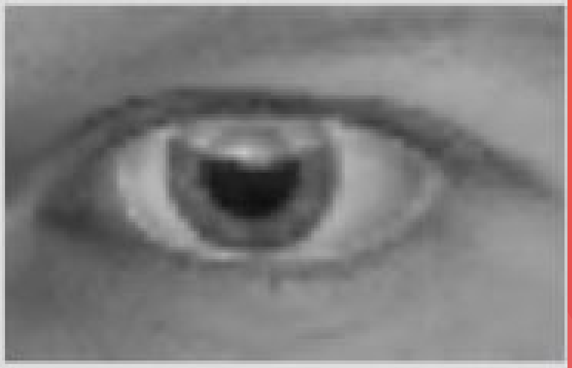
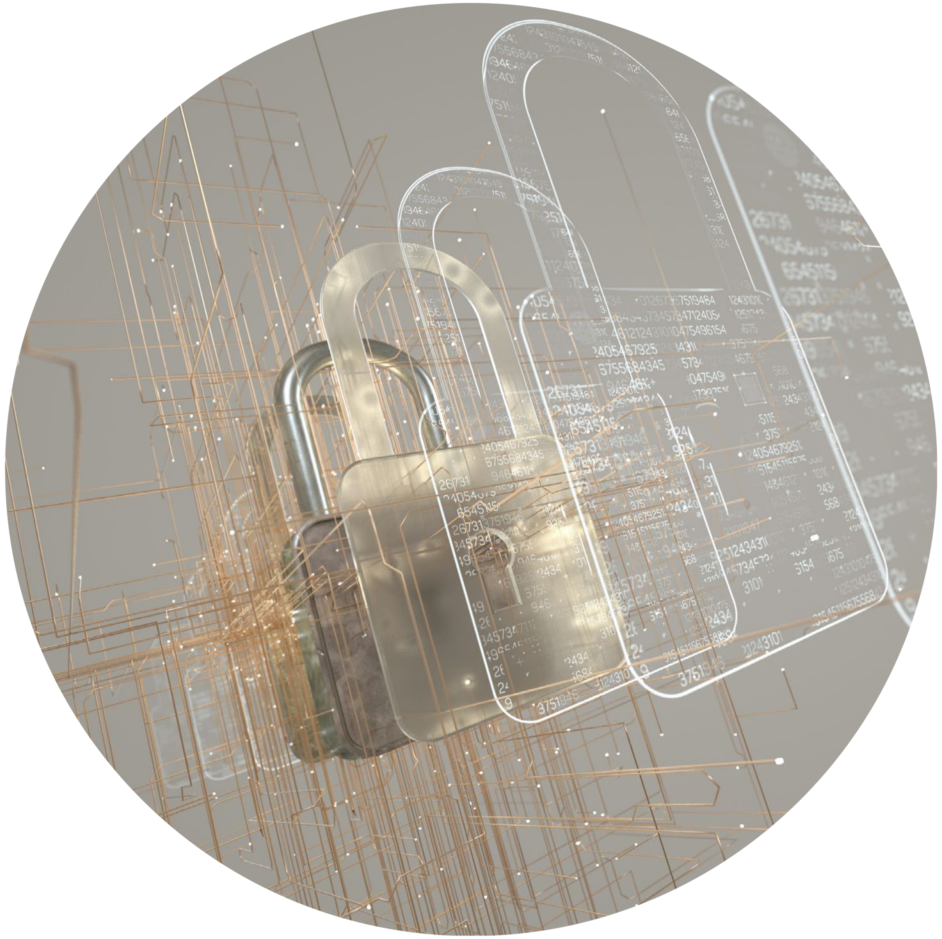


# LEARNING FROM SIMULATED DATA

# BACKGROUND

- Because of the size of many image datasets…



- … and the availability of better modern graphics hardware,

- Researchers at Apple have proposed a simulated and unsupervised training of images using adversarial networks[1].
- The goal of this process is to avoid the need for expensive and time-consuming annotation.

# MOTIVATION

- Large labeled datasets are increasingly needed because of the rise of deep neural networks.
- Labeling such datasets is expensive and time consuming.
- Annotation can be automated by creating simulated datasets.

