INSTITUT
Mines-Télécom

Style Transfer by Deep Learning

Julien Guillaumin (ex Télécom Bretagne)
julien.guillaumin@imt-atlantique.net

TELECOM Bretagne

MINES Albi-Carmaux MINES Alès MINES Douai MINES Nantes MINES ParisTech Ecole Nationale Supérieure des Mines SAINT-ETIENNE TELECOM Bretagne TELECOM Ecole de Management TELECOM ParisTech TELECOM SudParis EURECOM Sophia Antipolis TELECOM Lille MINES nancy

Style Transfer by Deep Learning : Motivations

content image

style image

?

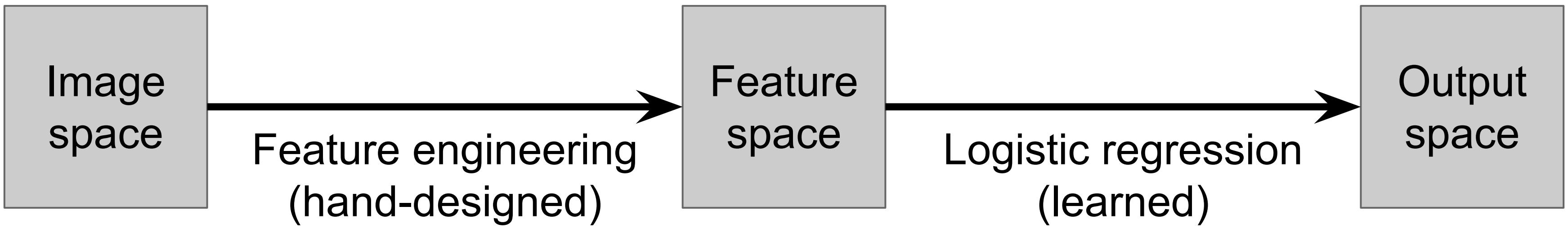
stylized image with content

- **Generative task**
  - From an image, generate a new one
- **Introduction to more complex tasks**
  - Super-resolution and colorisation
- **CNNs understanding is required**
  - Hierarchy of representations
  - Feature spaces

- Machine Learning vs Deep Learning
- CNNs visualizing and understanding
- Content and style representations
- Optimization-based style transfer
- Feed-forward method with learning
- Arbitrary style transfer

Machine Learning vs Deep Learning

Machine Learning pipeline

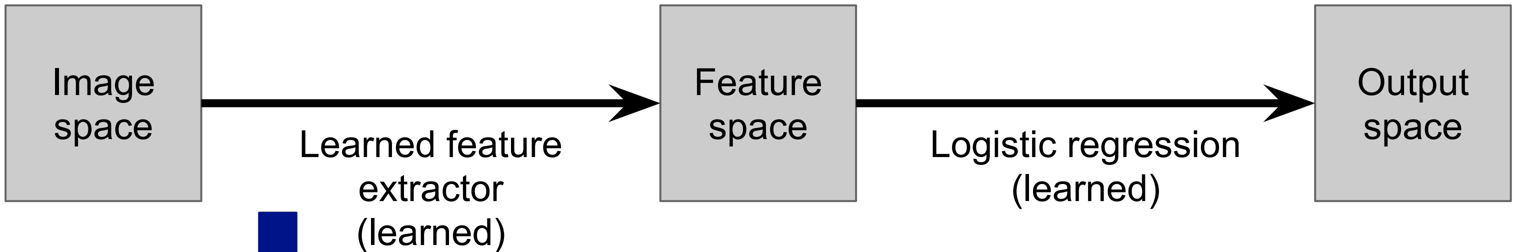


- Domain dependence
- Need expert domain to tune
- Hard to extract complex patterns

For images : HOG features, SIFT methods, Histograms, LBP features, ….

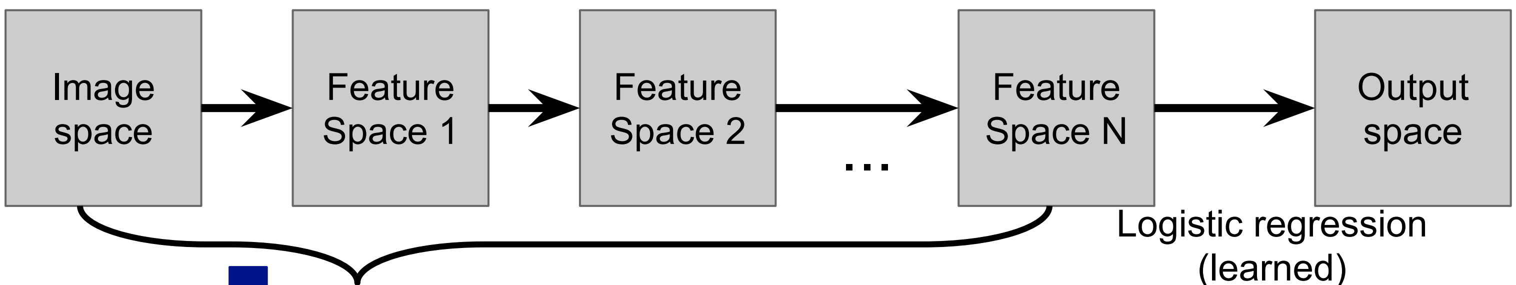
Representation Learning

Representation Learning approach



- Learns a new representation of the data
- Ex : PCA (Principal Component Analysis)

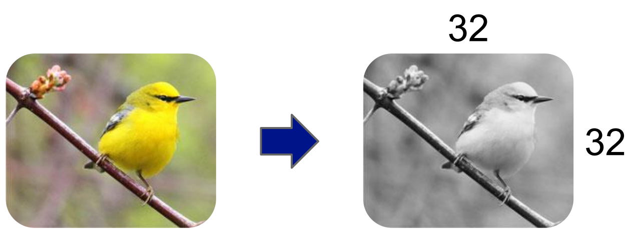
Deep Representation Learning approach

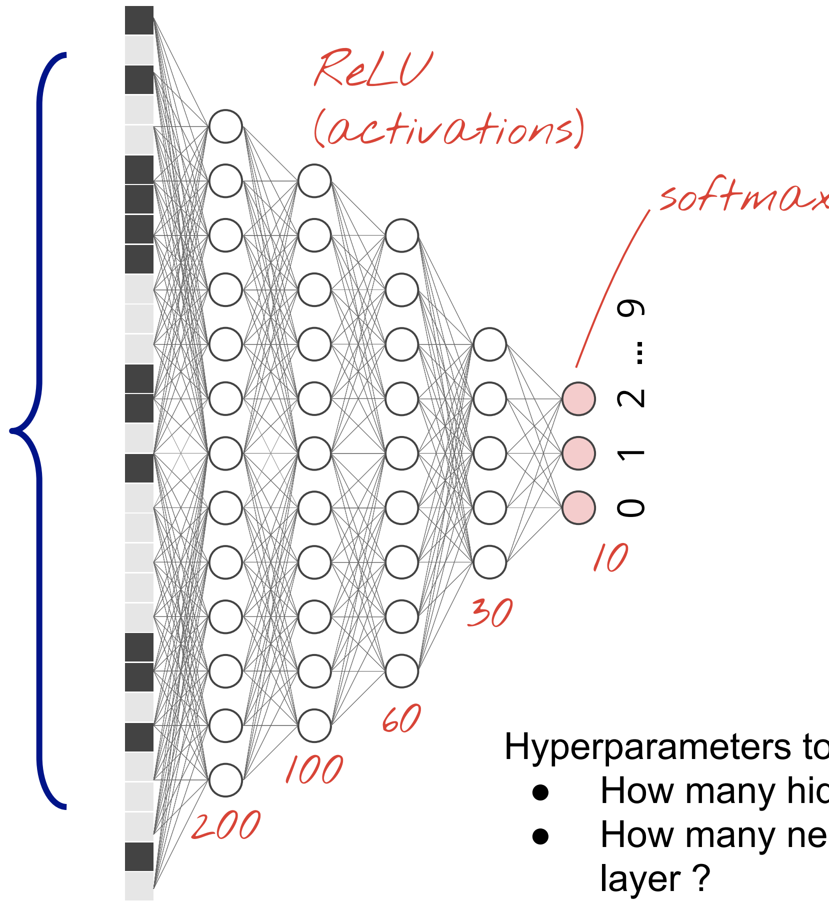


- **Learns a hierarchy of representations**
- **Can be done with Neural Networks**

