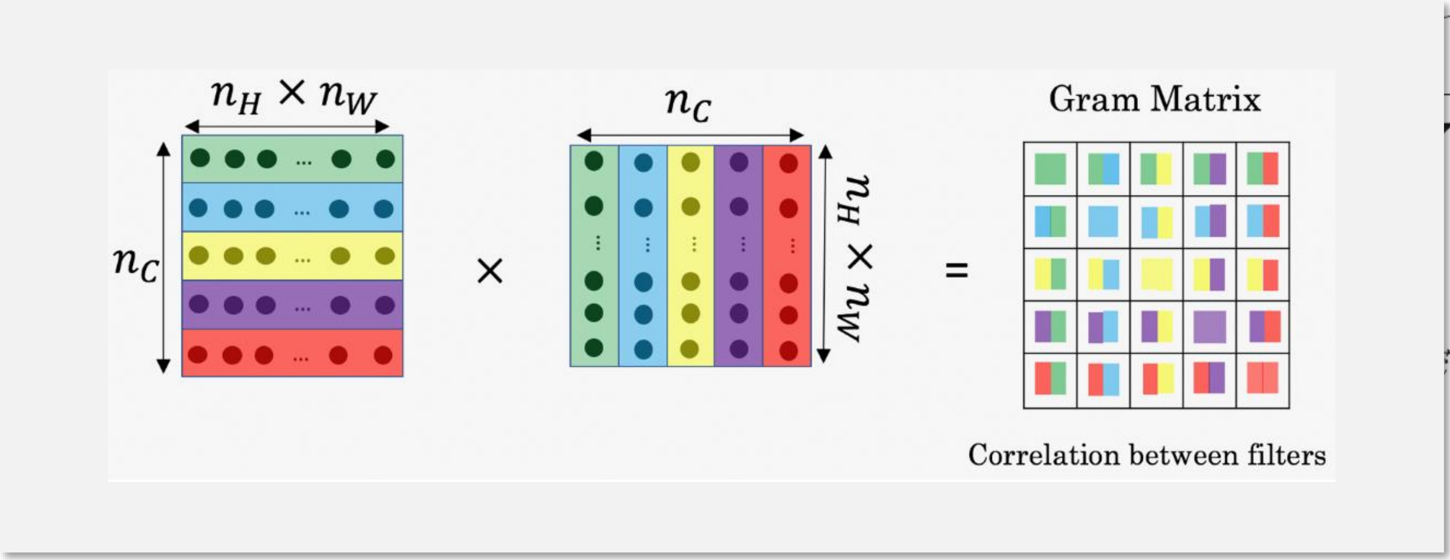

$$\ell_{total} = w_1 \ell_{content} + w_2 \ell_{style} + w_3 \ell_{tv}$$

# Exercise 1: Style Transfer

- Task 3:
  - Compute style loss:
    - Compute normalized Gram matrix

$$\hat{G}_{ij}^{L} = \frac{1}{(n_c n_h n_w)} \sum_k F_{ik}^{L} F_{jk}^{L}$$




$$\ell_{total} = w_1 \ell_{content} + w_2 \ell_{style} + w_3 \ell_{tv}$$

# Exercise 1: Style Transfer

- Task 3:
  - Compute style loss:
    - Compute normalized Gram matrix

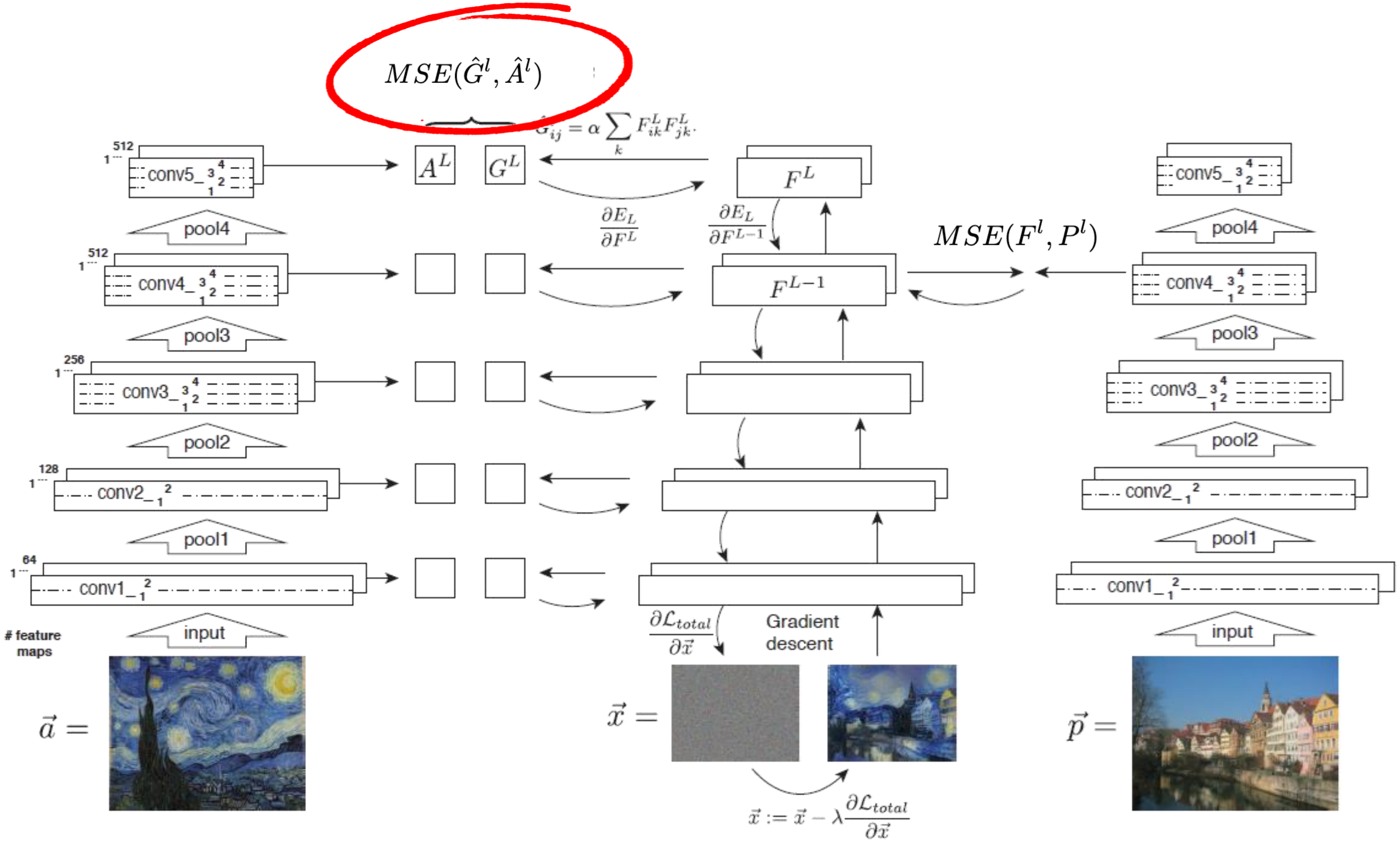
$$G_{ij}^{L} = \frac{1}{(n_c n_h n_w)} \sum_k F_{ik}^{L} F_{jk}^{L}$$

```python
def gram_matrix(x):
    """ Calculates the gram matrix for a given feature matrix.

    # Parameters:
        @x, torch.tensor of size (b, c, h, w)

    # Returns the gram matrix
    """
    # TODO: 3.2 Implement the calculation of the normalized gram matrix.
    # Do not use for-loops, make use of Pytorch functionalities.

    return x
```