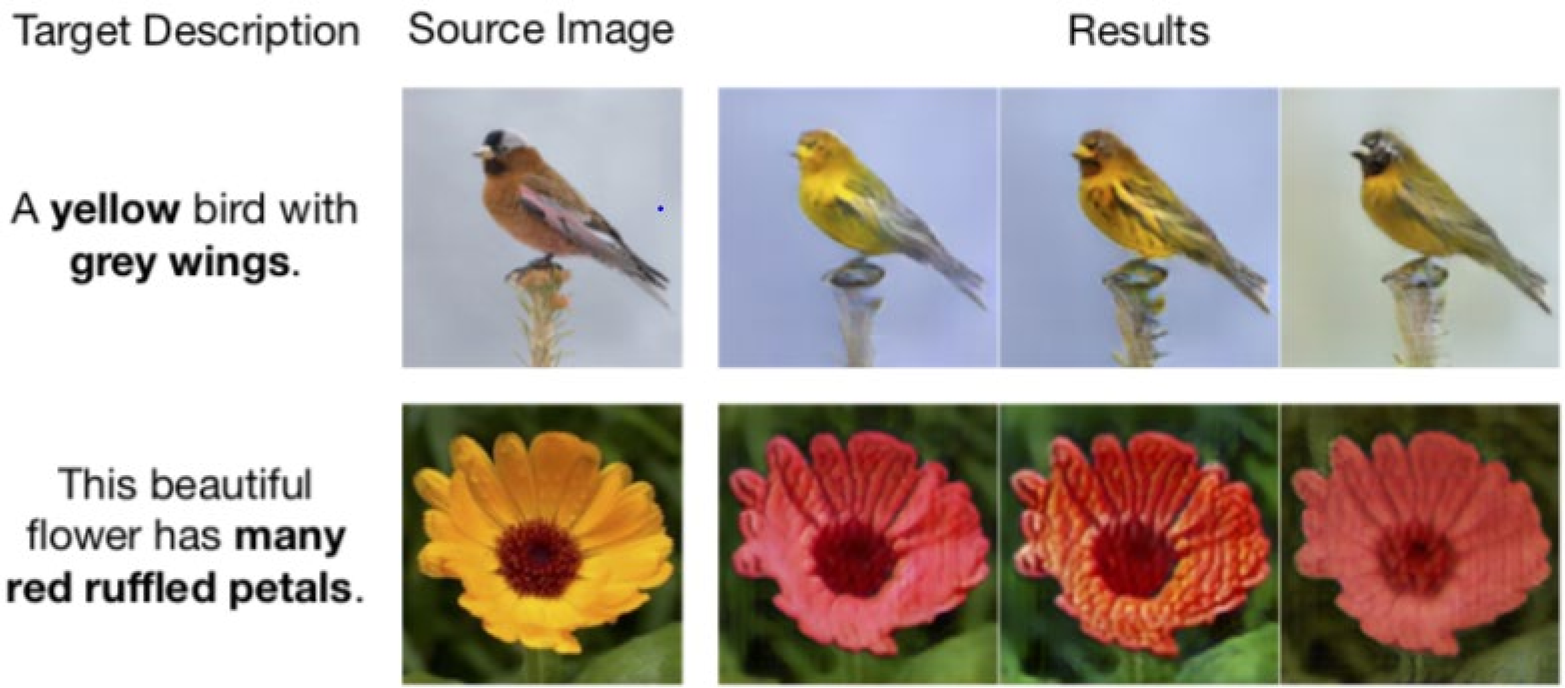
# THE PROBLEM

- Often, training with simulated data does not achieve desired performance standards because there is generally a large gap between the distributions found in simulated image data, and those found in real image data.

# ADDITIONALLY

GAN'S IN GENERAL OFTEN CREATE ARTIFACTS…..

# RELATED[2]

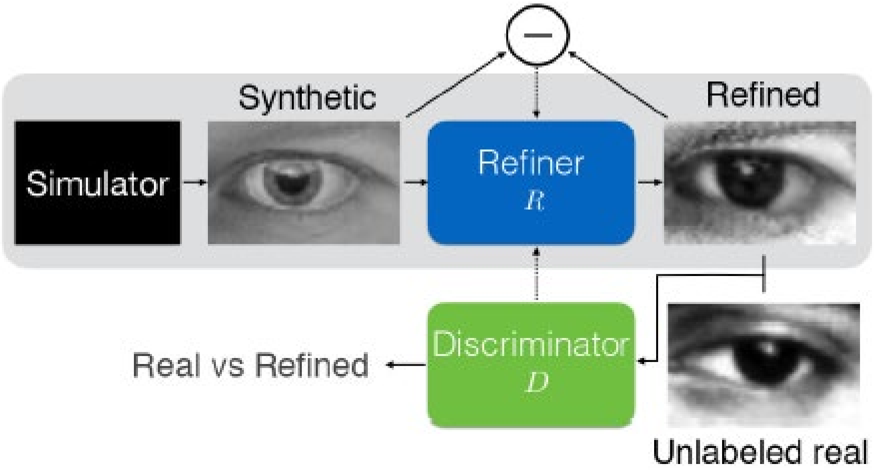

Target Description
Source Image
Results
A yellow bird with grey wings.
This beautiful flower has many red ruffled petals.

