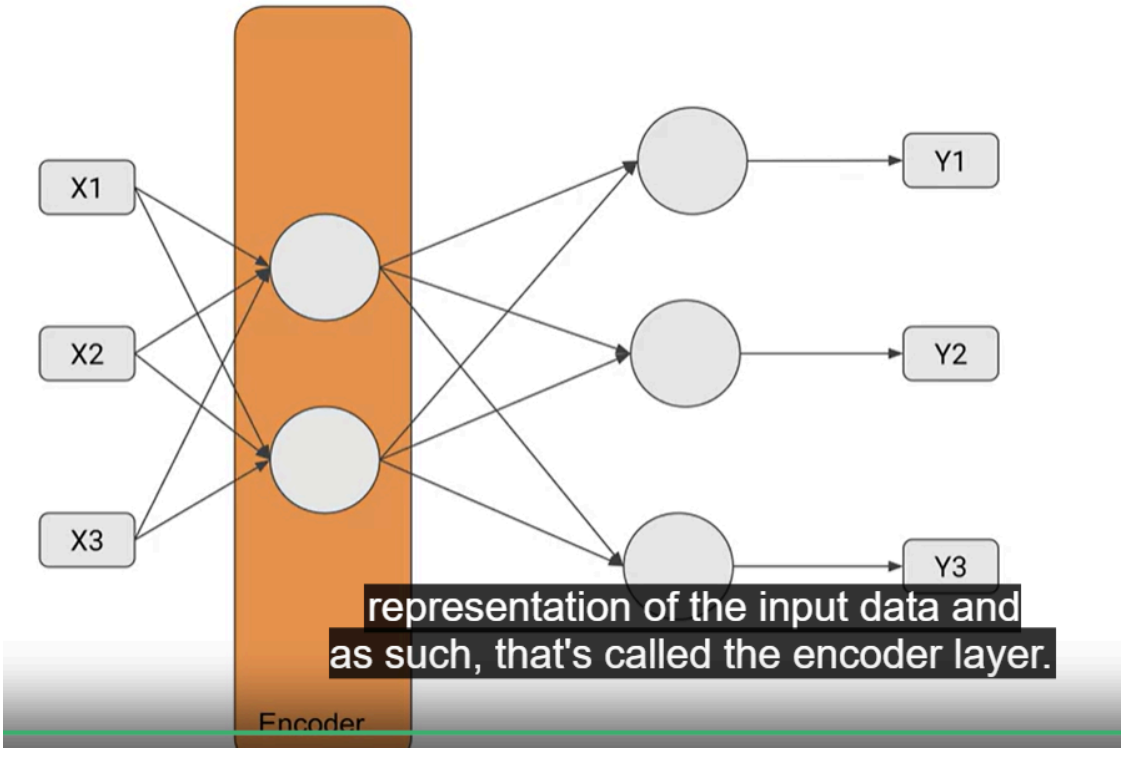
Week 2 - AutoEncoders

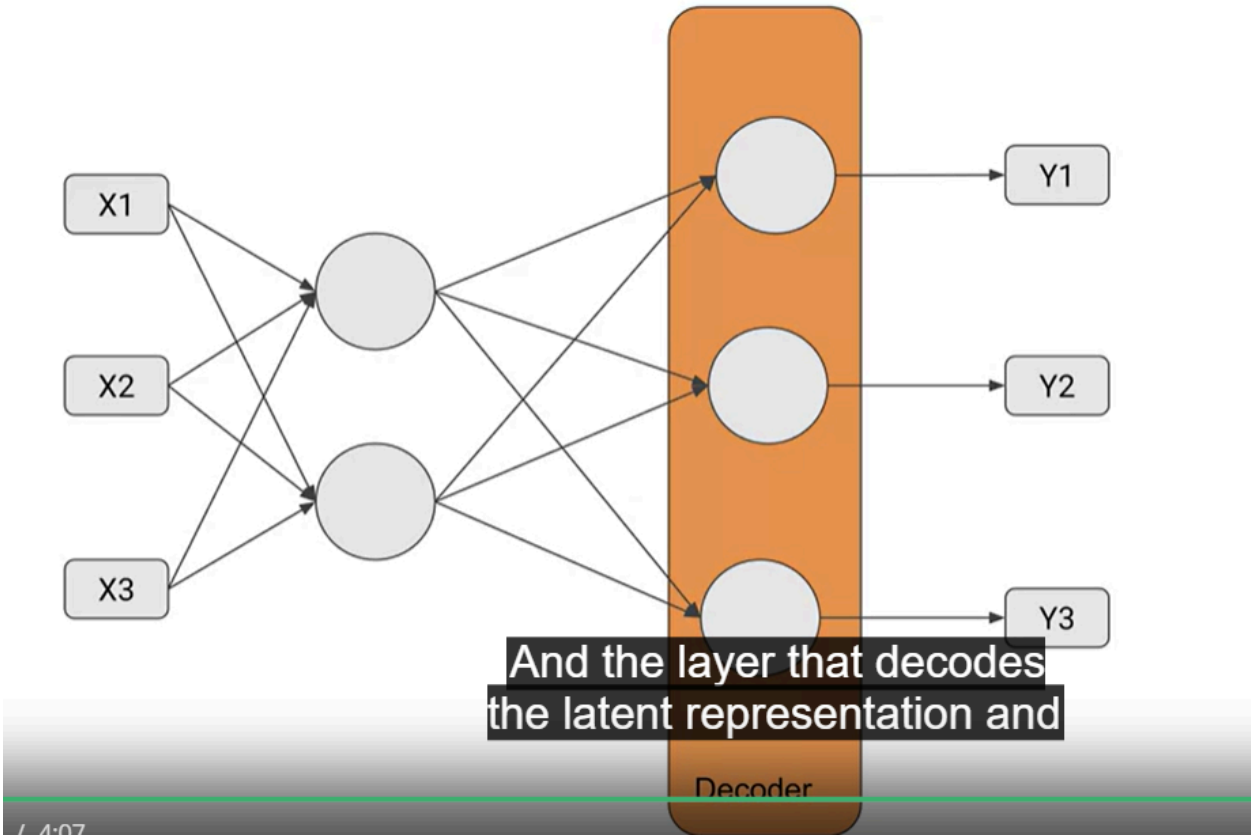
1. NN capable of learning dense representations of input data without supervision
    - Training data is not labeled
2. Useful for dimensionality reduction and for visualization
3. Can be used to generate new data that resembles input data
4. In practice they,
    a. Copy input to output
    b. Learn efficient ways to represent data

What are AutoEncoders?

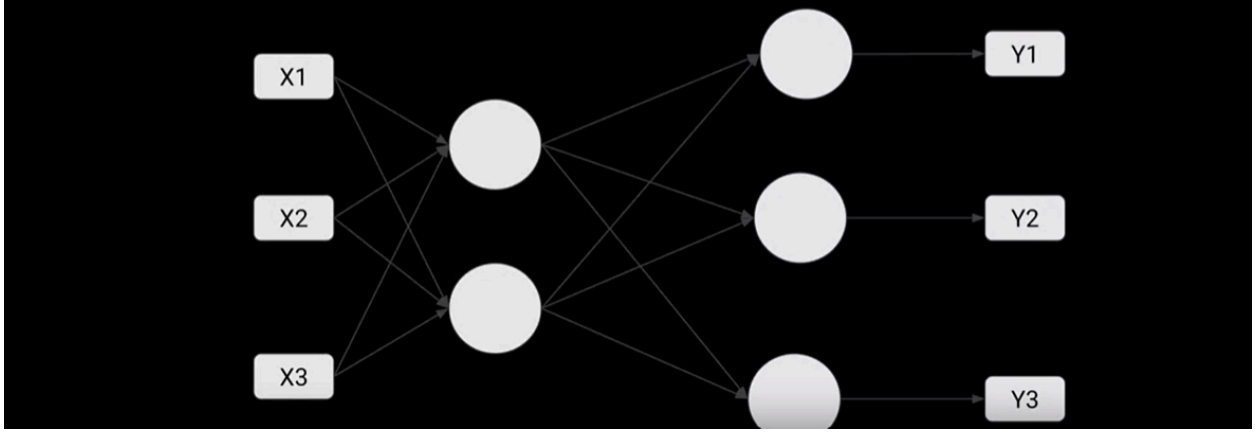
- Neural networks capable of learning dense representations of input data without supervision
    - Training data is not labelled
- Useful for dimensionality reduction and for visualization
- Can be used to generate new data that resembles input data
- In practice they
    - Copy input to output
    - They learn efficient ways to represent data