# Exercise 1: Style Transfer

- Task 3:
  - Compute style loss:
    - MSE based on gram matrices for the desired layers

$$\ell_{style} = \frac{1}{|L|}\sum_{l}^{|L|} MSE(\hat{G}^l, \hat{A}^l)$$




$$\ell_{total} = w_1\ell_{content} + w_2\ell_{style} + w_3\ell_{tv}$$

TU Delft

# Exercise 1: Style Transfer

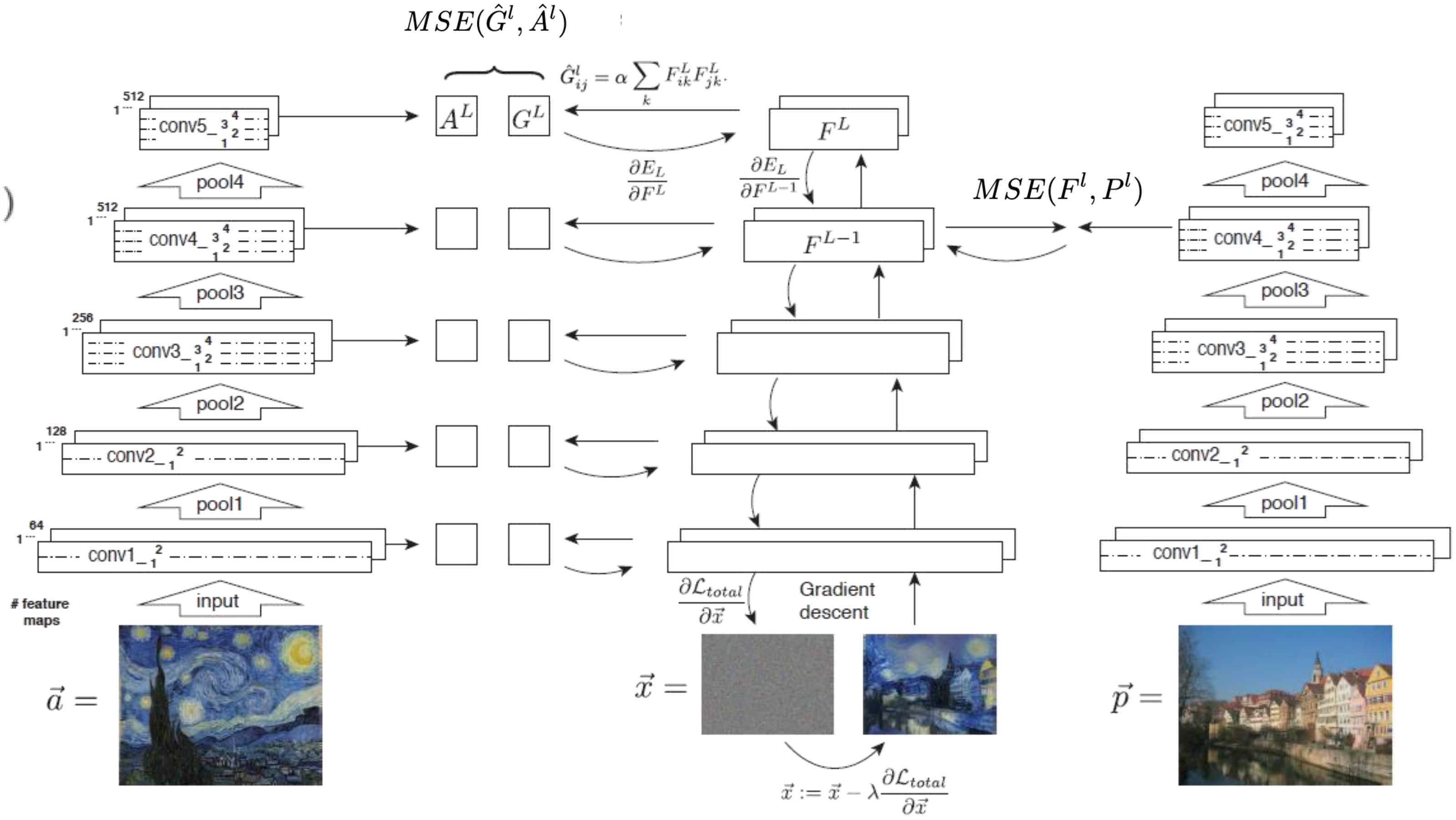
- Task 3:
  - Compute style loss:
    - Compute Gram matrices
    - Form weighted style loss

```python
def style_loss(input_features, style_features, style_layers):
    """ Calculates the style loss as in Gatys et al. 2016.

    # Parameters:
        @input_features, VGG features of the image to be optimized. It is a
            dictionary containing the layer names as keys and the corresponding
            features volumes as values.
        @style_features, VGG features of the style image. It is a dictionary
            containing the layer names as keys and the corresponding features
            volumes as values.
        @style_layers, a list containing which layers to consider for calculating
            the style loss.

    # Returns the style loss, a torch.tensor of size (1)
    """
    # TODO: 3.1 Implement the style loss given the input feature volume and the
    # style feature volume. Note that:
    # - Only the layers given in style_layers should be used for calculating this loss.
    # - Normalize the loss by the number of layers.
    # - Implement the gram_matrix function.

    return torch.rand((1), requires_grad=True)
```

# Exercise 1: Style Transfer

- Task 4:
  - Compute TV loss

$$\ell_{tv} = \frac{1}{c * K * J} \sum_{k,j} (|\vec{x}_{k,j+1} - \vec{x}_{k,j}| + |\vec{x}_{k+1,j} - \vec{x}_{k,j}|)$$




$$\ell_{total} = w_1 \ell_{content} + w_2 \ell_{style} + w_3 \ell_{tv}$$

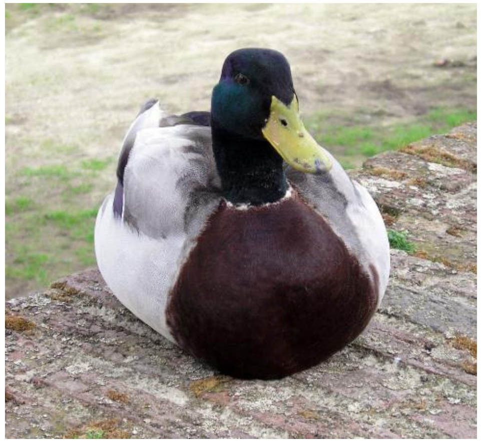
# Exercise 1: Style Transfer

- Task 4:
  - Compute TV loss

```
def total_variation_loss(y):
        """ Calculates the total variation across the spatial dimensions.

        # Parameters:
                @x, torch.tensor of size (b, c, h, w)
        # Returns the total variation, a torch.tensor of size (1)
  """
        # TODO: 4. Implement the total variation loss. Normalize by tensor dimension sizes

        return torch.rand((1), requires_grad=True) # Initialize placeholder such that the code runs
```

# Exercise 1: Style Transfer

- Task 5:
  - Given:
    - The following content and style images:



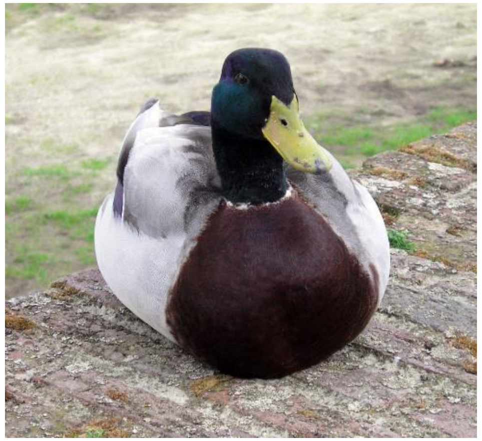
Content

Style 1

a. Perform style transfer for the content image and the Style 1 image as style.
   - Expected output:

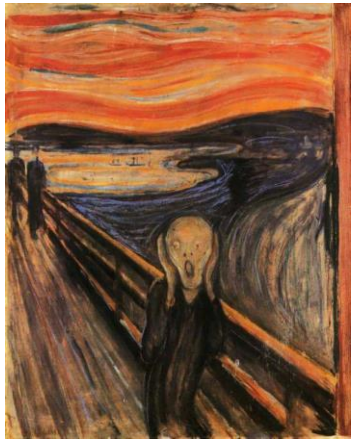

# Exercise 1: Style Transfer

- Task 5:
  - Given:
    - The following content and style images:



Content

Style 1



Style 2

b. Now perform style transfer by using 2 separate style losses for the given style images.

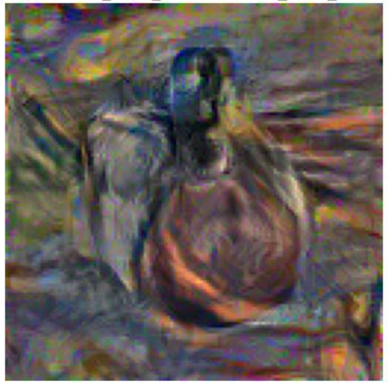
# Exercise 1: Style Transfer

- Task 5:
  b. Now perform style transfer by using 2 separate style losses for the given style images.

```
def run_double_image(
    vgg_mean, vgg_std, content_img, style_img_1, style_img_2, num_steps,
    random_init, w_style_1, w_style_2, w_content, w_tv, content_layers, style_layers, device):

    # TODO: 5. Implement style transfer for two given style images.

    return content_img
```

- Expected output:

double img_size-128 num_steps-400 w_style_1-100000.0 w_style_2-100000.0 w_content-2 w_tv-15



double img_size-256 num_steps-600 w_style_1-500000.0 w_style_2-500000.0 w_content-1 w_tv-15

TU Delft

# Additional Notes/Hints

- Here you can find the equations that we used to produce the results in the folder *expected_output*:

Gram Matrix

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l, \tag{1}$$

with $i = 1...n_c$, and $j = 1...n_h \times n_w$.