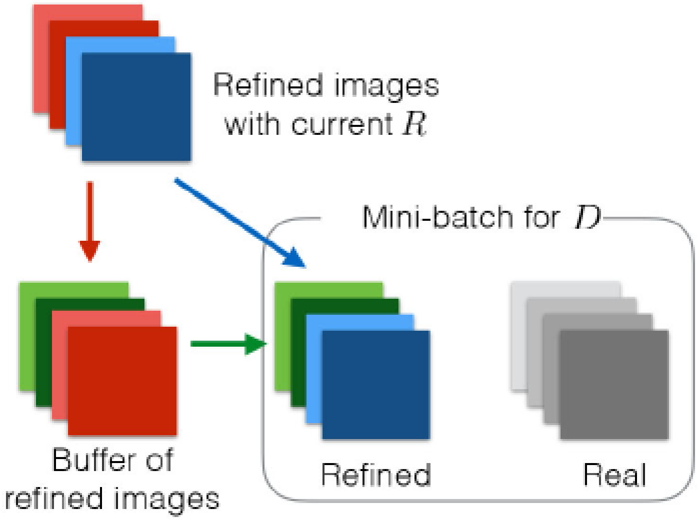
# THE METHOD

- Data can be both simulated and annotated using graphically powerful game engines such as UNITY. Specifically in this study, the UNITY Eyes application.

# LET ME EXPLAIN….

- The simulator (UNITY) creates the simulated image.
- The image is sent to the Refiner ( R) and treated with a set of weighted losses to more closely match real images.
- Small batches of the refined images are sent 1. to a buffer and 2. to the discriminator.






- The Descriminator (D) gets a partial batch from the buffer and a second partial batch from the currently refined images.
- This process is used to avoid spiking of the gradients between steps.

$$\tilde{\mathbf{x}} := R_{\boldsymbol{\theta}}(\mathbf{x})$$

# THE MATH PART 1 THE REFINER

- The key is for the refined image to look like a real one.
- In the first (lreal) part ~x corresponds to the refined image, while x is the raw simulated image. Theta is learned by minimizing the combination of losses.

$$\mathcal{L}_R(\boldsymbol{\theta}) = \sum_i \ell_{\text{real}}(\boldsymbol{\theta}; \tilde{\mathbf{x}}_i, \mathcal{Y}) + \lambda \ell_{\text{reg}}(\boldsymbol{\theta}; \tilde{\mathbf{x}}_i, \mathbf{x}_i),$$

$$\tilde{\mathbf{x}} := R_{\boldsymbol{\theta}}(\mathbf{x})$$

# THE REFINER CONTINUED

- In the second part preserves the annotation information by minimizing the differences between simulated and refined images.
- Lambda value is added here is to avoid unwanted artifacts.
- (Those can be scary)

$$\mathcal{L}_R(\boldsymbol{\theta}) = \sum_i \ell_{\text{real}}(\boldsymbol{\theta}; \tilde{\mathbf{x}}_i, \mathcal{Y}) + \lambda \ell_{\text{reg}}(\boldsymbol{\theta}; \tilde{\mathbf{x}}_i, \mathbf{x}_i),$$

# THE MATH PART 2: THE DISCRIMINATOR

- Basically cross-entropy for a two-class classification problem.
- D is the probability of an image being simulated. And 1-D, that of being a real one.
- Phi is updated with SGD for the batch.

