# Neural style transfer

## Compose Image in Style of Another -- Creating Art With Deep Learning

Xiaoyu Wang
Adhira Deogade

# Define functions for loading and showing the images

- Visualize the Input

```
[6] def load_img(path_to_img):
      max_dim = 512
      img = Image.open(path_to_img)
      long = max(img.size)
      scale = max_dim/long
      img = img.resize((round(img.size[0]*scale), round(img.size[1]*scale)), Image.ANTIALIAS)

      img = kp_image.img_to_array(img)

      # We need to broadcast the image array such that it has a batch dimension
      img = np.expand_dims(img, axis=0)
      return img
```

```
[7] def imshow(img, title=None):
      # Remove the batch dimension
      out = np.squeeze(img, axis=0)

      # Normalize for display
      out = out.astype('uint8')
      plt.imshow(out)

      if title is not None:
        plt.title(title)
      plt.imshow(out)
```

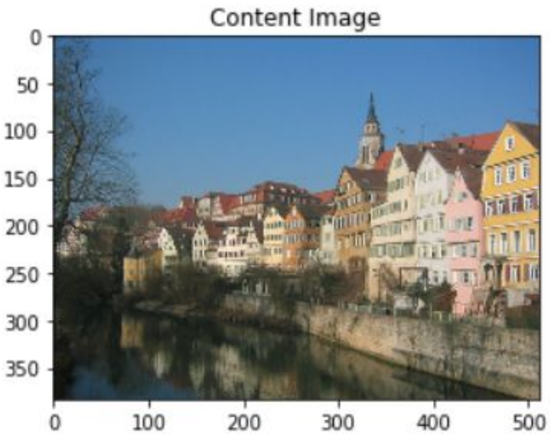
# Create Content Image & Style Image

```
[10] plt.figure(figsize=(10,10))

     content = load_img(content_path).astype('uint8')
     style = load_img(style_path).astype('uint8')

     plt.subplot(1, 2, 1)
     imshow(content, 'Content Image')

     plt.subplot(1, 2, 2)
     imshow(style, 'Style Image')
     plt.show()
```




# Prepare Images & Inverse Preprocessing

```
[12] def deprocess_img(processed_img):
       x = processed_img.copy()
       if len(x.shape) == 4:
         x = np.squeeze(x, 0)
       assert len(x.shape) == 3, ("Input to deprocess image must be an image of "
                                  "dimension [1, height, width, channel] or [height, width, channel]")
       if len(x.shape) != 3:
         raise ValueError("Invalid input to deprocessing image")

       # perform the inverse of the preprocessing step
       x[:, :, 0] += 103.939
       x[:, :, 1] += 116.779
       x[:, :, 2] += 123.68
       x = x[:, :, ::-1]

       x = np.clip(x, 0, 255).astype('uint8')
       return x
```

VGG are trained on image with each channel
Normalized by mean = [103.939, 116.779, 123.68]
And with channels BGR

Since our optimized image may take its values anywhere between $-\infty$ and $\infty$
We must clip to maintain our values from within the 0-255 range.

# Define Content & Style Representation

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

     # Style layer we are interested in
     style_layers = ['block1_conv1',
                     'block2_conv1',
                     'block3_conv1',
                     'block4_conv1',
                     'block5_conv1'
                    ]

     num_content_layers = len(content_layers)
     num_style_layers = len(style_layers)
```

As we reconstruct the original image from deeper layers
We still preserve the high-level content of the original
But lose the exact pixel information.

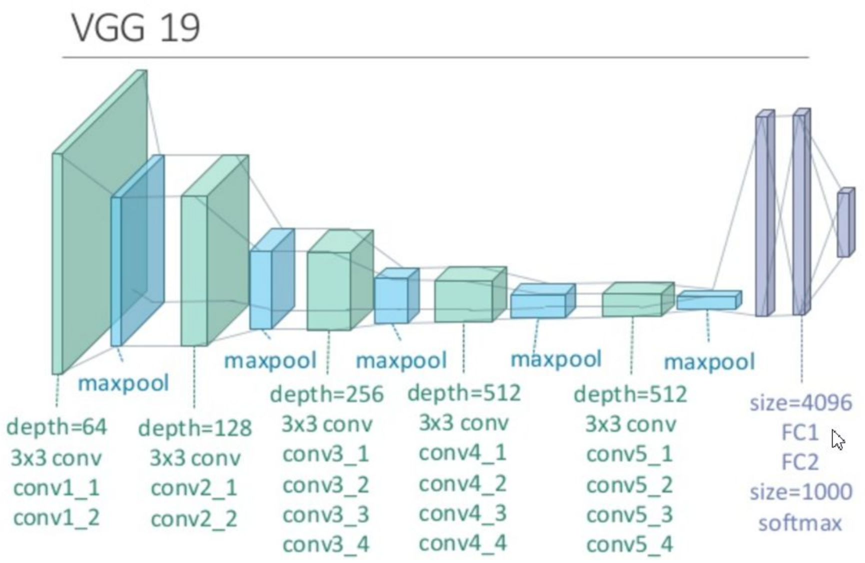
Best results were achieved by taking a combination of shallow and deep layers
As the style representation for an image.

By accessing intermediate layers, we're able to describe the content and style of input images

# Apply Pretrained VGG-19 Model