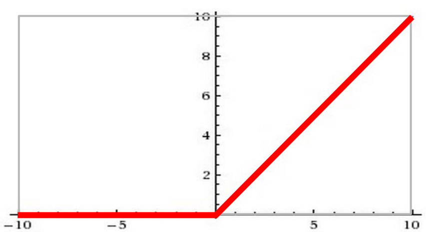
INSTITUT Mines-Télécom

Deep Neural Networks



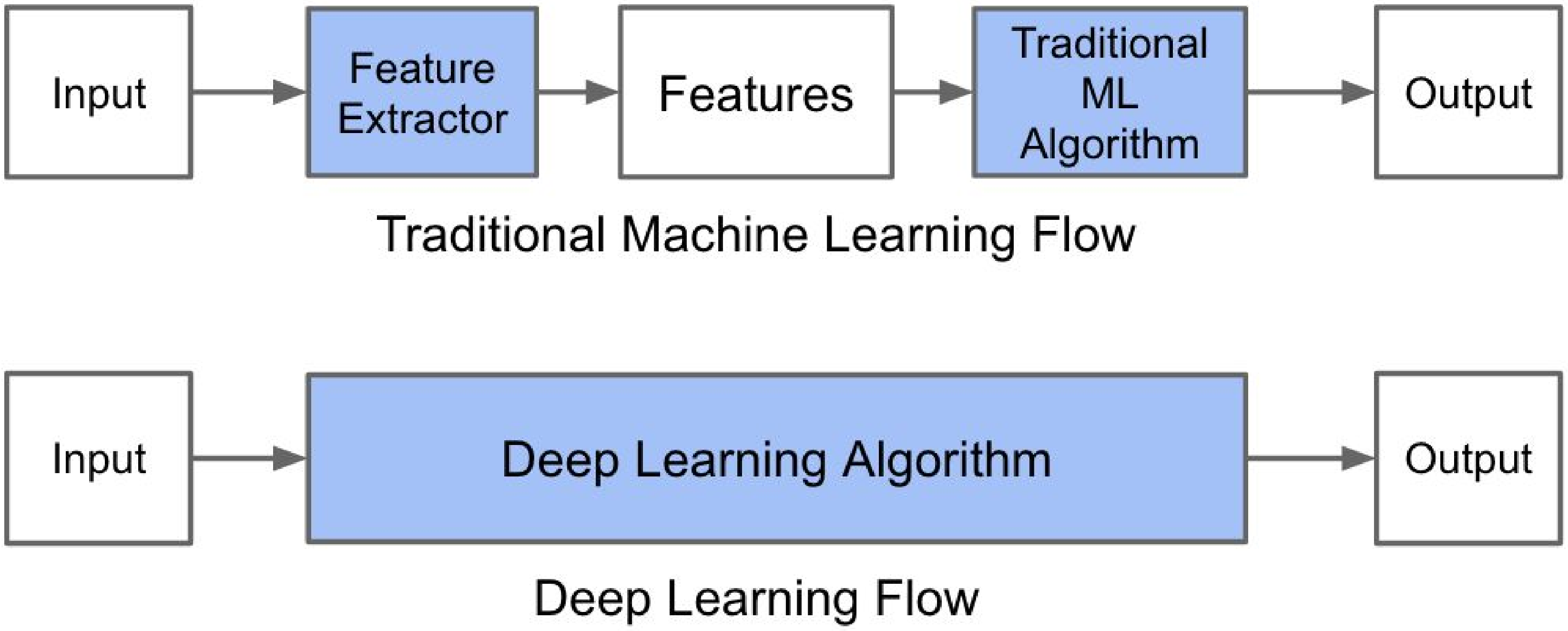


ReLU function



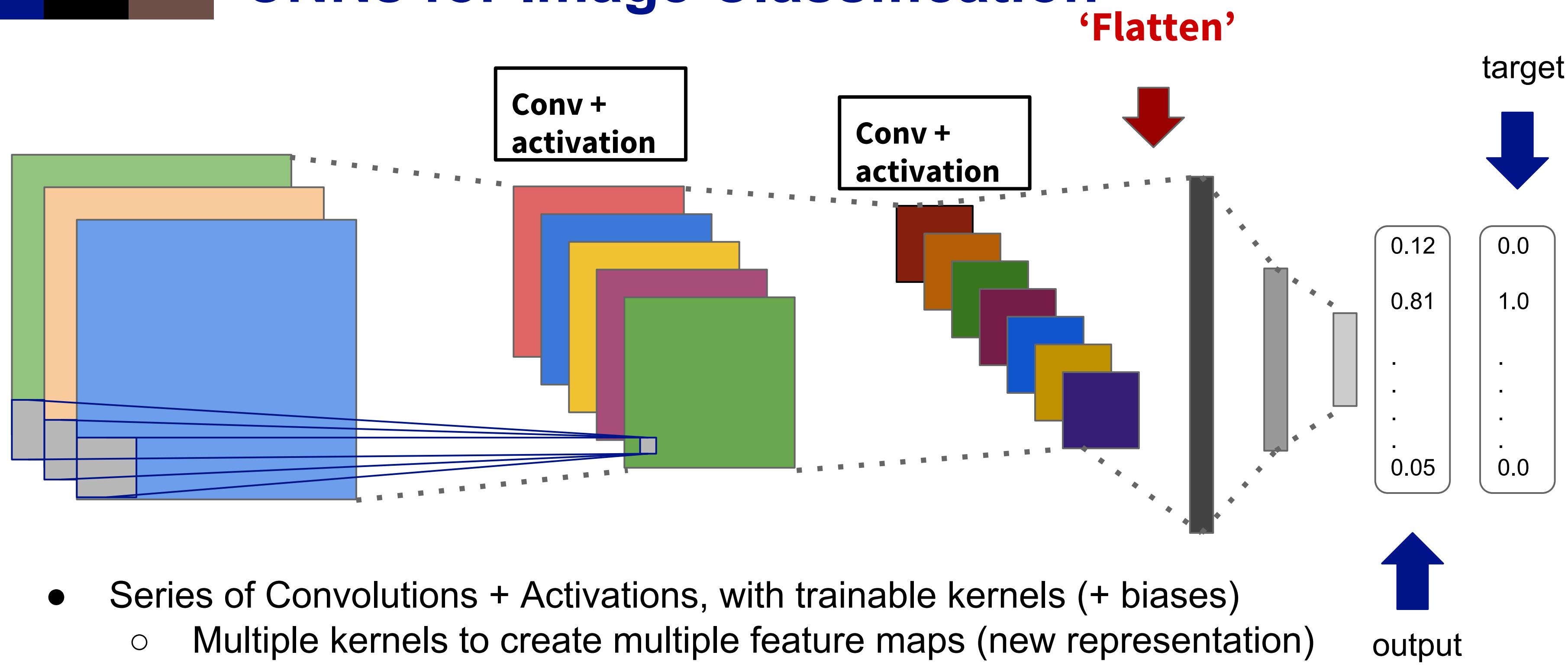
Hyperparameters to tune:

- How many hidden layers ?
- How many neurons per layer ?
- Which activation ?
- Regularization ?


Input
Feature Extractor
Features
Traditional ML Algorithm
Output
Traditional Machine Learning Flow
Input
Deep Learning Algorithm
Output
Deep Learning Flow

CNNs visualizing and understanding

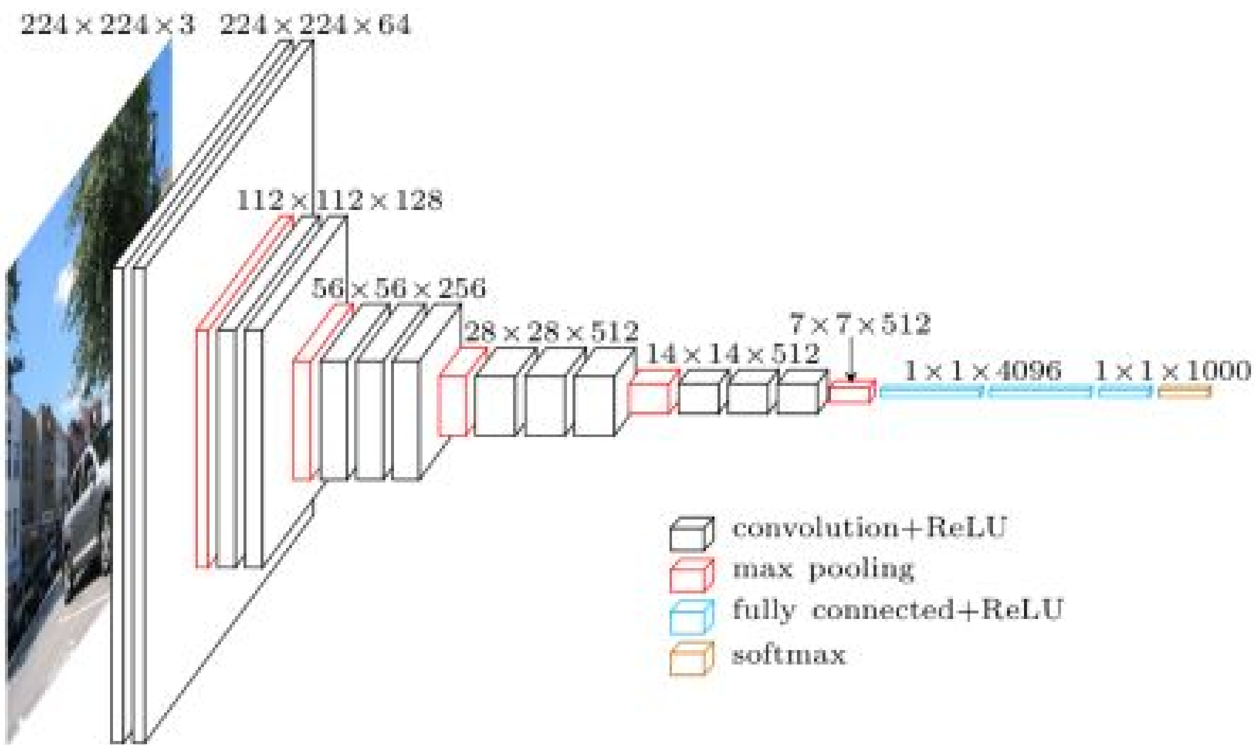
CNNs for Image Classification



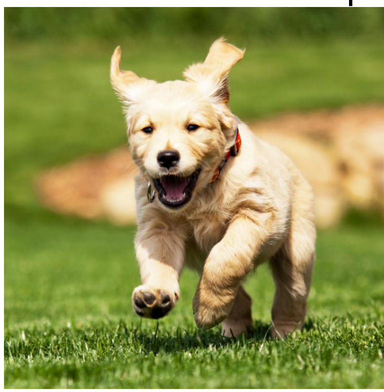
- Series of Convolutions + Activations, with trainable kernels (+ biases)
  - Multiple kernels to create multiple feature maps (new representation)
- Pooling operations to reduce the dimensions of the feature maps
- 'Flatten' operation, representation as a vector
- Fully-Connected layers (Multilayer Perceptron)
  - Learned : weight matrix and bias vector
- Training (weight-update) on error
  - Classification : cross entropy

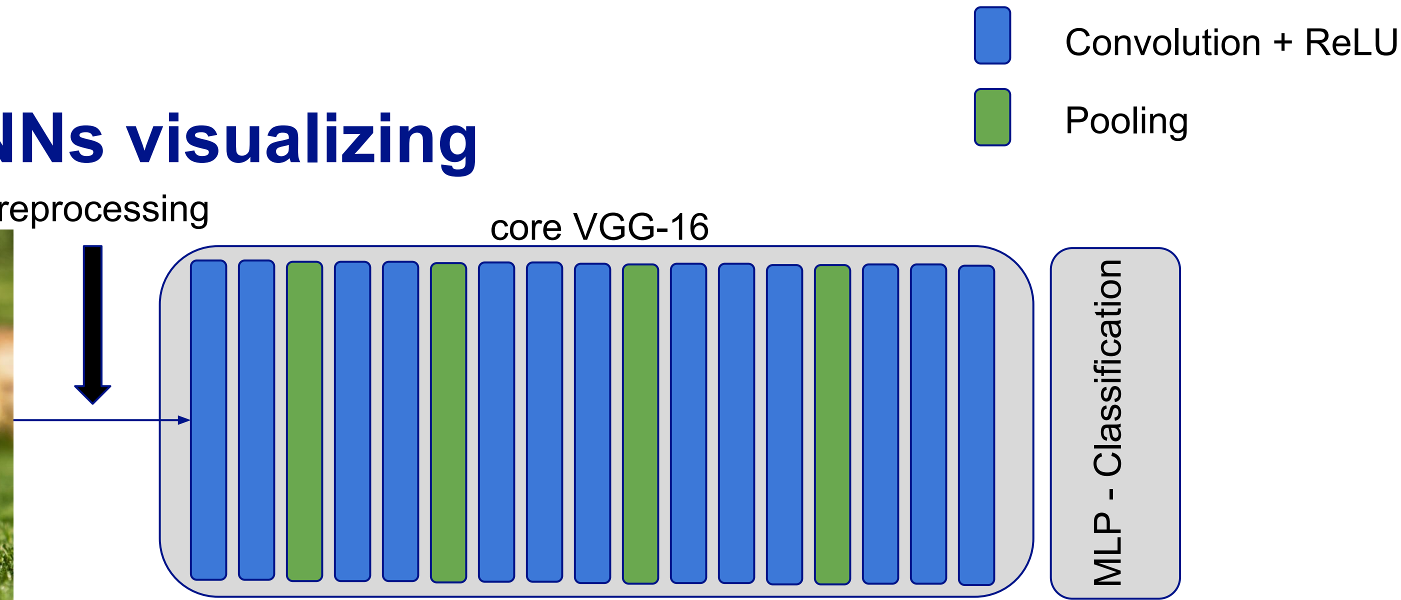
Deep Network : VGG-16

- Simple : no **inception modules** or **residual connections**
- Trained for **image classification** on ImageNet (1000 classes)
- State of the art in 2014 (92.7% top-5 test accuracy)
- 138,357,544 parameters (10% conv weights, 90% FC layers)
- No residual connections, or inception modules : Deep simple model

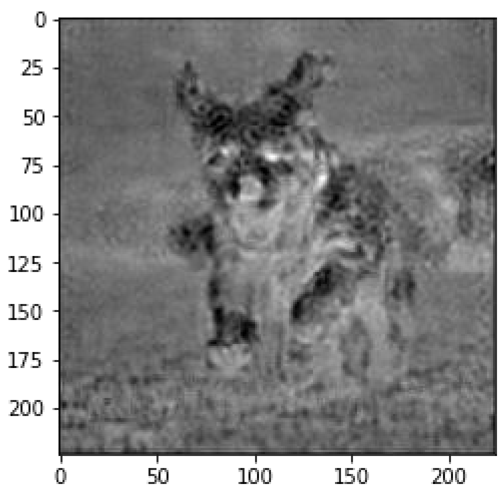


CNNs visualizing

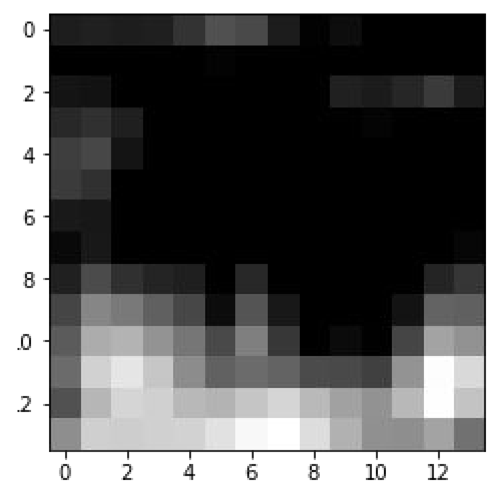




conv1_1 (224x224x64)



conv5_3 (14x14x512)