Normalized Gram Matrix:

$$\hat{G}^l = \frac{G^l}{n_c n_h n_w} \tag{2}$$

The style loss:

$$\ell_{style} = \frac{1}{|L|} \sum_l^{|L|} MSE(\hat{G}^l, \hat{A}^l), \tag{3}$$

with $|L|$ being the number of layers and $MSE$ being the mean squared error. $\hat{G}^l$ is the normalized Gram matrix for the image we are optimizing at layer $l$, while $\hat{A}^l$ is the normalized Gram matrix for the style image. Similarly for the content loss:

$$\ell_{content} = \frac{1}{|L|} \sum_l^{|L|} MSE(F^l, K^l), \tag{4}$$

where $F^l$ is feature volume at layer $l$ of the image that we are optimizing and $K^l$ the feature volume of our content image.

We further apply TV loss for piece-wise smoothness on the image $I$:

$$\ell_{tv} = \frac{1}{c * K * J} \sum_{k,j} (|\vec{x}_{k,j+1} - \vec{x}_{k,j}| + |\vec{x}_{k+1,j} - \vec{x}_{k,j}|) \tag{5}$$

where $K$ and $J$ are spatial dimensions and $c$ the channel dimension.

# Questions?

# Exercise 2: Point-based Editing on Generative Image Manifold

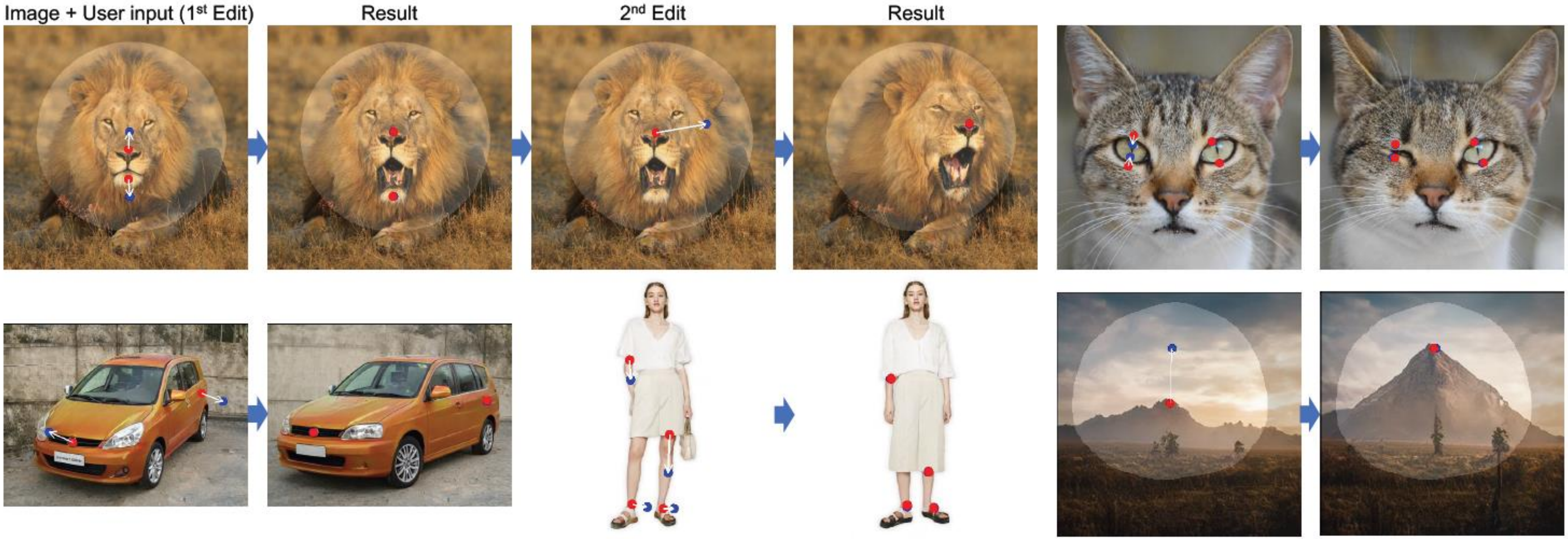
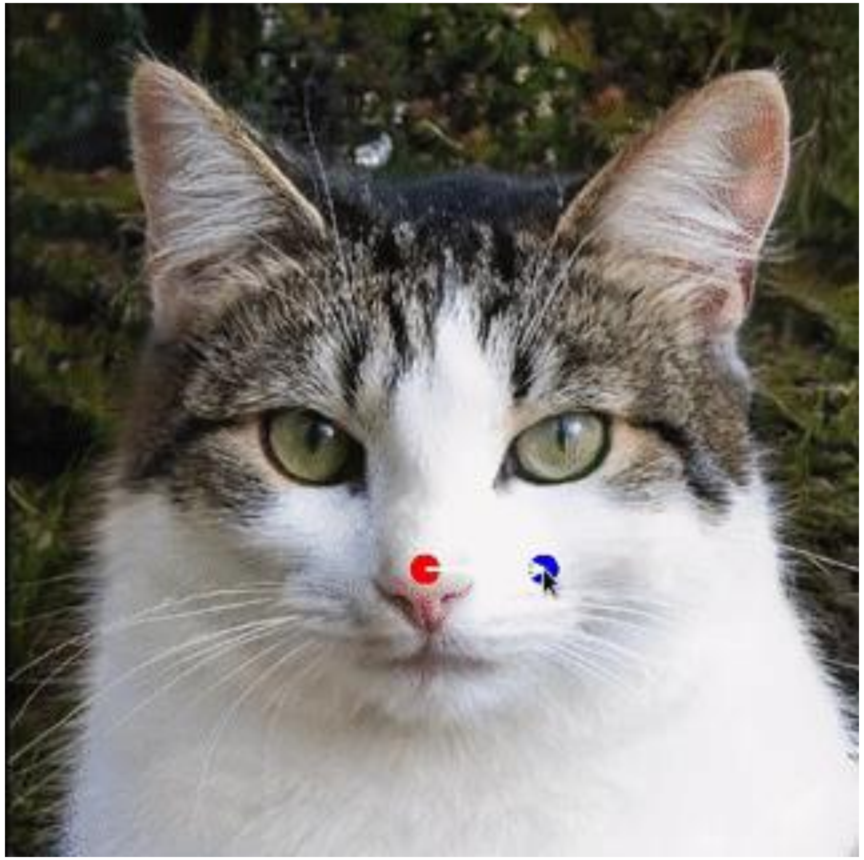



Fig. 1. Our approach *DragGAN* allows users to "drag" the content of any GAN-generated images. Users only need to click a few handle points (red) and target points (blue) on the image, and our approach will move the handle points to precisely reach their corresponding target points. Users can optionally draw a mask of the flexible region (brighter area), keeping the rest of the image fixed. This flexible point-based manipulation enables control of many spatial attributes like pose, shape, expression, and layout across diverse object categories. Project page: https://vcai.mpi-inf.mpg.de/projects/DragGAN/.