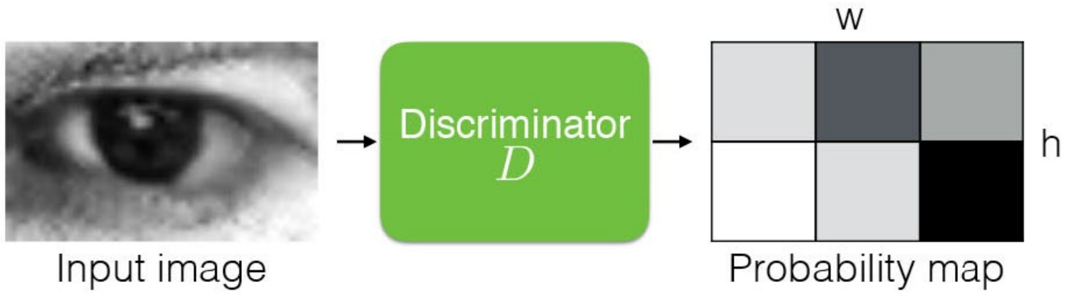
$$\mathcal{L}_D(\phi) = -\sum_i \log(D_\phi(\tilde{\mathbf{x}}_i)) - \sum_j \log(1 - D_\phi(\mathbf{y}_j)).$$

# THE MATH PART 3: REALISM LOSS UPDATED (DISCRIMINATOR FOOLED)

$$\ell_{\text{real}}(\boldsymbol{\theta}; \tilde{\mathbf{x}}_i, \mathcal{Y}) = -\sum_i \log(1 - D_{\boldsymbol{\phi}}(R_{\boldsymbol{\theta}}(\mathbf{x}_i)))$$

- By minimizing the real loss function (from part 1), the Discriminator is fooled into classifying a simulated image as real.

# AS A SIDE NOTE:




- Images are discriminated in sections, basically in groups of pixels in a method like patchGAN.
- Each section of the image is classified as either simulated or real.

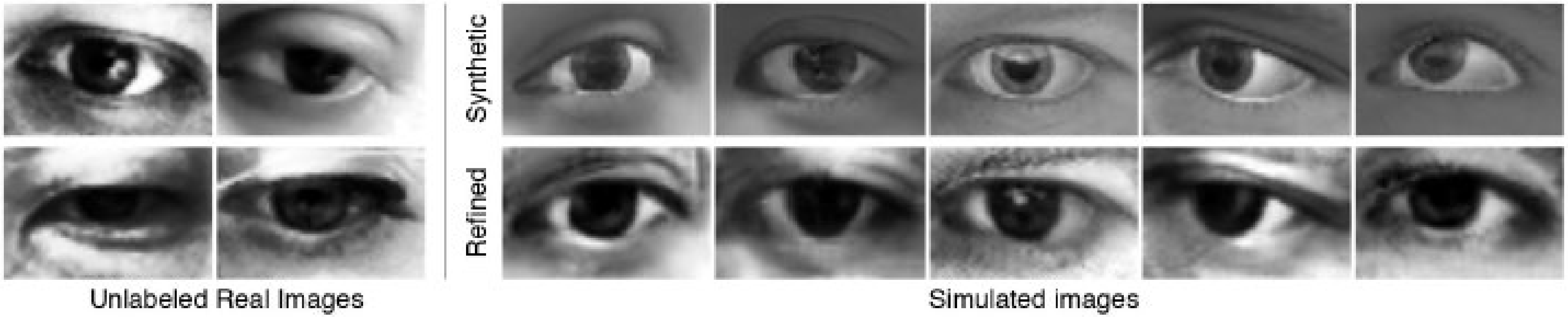
# THE RESULTING ALGORITHM

**Algorithm 1:** Adversarial training of refiner network $R_{\boldsymbol{\theta}}$

**Input**: Sets of synthetic images $\mathbf{x}_i \in \mathcal{X}$ , and real images $\mathbf{y}_j \in \mathcal{Y}$, max number of steps $(T)$, number of discriminator network updates per step $(K_d)$, number of generative network updates per step $(K_g)$.
**Output**: ConvNet model $R_{\boldsymbol{\theta}}$.
**for** $t = 1, \ldots, T$ **do**
  **for** $k = 1, \ldots, K_g$ **do**
    1. Sample a mini-batch of synthetic images $\mathbf{x}_i$.
    2. Update $\boldsymbol{\theta}$ by taking a SGD step on mini-batch loss $\mathcal{L}_R(\boldsymbol{\theta})$ in (4) .
  **end**
  **for** $k = 1, \ldots, K_d$ **do**
    1. Sample a mini-batch of synthetic images $\mathbf{x}_i$, and real images $\mathbf{y}_j$.
    2. Compute $\tilde{\mathbf{x}}_i = R_{\boldsymbol{\theta}}(\mathbf{x}_i)$ with current $\boldsymbol{\theta}$.
    3. Update $\boldsymbol{\phi}$ by taking a SGD step on mini-batch loss $\mathcal{L}_D(\boldsymbol{\phi})$ in (2).
  **end**
**end**

# SIDE NOTE 2

- A Visual Touring Test was given to evaluate the quality of the images from a human perspective.
- Without labels could you tell?