From low-level to high-level feature spaces

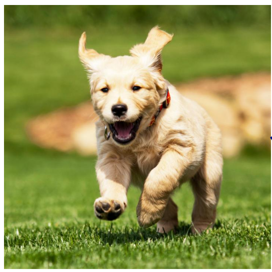
Additional visualization methods :

- Deep Dream approach [2]
  - Optimization-based
- Zeiler & Fergus [3]
  - Transposed convolutions and unpooling operations

Content & style representations

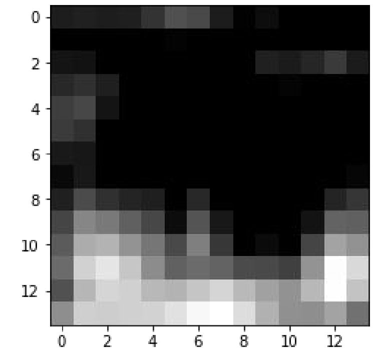
Content Representation/Reconstruction

x_c

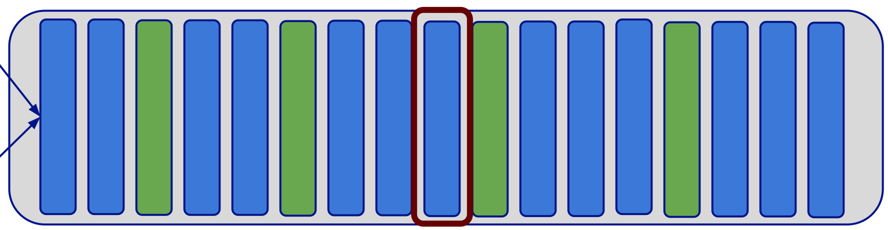


$\Phi_j(x)$ Activations of the *j*th layer

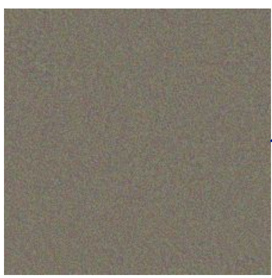
Eg : $\Phi_j(x)[40]$



conv3_3 : 56x56x256



Fixed VGG-16



x

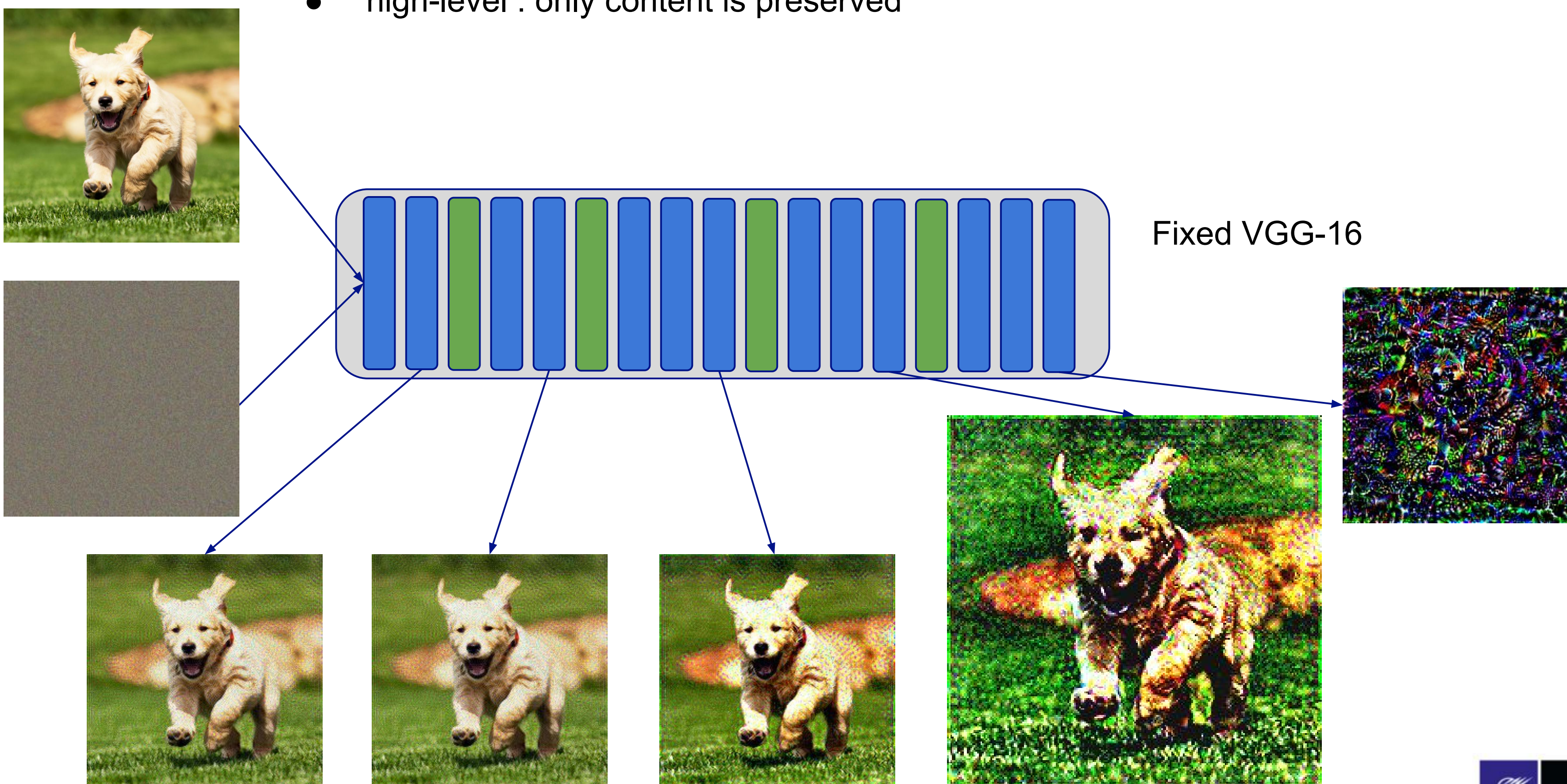
$$\hat{x} = \underset{x}{\operatorname{argmin}} \ ||\Phi_j(x) - \Phi_j(x_c)||_2^2$$

- Goal : find an image with the same activations at a given layer (all feature maps)
- Optimization problem, start from a random image

INSTITUT Mines-Télécom

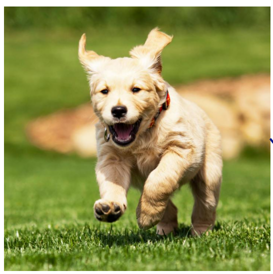
Content Representation/Reconstruction

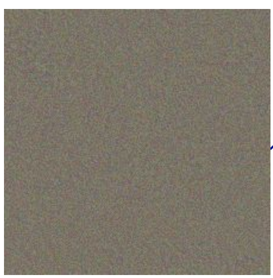
- gradient descent optimization on input image, network does not change
- loss = MSE on feature maps, 1000 iterations, Adam (lr=2.0)
- low-level : input image is correctly reconstructed, with pixel-level details
- high-level : only content is preserved

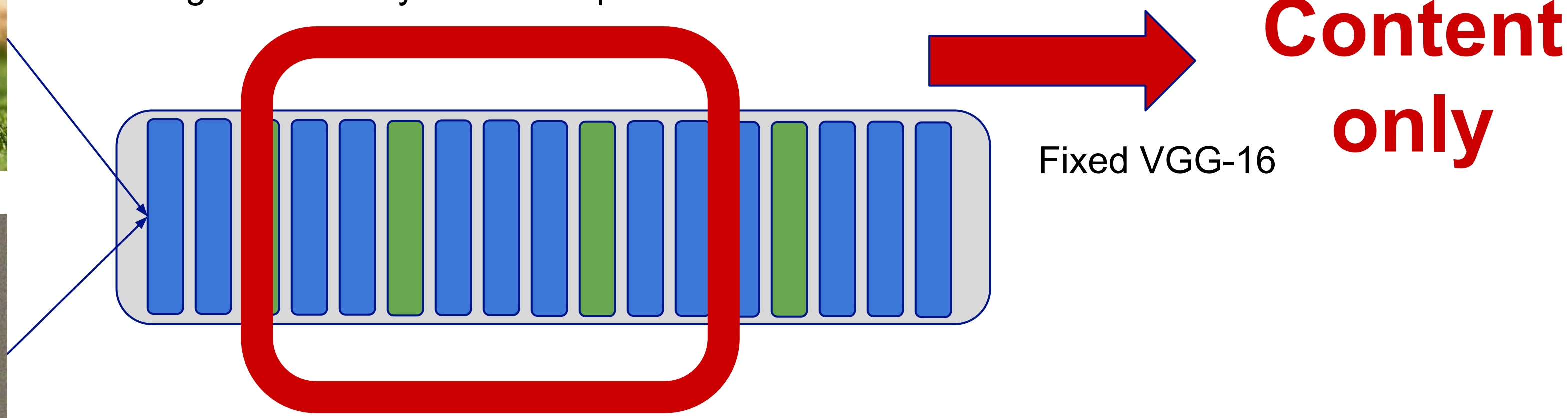


Content Representation/Reconstruction

- From a random image, reconstruct the feature maps obtained with a normal image, on a specific layer
- Gradient descent optimization on image input, network does not change
- Loss = MSE on feature maps, 1000 iterations, Adam (lr=2.0)
- Low-level : input image is correctly reconstructed, with pixel-level details
- High-level : only content is preserved







$$L_C(x, x_C) = \frac{1}{C_j H_j W_j} ||\phi_j(x_C) - \phi_j(x)||_2^2$$

Style Representation/Reconstruction

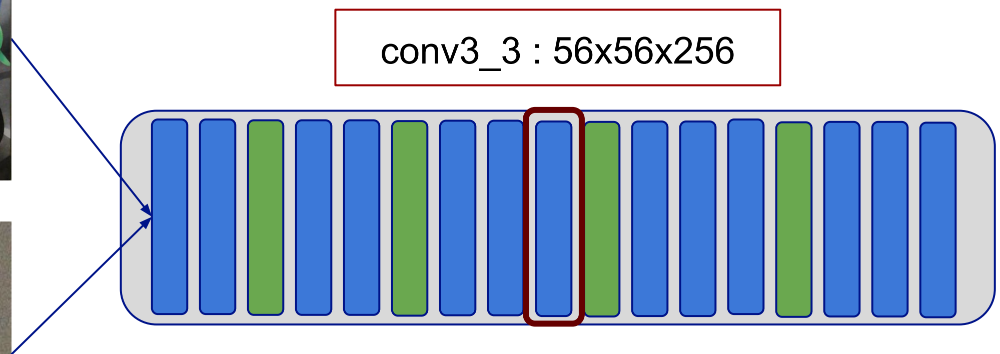
- Needs more complex statistics on feature maps : **Gram matrix**
  - Second-order statistics
  - Can capture texture information, no spatial information
- For a given layer j with C_j feature maps of size (W_j, H_j)
- The Gram matrix is a (C_j, C_j) matrix :