- **The goal of an autoencoder is to create a latent representation**
- **An autoencoder is a special type of neural network that is trained to copy its input to its output. For example, given an image of a handwritten digit, an autoencoder first encodes the image into a lower dimensional latent representation, then decodes the latent representation back to an image. An autoencoder learns to compress the data while minimizing the reconstruction error.**






```
encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])
autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1.5))
```



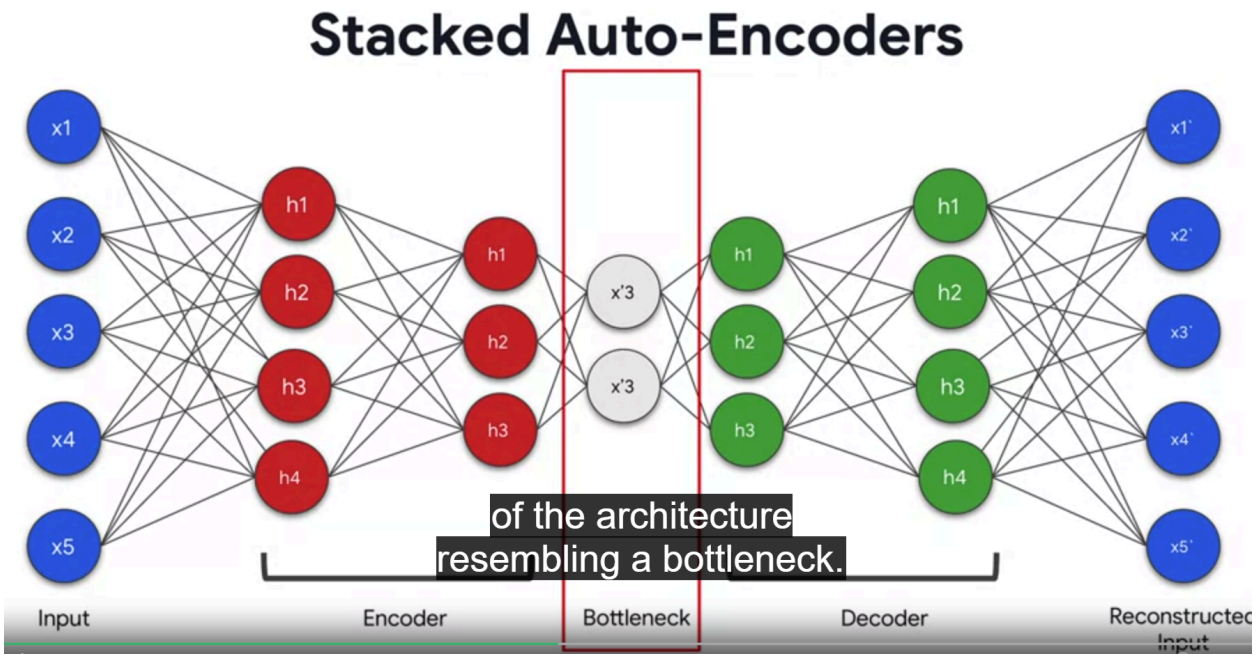
```
history = autoencoder.fit(X_train, X_train, epochs=200)
```

```
codings = encoder.predict(data)
```

```
inputs =  tf.keras.layers.Input(shape=(784,))

def simple_autoencoder():
    encoder = tf.keras.layers.Dense(units=32, activation='relu')(inputs)
    decoder = tf.keras.layers.Dense(units=784, activation='sigmoid')(encoder)
    return encoder, decoder

encoder_output, decoder_output = simple_autoencoder()

encoder_model = tf.keras.Model(inputs=inputs, outputs=encoder_output)

autoencoder_model = tf.keras.Model(inputs=inputs, outputs=decoder_output)
```

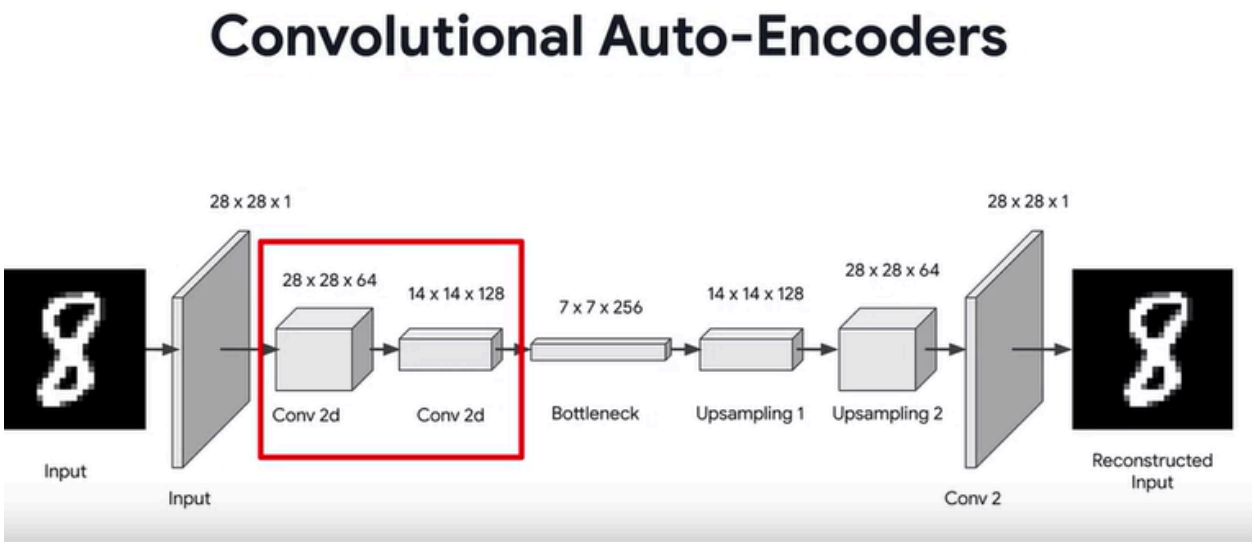
```
autoencoder_model.compile(
        optimizer=tf.keras.optimizers.Adam(),
        loss='binary_crossentropy')
```

Stacked AutoEncoders




```
inputs =  tf.keras.layers.Input(shape=(784,))

def deep_autoencoder():
 encoder = tf.keras.layers.Dense(units=128, activation='relu')(inputs)
 encoder = tf.keras.layers.Dense(units=64, activation='relu')(encoder)
 encoder = tf.keras.layers.Dense(units=32, activation='relu')(encoder)

 decoder = tf.keras.layers.Dense(units=64, activation='relu')(encoder)
 decoder = tf.keras.layers.Dense(units=128, activation='relu')(decoder)
 decoder = tf.keras.layers.Dense(units=784, activation='sigmoid')(decoder)

 return encoder, decoder

deep_encoder_output, deep_autoencoder_output = deep_autoencoder()

deep_encoder_model = tf.keras.Model(inputs=inputs, outputs=deep_encoder_output)
deep_autoencoder_model = tf.keras.Model(inputs=inputs, outputs=deep_autoencoder_output)
```

Convolutional AutoEncoders




```
def encoder(inputs):
  conv_1 = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3),
                                  activation='relu', padding='same')(inputs)

  max_pool_1 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_1)

  conv_2 = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3),
                                  activation='relu', padding='same')(max_pool_1)

  max_pool_2 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_2)

  return max_pool_2
```