```
[15] vgg = tf.keras.applications.vgg19.VGG19(include_top=False, weights='imagenet')
     layers = dict([(layer.name, layer.output) for layer in vgg.layers])
     layers
```

```
{'block1_conv1': <tf.Tensor 'Relu_16:0' shape=(?, ?, ?, 64) dtype=float32>,
 'block1_conv2': <tf.Tensor 'Relu_17:0' shape=(?, ?, ?, 64) dtype=float32>,
 'block1_pool': <tf.Tensor 'MaxPool_5:0' shape=(?, ?, ?, 64) dtype=float32>,
 'block2_conv1': <tf.Tensor 'Relu_18:0' shape=(?, ?, ?, 128) dtype=float32>,
 'block2_conv2': <tf.Tensor 'Relu_19:0' shape=(?, ?, ?, 128) dtype=float32>,
 'block2_pool': <tf.Tensor 'MaxPool_6:0' shape=(?, ?, ?, 128) dtype=float32>,
 'block3_conv1': <tf.Tensor 'Relu_20:0' shape=(?, ?, ?, 256) dtype=float32>,
 'block3_conv2': <tf.Tensor 'Relu_21:0' shape=(?, ?, ?, 256) dtype=float32>,
 'block3_conv3': <tf.Tensor 'Relu_22:0' shape=(?, ?, ?, 256) dtype=float32>,
 'block3_conv4': <tf.Tensor 'Relu_23:0' shape=(?, ?, ?, 256) dtype=float32>,
 'block3_pool': <tf.Tensor 'MaxPool_7:0' shape=(?, ?, ?, 256) dtype=float32>,
 'block4_conv1': <tf.Tensor 'Relu_24:0' shape=(?, ?, ?, 512) dtype=float32>,
 'block4_conv2': <tf.Tensor 'Relu_25:0' shape=(?, ?, ?, 512) dtype=float32>,
 'block4_conv3': <tf.Tensor 'Relu_26:0' shape=(?, ?, ?, 512) dtype=float32>,
 'block4_conv4': <tf.Tensor 'Relu_27:0' shape=(?, ?, ?, 512) dtype=float32>,
 'block4_pool': <tf.Tensor 'MaxPool_8:0' shape=(?, ?, ?, 512) dtype=float32>,
 'block5_conv1': <tf.Tensor 'Relu_28:0' shape=(?, ?, ?, 512) dtype=float32>,
 'block5_conv2': <tf.Tensor 'Relu_29:0' shape=(?, ?, ?, 512) dtype=float32>,
 'block5_conv3': <tf.Tensor 'Relu_30:0' shape=(?, ?, ?, 512) dtype=float32>,
 'block5_conv4': <tf.Tensor 'Relu_31:0' shape=(?, ?, ?, 512) dtype=float32>,
 'block5_pool': <tf.Tensor 'MaxPool_9:0' shape=(?, ?, ?, 512) dtype=float32>,
 'input_2': <tf.Tensor 'input_2:0' shape=(?, ?, ?, 3) dtype=float32>}
```



# Load VGG-19 Using Keras Functional API

```
[16] def get_model():
       """ Creates our model with access to intermediate layers.

       This function will load the VGG19 model and access the intermediate layers.
       These layers will then be used to create a new model that will take input image
       and return the outputs from these intermediate layers from the VGG model.

       Returns:
         returns a keras model that takes image inputs and outputs the style and
           content intermediate layers.
       """
       # Load our model. We load pretrained VGG, trained on imagenet data
       vgg = tf.keras.applications.vgg19.VGG19(include_top=False, weights='imagenet')
       vgg.trainable = False
       # Get output layers corresponding to style and content layers
       style_outputs = [vgg.get_layer(name).output for name in style_layers]
       content_outputs = [vgg.get_layer(name).output for name in content_layers]
       model_outputs = style_outputs + content_outputs
       # Build model
       return models.Model(vgg.input, model_outputs)
```

With the Functional API defining a model simply involves Defining the input and output:

*model = Model(inputs, outputs)*

# Computing Content Loss

Given a chosen content layer $l$, the content loss is defined as the euclidean distance between the feature map $F^l$ of our content image $C$ and the feature map $P^l$ of our generated image $Y$. When the content representation of $C$ and $Y$ are exactly the same this loss becomes $0$.

$$\mathcal{L}_{content} = \frac{1}{2} \sum_{i,j} (F^l_{ij} - P^l_{ij})^2$$