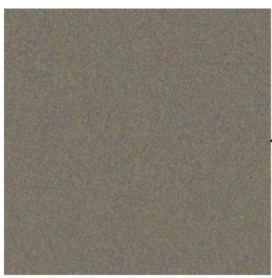
$$G_j(x)_{c_1,c_2} = \mathbb{E}[\ \Phi_j(x)[c_1] * \Phi_j(x)[c_2]\ ]$$

- Where $*$ is an element-wise operation between 2 feature maps (Hadamard product)
- Contains the correlation between every pair of feature maps

Style Representation/Reconstruction

x_s $\qquad G_j(x)$ Gram matrix of the jth layer (256 x 256)





x

$$\hat{x} = \underset{x}{\operatorname{argmin}} \; ||G_j(x) - G_j(x_s)||_2^2$$

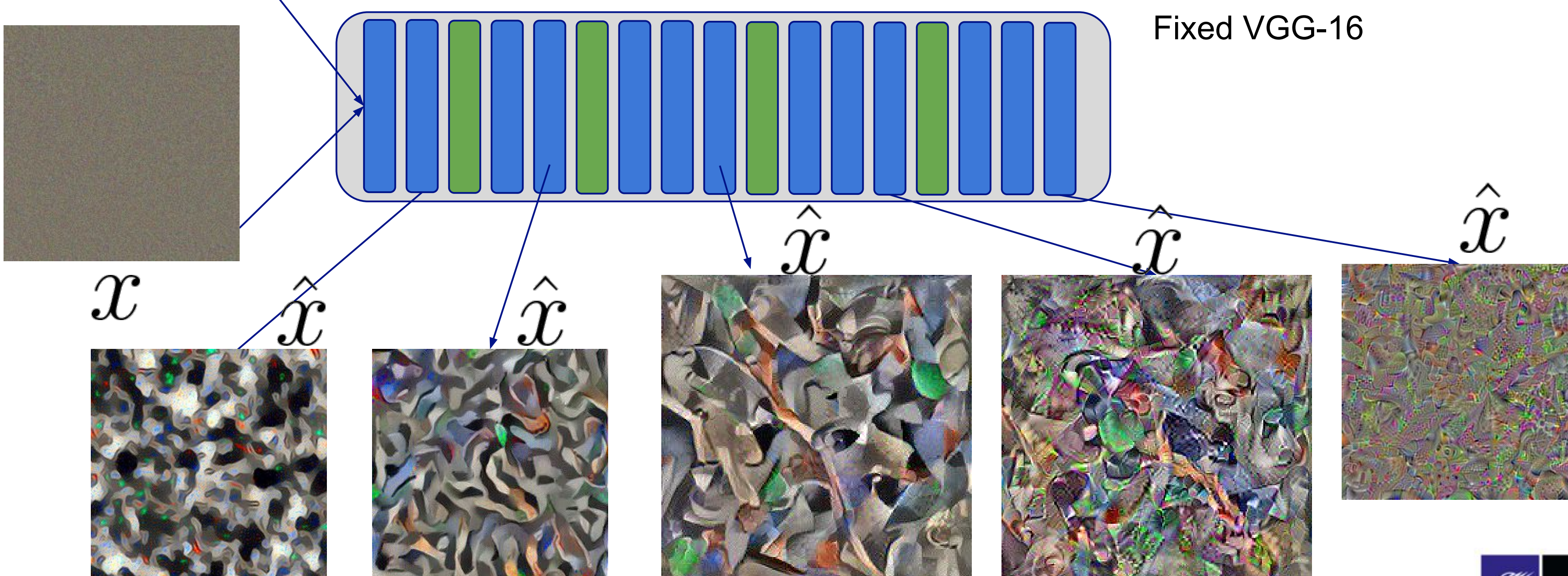
- Goal : To find an image with the same Gram matrix for a given layer
- Optimization problem: Start from a random image

Style Representation/Reconstruction

- Gradient descent optimization on image input, network is freezed
- Loss = MSE on feature maps, 1000 iterations, Adam (lr=2.0)
- Low-level : Small and simple patterns
- High-level : More complex patterns

x_S

$$L_S(x, x_S) = \sum_j \frac{\lambda_j}{C_j^2} ||G_j(x_S) - G_j(x)||_2^2$$



Content & Style Representations

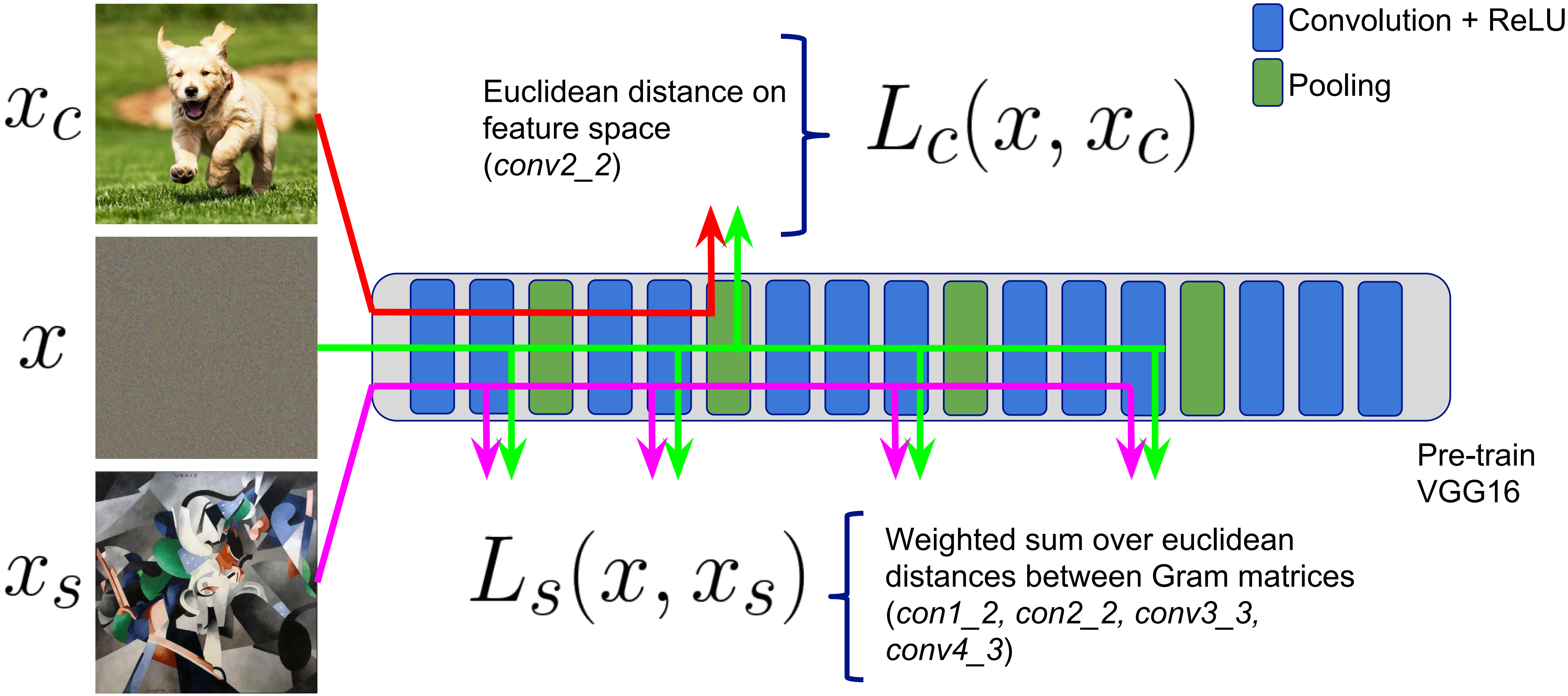
- Content is preserved in high level features
- Style is present in second-order statistics in low and medium levels
- Content and Style are separable
- *content_loss* and a *style_loss* are defined
- Combine style and loss from different images is possible, via feature extraction learned within a VGG network, trained for image classification

Optimization-based approach for style transfer

Approach proposed by

- Gatys et al [4, 10]
- Ruder et al [5]

Mix content & style via specific losses



$$L(x, x_s, x_c) = \alpha L_s(x, x_s) + \beta L_c(x, x_c)$$

Optimization process

- Compute *content_target* (feature maps) with *content_image*
- Compute *style_target* (Gram matrices) with *style_image*
- Start from a random image (*input_image*)
- Optimization process :
  - Compute *content_loss* and *style_loss* with targets + *input_image*
  - Minimize this loss by modifying *input_image*
  - Possible thanks to gradient-descent method (like Adam)

TensorFlow Implementation

- TensorFlow implementation (version 1.1.0, Python 3.5)
- With TensorBoard annotations (Graph and metrics visualization)
- Jupyter notebooks and Conda/Docker envs
- GitHub: JGuillaumin/style-transfer-workshop

```
sess = tf.InteractiveSession()

vgg = VGG.generate_model(weights_file=MODEL_WEIGHTS,
                         input_shape=(1, IMAGE_HEIGHT, IMAGE_WIDTH, COLOR_CHANNELS),
                         remove_top=True,
                         with_preprocessing=False)
# vgg is a Python dictionary
# vgg['input'] : tf.Variable(...)
# vgg['...'] : tensor after each block (conv or max_pooling)

content_image = load_image(CONTENT_IMAGE)
style_image = load_image(STYLE_IMAGE)
noise_image = generate_noise_image(IMAGE_HEIGHT, IMAGE_WIDTH)
```

TensorFlow implementation

```
# Content loss
with tf.name_scope('content_image'):
    # Construct content_loss using content_image.
    sess.run(vgg['input'].assign(content_image))

with tf.name_scope('content_loss'):
    content_target = sess.run(vgg['conv4_2'])
    N = content_target.shape[3] # number of feature maps
    M = content_target.shape[1] * content_target.shape[2] # number of feature per feature map
    content_loss = (1 / (4 * N * M)) * tf.reduce_sum(tf.pow(vgg['conv4_2'] - content_target, 2))
```

```
# Style loss
with tf.name_scope('style_image'):
    # Construct style_loss using style_image
    sess.run(vgg['input'].assign(style_image))

STYLE_LAYERS = [('conv1_1', 0.5), ('conv2_1', 1.0), ('conv3_1', 2.5),
                ('conv4_1', 3.0), ('conv5_1', 1.0)]
```

TensorFlow implementation

```
def _gram_matrix_tf(F, N, M):
    F = tf.reshape(F, (M, N))
    return tf.matmul(tf.transpose(F), F)

def _gram_matrix_np(F, N, M):
    F = np.reshape(F, (M, N))
    return np.matmul(np.transpose(F), F)

with tf.name_scope('style_loss'):
    style_loss = 0

    for layer_name, weight in STYLE_LAYERS :
        style_target = sess.run(vgg[layer_name])
        N = style_target.shape[3] # number of feature maps
        M = style_target.shape[1] * style_target.shape[2] # number of features per feature map

        # compute Gram matrices : target and tensor
        style_target = _gram_matrix_np(style_target, N, M) # works on Numpy array
        G = _gram_matrix_tf(vgg[layer_name], N, M) # works on Tensor

        style_loss += weight * (1 / (4 * N**2 * M**2)) * tf.reduce_sum(tf.pow(G - style_target, 2))
```

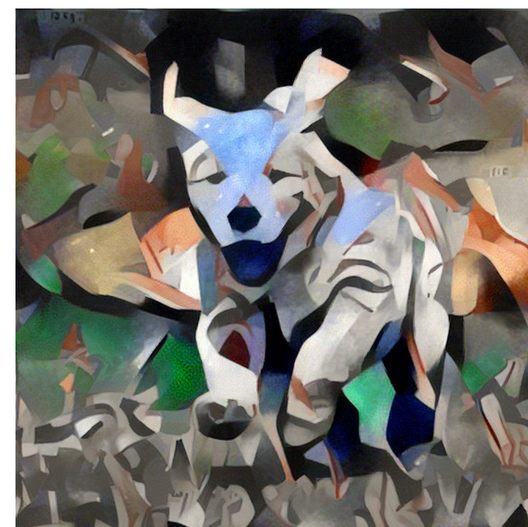
TensorFlow implementation

```
with tf.name_scope('total_loss'):
    total_loss = BETA * content_loss + ALPHA * style_loss

with tf.name_scope('train'):
    optimizer = tf.train.AdamOptimizer(2.0)
    train_step = optimizer.minimize(total_loss)

_ = sess.run(vgg['input'].assign(noise_image))
```