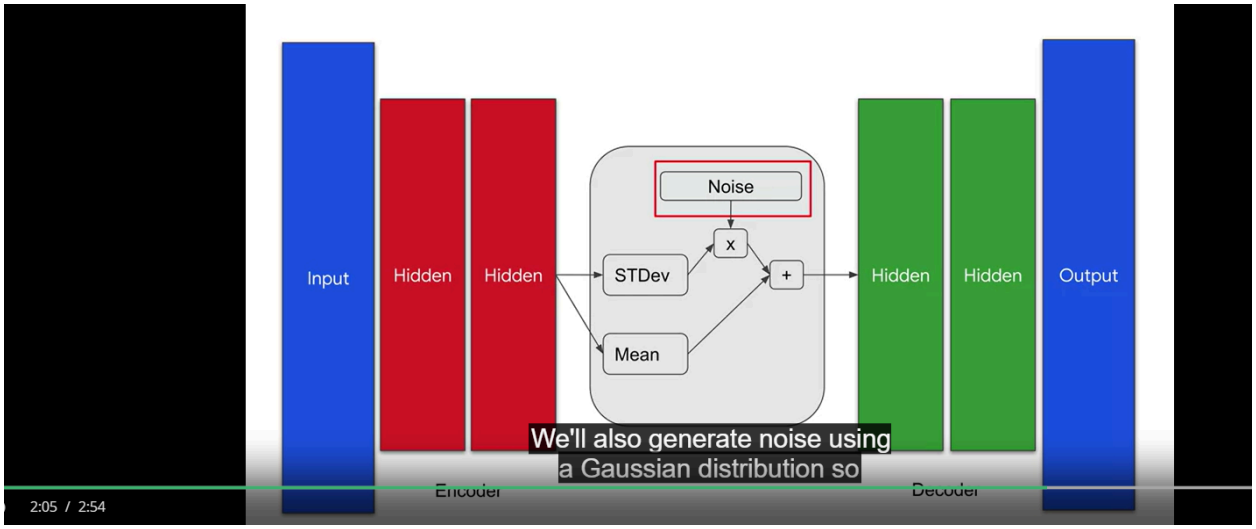
```
def bottle_neck(inputs):
  bottle_neck = tf.keras.layers.Conv2D(filters=256, kernel_size=(3,3),
                                       activation='relu', padding='same')(inputs)

  encoder_visualization = tf.keras.layers.Conv2D(filters=1, kernel_size=(3,3),
                                                 activation='sigmoid',
                                                 padding='same')(bottle_neck)

  return bottle_neck, encoder_visualization
```

```
def decoder(inputs):
  conv_1 = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3),
                                  activation='relu', padding='same')(inputs)
  up_sample_1 = tf.keras.layers.UpSampling2D(size=(2,2))(conv_1)

  conv_2 = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3),
                                  activation='relu', padding='same')(up_sample_1)
  up_sample_2 = tf.keras.layers.UpSampling2D(size=(2,2))(conv_2)

  conv_3 = tf.keras.layers.Conv2D(filters=1, kernel_size=(3,3),
                                  activation='sigmoid',
                                  padding='same')(up_sample_2)

  return conv_3
```

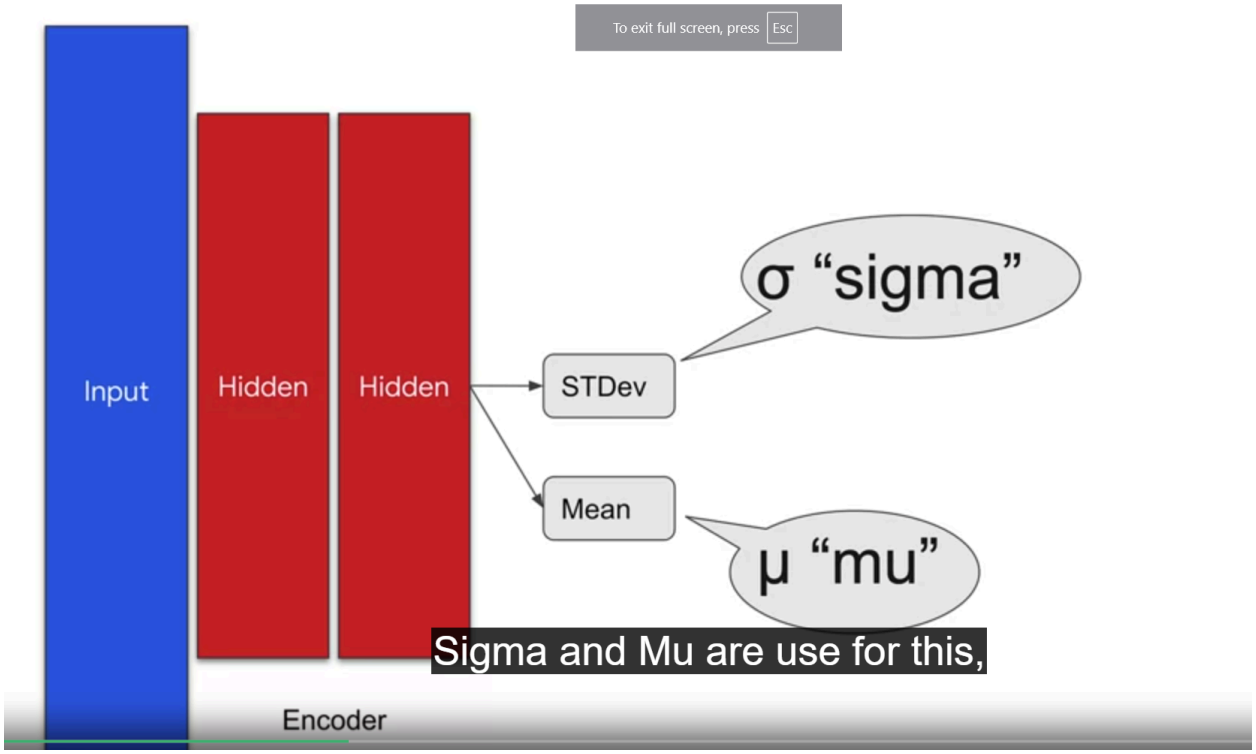
```
def convolutional_auto_encoder():
  inputs = tf.keras.layers.Input(shape=(28, 28, 1,))

  encoder_output = encoder(inputs)

  bottleneck_output, encoder_visualization = bottle_neck(encoder_output)

  decoder_output = decoder(bottleneck_output)

  model = tf.keras.Model(inputs =inputs, outputs=decoder_output)
  encoder_model = tf.keras.Model(inputs=inputs, outputs=encoder_visualization)

  return model, encoder_model
```

```
def map_image_with_noise(image, label):
  noise_factor = 0.5
  image = tf.cast(image, dtype=tf.float32)
  image = image / 255.0

  factor = noise_factor * tf.random.normal(shape=image.shape)
  image_noisy = image + factor
  image_noisy = tf.clip_by_value(image_noisy, 0.0, 1.0)

  return image_noisy, image
```

Variational AutoEncoder (VAE)



A Variational Autoencoder is a type of likelihood-based generative model. It consists of an encoder, that takes in data as input and transforms this into a latent representation , and a decoder, that takes a latent representation and returns a reconstruction . Inference is performed via variational inference to approximate the posterior of the model.



Probability Distribution

Gaussian probability density function or **Normal Distribution.**

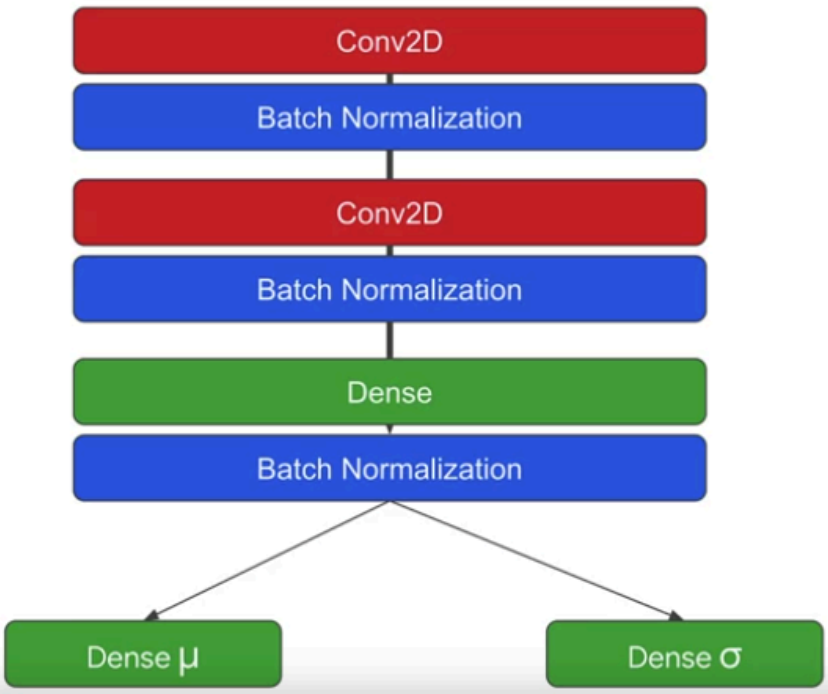
Normal Distribution is controlled by:

- **μ "mean"**
- **σ "standard deviation"**

$N(\mu, \sigma)$

what I refer to as noise earlier.