```
[17] def get_content_loss(base_content, target):
       return tf.reduce_mean(tf.square(base_content - target))
```

Pass the network both the desired content image and base input image.
This will return the intermediate layer outputs from our model.
Take the euclidean distance between the two intermediate representations of those images.

# Computing Style Loss

Computing style loss is a bit more involved, but follows the same principle, this time feeding our network the base input image and the style image. However, instead of comparing the raw intermediate outputs of the base input image and the style image, we instead compare the Gram matrices of the two outputs.

Mathematically, we describe the style loss of the base input image, $x$, and the style image, $a$, as the distance between the style representation (the gram matrices) of these images. We describe the style representation of an image as the correlation between different filter responses given by the Gram matrix $G^l$, where $G^l_{ij}$ is the inner product between the vectorized feature map $i$ and $j$ in layer $l$. We can see that $G^l_{ij}$ generated over the feature map for a given image represents the correlation between feature maps $i$ and $j$.

To generate a style for our base input image, we perform gradient descent from the content image to transform it into an image that matches the style representation of the original image. We do so by minimizing the mean squared distance between the feature correlation map of the style image and the input image. The contribution of each layer to the total style loss is described by

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G^l_{ij} - A^l_{ij})^2$$

where $G^l_{ij}$ and $A^l_{ij}$ are the respective style representation in layer $l$ of $x$ and $a$. $N_l$ describes the number of feature maps, each of size $M_l = height * width$. Thus, the total style loss across each layer is

$$L_{style}(a, x) = \sum_{l \in L} w_l E_l$$

where we weight the contribution of each layer's loss by some factor $w_l$. In our case, we weight each layer equally ($w_l = \frac{1}{|L|}$)

# Computing Style Loss

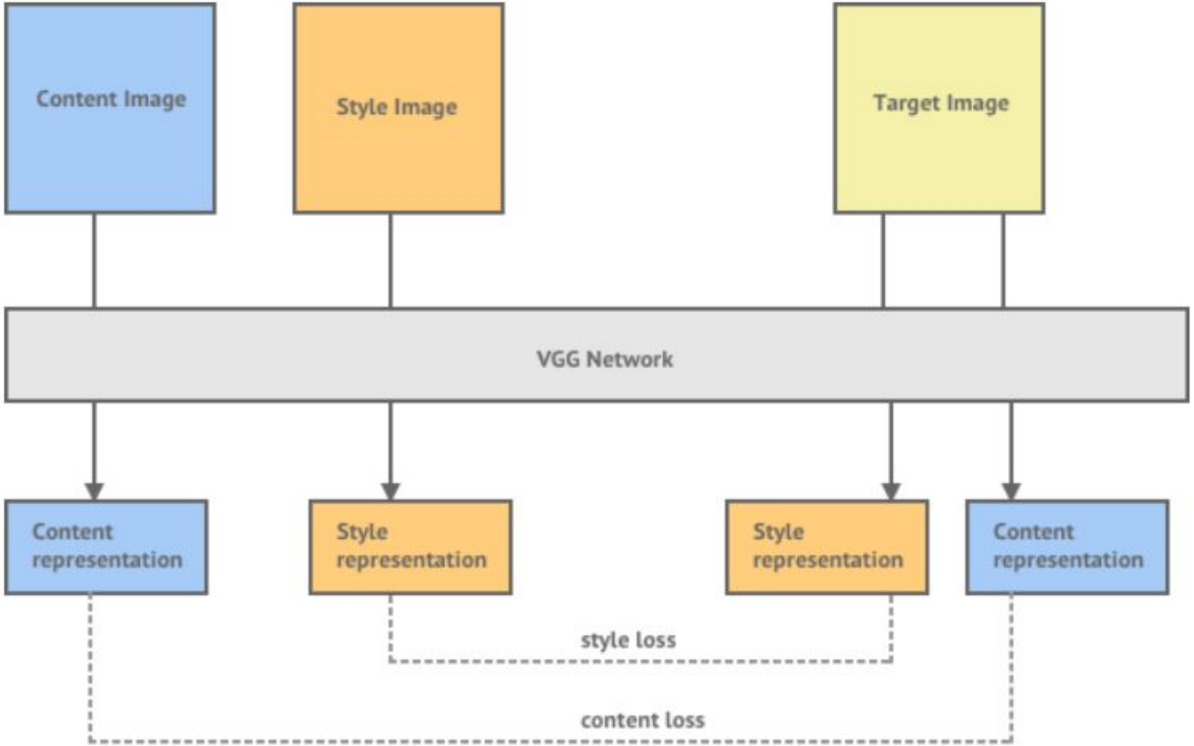
```
[ ]  def gram_matrix(input_tensor):
       # We make the image channels first
       channels = int(input_tensor.shape[-1])
       a = tf.reshape(input_tensor, [-1, channels])
       n = tf.shape(a)[0]
       gram = tf.matmul(a, a, transpose_a=True)
       return gram / tf.cast(n, tf.float32)

     def get_style_loss(base_style, gram_target):
       """Expects two images of dimension h, w, c"""
       # height, width, num filters of each layer

