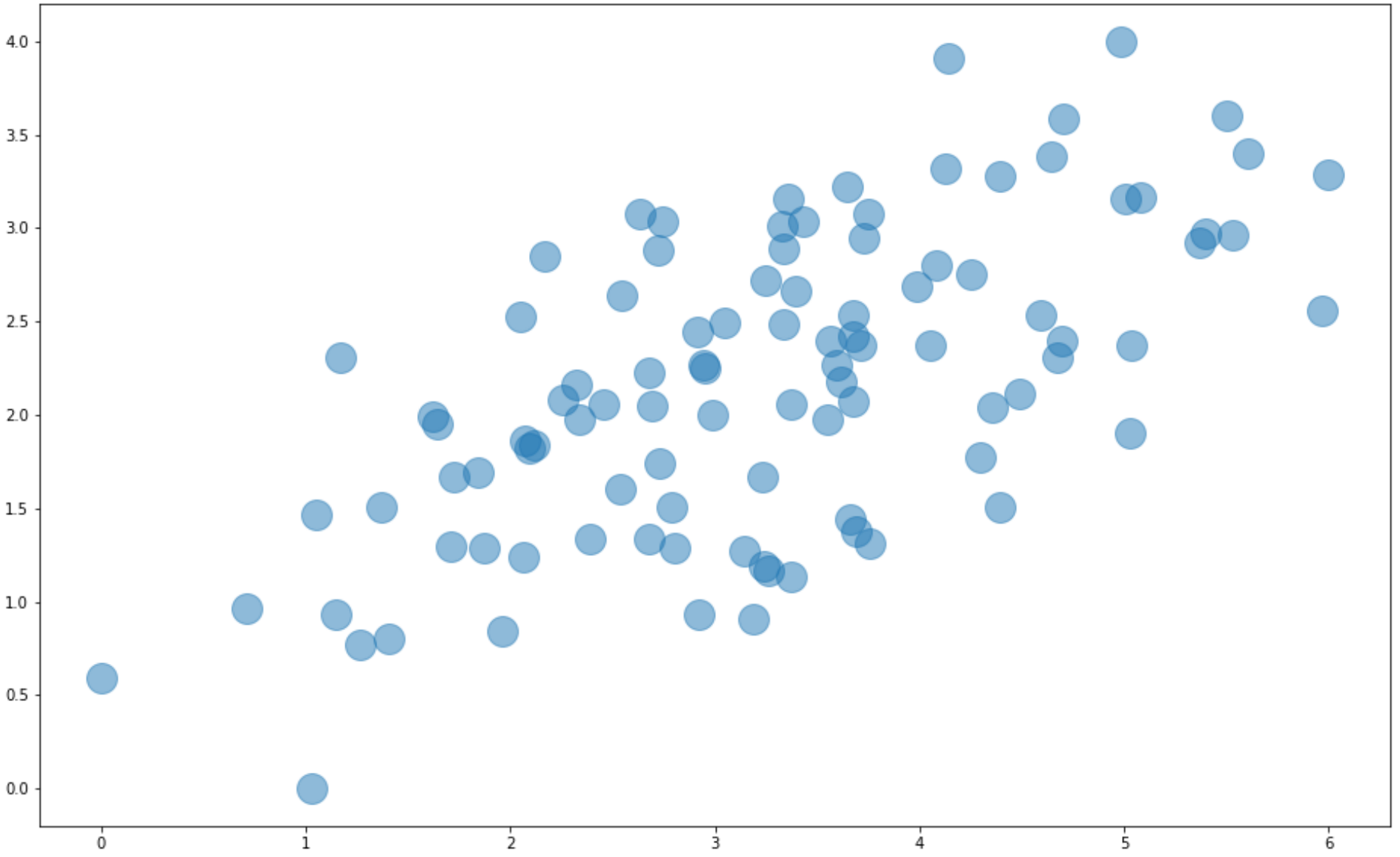
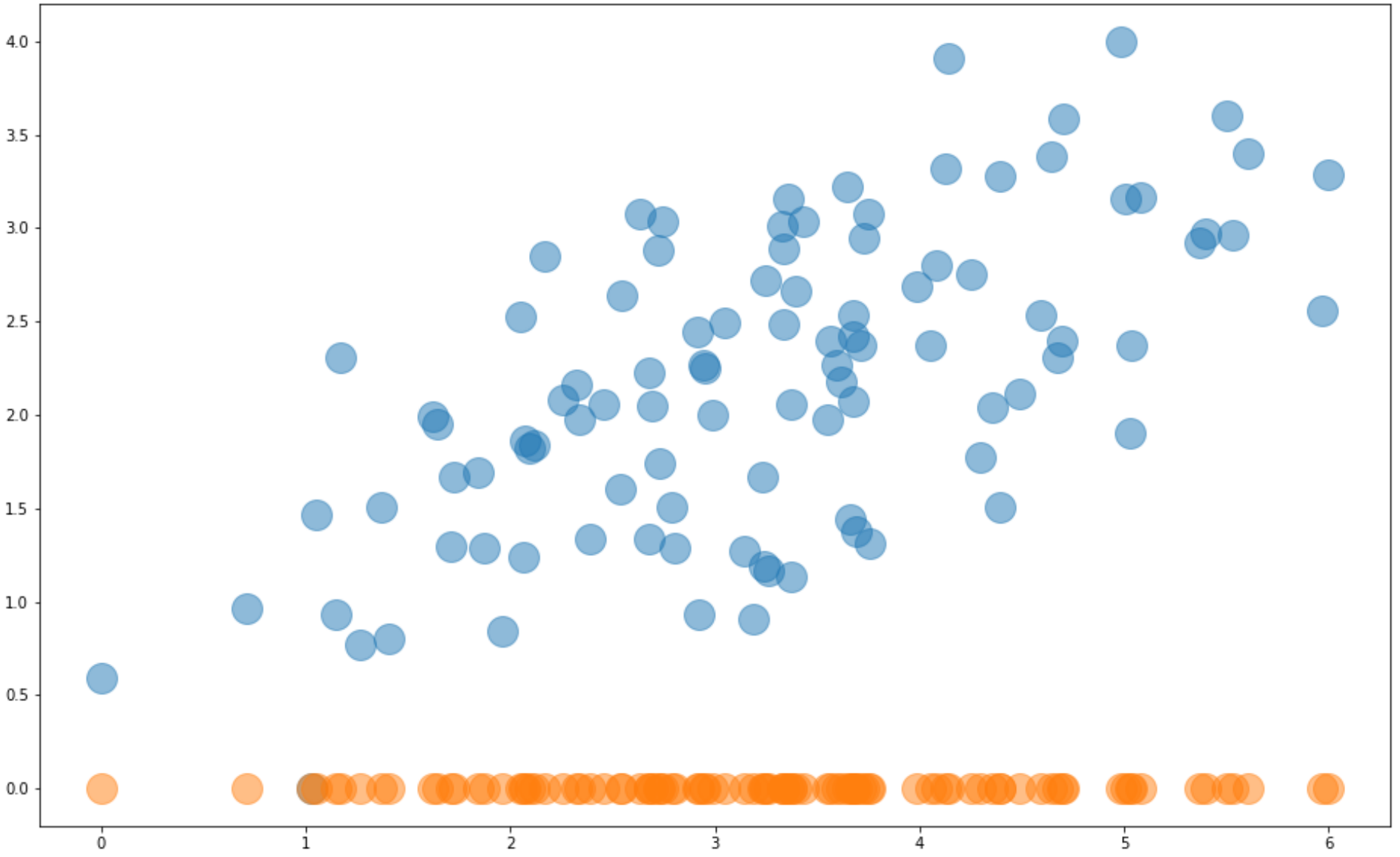