TensorFlow implementation

```python
ITERATIONS = 1000

for it in range(ITERATIONS):
    _ = sess.run(train_step)

    if it%100 == 0:
        _image = sess.run(vgg['input'])
        filename = 'output/stylized_gatys_iter{}.png'.format(it)
        save_image(filename, _image)
```



it = [100, 400, 800, 3000]

Results

- **Produce high-quality images**
- **Easy to tune effects (more content ? more style ?)**
- **Any input/output size**

- **Running time (1000 #iter)**
  - GPU (GTX 1070) : ~ 5 min (1920 CUDA cores)
  - CPU (i7-7700K) : ~ 150 min (4 cores x 2 threads)

- **Avoid any real-time applications**
- **But perceptual loss (content+style) is defined**

Improvements

- Time dependency for video transformation (see [5])
- Change optimizer : L-BFGS !
- Tune weights between style and content loss
- Start from : content image, style image, noisy image, or a mix.
- Color constraint : preserve color from content image ! (see [10])

from : github.com/tensorflow/magenta

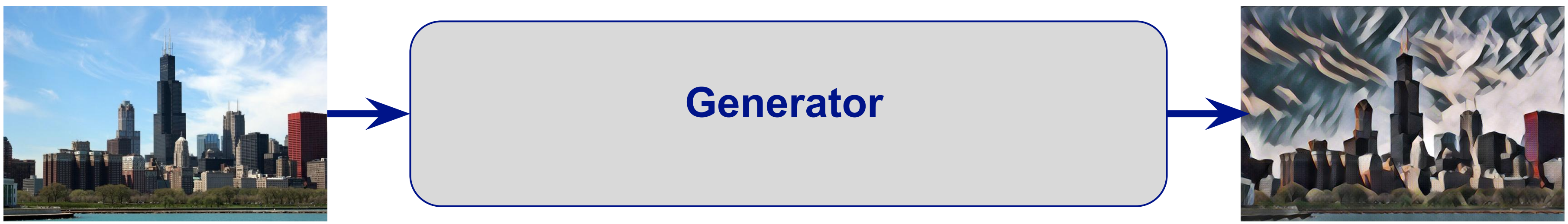
Feed-forward method with learning

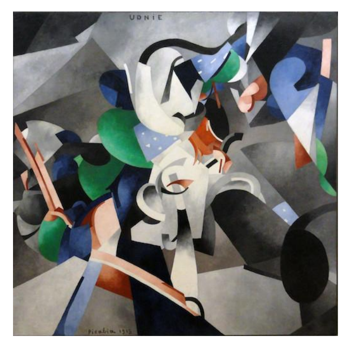
Approach proposed by

- Ulyanov et al [6, 7]
- Johnson et al [8]

Feed-forward method

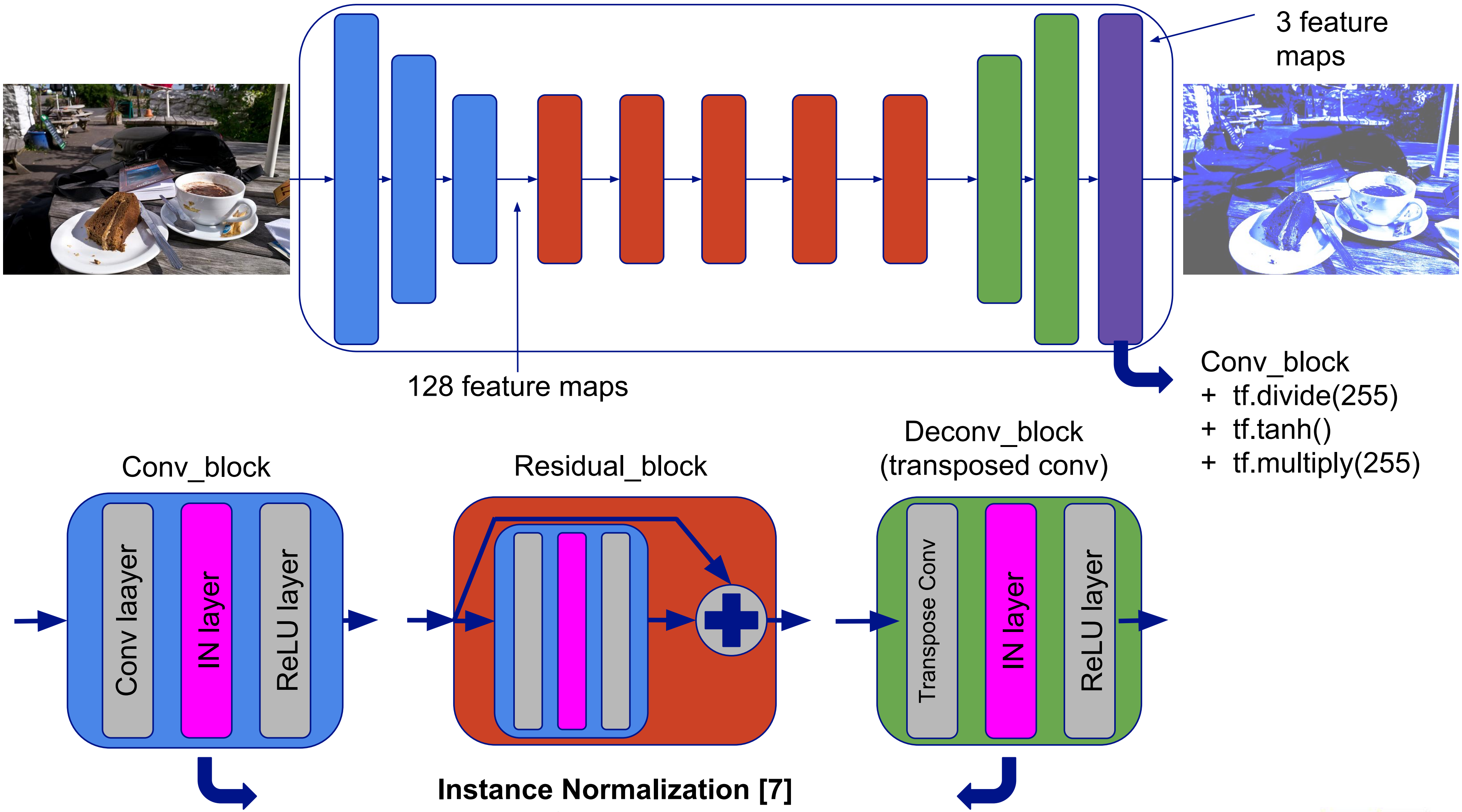
- Train a network to obtain a stylized image in one pass as an output
- Used for one specific style (fixed)





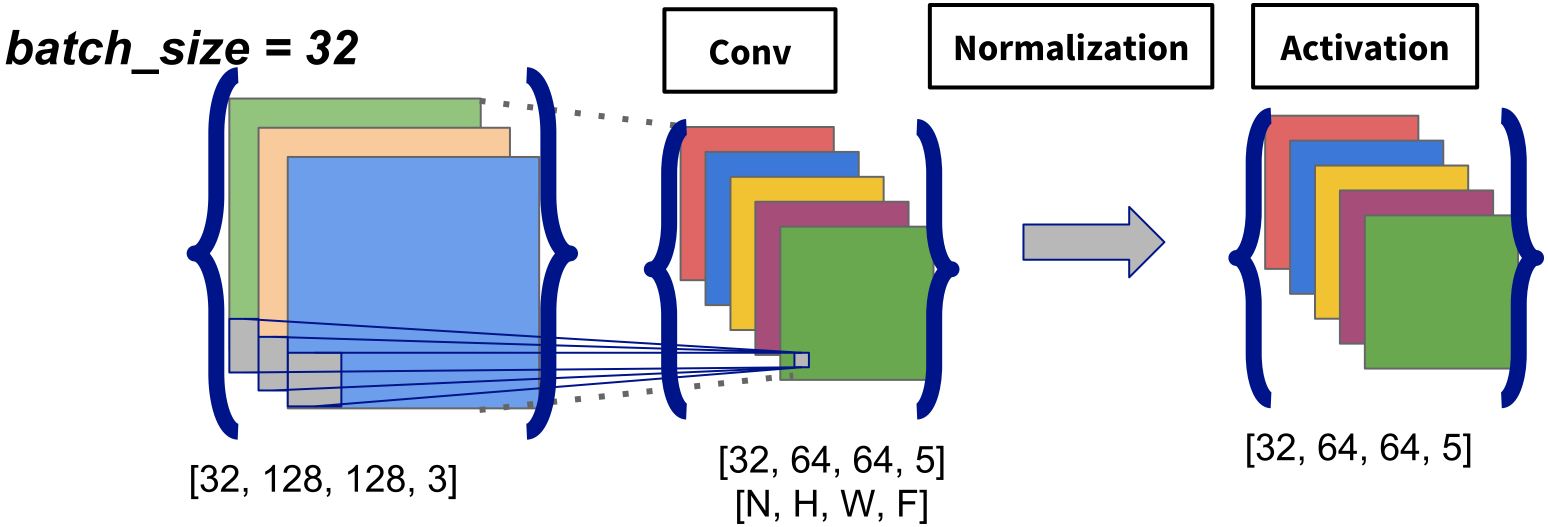
- Trained to add this style
- With a dataset of content images
- Same input/output size

What type of structures for the generator ?



Instance Normalization [7]
Variant of Batch Normalization

Batch Normalization vs. Instance Normalization



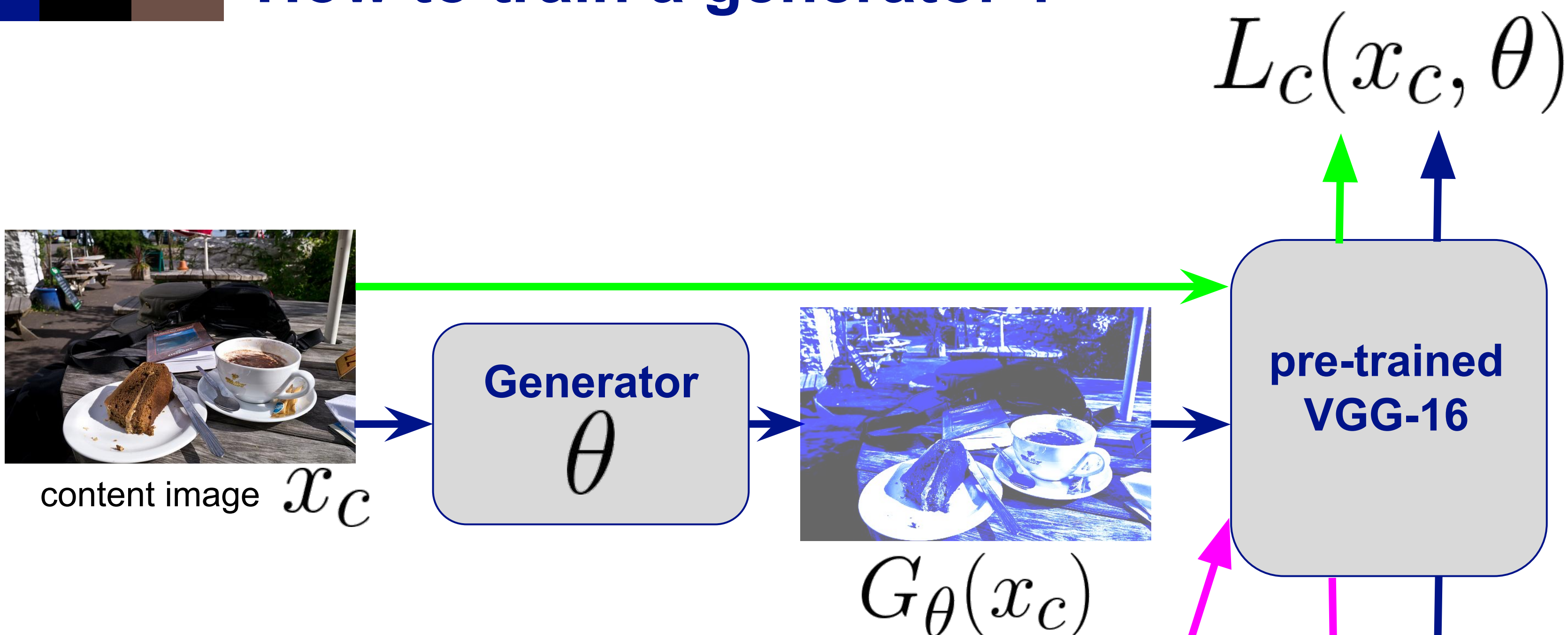
$$BN_f(x) = \gamma \frac{x - \mu_f(x)}{\sigma_f(x)} + \beta$$