# Exercise 2: Point-based Editing on Generative Image Manifold

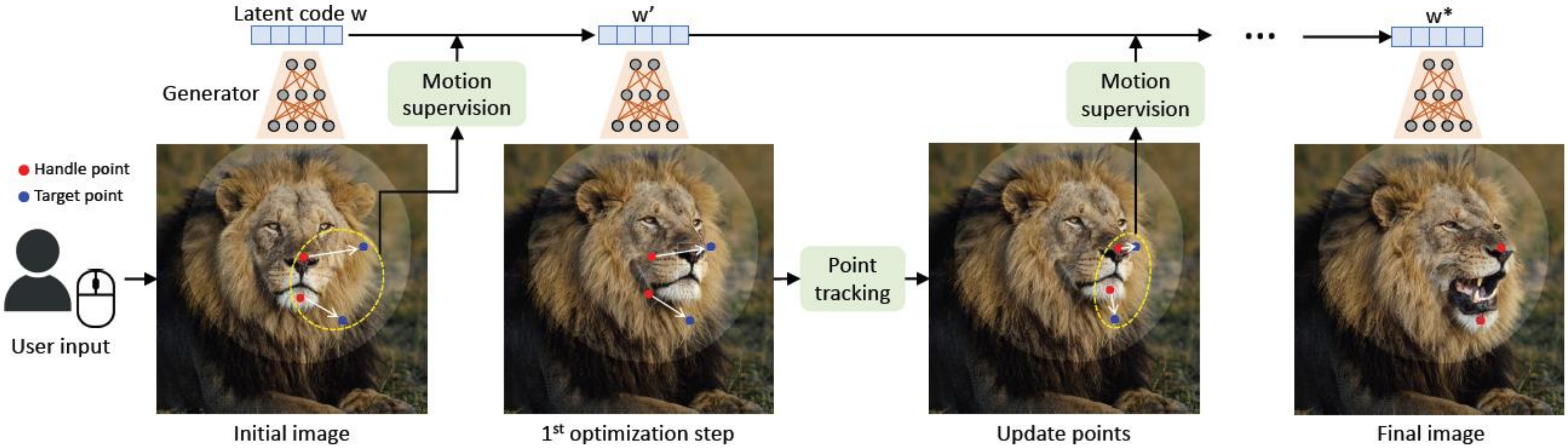


# Exercise 2: Point-based Editing on Generative Image Manifold

- In the second part of the assignment, we will focus on image editing based on the approach by Pan et al. (2023), „Drag Your GAN: Interactive Point-based Manipulation on the Generative Image Manifold".
- The major subtasks will be …
  - … the extraction of the features at selected points within a feature map via bilinear sampling
  - … the computation of the nearest neighbor in a local environment for point tracking
  - … the computation of the mask loss used for motion supervision
  - … conducting respective experiments

# Exercise 2: Point-based Editing on Generative Image Manifold

› Overview:



# Exercise 2: Point-based Editing on Generative Image Manifold

- Algorithm:
  - Setup:
    - Choose initial latent vector $w$ and initialize feature block $\boldsymbol{F}$
    - Define handle point $p$ and target point $t$
    - Sample feature $f_0$ at initial $p$

1. Shift $p$ by $d = \dfrac{t - p}{\| t - p \|_2}$
2. Optimize $w$ such that neighborhood features around $p$ appear at $p + d$ to retrieve $\boldsymbol{F}'$ *(Motion Supervision)*
3. Find new location of $p$ in $\boldsymbol{F}'$ *(Point Tracking)*
4. Repeat 1 - 3 until distance between current handle point and target point is small enough

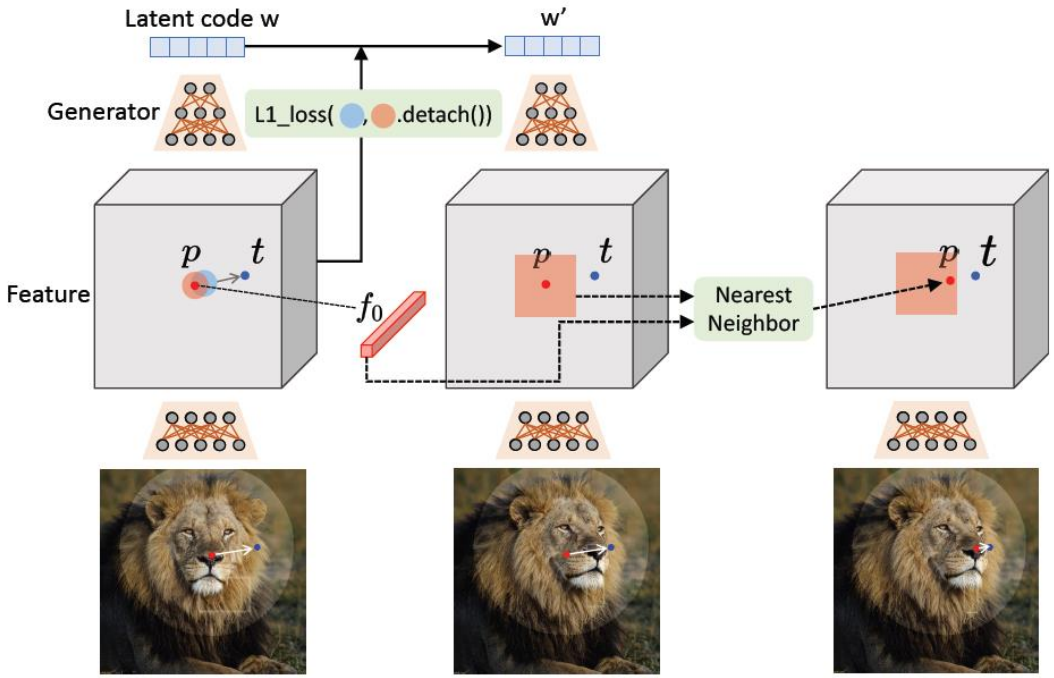



Fig. 3. Method. Our motion supervision is achieved via a shifted patch loss on the feature maps of the generator. We perform point tracking on the same feature space via the nearest neighbor search.

# Exercise 2: Point-based Editing on Generative Image Manifold

- Task 1:
  - Extract a neighborhood of points $q_i$ and $q_i + d$ based on $p$ and $r$

$$\mathcal{L} = \sum_{\mathbf{q}_i \in \Omega_1(\mathbf{p}, r_1)} \frac{1}{cn} \|\mathbf{F}(\mathbf{q}_i) - \mathbf{F}(\mathbf{q}_i + \mathbf{d})\|_1$$

$c :=$ channel dimension of $\mathbf{F}$
$n :=$ neighbourhood size

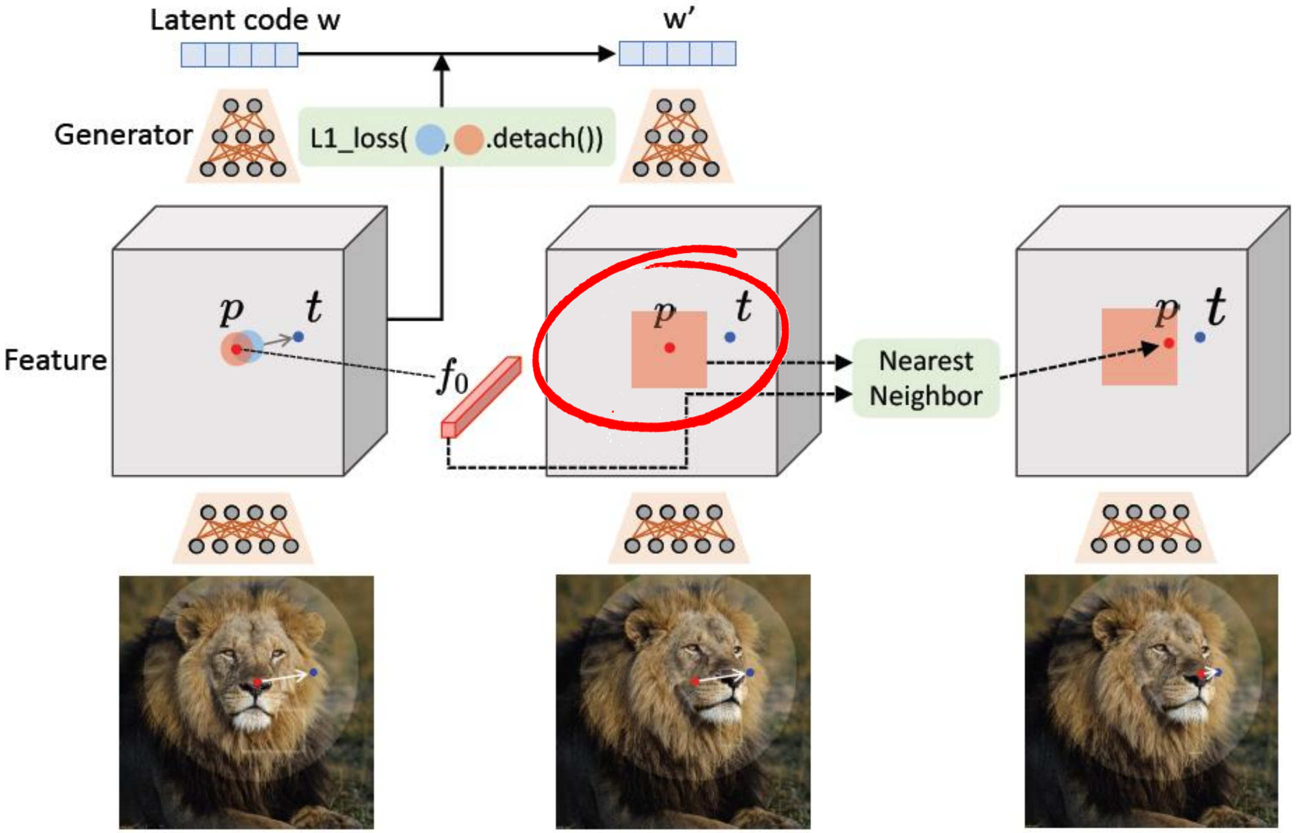
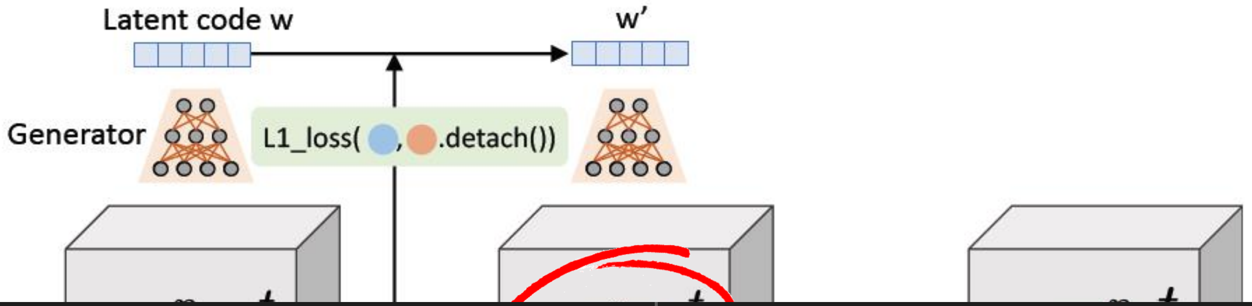