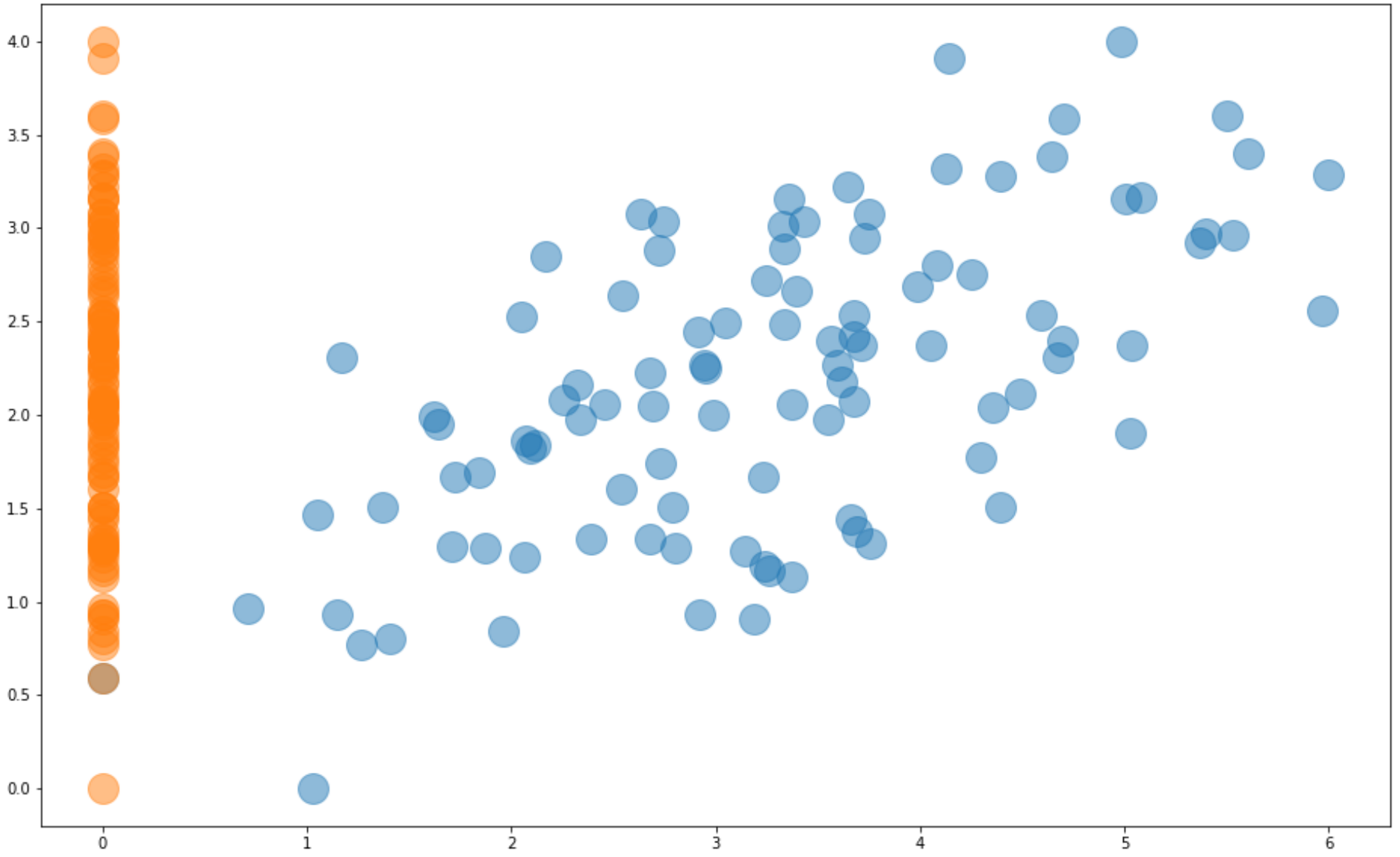
Image Generation