0:47 / 2:44



```python
# This function defines the encoder's layers
def encoder_layers(inputs, latent_dim):
  x = tf.keras.layers.Conv2D(filters=32, kernel_size=3, strides=2,
                             padding="same", activation='relu',
                             name="encode_conv1")(inputs)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Conv2D(filters=64, kernel_size=3, strides=2,
                             padding='same', activation='relu',
                             name="encode_conv2")(x)
  batch_2 = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Flatten(name="encode_flatten")(batch_2)
  x = tf.keras.layers.Dense(20, activation='relu', name="encode_dense")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  mu = tf.keras.layers.Dense(latent_dim, name='latent_mu')(x)
  sigma = tf.keras.layers.Dense(latent_dim, name='latent_sigma')(x)

  return mu, sigma, batch_2.shape
```

little to understand what's going on.

```python
class Sampling(tf.keras.layers.Layer):
  def call(self, inputs):
    mu, sigma = inputs
    batch = tf.shape(mu)[0]
    dim = tf.shape(mu)[1]
    epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
    return mu + tf.exp(0.5 * sigma) * epsilon
```

```python
def encoder_model(LATENT_DIM, input_shape):
  inputs = tf.keras.layers.Input(shape=input_shape)
  mu, sigma, conv_shape = encoder_layers(inputs, latent_dim=LATENT_DIM)
  z = Sampling()((mu, sigma))
  model = tf.keras.Model(inputs, outputs=[mu, sigma, z])
  return model, conv_shape
```

```
def decoder_layers(inputs, conv_shape):
  units = conv_shape[1] * conv_shape[2] * conv_shape[3]
  x = tf.keras.layers.Dense(units, activation = 'relu',
                            name="decode_dense1")(inputs)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Reshape((conv_shape[1], conv_shape[2], conv_shape[3]),
                              name="decode_reshape")(x)
  x = tf.keras.layers.Conv2DTranspose(filters=64, kernel_size=3, strides=2,
                                      padding='same', activation='relu',
                                      name="decode_conv2d_2")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Conv2DTranspose(filters=32, kernel_size=3, strides=2,
                                      padding='same', activation='relu',
                                      name="decode_conv2d3")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Conv2DTranspose(filters=1, kernel_size=3, strides=1, padding='same',
                                      name="decode_final")(x)

  return x
```

That's the job of the decoder

```
def decoder_model(latent_dim, conv_shape):
  inputs = tf.keras.layers.Input(shape=(latent_dim,))
  outputs = decoder_layers(inputs, conv_shape)
  model = tf.keras.Model(inputs, outputs)
  return model
```

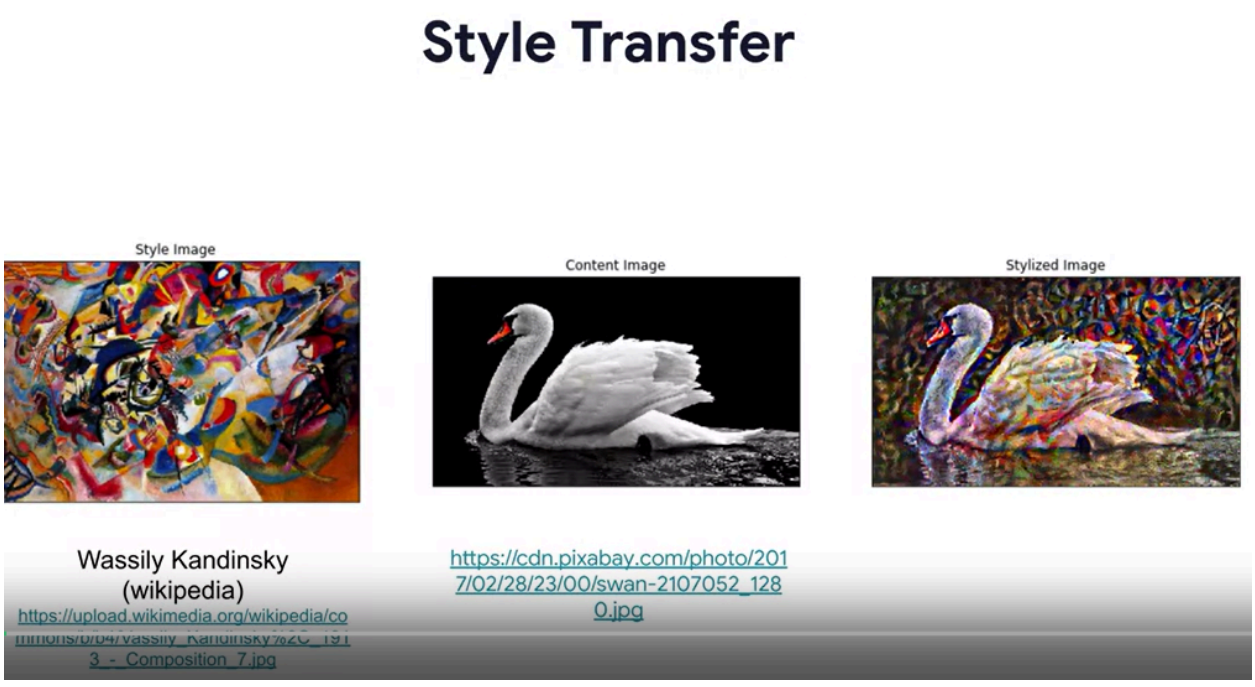
```python
# Define a kl reconstruction loss function
def kl_reconstruction_loss(mu, sigma):
  kl_loss = 1 + sigma - tf.square(mu) - tf.math.exp(sigma)
  return tf.reduce_mean(kl_loss) * -0.5
```

We'll use a Kullback-Leibler cost function here.

```python
def vae_model(encoder, decoder, input_shape):
  inputs = tf.keras.layers.Input(shape=input_shape)
  mu = encoder(inputs)[0]
  sigma = encoder(inputs)[1]
  z = encoder(inputs)[2]
  reconstructed = decoder(z)
  model = tf.keras.Model(inputs=inputs, outputs=reconstructed)
  loss = kl_reconstruction_loss(mu, sigma)
  model.add_loss(loss)
  return model
```

```python
for epoch in range(epochs):
  for step, x_batch_train in enumerate(train_dataset):
    with tf.GradientTape() as tape:
      reconstructed = vae(x_batch_train)
      flattened_inputs = tf.reshape(x_batch_train, shape=[-1])
      flattened_outputs = tf.reshape(reconstructed, shape=[-1])
      loss = bce_loss(flattened_inputs, flattened_outputs) * 784
      loss += sum(vae.losses)  # Add KLD regularization loss

    grads = tape.gradient(loss, vae.trainable_weights)
    optimizer.apply_gradients(zip(grads, vae.trainable_weights))
```

Style Transfer



Style transfer is an image processing technique by which the style and textures from one image can be transferred onto another image. For example, if we take the image of the swan in the middle of the screen, we can combine it with the image of an impressionist painting by Wassily Kandinsky on the left. The results will be a stylized version of the swan as you can see on the right-hand side

Approaches to Style Transfer

1. Supervised Learning.
2. Neural Style Transfer
3. Fast Neural Style Transfer

There are typically three ways that you can

Neural Style Transfer

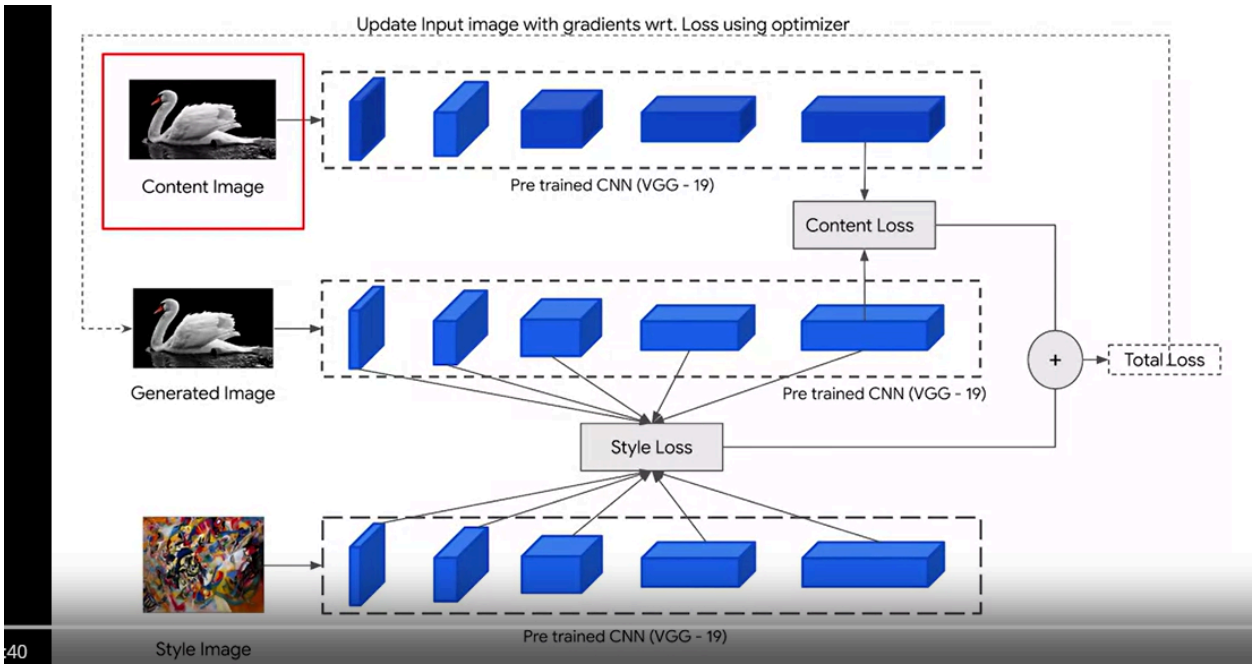
- **That's what you don't need to train a network to understand how one image should map to another. Instead, we extract features using a network and apply those features as styles. It's analogous to transfer learning. What's nice is we don't need lots and lots of training samples. We can do it with just a single pair of images.**

1. Supervised Learning

- Pairs: original & stylized image.
  - Need lots of pairs!