Fig. 3. Method. Our motion supervision is achieved via a shifted patch loss on the feature maps of the generator. We perform point tracking on the same feature space via the nearest neighbor search.

# Exercise 2: Point-based Editing on Generative Image Manifold

- Task 1:
  - Extract a neighborhood of points $q_i$ and $q_i + d$ based on $p_i$

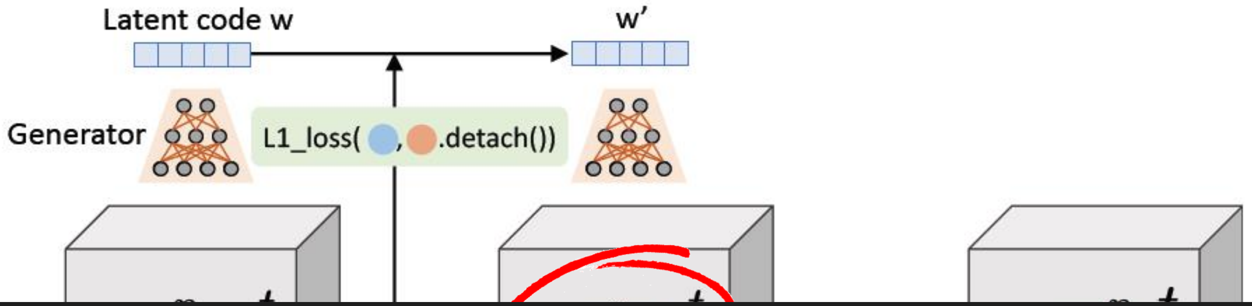



```python
def get_neighbourhood(p, radius):
    """ Returns a neighbourhood of points around p.

    # Parameters:
        @p: torch.tensor size [2], the current handle point p
        @radius: int, the radius of the neighbourhood to return

    # Returns: torch.tensor size [radius * radius, 2], the neighb
    """
    # TODO: 1. Get Neighbourhood
    # Note that the order of the points in the neighbourhood does not matter.
    # Do not use for-loops, make use of Pytorch functionalities.

    return torch.zeros(radius * radius, 2), device=p.device) # Initialize placeholder such that the code runs
```

The marked parts in the provided framework are meant to specify regions of a size of
**(2*radius+1) x (2*radius+1)**
and have to be changed according to the following slide.

# Exercise 2: Point-based Editing on Generative Image Manifold

- Task 1:
  - Extract a neighborhood of points $q_i$ and $q_i + d$ based on $p_i$




```python
def get_neighbourhood(p, radius):
    """ Returns a neighbourhood of points around p.

    # Parameters:
        @p: torch.tensor size [2], the current handle point p
        @radius: int, the radius of the neighbourhood to return

    # Returns: torch.tensor size [ (2*radius+1)**2 , 2], the neighb
    """
    # TODO: 1. Get Neighbourhood
    # Note that the order of the points in the neighbourhood does n
    # Do not use for-loops, make use of Pytorch functionalities.

    return torch.zeros(( (2*radius+1)**2 , 2), device=p.device) # Initialize placeholder such that the code runs
```

The marked parts in the provided framework are meant to specify regions of a size of
**(2*radius+1) x (2*radius+1)**
and have to be changed according to this slide.

# Exercise 2: Point-based Editing on Generative Image Manifold

- Task 2:
  - Sample the features at points $q_i$ and $q_i + \mathrm{d}$ within a feature map

$$\mathcal{L} = \frac{1}{cn} \sum_{\mathbf{q}_i \in \Omega_1(\mathbf{p}, r_1)} \|\mathbf{F}(\mathbf{q}_i) - \mathbf{F}(\mathbf{q}_i + \mathbf{d})\|_1$$

$c :=$ length of feature vector, $n :=$ neighbourhood size

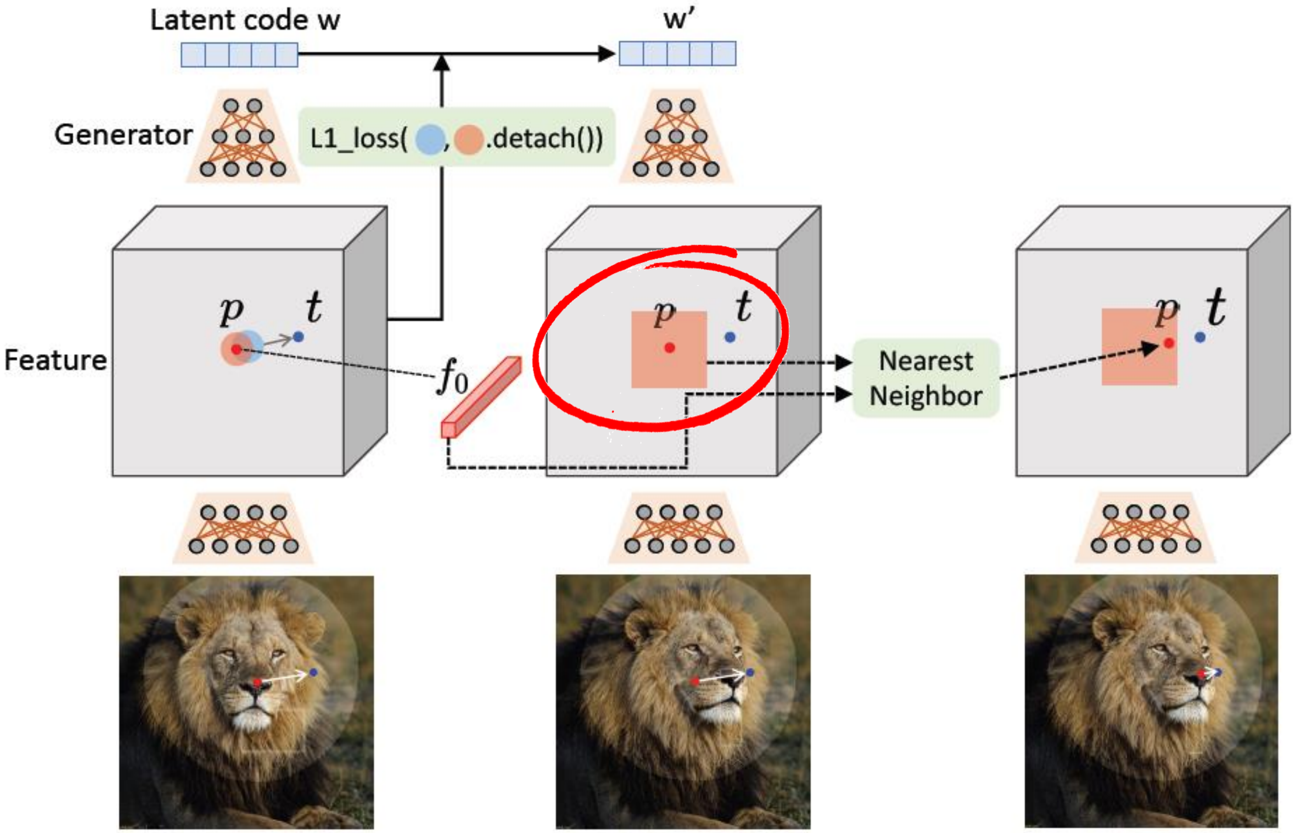
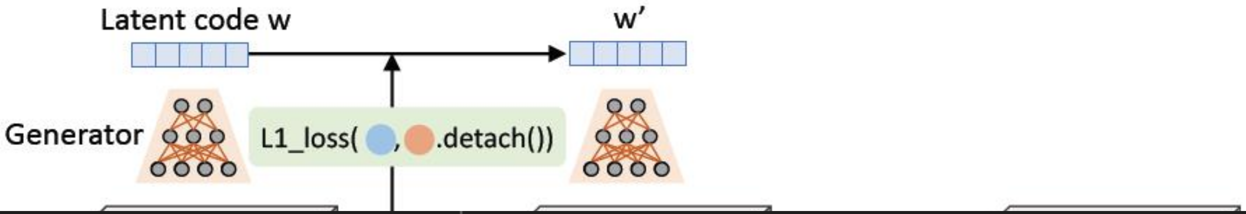