# MY REPLICATION (WITH THE HELP OF KAGGLE)

https://www.kaggle.com/soundpoet/simgan-implementation-using-tensorflow-keras

## Loss Functions

```
In [26]:  def self_regularisation_loss(y_true, y_pred):
              return tf.multiply(0.0002, tf.reduce_sum(tf.abs(y_pred - y_true)))

          # reduce_sum: Computes the sum of elements across dimensions of a tensor.
```

```
In [27]:  def local_adversarial_loss(y_true, y_pred):
              truth = tf.reshape(y_true, (-1, 2))
              predicted = tf.reshape(y_pred, (-1, 2))

              computed_loss = tf.nn.softmax_cross_entropy_with_logits(labels=truth, logits=predicted)
              output = tf.reduce_mean(computed_loss)

              return output
```

```
#Refiner
```

In [28]:

```python
def refiner_model(width = 55, height = 35, channels = 1):
    """
    The refiner network, Rθ, is a residual network (ResNet). It modifies the synthetic image on a pixel level
    than holistically modifying the image content, preserving the global structure and annotations.

    :param input_image_tensor: Input tensor that corresponds to a synthetic image.
    :return: Output tensor that corresponds to a refined synthetic image.
    """

    def resnet_block(input_features, nb_features=64, kernel_size=3):
        """
        A ResNet block with two `kernel_size` x `kernel_size` convolutional layers,
        each with `nb_features` feature maps.

        See Figure 6 in https://arxiv.org/pdf/1612.07828v1.pdf.

        :param input_features: Input tensor to ResNet block.
        :return: Output tensor from ResNet block.
        """
        y = Conv2D(nb_features, kernel_size=kernel_size, padding='same')(input_features)
        y = Activation('relu')(y)
        y = Conv2D(nb_features, kernel_size=kernel_size, padding='same')(y)

        y = Add()([y, input_features])
        y = Activation('relu')(y)

        return y

    input_layer = Input(shape=(height, width, channels))
    # an input image of size w × h is convolved with 3 × 3 filters that output 64 feature maps
    x = Conv2D(64, kernel_size=3, padding='same', activation='relu')(input_layer)

    for _ in range(4):
        x = resnet_block(x)

    output_layer = Conv2D(channels, kernel_size=1, padding='same', activation='tanh')(x)

    return Model(input_layer, output_layer, name='refiner')
```

## Discriminator

In [29]:

```
def discriminator_model(width = 55, height = 35, channels = 1):
    input_layer = Input(shape=(height, width, channels))

    x = Conv2D(96, kernel_size=3, strides=2, padding='same', activation='relu')(input_layer)
    x = Conv2D(64, kernel_size=3, strides=2, padding='same', activation='relu')(x)
    x = MaxPooling2D(pool_size=3, strides=1, padding='same')(x)
    x = Conv2D(32, kernel_size=3, strides=1, padding='same', activation='relu')(x)
    x = Conv2D(32, kernel_size=1, strides=1, padding='same', activation='relu')(x)
    x = Conv2D(2, kernel_size=1, strides=1, padding='same', activation='relu')(x)
    output_layer = Reshape(target_shape=(-1, 2))(x)

    return Model(input_layer, output_layer, name='discriminator')
```

```python
class ImageHistoryBuffer():
    def __init__(self, shape, max_size, batch_size):
        """
        :param shape: Shape of the data to be stored in the image history buffer
                      (i.e. (0, img_height, img_width, img_channels)).
        :param max_size: Maximum number of images that can be stored in the image history buffer.
        :param batch_size: Batch size used to train GAN.
        """
        self.image_history_buffer = np.zeros(shape=shape)
        self.max_size = max_size
        self.batch_size = batch_size

    def add_to_history_img_buffer(self, images, nb_to_add=0):
        if not nb_to_add:
            nb_to_add = self.batch_size // 2

        if len(self.image_history_buffer) < self.max_size:
            np.append(self.image_history_buffer, images[:nb_to_add], axis=0)
        elif len(self.image_history_buffer) == self.max_size:
            self.image_history_buffer[:nb_to_add] = images[:nb_to_add]
        else:
            assert False

        np.random.shuffle(self.image_history_buffer)

    def get_from_image_history_buffer(self, nb_to_get=None):
        """
        Get a random sample of images from the history buffer.

        :param nb_to_get: Number of images to get from the image history buffer (batch_size / 2 by default).
        :return: A random sample of `nb_to_get` images from the image history buffer, or an empty np array if
                 history buffer is empty.
        """
        if not nb_to_get:
            nb_to_get = self.batch_size // 2

        try:
            return self.image_history_buffer[:nb_to_get]
        except IndexError:
            return np.zeros(shape=0)
```