       # We scale the loss at a given layer by the size of the feature map and the number of filters
       height, width, channels = base_style.get_shape().as_list()
       gram_style = gram_matrix(base_style)

       return tf.reduce_mean(tf.square(gram_style - gram_target))# / (4. * (channels ** 2) * (width * height) ** 2)
```

# Computing Total Loss



# Computing Total Loss

```
[20] def compute_loss(model, loss_weights, init_image, gram_style_features, content_features):

       style_weight, content_weight = loss_weights

       # Feed our init image through our model. This will give us the content and
       # style representations at our desired layers. Since we're using eager
       # our model is callable just like any other function!
       model_outputs = model(init_image)

       style_output_features = model_outputs[:num_style_layers]
       content_output_features = model_outputs[num_style_layers:]

       style_score = 0
       content_score = 0

       # Accumulate style losses from all layers
       # Here, we equally weight each contribution of each loss layer
       weight_per_style_layer = 1.0 / float(num_style_layers)
       for target_style, comb_style in zip(gram_style_features, style_output_features):
         style_score += weight_per_style_layer * get_style_loss(comb_style[0], target_style)

       # Accumulate content losses from all layers
       weight_per_content_layer = 1.0 / float(num_content_layers)
       for target_content, comb_content in zip(content_features, content_output_features):
         content_score += weight_per_content_layer* get_content_loss(comb_content[0], target_content)

       style_score *= style_weight
       content_score *= content_weight

       # Get total loss
       loss = style_score + content_score
       return loss, style_score, content_score
```

# Computing Total Loss

The total loss can then be written as a weighted sum of the both the style and content losses, where the weights can be adjusted to preserve more of the style or more of the content.

$$\mathcal{L}_{total} = \alpha\mathcal{L}_{content} + \beta\mathcal{L}_{style}$$

Performing the task of style transfer can now be reduced to the task of trying to generate an image $Y$ which minimises the loss function.

```
[21] # Compute the gradients:
     def compute_grads(cfg):
       with tf.GradientTape() as tape:
         all_loss = compute_loss(**cfg)

       # Compute gradients wrt input image
       total_loss = all_loss[0]
       return tape.gradient(total_loss, cfg['init_image']), all_loss
```

# Generate Optimization Loop

```
[22]  import IPython.display

      def run_style_transfer(content_path,
                             style_path,
                             num_iterations=1000,
                             content_weight=1e3,
                             style_weight=1e-2):
        # We don't need to (or want to) train any layers of our model, so we set their
        # trainable to false.
        model = get_model()
        for layer in model.layers:
          layer.trainable = False

        # Get the style and content feature representations (from our specified intermediate layers)
        style_features, content_features = get_feature_representations(model, content_path, style_path)
        gram_style_features = [gram_matrix(style_feature) for style_feature in style_features]

        # Set initial image
        init_image = load_and_process_img(content_path)
        init_image = tfe.Variable(init_image, dtype=tf.float32)
        # Create our optimizer
        opt = tf.train.AdamOptimizer(learning_rate=5, beta1=0.99, epsilon=1e-1)

        # For displaying intermediate images
        iter_count = 1

        # Store our best result
        best_loss, best_img = float('inf'), None

        # Create a nice config
        loss_weights = (style_weight, content_weight)
        cfg = {
            'model': model,
            'loss_weights': loss_weights,
            'init_image': init_image,
            'gram_style_features': gram_style_features,
            'content_features': content_features
        }
```

# Generate Optimization Loop

```
[22]
    imgs = []
    for i in range(num_iterations):
      grads, all_loss = compute_grads(cfg)
      loss, style_score, content_score = all_loss
      opt.apply_gradients([(grads, init_image)])
      clipped = tf.clip_by_value(init_image, min_vals, max_vals)
      init_image.assign(clipped)
      end_time = time.time()

      if loss < best_loss:
        # Update best loss and best image from total loss.
        best_loss = loss
        best_img = deprocess_img(init_image.numpy())

      if i % display_interval== 0:
        start_time = time.time()

        # Use the .numpy() method to get the concrete numpy array
        plot_img = init_image.numpy()
        plot_img = deprocess_img(plot_img)
        imgs.append(plot_img)
        IPython.display.clear_output(wait=True)
        IPython.display.display_png(Image.fromarray(plot_img))
        print('Iteration: {}'.format(i))
        print('Total loss: {:.4e}, '
              'style loss: {:.4e}, '
              'content loss: {:.4e}, '
              'time: {:.4f}s'.format(loss, style_score, content_score, time.time() - start_time))
    print('Total time: {:.4f}s'.format(time.time() - global_start))
    IPython.display.clear_output(wait=True)
    plt.figure(figsize=(14,4))
    for i,img in enumerate(imgs):
        plt.subplot(num_rows,num_cols,i+1)
        plt.imshow(img)
        plt.xticks([])
        plt.yticks([])