Fig. 3. Method. Our motion supervision is achieved via a shifted patch loss on the feature maps of the generator. We perform point tracking on the same feature space via the nearest neighbor search.

# Exercise 2: Point-based Editing on Generative Image Manifold

- Task 2:
  - Sample the features




```python
def sample_p_from_feature_map(q_N, F):
    """ Samples the feature map F at the points q_N.

    # Parameters:
        @q_N: torch.tensor size [N, 2], the points to sample from the feature map
        @F: torch.tensor size [1, C, H, W], the feature map of the current image

    # Returns: torch.tensor size [N, C], the sampled features at q_N
    """
    assert F.shape[-1] == F.shape[-2]

    # TODO: 2. Sample features from neighbourhood
    # NOTE: As the points in q_N are floats, we can not access the points from the feature map via indexing.
    # Bilinear interpolation is needed, PyTorch has a function for this: Func.grid_sample.
    # NOTE: To check whether you are using grid_sample correctly, you can pass an index matrix as the feature map F_i
    # where each entry corresponds to its x,y index. If you sample from this feature map, you should get the same points back.

    return torch.zeros((q_N.shape[0], F.shape[1]), device=q_N.device) # Initialize placeholder such that the code runs
```

# Exercise 2: Point-based Editing on Generative Image Manifold

- Task 3:
  - After optimization step based on task 1 and 2 we receive optimized feature map $F'$
  - Compute the position of the nearest neighbor in feature space to find point that most closely matches the feature vector $f_0$ of the initial handle point ($f_p = f_0$)

$$\mathbf{p} := \arg\min_{q_i \in \Omega_2(\mathbf{p}, r_2)} \|\mathbf{F}'(\mathbf{q}_i) - \mathbf{f}_0\|_1$$

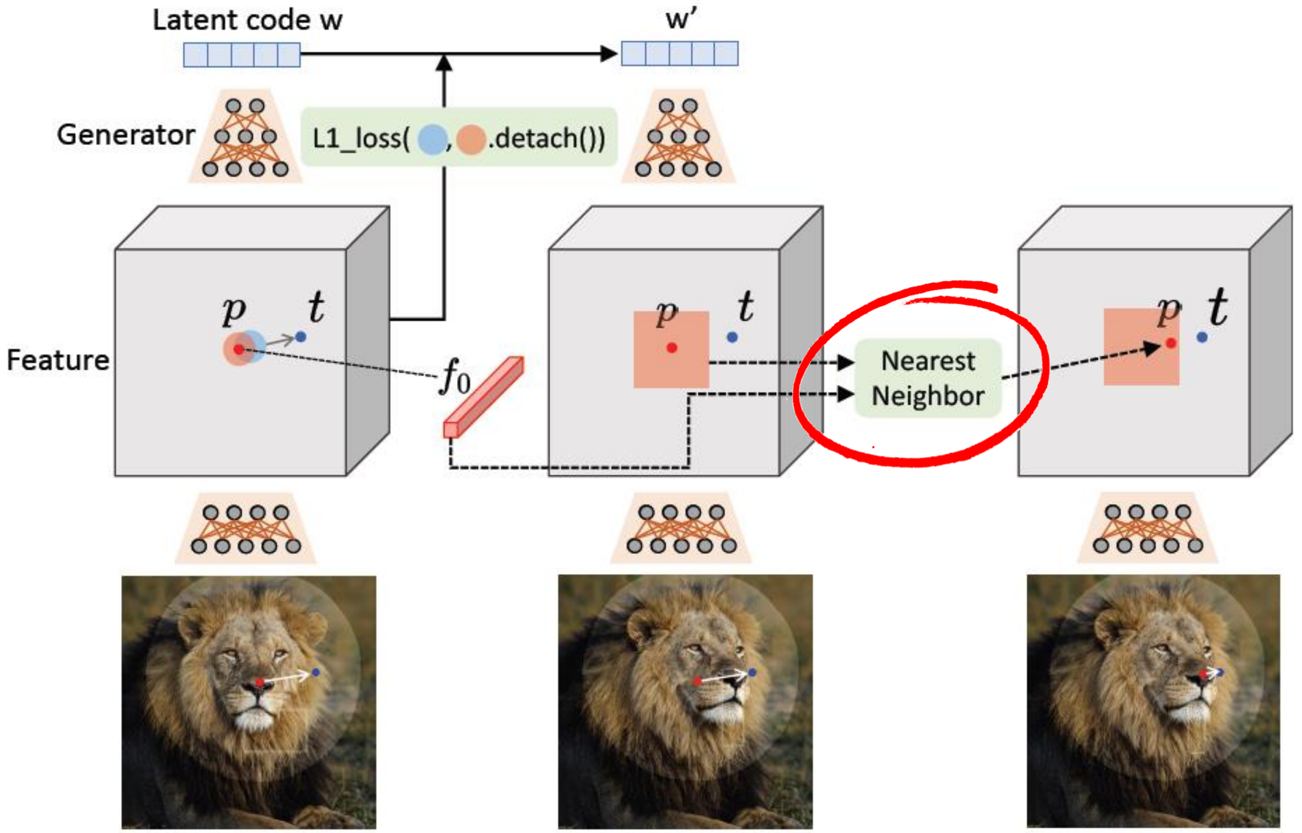
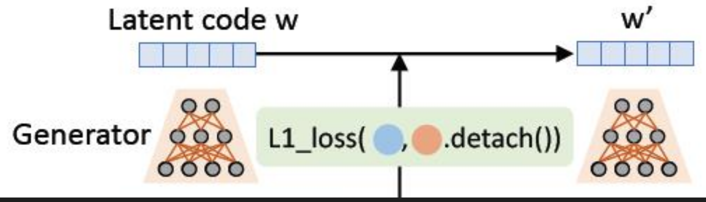



Fig. 3. Method. Our motion supervision is achieved via a shifted patch loss on the feature maps of the generator. We perform point tracking on the same feature space via the nearest neighbor search.

# Exercise 2: Point-based Editing on Generative Image Manifold

- Task 3:
  - After optimization step based




```python
def nearest_neighbour_search(f_p, F_q_N, q_N):
    """ Does a nearest neighbourhood search in feature space to find the new handle point position.

    # Parameters:
        @f_p: torch.tensor size [1, C], the feature vector of the handle point p
        @F: torch.tensor size [1, C, H, W], the feature map of the current image
        @q_N: torch.tensor size [N, 2], corresponding points to F_q_N in the image space

    # Returns: torch.tensor size [2], the new handle point p
    """
    # TODO: 3. Neighbourhood search

    return torch.rand((2)) # Initialize placeholder such that the code runs
```

same feature space via the nearest neighbor search.