2. Neural Style Transfer

- pre-trained model
- Inputs: a single pair of images
  - Extract style from image 1
  - Extract content from image 2
- Generate image to match style and content.
  - In a loop: minimize loss



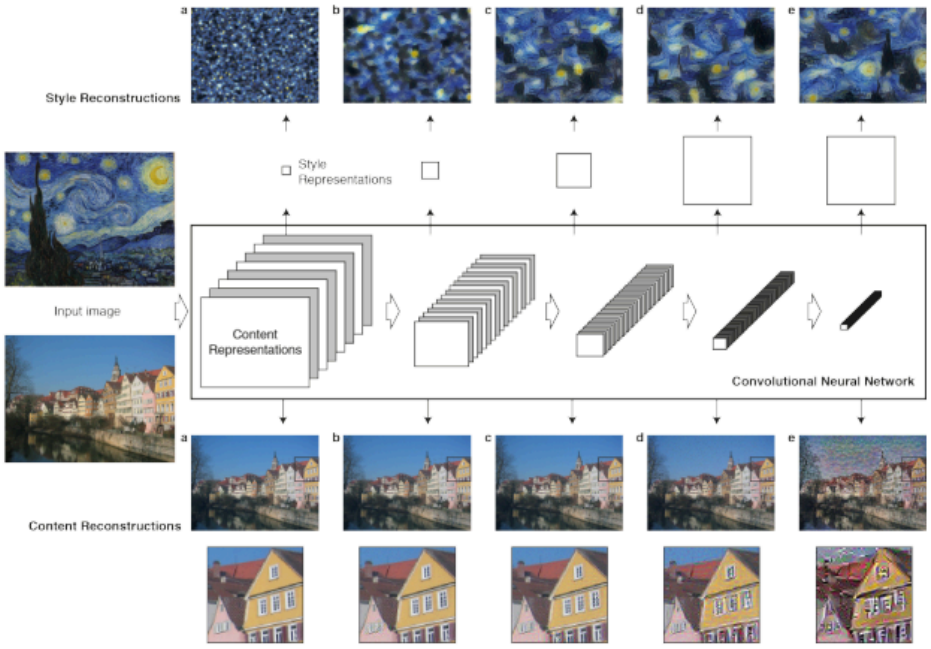


Figure 1: Convolutional Neural Network (CNN). **A given input image is represented as a set of filtered images at each processing stage in the CNN**. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (e.g. max-pooling) leading to a decrease in the total number of units per layer of the network. **Content Reconstructions. We can visualise the information at different processing stages in the CNN by reconstructing the input image from only knowing the network's responses in a particular layer. We reconstruct the input image from from layers 'conv1 1' (a), 'conv2 1' (b), 'conv3 1' (c), 'conv4 1' (d) and 'conv5 1' (e) of the original VGG-Network**. We find that reconstruction from lower layers is almost perfect (a,b,c). In higher layers of the network, detailed pixel information is lost while the high-level content of the image is preserved (d,e). **Style Reconstructions. On top of the original CNN representations we built a new feature space that captures the style of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from style representations built on different subsets of CNN layers ('conv1 1' (a), 'conv1 1' and 'conv2 1' (b), 'conv1 1', 'conv2 1' and 'conv3 1' (c), 'conv1 1', 'conv2 1', 'conv3 1' and 'conv4 1' (d), 'conv1 1', 'conv2 1', 'conv3 1', 'conv4 1' and 'conv5 1' (e)).** This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.

Steps to Develop Neural Style Transfer

1. Preprocess, content & style images
2. Load pre trained model, define loss functions
3. Loop:
    a. Optimize and generate new image
    b. Visualize Outputs