    return best_img, best_loss
```

# Apply Style Transfer To Test Image

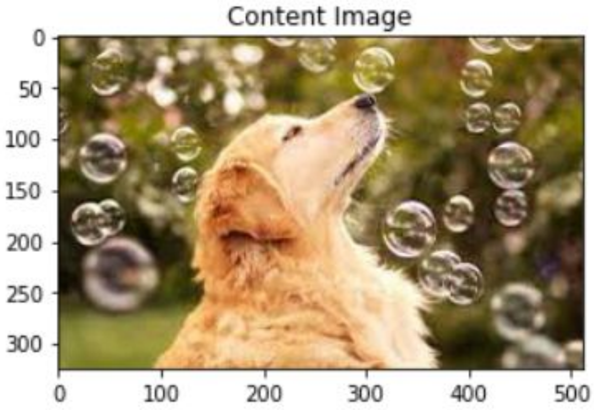
```
best, best_loss = run_style_transfer(content_path,
                                     style_path, num_iterations=1000)
```

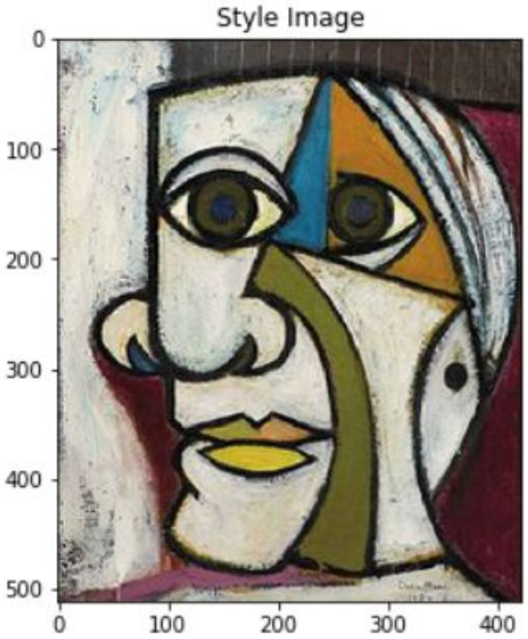
# Apply Style Transfer To Test Image

```