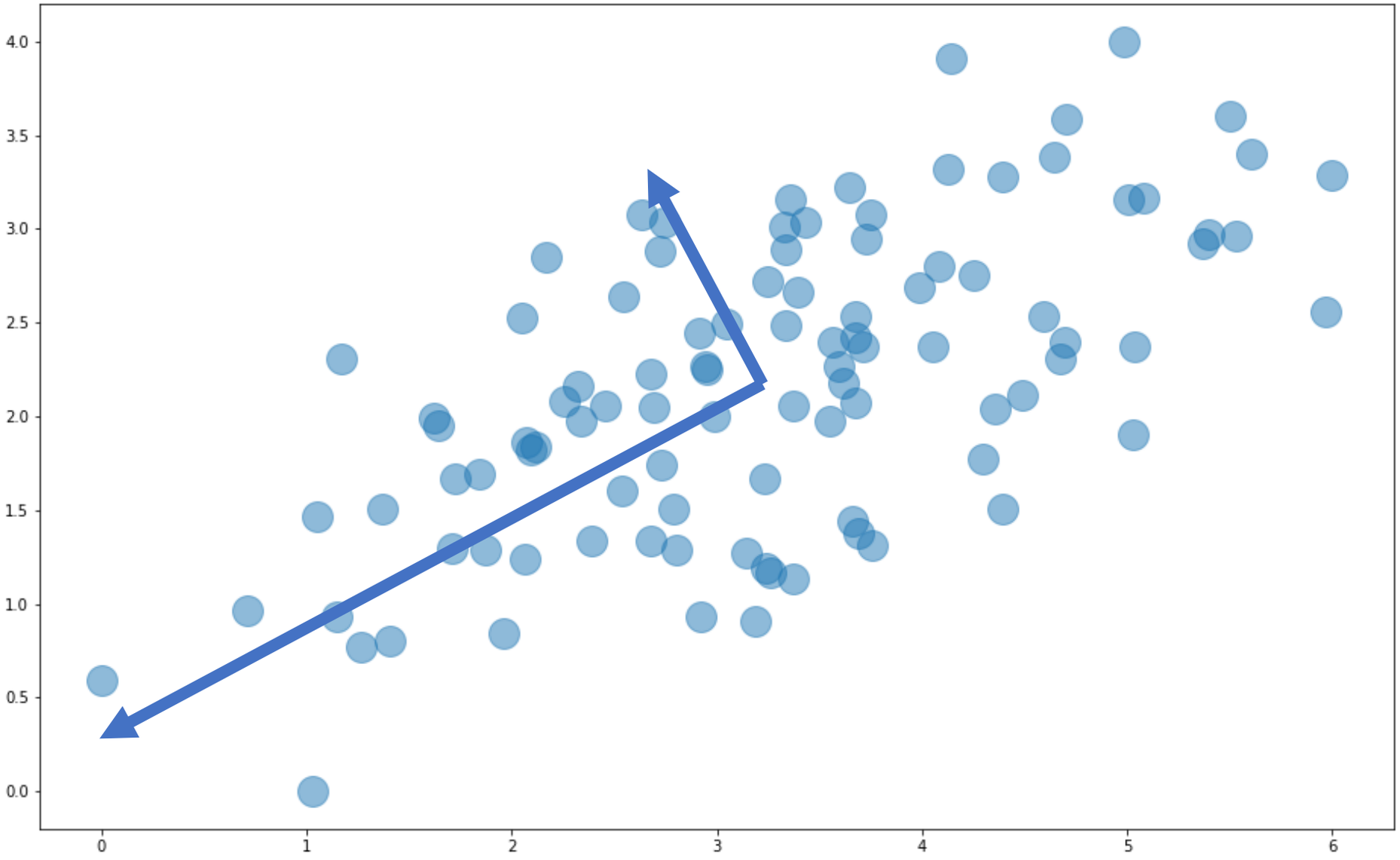
Jeff Prosise

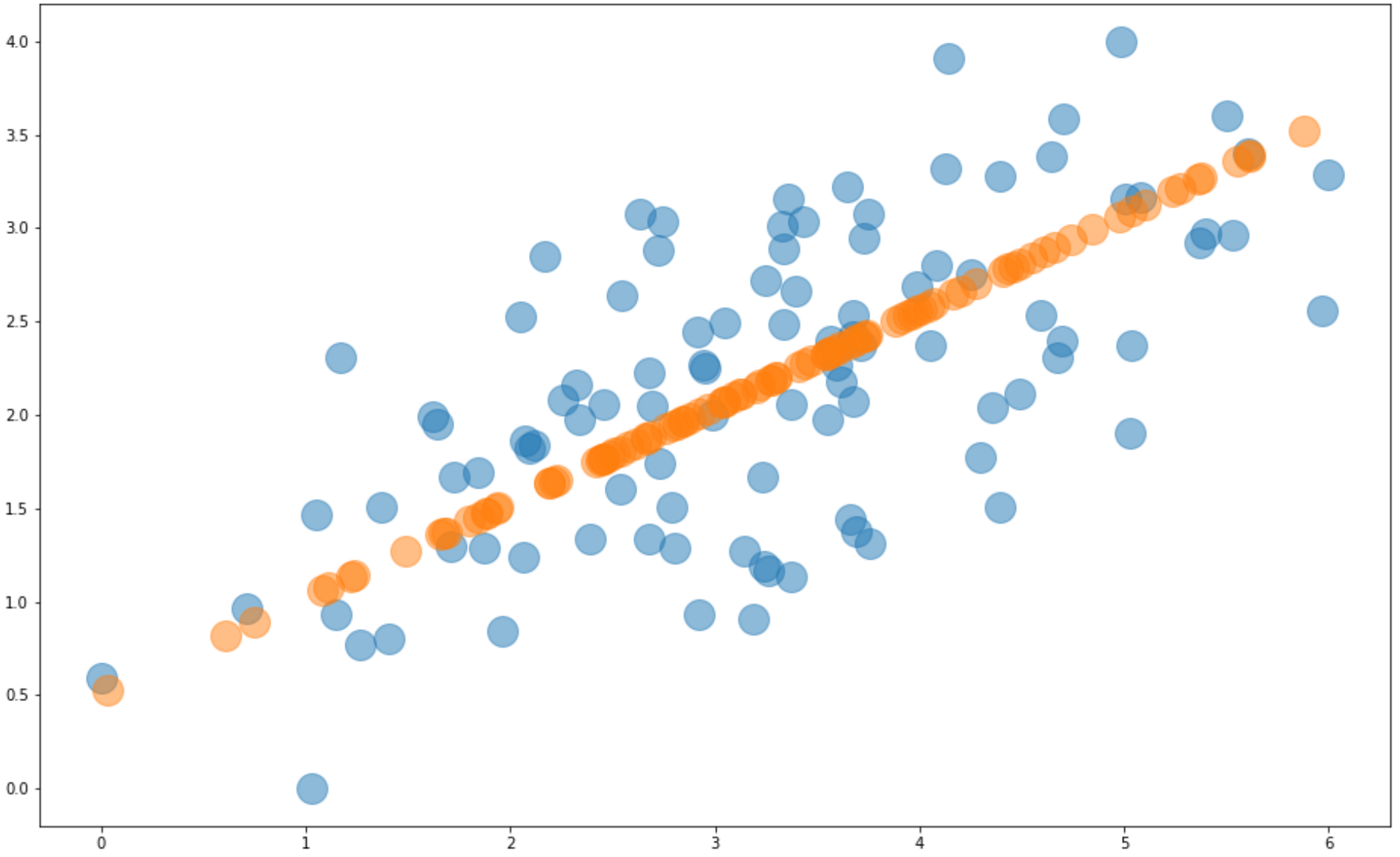
@jprosise


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
5
6


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
5
6


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
5
6


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
5
6

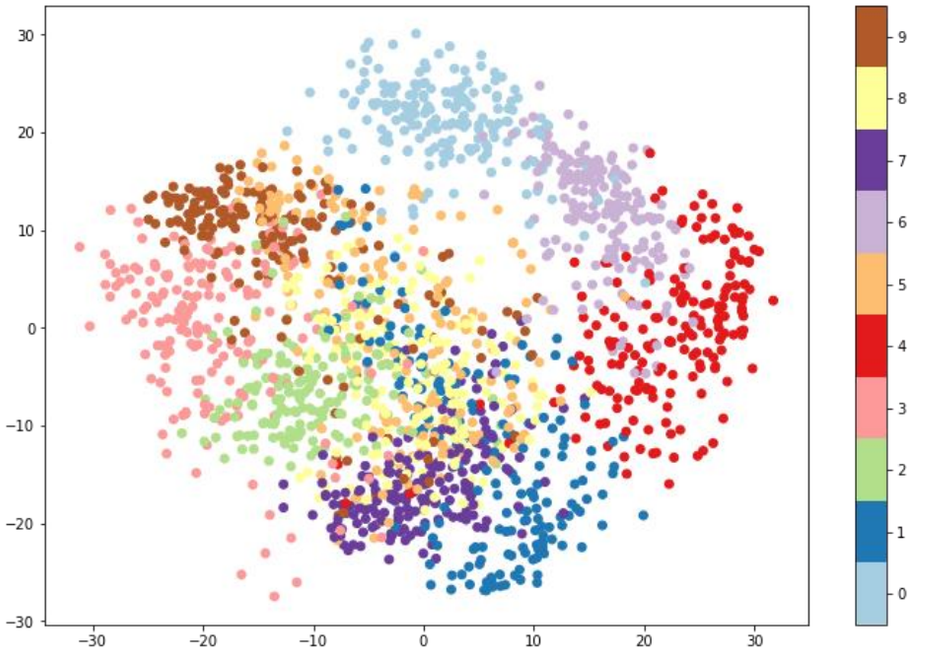