channel-wise

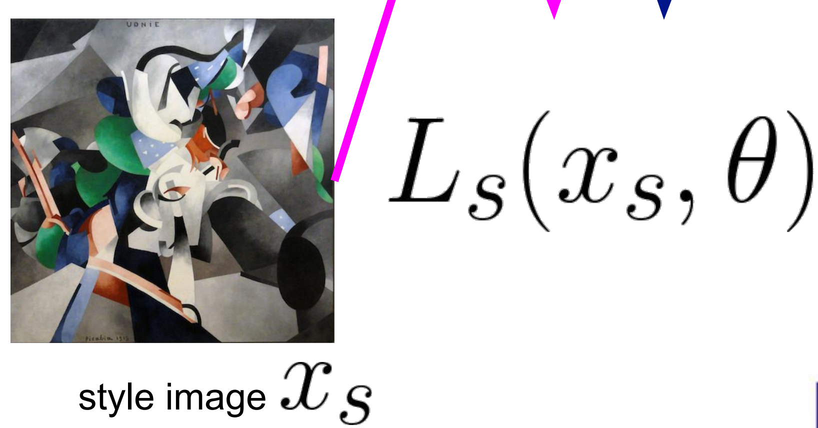
$$IN_{n,f}(x) = \gamma \frac{x - \mu_{n,f}(x)}{\sigma_{n,f}(x)} + \beta$$

(sample,channel)-wise

How to train a generator ?



- Train with batch of content images
- Minimize total loss w.r.t. theta



INSTITUT Mines-Télécom

Need a dataset of content images

- COCO dataset, about 80k images
- Only 1 style image

Training process (loop) :
- Take a batch of samples from COCO
- Pass this batch through the generator to get generated images
- Compute *style_loss* between the generated images and the style image
- Compute *content_loss* between the generated images and the original ones
- Minimize the *total_loss* by updating the weights from the generator

Training information :
- Adam optimizer (*lr=0.05*)
- Only 20k iterations (with *batch_size=4*)
- For 512x512x3:
  - Training time (on GTX 1070) : 10 hours
  - Inference time : 330 ms (GTX 1070)

TensorFlow implementation

```
sess = tf.InteractiveSession()

generator={}

with tf.name_scope('input_images'):
    input_images = tf.placeholder(tf.float32, shape=(BATCH_SIZE, IMAGE_HEIGHT, IMAGE_WIDTH, 3),
                                  name='input_images')

with tf.name_scope('generator'):
    generator['conv_block1'] =  _conv_block(input_images, 9, 32, 1, 'conv_block1')
    generator['conv_block2'] =  _conv_block(generator['conv_block1'], 3, 64, 2, 'conv_block2')
    generator['conv_block3'] =  _conv_block(generator['conv_block2'], 3, 128, 2, 'conv_block3')
    generator['residual_block1'] = _residual_block(generator['conv_block3'], 'residual_block1')
    generator['residual_block2'] = _residual_block(generator['residual_block1'], 'residual_block2')
    generator['residual_block3'] = _residual_block(generator['residual_block2'], 'residual_block3')
    generator['residual_block4'] = _residual_block(generator['residual_block3'], 'residual_block4')
    generator['residual_block5'] = _residual_block(generator['residual_block4'], 'residual_block5')
    generator['deconv_block1'] = _deconv_block(generator['residual_block5'], 3, 64, 2, 'deconv_block1')
    generator['deconv_block2'] = _deconv_block(generator['deconv_block1'], 3, 32, 2, 'deconv_block2')
    generator['final_conv'] = _conv_block(generator['deconv_block2'], 9, 3, 1, 'final_conv', relu=False)
    generator['output'] = tf.multiply(tf.tanh(generator['final_conv']/255.0), 255, name="output")
```

TensorFlow implementation

```
with tf.variables_scope('VGGs', reuse=True):
    vgg = VGG.generate_model(weights_file=MODEL_WEIGHTS,
                             input=generator['output'],
                             remove_top=True,
                             with_preprocessing=False)

    vgg_content = VGG.generate_model(weights_file=MODEL_WEIGHTS,
                                     input=input_images,
                                     remove_top=True,
                                     with_preprocessing=False)

# variable sharing between 'vgg' and 'vgg_content'


# Content loss
with tf.name_scope('content_loss'):
    content_loss = tf.reduce_mean(tf.pow(vgg_content['conv4_2'] - vgg['conv4_2'], 2))
```

TensorFlow implementation

```python
STYLE_LAYERS = [('conv1_1', 0.5), ('conv2_1', 1.0), ('conv3_1', 2.5),
                ('conv4_1', 3.0), ('conv5_1', 1.0)]

def _gram_matrix_np(F, B, N, M):
    F = np.reshape(F, (B, M, N))
    return (1 / M) * np.matmul(np.transpose(F, (0, 2, 1)), F)

def _gram_matrix_tf(F, B, N, M):
    F = tf.reshape(F, (B, M, N))
    return (1 / M) * tf.matmul(tf.transpose(F, perm=[0,2,1]), F)

style_image = load_image(STYLE_IMAGE)
style_images = np.asarray([style_image[0]]*BATCH_SIZE)
feed_dict = {model['input']: style_images}
```

TensorFlow implementation

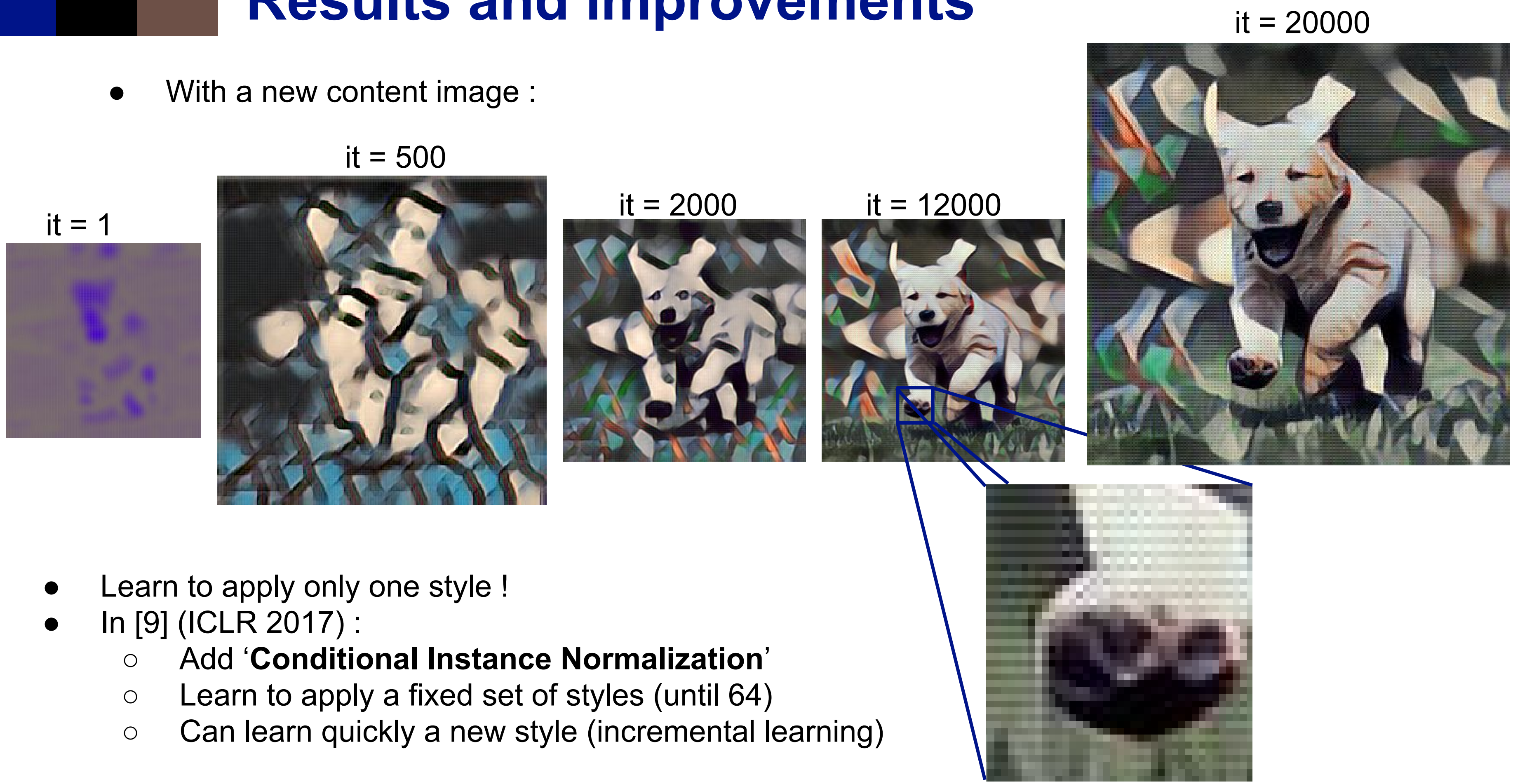
```
with tf.name_scope('style_loss'):
    style_loss = 0

    for layer_name, weight in STYLE_LAYERS :

        style_target = sess.run(vgg[layer_name], feed_dict=feed_dict)
        B = style_target.shape[0] # batch_size
        N = style_target.shape[3] # number of feature maps
        M = style_target.shape[1] * style_target.shape[2] # number of features per feature map

        # compute Gram matrices : target and tensor
        style_target = _gram_matrix_np(style_target, B, N, M) # works on Numpy array
        G = _gram_matrix_tf(vgg[layer_name],B , N, M) # works on Tensor

        style_loss += weight * tf.reduce_mean(tf.pow(G - style_target, 2))

with tf.name_scope('total_loss'):
    total_loss = BETA * content_loss + ALPHA * style_loss
```

TensorFlow implementation

```
sess.run(tf.global_variables_initializer())
ITERATIONS = 20000
feed={}

for it in range(ITERATIONS):
    batch = COCO_batch_generator.next()
    feed[input_images] = batch

    _ = sess.run([train_step], feed_dict=feed)

    if it%500 == 0:
        _image = sess.run(generator['output'], feed_dict={input_images:[content_image[0]]*BATCH_SIZE})
        save_image(filename, _image)
```

Results and improvements

- With a new content image :



- Learn to apply only one style !
- In [9] (ICLR 2017) :
  - Add ‘**Conditional Instance Normalization**’
  - Learn to apply a fixed set of styles (until 64)
  - Can learn quickly a new style (incremental learning)