Image.fromarray(best)
```

# Combine Show Result Functions

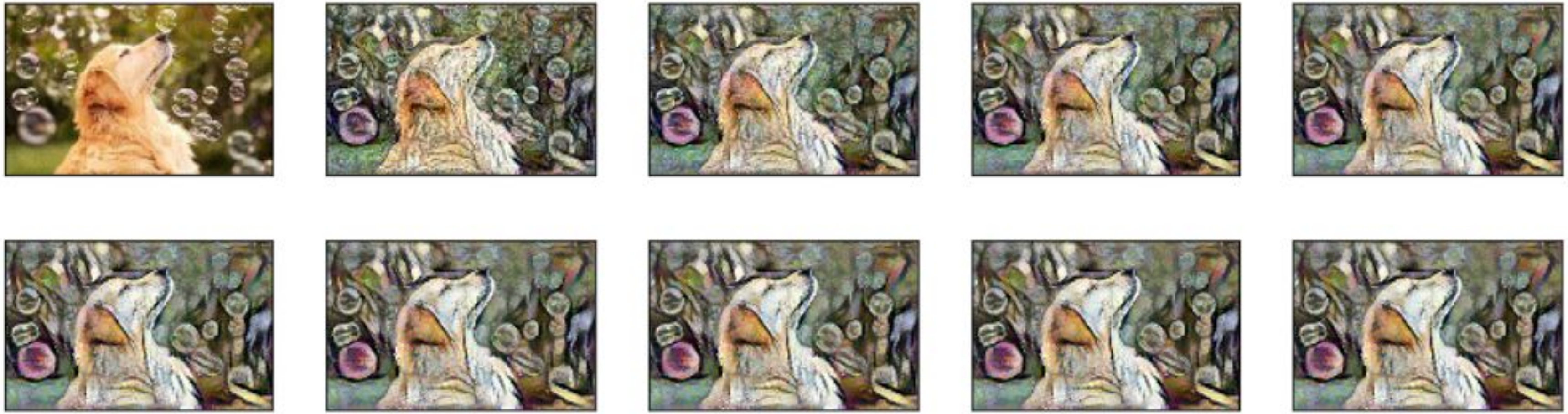
```
show_results(best_dog_styled, 'drive/My Drive/Team 3 Final Project/dog.jpg',
             'drive/My Drive/Team 3 Final Project/dora-maar-picasso.jpg')
```



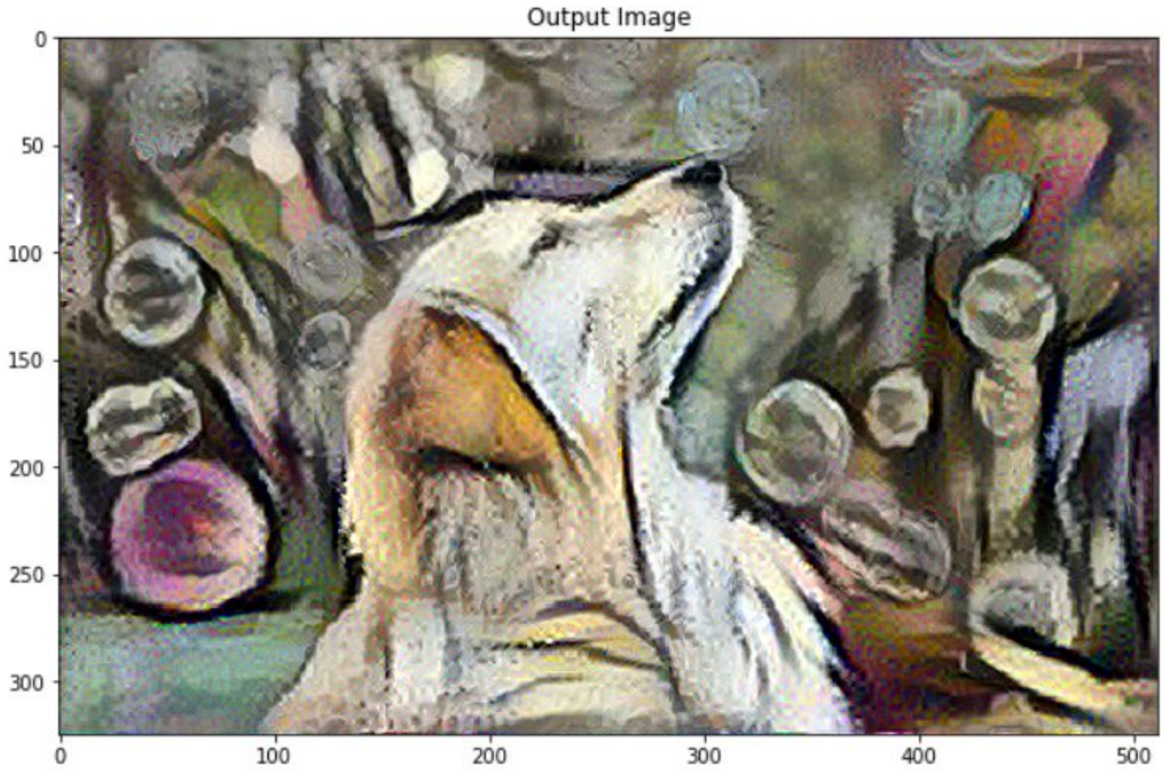



# Visualize the Final Output