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
5
6

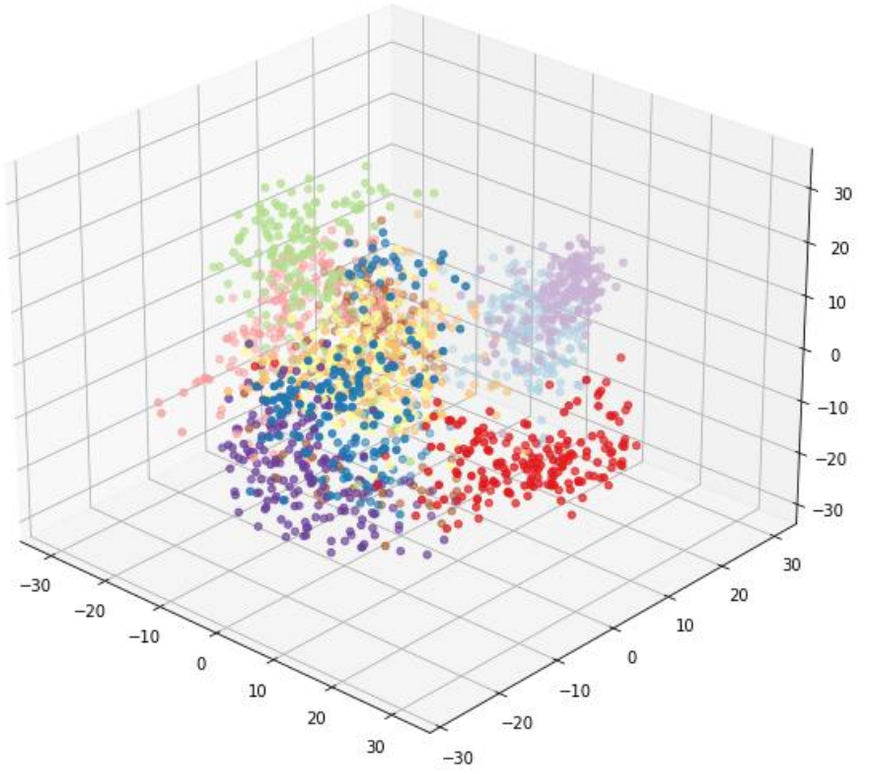
Principal Component Analysis (PCA)

- Commonly used dimensionality-reduction algorithm
  - Reduces number of dimensions without commensurate loss of information
    - Example: Reduce number of dimensions by 90% while retaining 90% of the information
  - Works best with dense data (fewer zeroes); use other algorithms such as Singular Value Decomposition (SVD) for sparse datasets
- Applications include increasing samples/dimensions ratio for small datasets, obfuscating data, filtering noise, eliminating multicollinearity, eliminating irrelevant features, reducing data to 2 or 3 dimensions for plotting and visualization, and anomaly detection
- Scale of all dimensions should be the same before applying PCA