- Use **resized convolutions** instead of transposed convolutions : Improves quality
- Add **variational loss** to encourage spatial smoothness

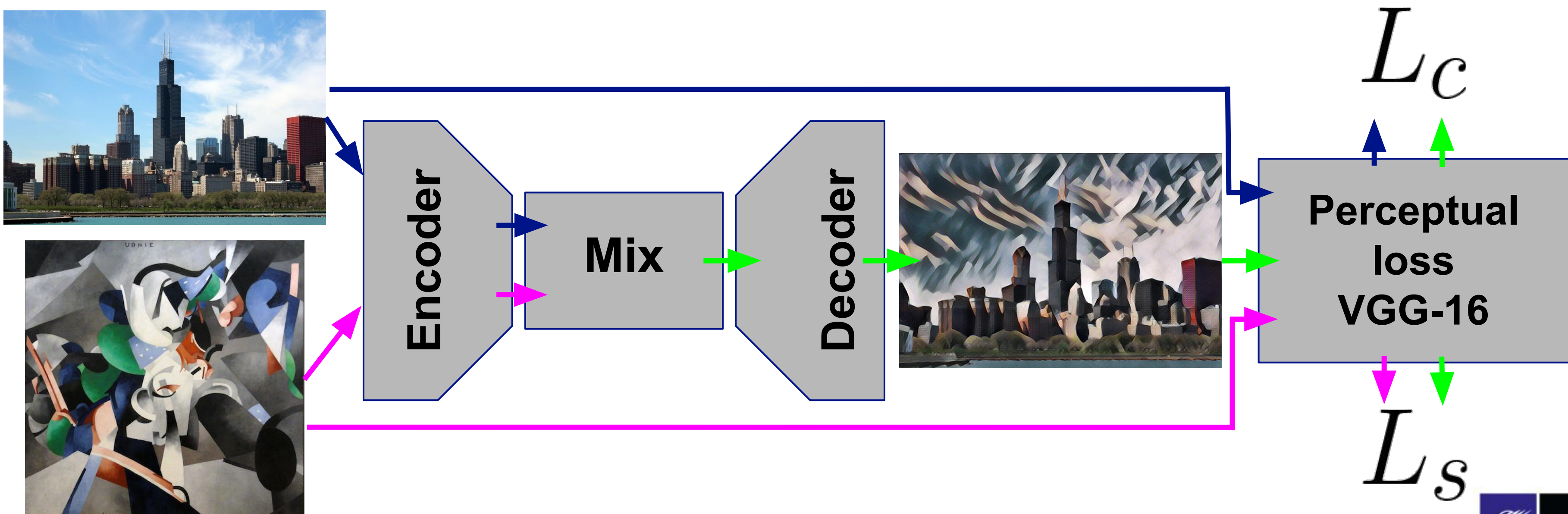
Arbitrary style transfer

Approach proposed by :

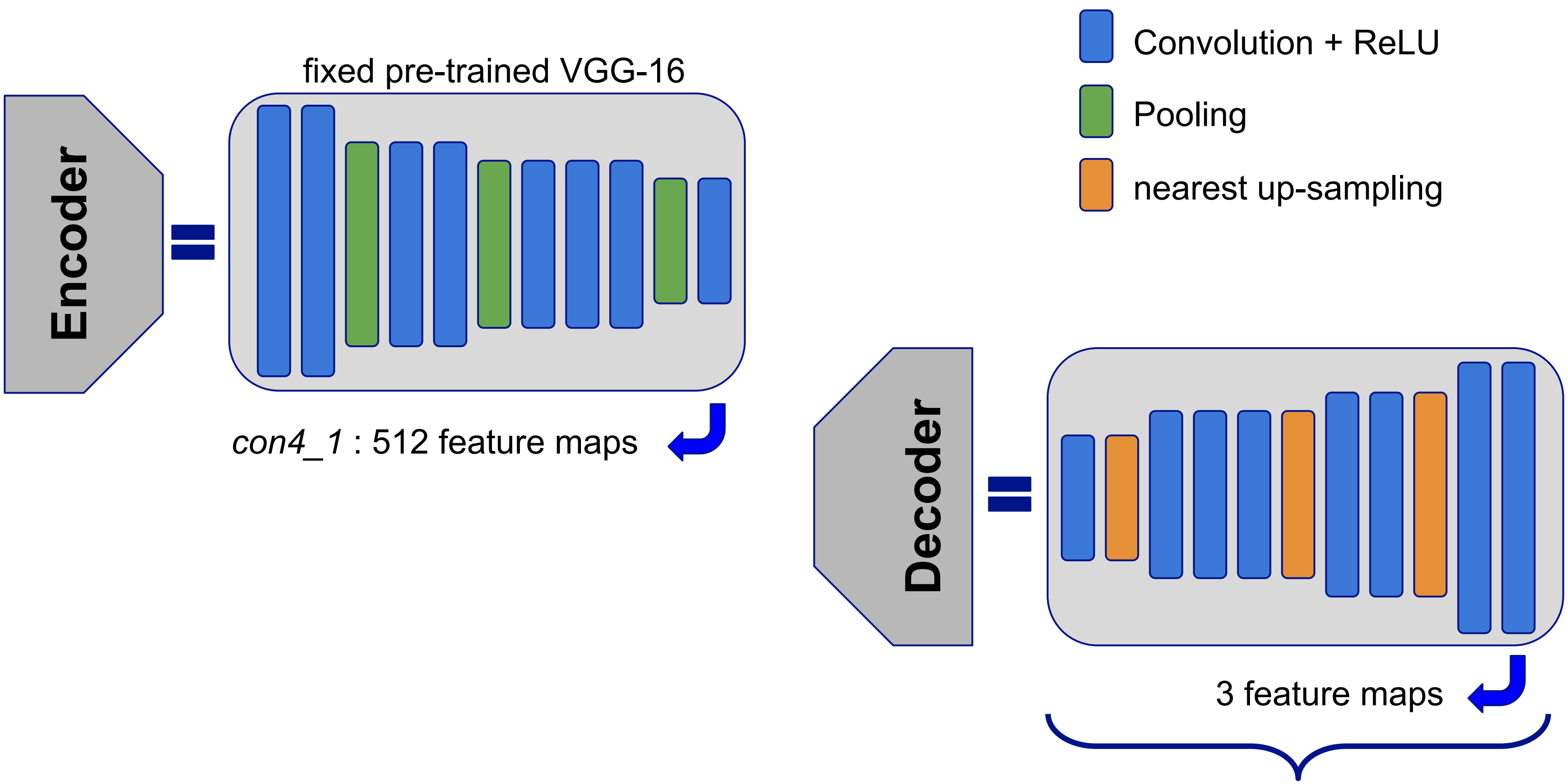
- X. Huang, S. Belongie [11] (ICLR 2017)

Mix content & style images within the generator

- **Goal** : train a generator to produce stylized images **from any style** with any content
- **Previous approach** : style is learned in the generator via the *style_loss*
- **New approach : mix content and style images in feature space !**
  - Use **encoder-decoder** structure
  - Mix encoded content and style images
  - Use the **same perceptual loss** (*content_loss* + *style_loss*)
  - Trained on a **content dataset** (COCO) and a **painting dataset** (WikiArt)

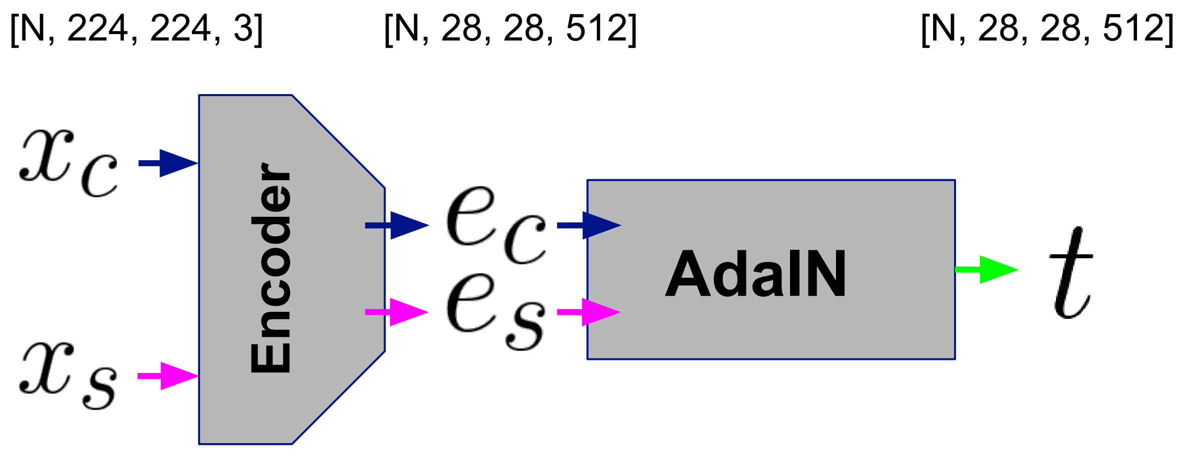


Encoder/Decoder + Adaptive IN



- the decoder mirrors the encoder
- to train !

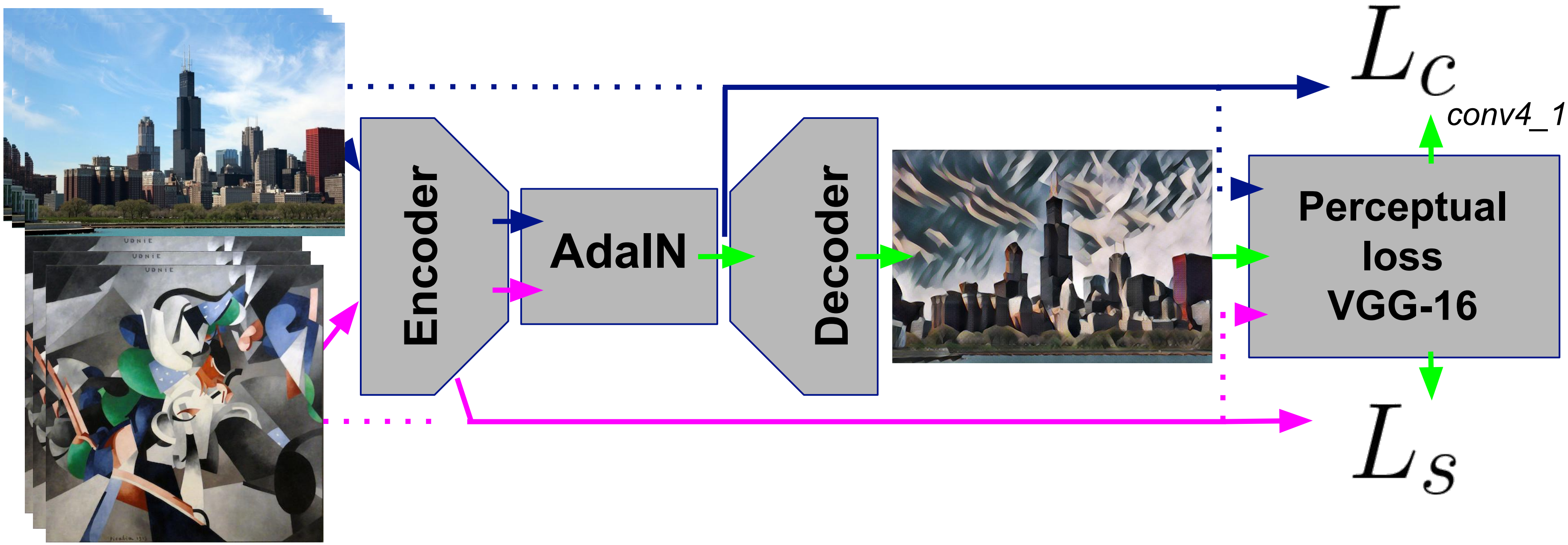
Adaptive Instance Normalization



$$AdaIN_{n,f}(e_c, e_s) = \sigma_{n,f}(e_s)\frac{e_c - \mu_{n,f}(e_c)}{\sigma_{n,f}(e_c)} + \mu_{n,f}(e_s)$$

- per sample (n) and per channel (f) statistics alignment
- producing the target feature maps

How to train the decoder ?