```
best_dog_styled, best_loss = run_style_transfer('drive/My Drive/Team 3 Final Project/dog.jpg',
                                                'drive/My Drive/Team 3 Final Project/dora-maar-picasso.jpg')
```

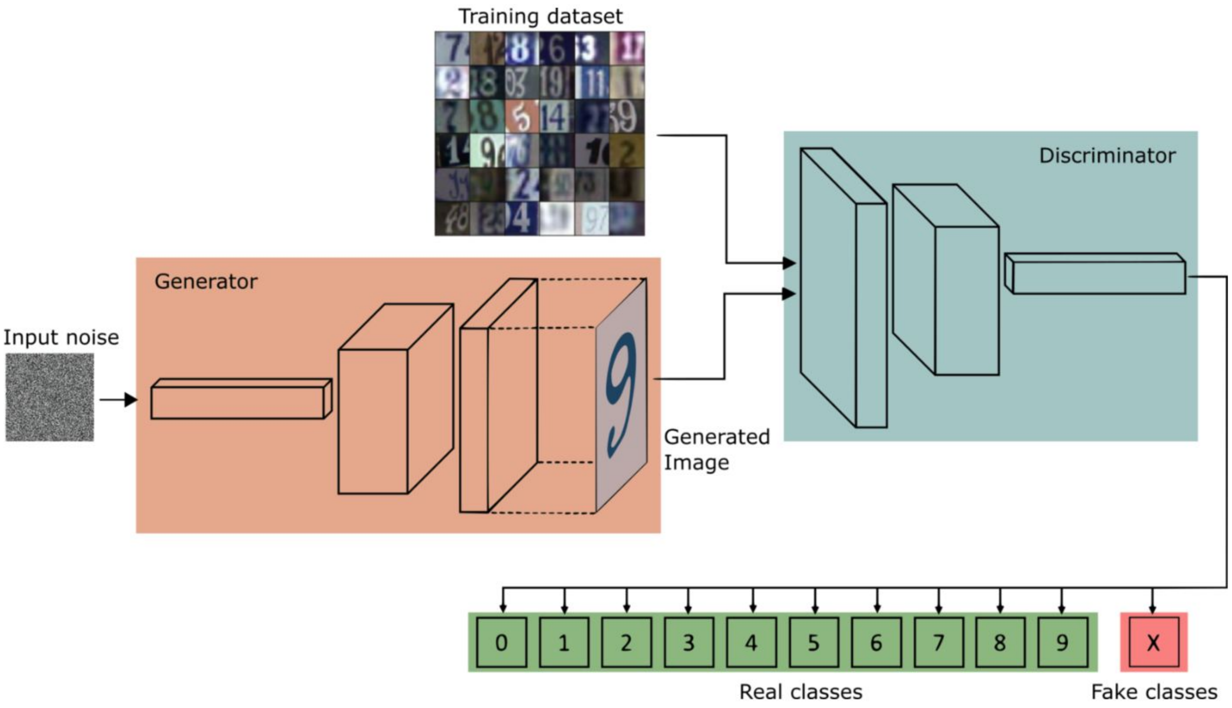


# Visualize the Final Output



# Generative Adversarial Network (GANs)

1. Training semi-supervised classifiers
2. Generating data, learning to create output closer to desired value by generating the desired value
3. Both labelled and unlabeled data are used to train a classifier
4. This takes a small portion of labeled data and a large portion of unlabeled data, combine to train a deep convolutional neural network to learn an inferred function capable of mapping a new data point to its desirable outcome




Semi-supervided learning GAN architecture for an 11 class classification problem.

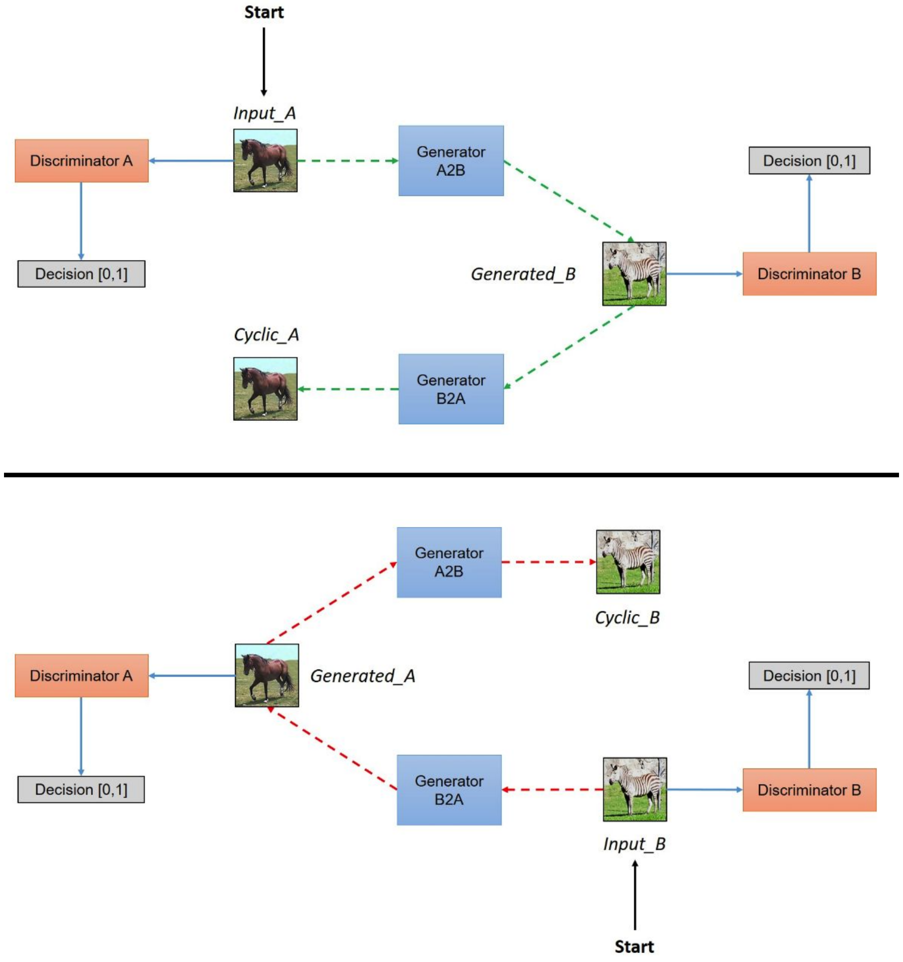
# Building GANs

1. When building GANs for generating images, train both generator and discriminator simultaneously.
2. After training, discard discriminator, as it has now trained the generator and was used for the same purpose