Visualizing High-Dimensional Data

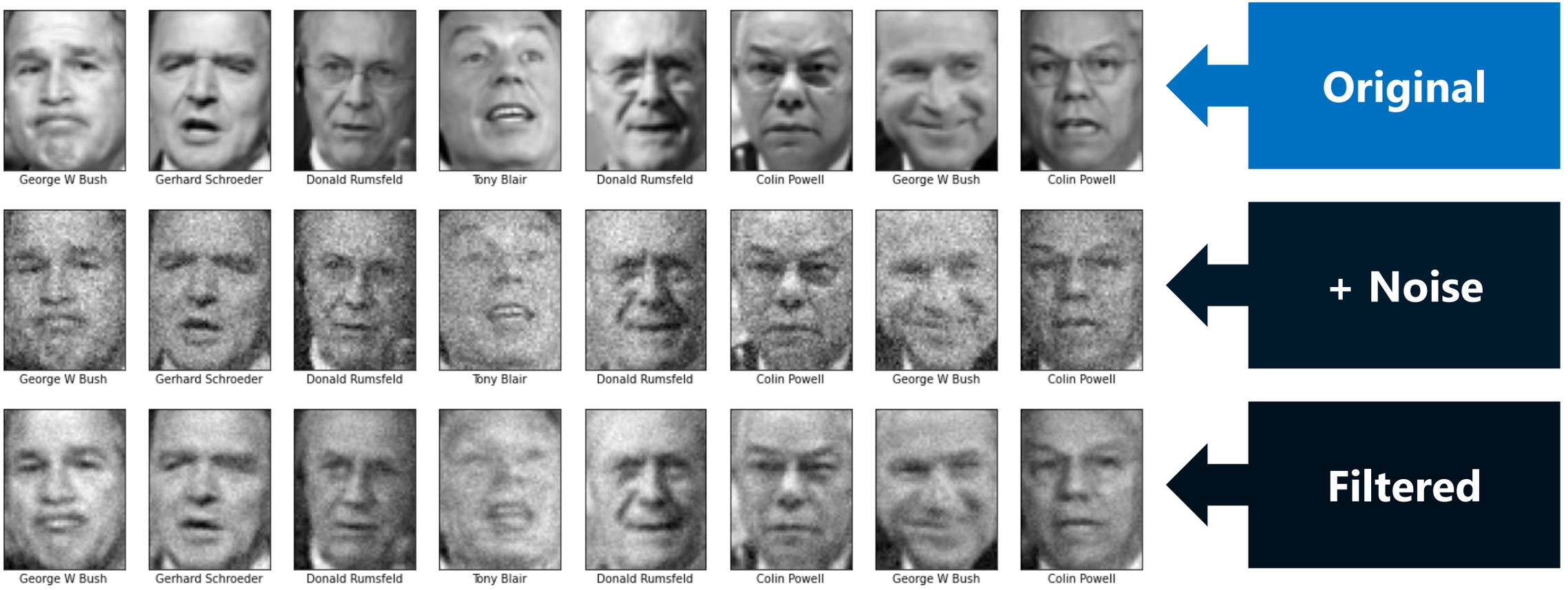
- Humans can't visualize data in more than three dimensions
- Solution: "Squeeze" data down to two or three dimensions and plot it