## Data Generators

In [63]:
```
datagen = image.ImageDataGenerator(preprocessing_function=applications.xception.preprocess_input, data_format
```

In [64]:
```
syn_gen = datagen.flow(x=syn_img_stack, batch_size=batch_size)
real_gen = datagen.flow(x=real_img_stack, batch_size=batch_size)
```

In [65]:
```
def get_image_batch(generator):
    """keras generators may generate an incomplete batch for the last batch"""
    img_batch = generator.next()
    if len(img_batch) != batch_size:
        img_batch = generator.next()

    assert len(img_batch) == batch_size

    return img_batch
```

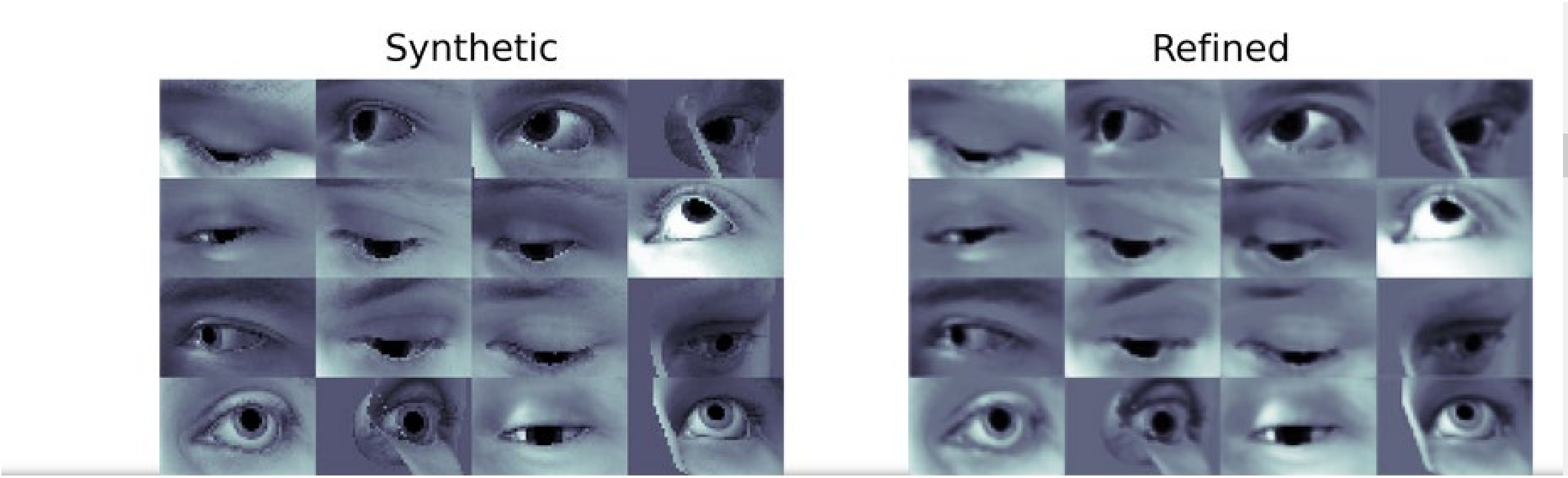
In [66]:
```
disc_output_shape = disc.output_shape
```

In [67]:
```
y_real = np.array([[[1.0, 0.0]] * disc_output_shape[1]] * batch_size)
y_refined = np.array([[[0.0, 1.0]] * disc_output_shape[1]] * batch_size)

assert y_real.shape == (batch_size, disc_output_shape[1], 2)
assert y_refined.shape == (batch_size, disc_output_shape[1], 2)

batch_out = get_image_batch(syn_gen)
assert batch_out.shape == (batch_size, img_height, img_width, channels), "Image dimension do not match, {} !=
    .format(batch_out.shape, (batch_size, img_height, img_width, img_channels))
```

# PRETRAINING (MY RESULTS)



## References

Shrivastava, A. P. (2016). Learning from Simulated and Unsupervised Images using Adversarial Networks . *ARXIV*.

Yu, S., Dong, H., Liang, F., Mo, Y., Wu, C., & Guo, Y. (2019). SimGAN: Photo-Realistic Semantic Image Manipulation Using Generative Adversarial Networks. *ICIP*.