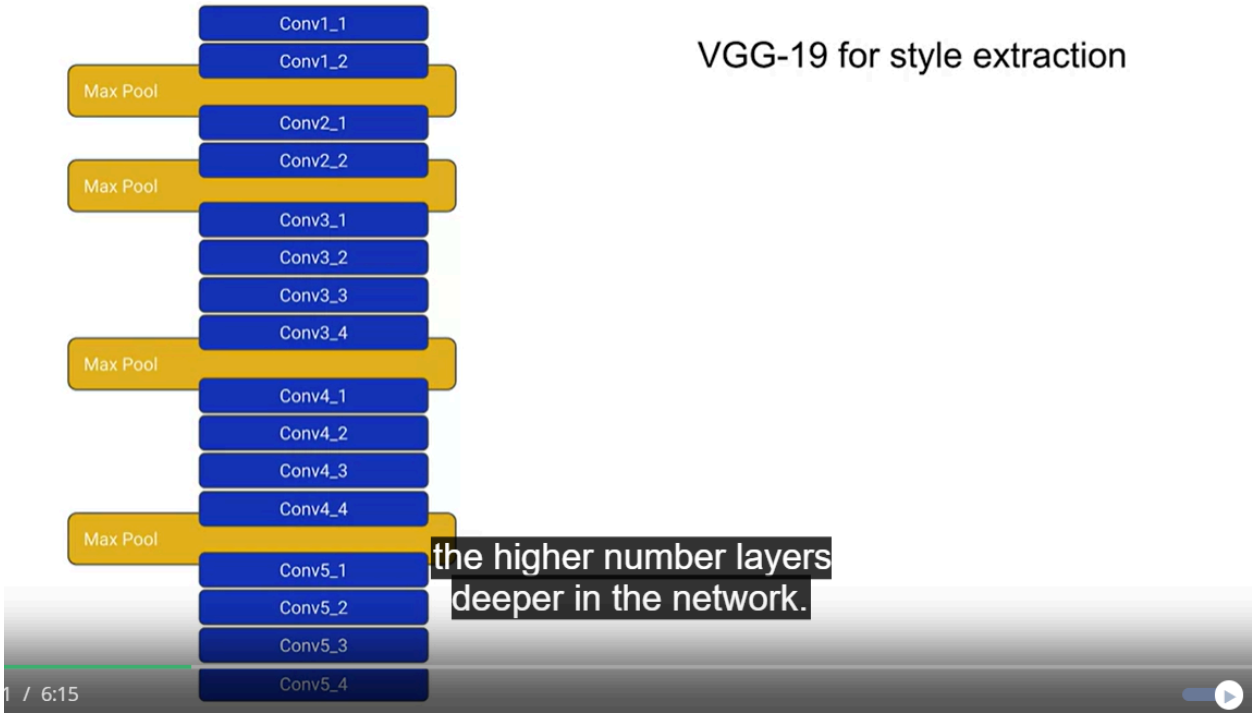
Load and Preprocess Images

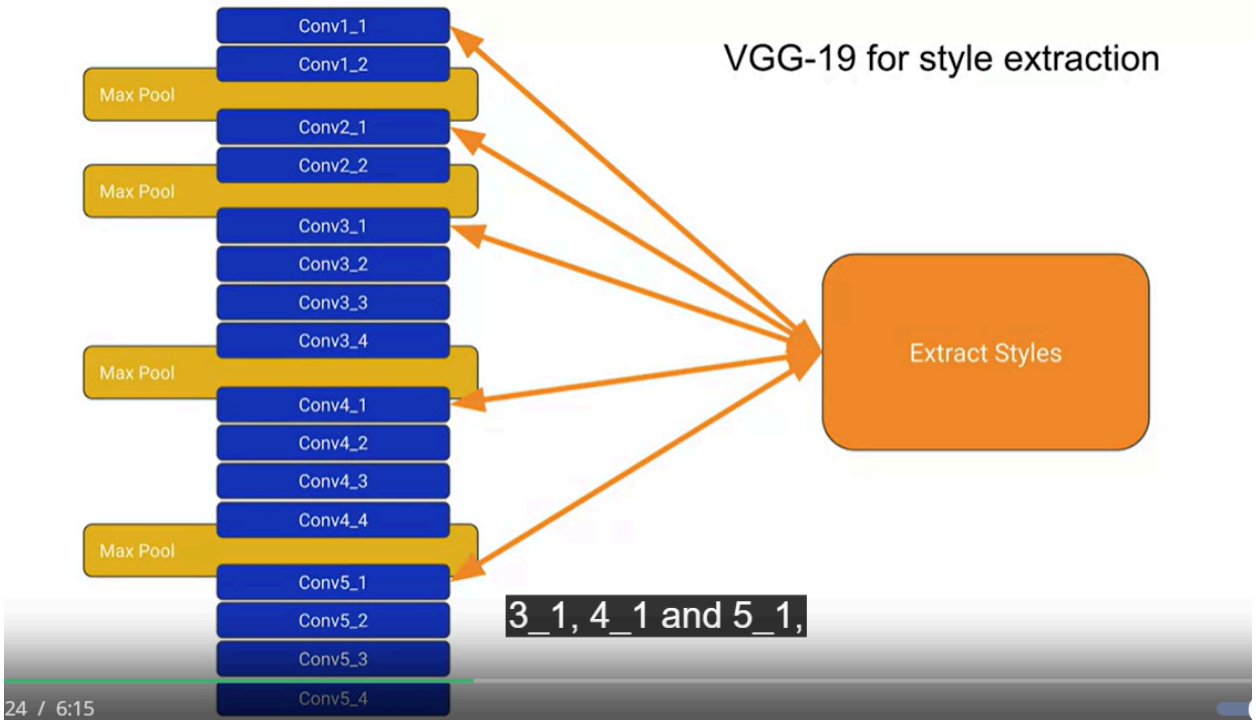
- VGG-19 pre trained on ‘imagenet’
- Accepts pixels values [0...255]
    - Don’t normalize to [0 ... 1]
- Expects centered pixel values

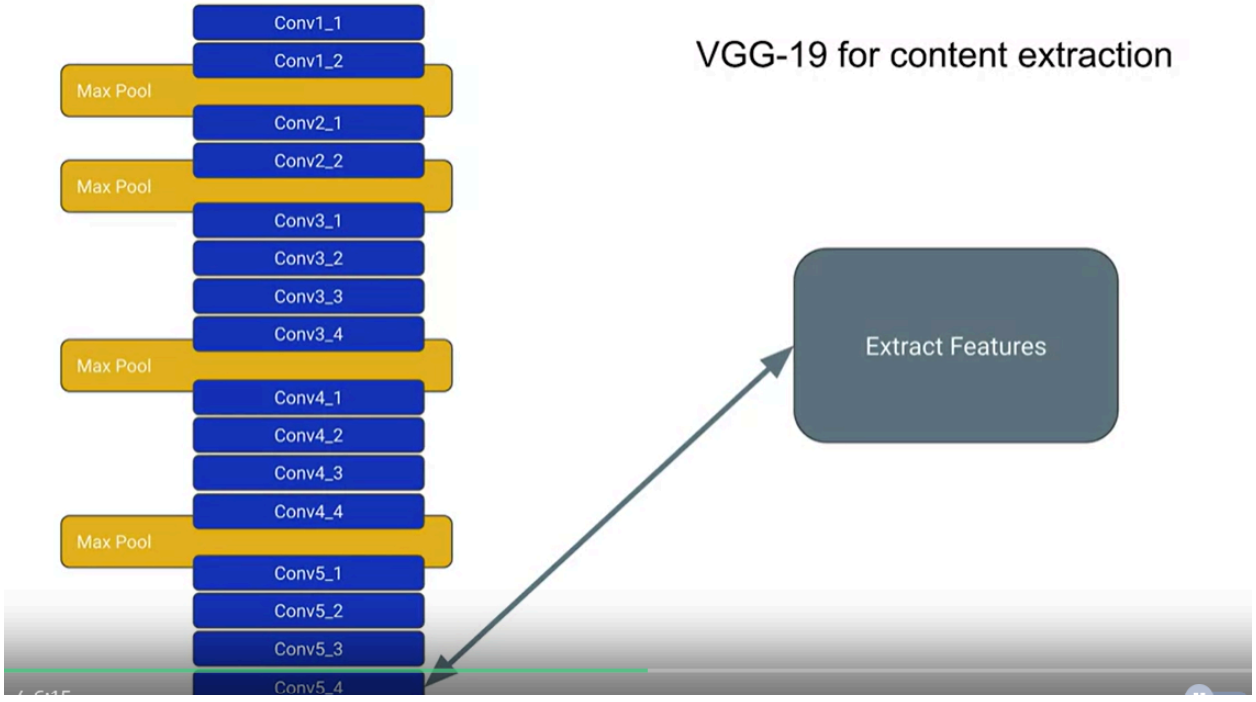
the images are formatted to be

Keras Preprocess_input

```
def preprocess_image(image):
    ...
    image = tf.keras.applications.vgg19.preprocess_input(image)

    return image
```




VGG-19 for style extraction
Conv1_1
Conv1_2
Max Pool
Conv2_1
Conv2_2
Max Pool
Conv3_1
Conv3_2
Conv3_3
Conv3_4
Max Pool
Conv4_1
Conv4_2
Conv4_3
Conv4_4
Max Pool
Conv5_1
Conv5_2
Conv5_3
Conv5_4
Extract Styles
3_1, 4_1 and 5_1,
24 / 6:15


VGG-19 for content extraction
Conv1_1
Conv1_2
Max Pool
Conv2_1
Conv2_2
Max Pool
Conv3_1
Conv3_2
Conv3_3
Conv3_4
Max Pool
Conv4_1
Conv4_2
Conv4_3
Conv4_4
Max Pool
Conv5_1
Conv5_2
Conv5_3
Conv5_4
Extract Features

```
# Content layer where will pull our feature maps
content_layers = ['block5_conv2']

# Style layer of interest
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'Block5_conv1']

layer_names = content_layers + style_layers
```

Block 5 Conv 2 - 2nd convolutional layer at fifth block - Used to extract content features

```
def vgg_model(layer_names):
  """ Creates a vgg model that returns a list of intermediate output values."""
  vgg = tf.keras.applications.vgg19.VGG19(include_top=False, weights='imagenet')
  vgg.trainable = False

  outputs = [vgg.get_layer(name).output for name in layer_names]

  model = tf.keras.Model(inputs=vgg.input, outputs=outputs)

  return model
```

Total Loss

$$L_{total} = L_{content} + L_{style}$$

Total Loss

$$L_{total} = \alpha L_{content} + \beta L_{style}$$

α : Content Weight

β : Style Weight

Total Loss

$$L_{total} = \alpha L_{content} + \beta L_{style}$$

α : Content Weight

β : Style Weight

Total Loss

$$L_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha L_{content}(\vec{p}, \vec{x}) + \beta L_{style}(\vec{a}, \vec{x})$$

Total Loss

$$L_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha L_{content}(\vec{p}, \vec{x}) + \beta L_{style}(\vec{a}, \vec{x})$$

$\vec{p}$: Content Image (Original Photograph)

$\vec{x}$: Generated Image initialized to Input Image

α : Content Weight

β : Style Weight

Total Loss

$$L_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha L_{content}(\vec{p}, \vec{x}) + \beta L_{style}(\vec{a}, \vec{x})$$

$\vec{a}$: Style Image

$\vec{x}$: Generated Image initialized to Input Image

α : Content Weight

β : Style Weight

Total Loss

$$L_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha L_{content}(\vec{p}, \vec{x}) + \beta L_{style}(\vec{a}, \vec{x})$$

$\vec{p}$: Content Image (Original Photograph)