# Exercise 2: Point-based Editing on Generative Image Manifold

- Task 4:
  - Compute mask loss
    used for motion supervision

$$\mathcal{L} = \sum_{\mathbf{q}_i \in \Omega_1(\mathbf{p}, r_1)} \frac{1}{cn} \|\mathbf{F}(\mathbf{q}_i) - \mathbf{F}(\mathbf{q}_i + \mathbf{d}_i)\|_1 + \lambda \frac{1}{chw} \|(\mathbf{F} - \mathbf{F}_0) \cdot (1 - \mathbf{M})\|_1$$

$c :=$ channel dimension of $\mathbf{F}$
$n :=$ neighbourhood size
$h, w$ are spatial dimensions of $\mathbf{F}$

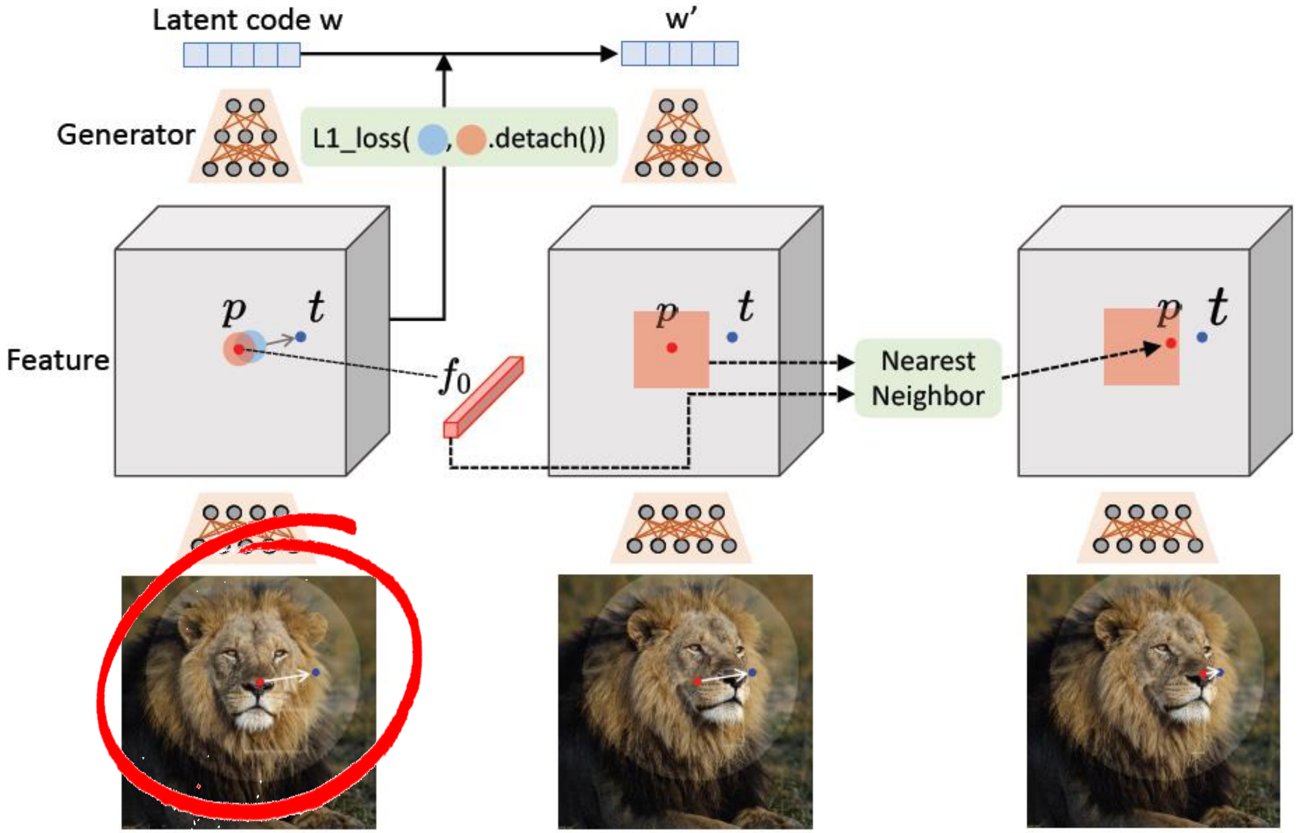
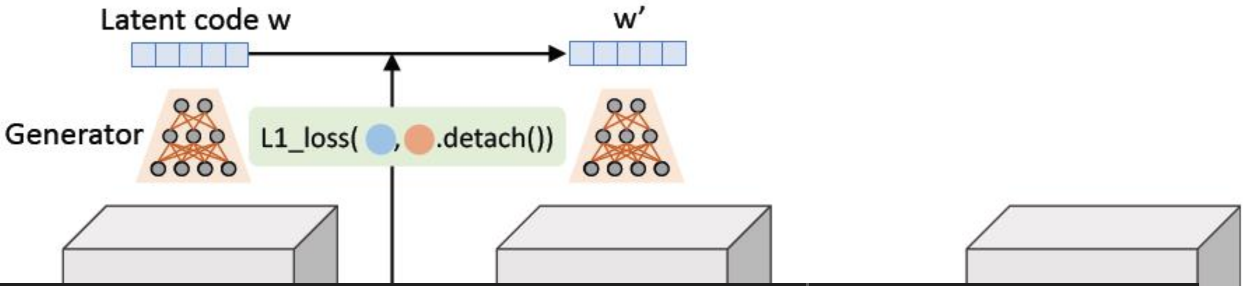



Fig. 3. Method. Our motion supervision is achieved via a shifted patch loss on the feature maps of the generator. We perform point tracking on the same feature space via the nearest neighbor search.

# Exercise 2: Point-based Editing on Generative Image Manifold

- Task 4:
  - Compute mask loss
    used for motion supervision

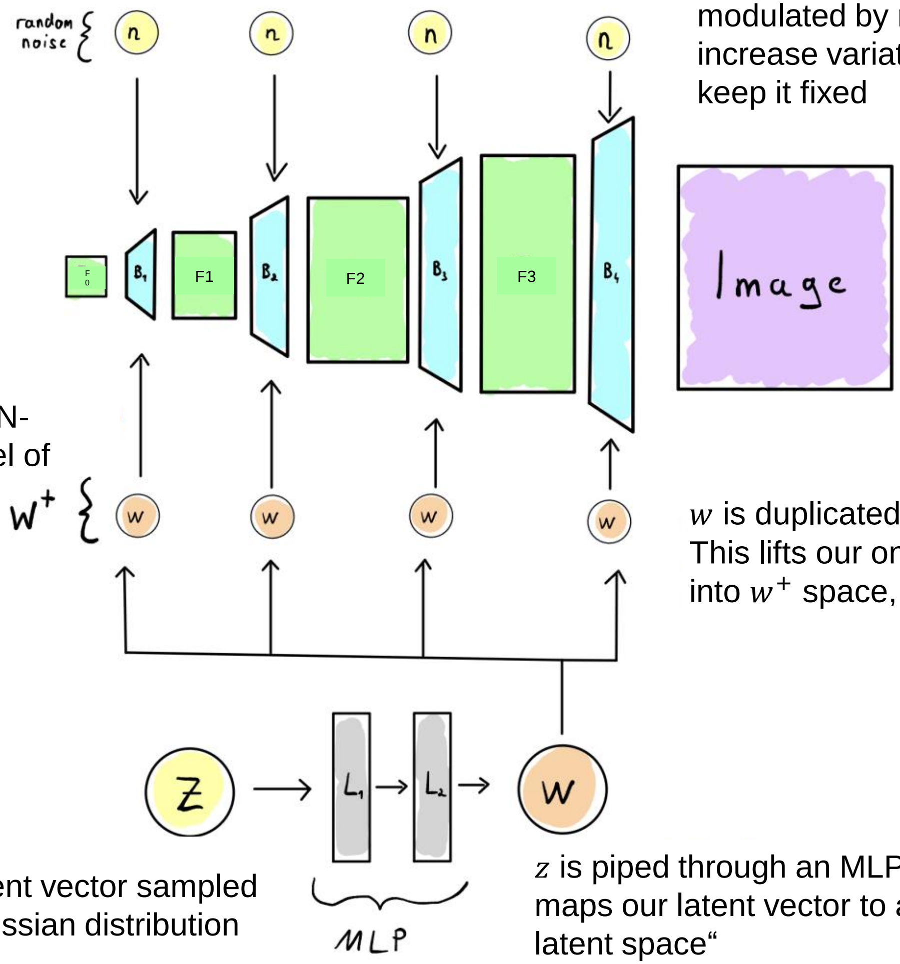



```python
def get_mask_loss(F_1, F_2, mask):
    """ Returns the mask loss.

    # Parameters:
        @F_1: torch.tensor size [1, C, H, W], the feature map of the first image
        @F_2: torch.tensor size [1, C, H, W], the feature map of the second image
        @mask: torch.tensor size [H, W], the segmentation mask.
            NOTE: 1 encodes what areas should move and 0 what areas should stay fixed.

    # Returns: torch.tensor of size [1], the mask loss
    """

    # TODO: 4. Calculate mask loss
    return torch.rand((1), requires_grad=True) # Initialize placeholder such that the code runs
```