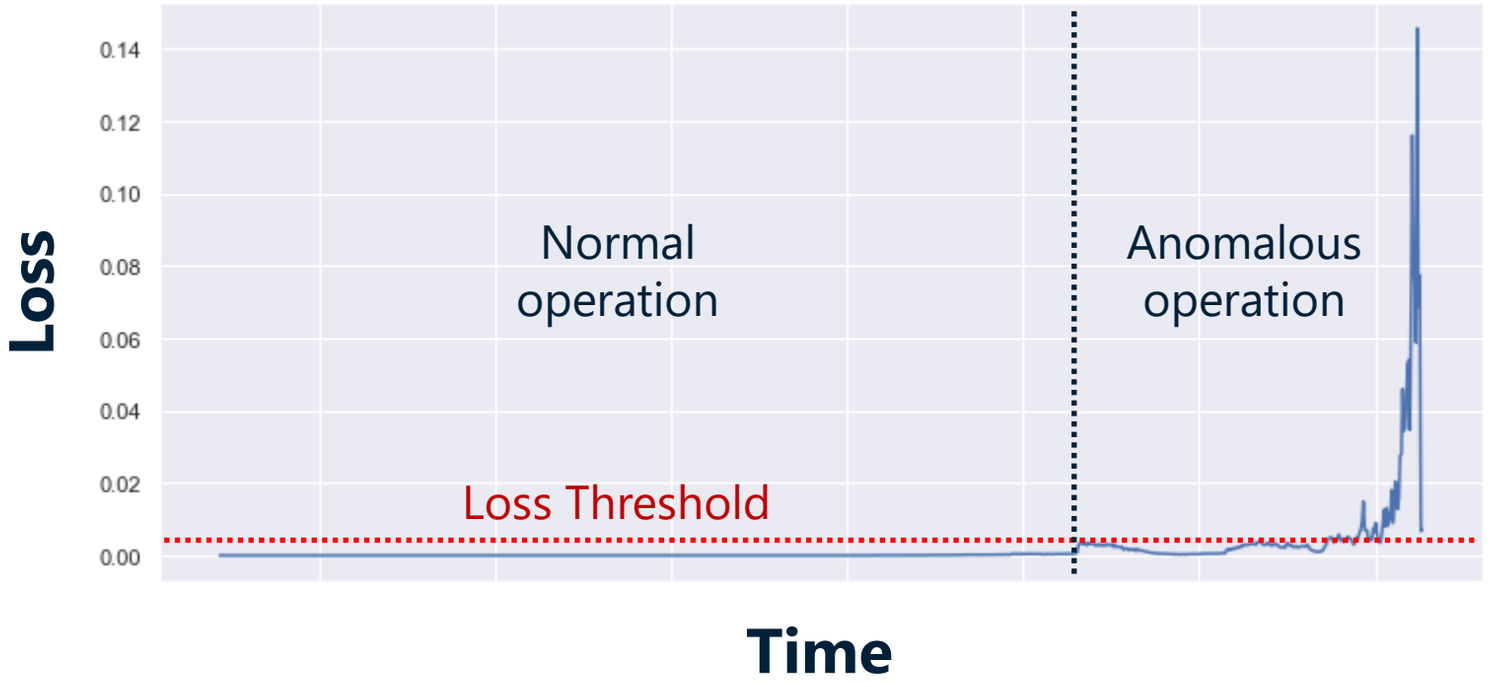
Filtering Noise

- Discard 10%-20% of the data, and then reverse the transform



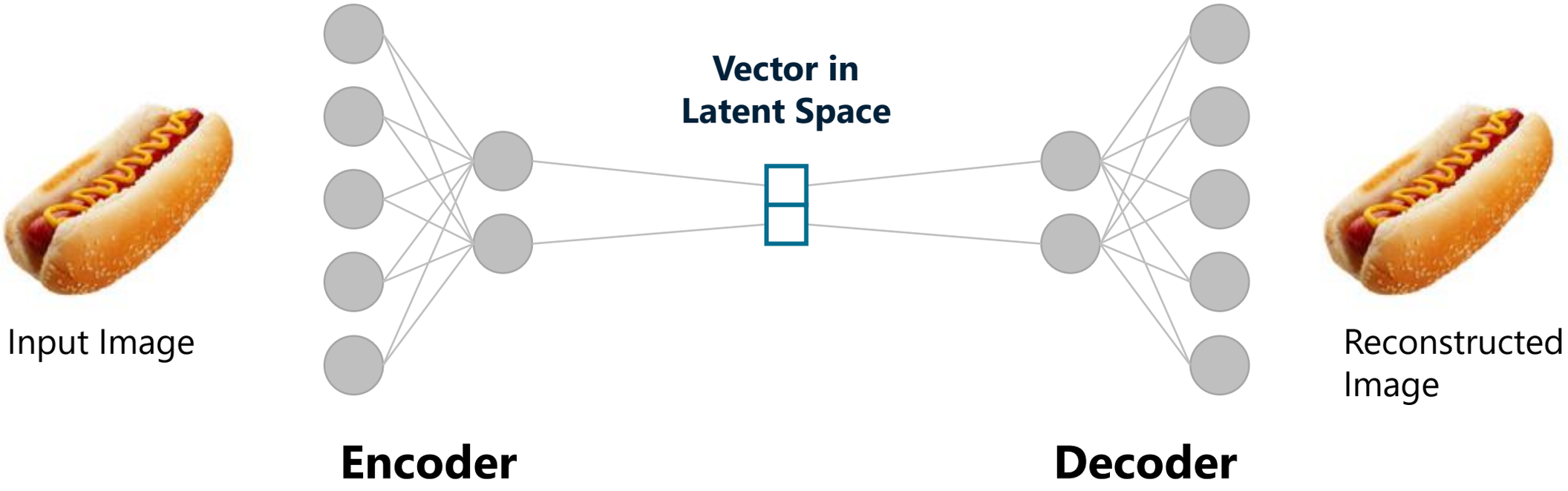
Detecting Anomalies

- Collapse signals to one dimension
- Invert the transform and measure reconstruction error (loss)



Autoencoders

- Use encoders to reduce inputs to lower-dimensional "latent space" and decoders to reconstruct the original inputs from latent space