$\vec{a}$: Style Image

$\vec{x}$: Generated Image initialized to Input Image

α : Content Weight

β : Style Weight

Content Loss

Generated image

1	2
3	4

−

Content image

2	2
2	2

=

1-2	2-2
3-2	4-2

Element-wise subtraction

$(1-2)^2$	$(2-2)^2$
$(3-2)^2$	$(4-2)^2$

Element-wise square

$1^2 + 0^2 + 1^2 + 2^2 =$ 6

Reduce sum

(½) 6 = 3

weight

Content Loss

$$L_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F^l_{i,j} - P^l_{i,j})^2$$

l: layer *l*

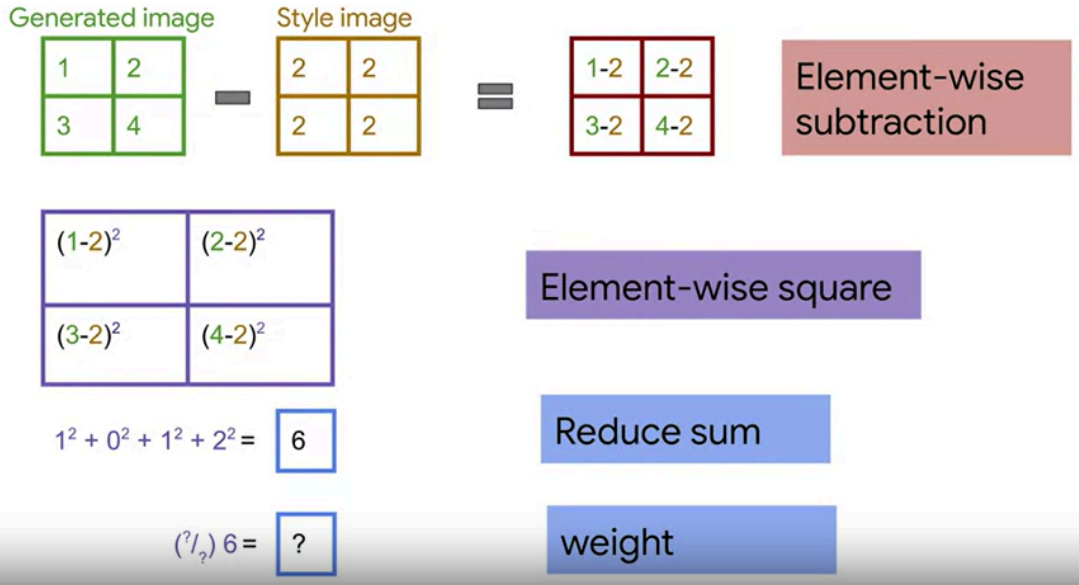
F^l_{ij}: Content representation of generated image x in layer l (Activation of *i*th feature map at position *j* in layer *l of generated image).*

P^l_{ij}: Content representation of content image p in layer l. (Activation of the *i* th feature map at position *j* in layer *l* of content image.)

Content Loss - In Practice

```
def get_content_loss(features, targets):
   return 0.5 * tf.reduce_sum(tf.square(features - targets))
```

Style Loss



Style Loss

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{i,j}^l - A_{ij}^l)^2$$

l: layer *l*

A^l_{ij} : Style Representation(Gram Matrix) of style image a.

G^l_{ij} : Style Representation(Gram Matrix) of generated image x

First, you take the difference

Gram Matrix

$$G_{ij}^{l} = \sum_{k} F_{ik}^{l} F_{jk}^{l}$$

G_{ij}^{l}: inner product between vectorized feature maps i and j in layer l.

and you can also check out the optional sections of

Gram Matrix - In Practice

```
def gram_matrix(input_tensor):
  result = tf.linalg.einsum('bijc,bijd->bcd', input_tensorT, input_tensor)
  input_shape = tf.shape(input_tensor)
  num_locations = tf.cast(input_shape[1]*input_shape[2], tf.float32)
  return result/(num_locations)
```

Get Style Feature Representation at Layers

```
def get_style_image_features(image):
  preprocessed_style_image = preprocess_image(image)
  style_outputs = vgg(preprocessed_style_image)
  gram_style_features =
     [gram_matrix(style_layer) for style_layer in style_outputs[:num_style_layers]]

  return gram_style_features
```

Get Style Feature Representation at Layers

```
def get_style_loss(features, targets):
  return tf.reduce_mean(tf.square(features - targets))
```

Total Loss

$$L_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha L_{content}(\vec{p}, \vec{x}) + \beta L_{style}(\vec{a}, \vec{x})$$

```
def get_style_content_loss(style_targets, style_outputs,
                           content_targets, content_outputs,
                           style_weight, content_weight):

  style_loss = tf.add_n([get_style_loss(style_output, style_target)
                           for style_output, style_target in zip(style_outputs, style_targets)])

  style_loss *= style_weight / num_style_layers

  content_loss = tf.add_n([get_content_loss(content_output, content_target)
                for content_output, content_target in zip(content_outputs, content_targets)])
  content_loss *= content_weight / num_content_layers

  loss = style_loss + content_loss
  return loss
```

Let's look at the code for this

```
def calculate_gradients(image, content_targets,
                        style_targets, style_weight,
                        content_weight,with_regularization=False):

    with tf.GradientTape() as tape:
        style_features = get_style_image_features(image)
        content_features = get_content_image_features(image)
        loss = get_style_content_loss(style_targets, style_features,
                                      content_targets, content_features,
                                      style_weight, content_weight)

    gradients = tape.gradient(loss, image)

    return gradients
```

```
def update_image_with_style(image, content_targets, style_targets,
                            optimizer, style_weight, content_weight,
                            with_regularization=False):

  gradients = calculate_gradients(image, content_targets,
                                  style_targets, style_weight,
                                  content_weight,with_regularization)

  optimizer.apply_gradients([(gradients, image)]) #Apply gradients on image
```

just saw to calculate
the gradients.

Style Loss

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{i,j}^l - A_{ij}^l)^2$$

l: layer *l*

A^l_{ij}: Style Representation(Gram Matrix) of style image a.

G^l_{ij}: Style Representation(Gram Matrix) of generated image x

Gram Matrix

$$G_{ij}^{l} = \sum_{k} F_{ik}^{l} F_{jk}^{l}$$

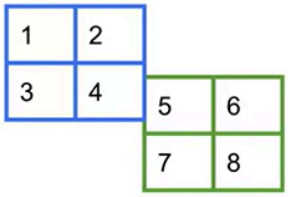
G_{ij}^{l}: inner product between vectorized feature maps i and j in layer l.

use gram matrices to

Style Loss (one layer)

Style layer

H =2
W = 2
F = 2



Gram matrix

$$A = \begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix} \quad G = \begin{bmatrix} a_1 \cdot a_1 & a_1 \cdot a_2 \\ a_2 \cdot a_1 & a_2 \cdot a_2 \end{bmatrix} = A^T A$$

a_1 a_2

which is A transpose A superscript T times A.

Style Loss (code)

```
style_layer = tf.constant([1,2,3,4,5,6,7,8],
                          shape=(2,2,2))

A = tf.transpose(
        tf.reshape(style_layer,
                   shape=(2,4)))

AT = tf.transpose(A)

G = tf.matmul(AT,A)
```

```
[[[1, 2],
  [3, 4]],

 [[5, 6],
  [7, 8]]]

[[1, 5],
 [2, 6],
 [3, 7],
 [4, 8]]

[[1, 2, 3, 4],
 [5, 6, 7, 8]]

[[ 30,  70],
 [ 70, 174]]
```

Here's what that looks like

Style Loss (code)

```
style_layer = tf.constant([1,2,3,4,5,6,7,8],
                          shape=(2,2,2))

A = tf.transpose(
        tf.reshape(style_layer,
                   shape=(2,4)))

AT = tf.transpose(A)

G = tf.matmul(AT,A)

G = tf.linalg.einsum('cij,dij->cd',
                     style_layer,
                     style_layer)
```

```
[[[1, 2],
  [3, 4]],

 [[5, 6],
  [7, 8]]]

[[1, 5],
 [2, 6],
 [3, 7],
 [4, 8]]

[[1, 2, 3, 4],
 [5, 6, 7, 8]]

[[ 30,  70],
 [ 70, 174]]
```

Einstein notation

`tf.linalg.einsum`

Einstein notation

```
x = tf.constant([1,2,3,4],shape=(2,2))
```

```
[1, 2],
[3, 4]
```

```
xT = tf.transpose(x)
```

```
[1, 3],
[2, 4]
```

```
tf.linalg.einsum('ij,ij->ij', x, x)
```