3. For semi-supervised learning, train the discriminator as a multi-class classifier on the given small labelled dataset.
4. At this time, the generator is discarded as it was used only for training the discriminator

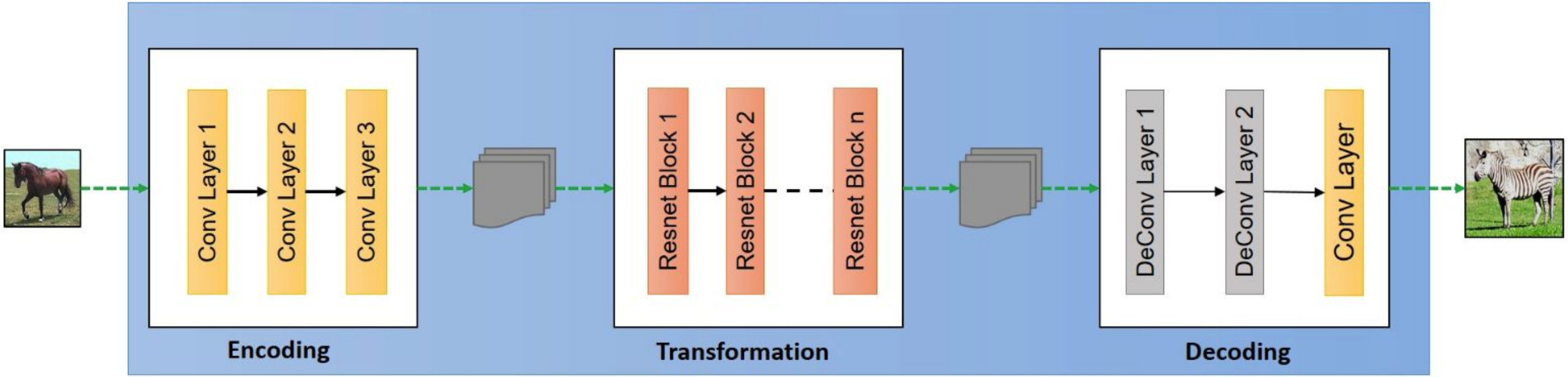
# Semi-supervised learning

1. The generator creates unlabeled data
2. This is fed to the discriminator
3. It trains on this unlabeled data
4. This data is used for improving discriminator performance
5. For regular image generation from GAN, the discriminator has the role of computing probability of the image being true or false
6. For discriminator to act as a semi-supervised classifier, it has to learn the probabilities of each of the original classes

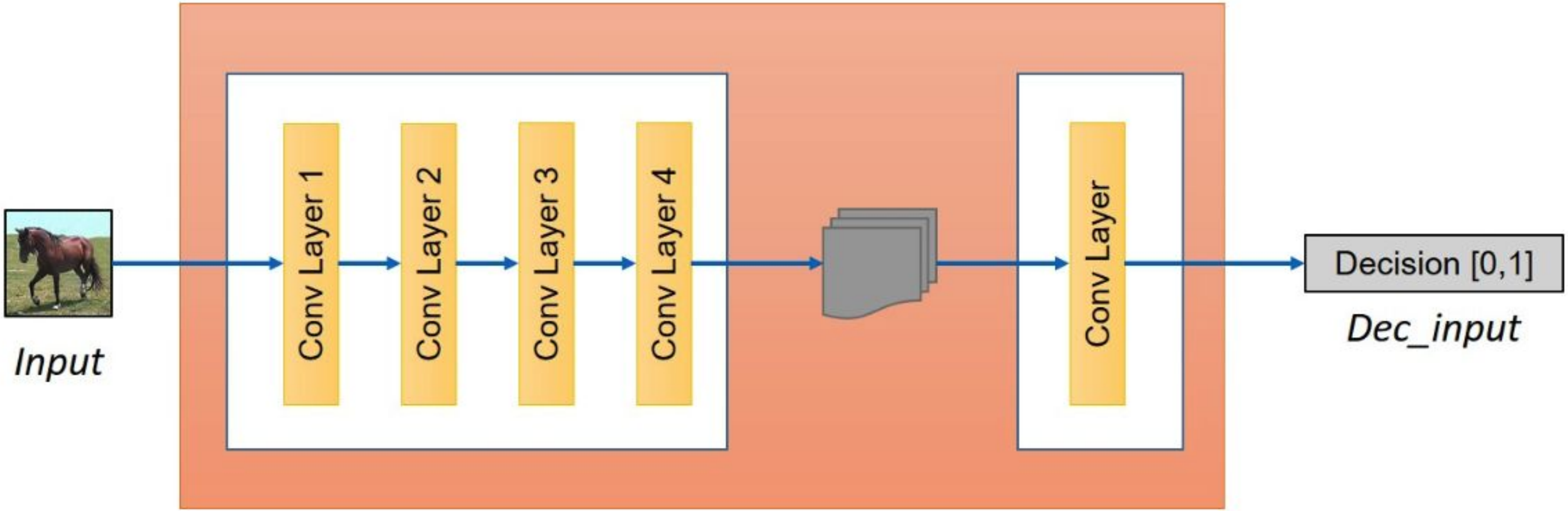
# Network architecture



# Building the generator



# Building the discriminator



# Loss

1. Discriminator loss
2. Generator loss
3. Cyclic loss
4. All together

# Evaluation metric

The FCN metric evaluates how interpretable the generated photos are according to an off-the-shelf semantic segmentation algorithm