- trained with batches of content-styles image pairs
- different contents and styles within the same batch !
- 80k content images (MS COCO) + 80k paintings (WikiArt.org)

TensorFlow Implementation

```
sess = tf.InteractiveSession()

generator={}

with tf.name_scope('input_images'):
    input_content = tf.placeholder(tf.float32, shape=(BATCH_SIZE, IMAGE_HEIGHT, IMAGE_WIDTH, 3),
                                  name='input_content')
    input_style= tf.placeholder(tf.float32, shape=(BATCH_SIZE, IMAGE_HEIGHT, IMAGE_WIDTH, 3),
                                  name='input_style')

with tf.variables_scope('VGGs', reuse=True):
    content_encoder = VGG.generate_model(weights_file=MODEL_WEIGHTS,
                             input=input_content,
                             remove_top=True,
                             with_preprocessing=False)

    style_encoder = VGG.generate_model(weights_file=MODEL_WEIGHTS,
                                 input=input_style,
                                 remove_top=True,
                                 with_preprocessing=False)


encoded_content = content_encoder['conv4_1']
encoded_style = style_encoder['conv4_1']
```

TensorFlow Implementation

```
with tf.name_scope('AdaIn') :
    eps = 1e-6
    mean_c, var_c = tf.nn.moments(encoded_content, [1, 2], keep_dims=True)
    mean_s, var_s = tf.nn.moments(encoded_style, [1, 2], keep_dims=True)

    target = mean_s * ((encoded_content - mean_c)/(tf.sqrt(var_c) + eps)) + mean_s


with tf.name_scope('decoder'):
    decoder = {}
    decoder['conv_block1_1'] = _conv_block(taget, 3, 256 , 1, 'conv_block1_1')
    decoder['up_sampling1'] = _up_sampling(decoder['conv_block1_1'], 2, 'up_sampling1')

    decoder['conv_block2_1'] = _conv_block(decoder['up_sampling1'], 3, 256 , 1, 'conv_block2_1')
    decoder['conv_block2_2'] = _conv_block(decoder['conv_block2_1'], 3, 256 , 1, 'conv_block2_2')
    decoder['conv_block2_3'] = _conv_block(decoder['conv_block2_2'], 3, 128 , 1, 'conv_block2_3')
    decoder['up_sampling2'] = _up_sampling(decoder['conv_block2_3'], 2, 'up_sampling2')

    decoder['conv_block3_1'] = _conv_block(decoder['up_sampling2'], 3, 128 , 1, 'conv_block3_1')
    decoder['conv_block3_2'] = _conv_block(decoder['conv_block3_1'], 3, 64 , 1, 'conv_block3_2')
    decoder['up_sampling3'] = _up_sampling(decoder['conv_block3_2'], 2, 'up_sampling3')

    decoder['conv_block4_1'] = _conv_block(decoder['up_sampling3'], 3, 64 , 1, 'conv_block3_1')
    decoder['conv_block4_2'] = _conv_block(decoder['conv_block4_1'], 3, 64 , 1, 'conv_block4_2')

    decoder['final_conv'] = _conv_block(decoder['conv_block4_1'], 9, 3, 1, 'final_conv', relu=False)
    decoder['output'] = tf.multiply(tf.tanh(decoder['final_conv']/255.0), 255, name="output")
```

TensorFlow Implementation

```
with tf.variables_scope('VGGs', reuse=True):
    vgg = VGG.generate_model(weights_file=MODEL_WEIGHTS,
                             input=decoder['output'],
                             remove_top=True,
                             with_preprocessing=False)

with tf.name_scope('content_loss'):
    content_loss = tf.reduce_mean(tf.pow(vgg['conv4_1'] - target, 2))

with tf.name_scope('style_loss'):
    style_loss = 0

    for layer_name, weight in STYLE_LAYERS :

        shape = vgg[layer_name].get_shape().as_list()
        B = style_target.shape[0] # batch_size
        N = style_target.shape[3] # number of feature maps
        M = style_target.shape[1] * style_target.shape[2] # number of features per feature map

        G_style = _gram_matrix_tf(style_encoder[layer_name], B, N, M) # works on Numpy array
        G = _gram_matrix_tf(vgg[layer_name],B , N, M) # works on Tensor

        style_loss += weight * tf.reduce_mean(tf.pow(G - G_style, 2))
```

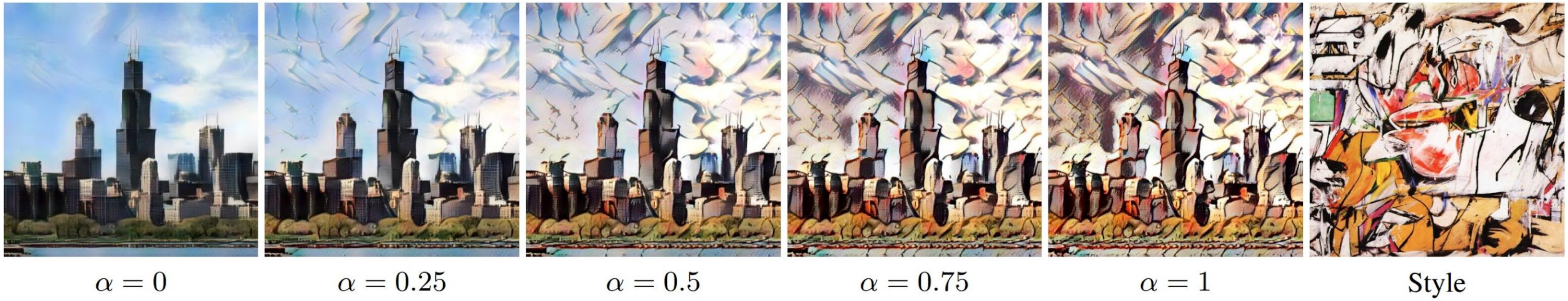
TensorFlow Implementation

```
with tf.name_scope('total_loss'):
    total_loss = BETA * content_loss + ALPHA * style_loss

with tf.name_scope('train'):
    optimizer = tf.train.AdamOptimizer(0.02)
    train_step = optimizer.minimize(total_loss)

sess.run(tf.global_variables_initializer())

ITERATIONS = 20000
feed={}

for it in range(ITERATIONS):
    batch_c = COCO_batch_generator.next()
    batch_s = WikiArt_batch_generator.next()
    feed[input_content] = batch_c
    feed[input_style] = batch_s

    _ = sess.run([train_step], feed_dict=feed)

    if it%500 == 0:
        _image = sess.run(generator['output'], feed_dict={input_content:[content_image[0]]*BATCH_SIZE,
                                                          input_style:[style_image[0]]*BATCH_SIZE})
        save_image(filename, _image)
```

Results

- Training time : about 24 hours (GTX 1070, 200k iterations)

- Decoder-AdaIN-decoder can apply any new style !
- High-quality images (similar to optimization-based approach)
- Inference time : ~ 450ms (256x256)



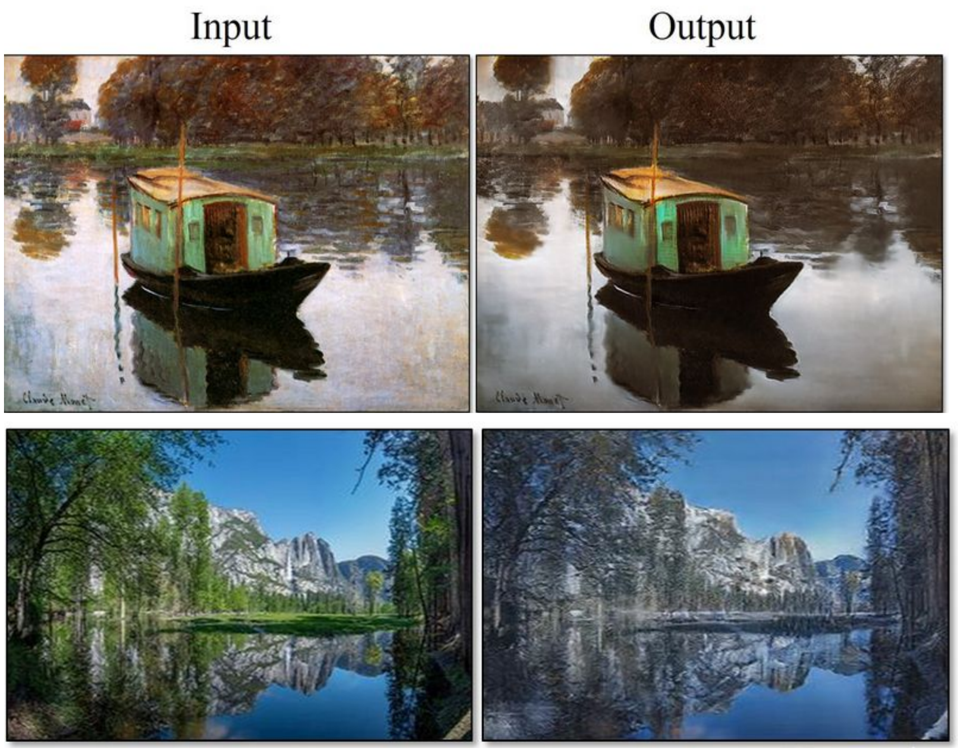
from github.com/xunhuang1995/AdaIN-style

Conclusion - Style Transfer

	Content	Style	Production	Training
Gatys et al (2015)	1	1	~ 5 min	X
Johnson Ulyanov (2016)	infinite	fixed number, until 64	~ 300ms	~ 4 hours
X Huang (2017)	infinite	infinite	~ 450 ms	~ days

Conclusion

- Resolve complex task by working on feature spaces
- Introduction to more complex tasks
  - Colorisation
  - Super-resolution
  - Inverse style transfer
  - Season transfer
  - Color transfer



from github.com/junyanz/CycleGAN

Online resources

github.com/JGuillaumin/style-transfer-workshop

- Several Jupyter notebooks
- All methods presented here and more
- Implementation with TensorFlow 1.1, Python 3.5
- With TensorBoard annotations
- Conda env and Dockerfiles (CPU and GPU)

Online in few days

Thank you

?

Resources

[1] : K. Simonyan, A. Zisserman : "Very Deep Convolutional Networks for Large-Scale Image Recognition", 2014, arXiv:1409.1556

[2] : About Deep Dream visualization technique : "Inceptionism: Going Deeper into Neural Networks"

[3] : M. Zeiler, R. Fergus: Visualizing and Understanding Convolutional Networks, 2013 arXiv:1311.2901

[4] : L. Gatys, A. Ecker, M. Bethge : A neural algorithm of artistic style, 2015, arXiv:1508.06576

[5] : M. Ruder, A. Dosovitskiy, T. Brox : Artistic style transfer for video, 2016, arXiv:1604.08610

[6] : D. Ulyanov et al : Texture Networks: Feed-forward Synthesis of Textures and Stylized Images, 2016, arXiv:1603.03417

[7] : D. Ulyanov et al : Instance Normalization: The Missing Ingredient for Fast Stylization, 2016, arXiv:1607.08022

[8] : J. Johnson et al : Perceptual losses for real-time style transfer and super-resolution, 2016, arXiv:1603.08155

[9] : V. Dumoulin et al : A learned representation for artistic style, 2017, arXiv:1610.07629

[10] : Gatys et al : Preserving color in Neural Artistic Style Transfer, 2016, arXiv:1606.05897

[11] X. Huang and S. Belongie : Arbitrary Style Transfer in real-time with AdaIN, 2017, arXiv:1703.06868

INSTITUT Mines-Télécom