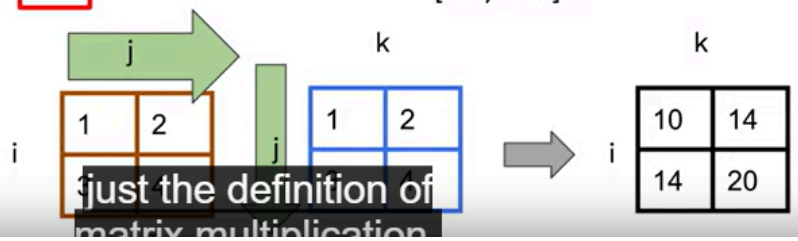
```
[ 1,  4],
[ 9, 16]
```

```
tf.square(x)
```

```
tf.linalg.einsum('ij,jk->ik', xT, x)
```

```
[10, 14],
[14, 20]
```

```
tf.matmul(xT, x)
```

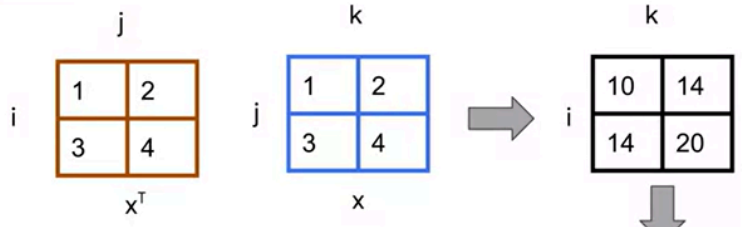


just the definition of
matrix multiplication

Reduce sum

```
tf.linalg.einsum('ij,jk->ik', xT, x)
```

```
[10, 14],
[14, 20]
```




```
tf.linalg.einsum('ij,jk->', xT, x)
```

```
[58]
```

Einstein notation

```
x = tf.constant([1,2,3,4],shape=(2,2))
```

```
[1, 2],
[3, 4]
```

```
v = tf.reshape(x, shape=(4,1) )
```

```
[1,
 2,
 3,
 4]
```

```
vT = tf.transpose(v)
```

```
[1,2,3,4]
```

```
tf.matmul(vT,v)
```

```
[30]
```

```
tf.linalg.einsum('ij,ij->', x, x)
```

```
[30]
```

Weighting the style loss

Style layer (height, width, filters)

1	2
3	4

5	6
7	8

Height =2
Width = 2
Filters = 2

Style weight

$$\frac{1}{(2 \times h \times w \times f)^2}$$

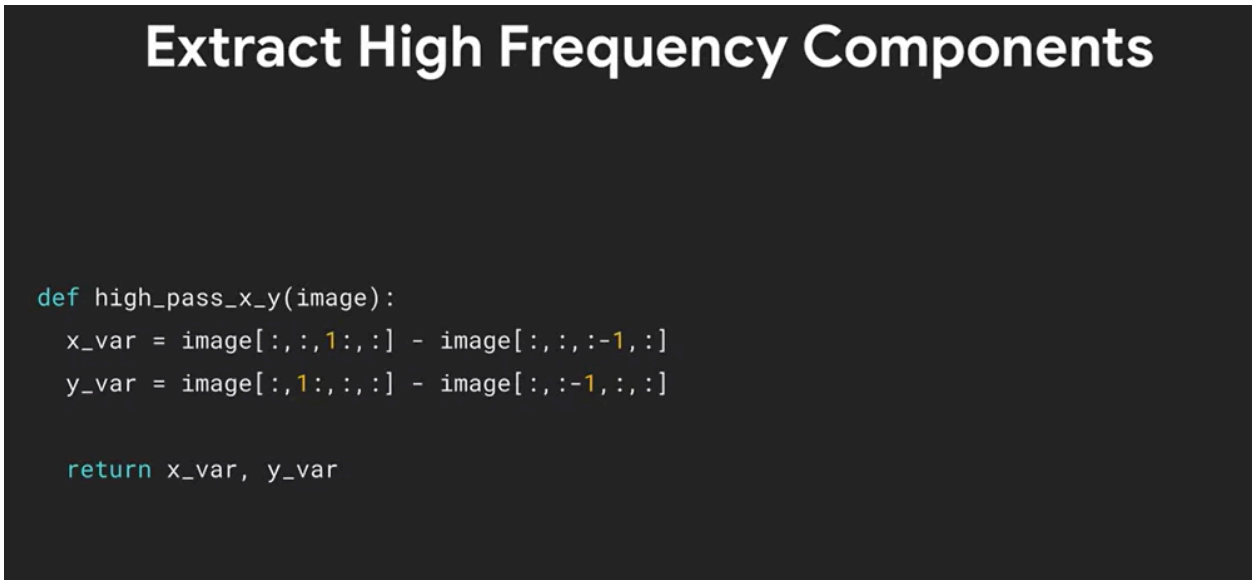
$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{i,j}^l - A_{ij}^l)^2$$

Gram Matrix - In Practice

(batch, height, width, filters) (batch, height, width, filters) (batch, filters, filters)

b i j c b i j d b c d

```
def gram_matrix(input_tensor):
  input_tensorT = tf.transpose(input_tensor,perm[0,2,1,3])
  result = tf.linalg.einsum('bijc,bijd->bcd', input_tensorT, input_tensor)
  input_shape = tf.shape(input_tensor)
  num_locations = tf.cast(input_shape[1]*input_shape[2], tf.float32)
  return result/(num_locations)
```



Neural Style Transfer paper:

1. When Convolutional Neural Networks are trained on object recognition, they develop a representation of the image that makes object information increasingly explicit along the processing hierarchy.8 Therefore, along the processing hierarchy of the network, the input image is transformed into representations that increasingly care about the actual content of the image compared to its detailed pixel values
2. We can directly visualise the information each layer contains about the input image by reconstructing the image only from the feature maps in that layer9 (Fig 1, content reconstructions, see Methods for details on how to reconstruct the image). Higher layers in the network capture the high-level content in terms of objects and their arrangement in the input image but do not constrain the exact pixel values of the reconstruction. (Fig 1, content reconstructions d,e). In contrast, reconstructions from the lower layers simply reproduce the exact pixel values of the original image (Fig 1, content reconstructions a,b,c). We therefore refer to the feature responses in higher layers of the network as the content representation.

In a Convolutional Neural Network (CNN), **higher layers** and **lower layers** refer to different stages of the network:

- **Lower Layers**: These are the initial layers of the CNN, closer to the input image. These layers capture basic features like edges, textures, and simple patterns. The feature maps generated in these layers retain more detailed information about the exact pixel values of the input image, making them more closely related to the original image in terms of appearance.
- **Higher Layers**: These are the deeper layers of the CNN, further away from the input. As you move deeper into the network, the layers capture more abstract and complex features, such as shapes, objects, and their spatial relationships. The feature maps in these layers represent high-level concepts rather than exact pixel values, focusing more on the overall content or structure of the image rather than the fine details.

In summary, lower layers capture fine-grained, low-level features (like edges and textures), while higher layers capture abstract, high-level content (like objects and their arrangement) in the input image.

Lower Layers

- **Feature Information**: Capture specific, detailed features such as edges, textures, and simple patterns.
- **Spatial Information**: Retain precise spatial details, including the exact location of edges and fine-grained textures.
- **Role**: Provide the foundational details needed for more complex feature recognition and understanding. They focus on the granular aspects of the image.

Higher Layers

- **Feature Information**: Capture generalized, abstract features such as objects, shapes, and complex patterns.
- **Spatial Information**: Integrate detailed spatial information from lower layers to understand the overall arrangement and context of objects.
- **Role**: Offer a comprehensive understanding of the image's content by synthesizing and interpreting detailed features. They focus on the high-level structure and meaning of the image.

Combined Summary:

- **Lower Layers**: Focus on detailed features (edges, textures) and precise spatial information, providing the building blocks for feature recognition.
- **Higher Layers**: Focus on generalized features (objects, shapes) and integrated spatial information, assembling detailed features into a holistic understanding of the image's content and structure.

Sure, let's summarize the high-level and low-level feature representations, including their relation to spatial and feature information:

- **Low-Level Representations**:
    - **Feature Information**: Capture specific, detailed features like edges, textures, and simple patterns.
    - **Spatial Information**: Retain precise spatial details of the image, such as the exact location of edges and texture details.
    - **Role**: Provide the foundational information needed to build more complex representations. They focus on the fine-grained aspects of the image.
- **High-Level Representations**:
    - **Feature Information**: Capture more abstract and generalized features like objects, shapes, and their spatial relationships.

    - **Spatial Information**: Integrate and interpret the detailed spatial information from lower layers to understand the overall arrangement and context of objects in the image.
    - **Role**: Offer a comprehensive understanding of the image's content by assembling and interpreting the detailed features from the lower layers.

Summary:

- **Low-Level Representations**: Detailed features and precise spatial information (e.g., edges, textures).
- **High-Level Representations**: Generalized features and integrated spatial information (e.g., objects, overall arrangement).