TU Delft

# Exercise 2: Point-based Editing on Generative Image Manifold

Each feature block is further modulated by random noise to increase variation, however we keep it fixed



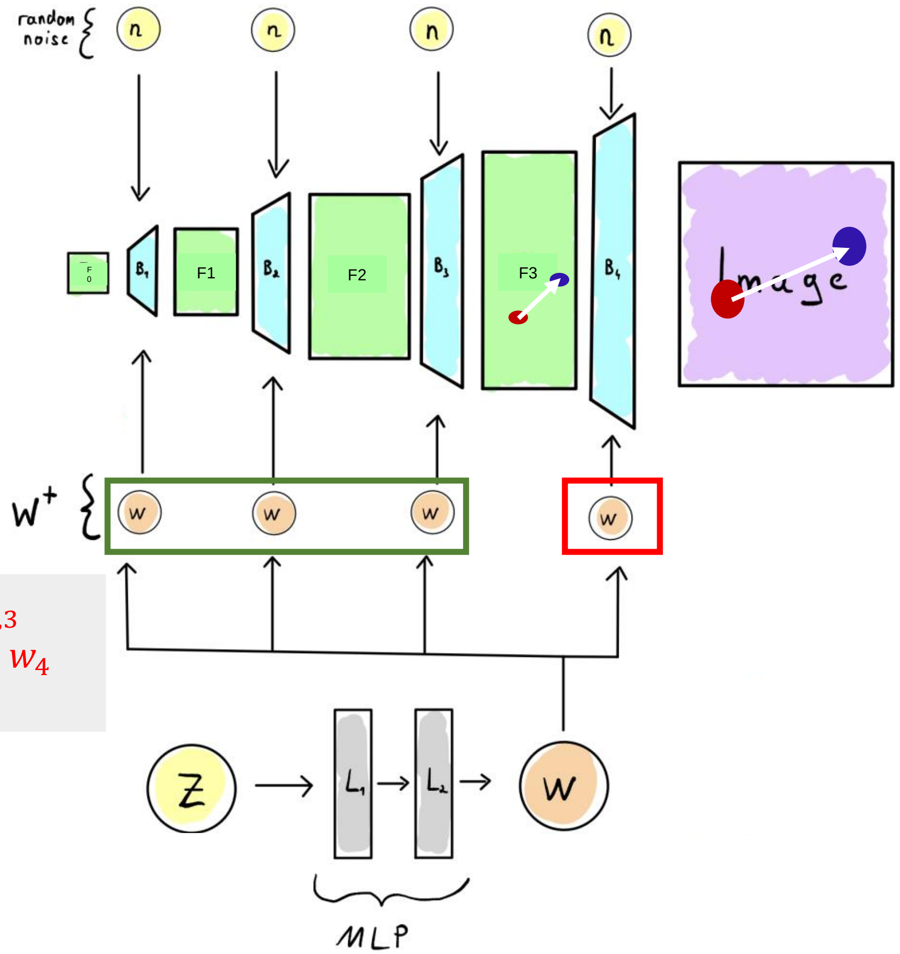

Each $w_i$ modulates the feature block within the corresponding StyleGAN-Block on a different level of resolution

$w$ is duplicated for each StyleGAN-Block. This lifts our one dimension vector $w \in \mathbb{R}^{512}$ into $w^+$ space, with $w^+ \in \mathbb{R}^{4x512}$

$z$ is our latent vector sampled from a Gaussian distribution

$z$ is piped through an MLP which maps our latent vector to a „learned latent space“

# Exercise 2: Point-based Editing on Generative Image Manifold




- Define handle point and target point in image space
- Scale $F_3$ to the same dimension as the image
- Motion supervision and point tracking on $F_3$

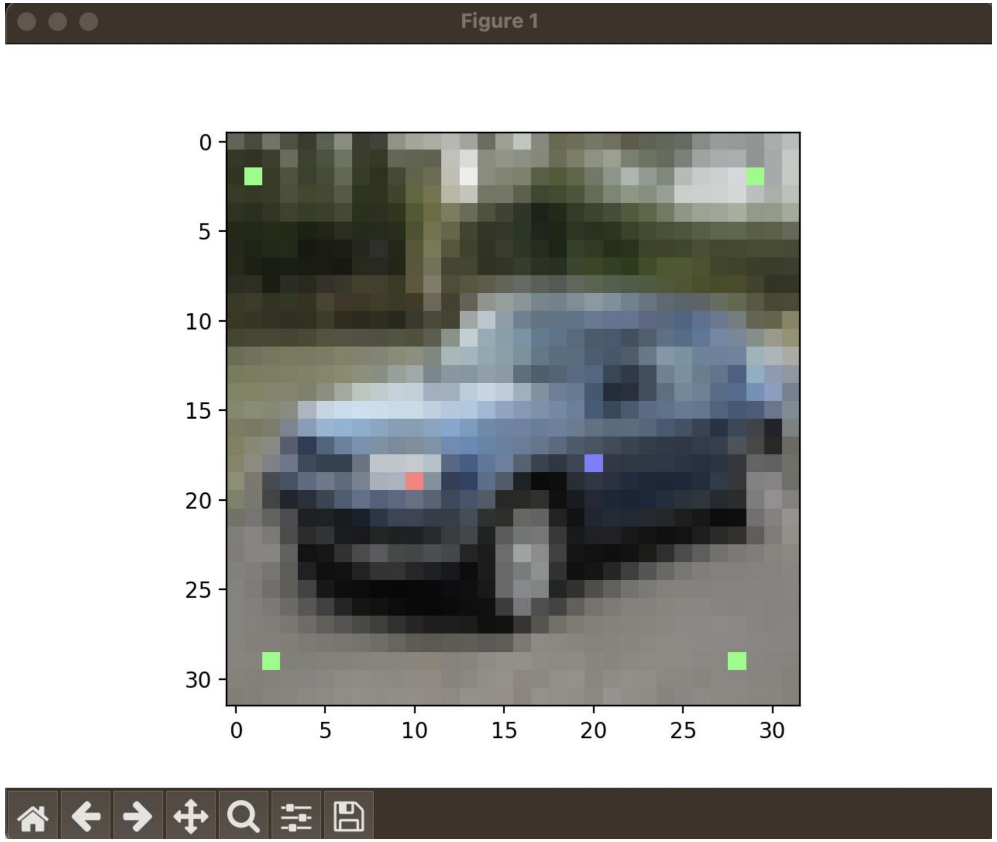
Optimise $w_{1,2,3}$ vectors, keep $w_4$ fixed

# Exercise 2: Point-based Editing on Generative Image Manifold

- Expected Outputs:
  - We have two set of sample inputs *car (w.o mask loss)* and *horse (w. mask loss)* to test your input

# Exercise 2: Point-based Editing on Generative Image Manifold

- We also provide a GUI to define your own points
  - **First click**: handle point (red)
  - **Second click**: target point (blue)
  - **All other clicks**: mask points. To finish mask, click on first mask point again (green)



# Assignment 2: Grading

| Feature | Maximum achievable points |
| --- | --- |
| **Exercise 1: Style Transfer** | |
| - Tensor normalization | 1 |
| - Content loss | 2 |
| - Computation of Gram matrix | 1 |
| - Style loss | 2 |
| - Total Variation loss | 1 |
| - Optimization using 2 style images | 1 |
| **Exercise 2: Point-based Editing on Generative Image Manifold** | |
| - Extraction of local neighborhood | 1 |
| - Extraction of features for given locations (grid sampling) | 2 |
| - Nearest Neighbor search | 1 |
| - Mask loss | 1 |
| $\Sigma$ | **13** |

# Questions?