Implementing an Autoencoder

```
encoder = Sequential()
encoder.add(Flatten()) # Reshape 28x28 images to 1D
encoder.add(Dense(128, activation='relu'))
encoder.add(Dense(32, activation='relu'))

decoder = Sequential()
decoder.add(Dense(128, activation='relu'))
decoder.add(Dense(28 * 28, activation='relu'))
decoder.add(Reshape((28, 28))) # Reshape 1D array into 28x28 image

model = Sequential([encoder, decoder])
model.compile(loss='mse', optimizer='adam')
model.fit(x_train, x_train, epochs=10, validation_data=(x_test, x_test))
```

Implementing a Convolutional Autoencoder

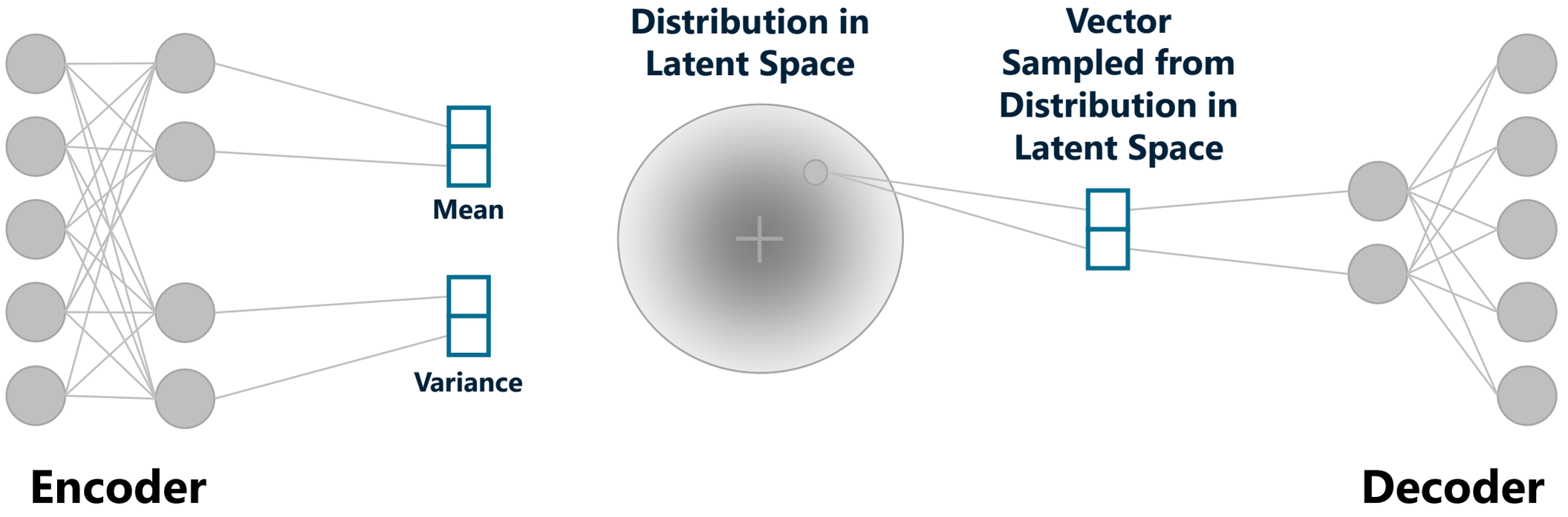
```
encoder = Sequential()
encoder.add(Conv2D(16, (3, 3), activation='relu', strides=2, padding='same',
                   input_shape=(28, 28, 1)))
encoder.add(Conv2D(32, (3, 3), activation='relu', strides=2, padding='same'))
encoder.add(GlobalAveragePooling2D())

decoder = Sequential()
decoder.add(Dense(7 * 7 * 16), activation='relu')
decoder.add(Reshape((7, 7, 16)))
decoder.add(Conv2DTranspose(32, (3, 3), strides=2, activation='relu', padding='same'))
decoder.add(Conv2DTranspose(1, (3, 3), strides=2, activation='relu', padding='same'))
decoder.add(Reshape((28, 28)))

model = Sequential([encoder, decoder])
model.compile(loss='mse', optimizer='adam')
model.fit(x_train, x_train, epochs=10, validation_data=(x_test, x_test))
```

Demo

Autoencoders

Variational Autoencoders (VAEs)

- Provide continuity in latent space by introducing randomness
- Encoder generates probabilistic distributions rather than points



Implementing a Variational Encoder

```
encoder_input = Input(shape=(28, 28))
x = Flatten()(encoder_input)
x = Dense(128, activation='relu')(x)
x = Dense(32, activation='relu')(x)
z_mean = Dense(8)(x) # Outputs vector representing mean in normal distribution
z_log_var = Dense(8)(x) # Outputs vector representing variance in normal distribution

encoder_output = Sampling()([z_mean, z_log_var]) # Custom layer with two inputs that
                                                 # outputs vector sampled from normal
                                                 # distribution defined by the inputs
encoder = Model(encoder_input, encoder_output)
```

Computing Loss

- Loss is the sum of reconstruction loss (MSE) and latent loss (Kullback-Liebler divergence between target distribution and actual distribution)

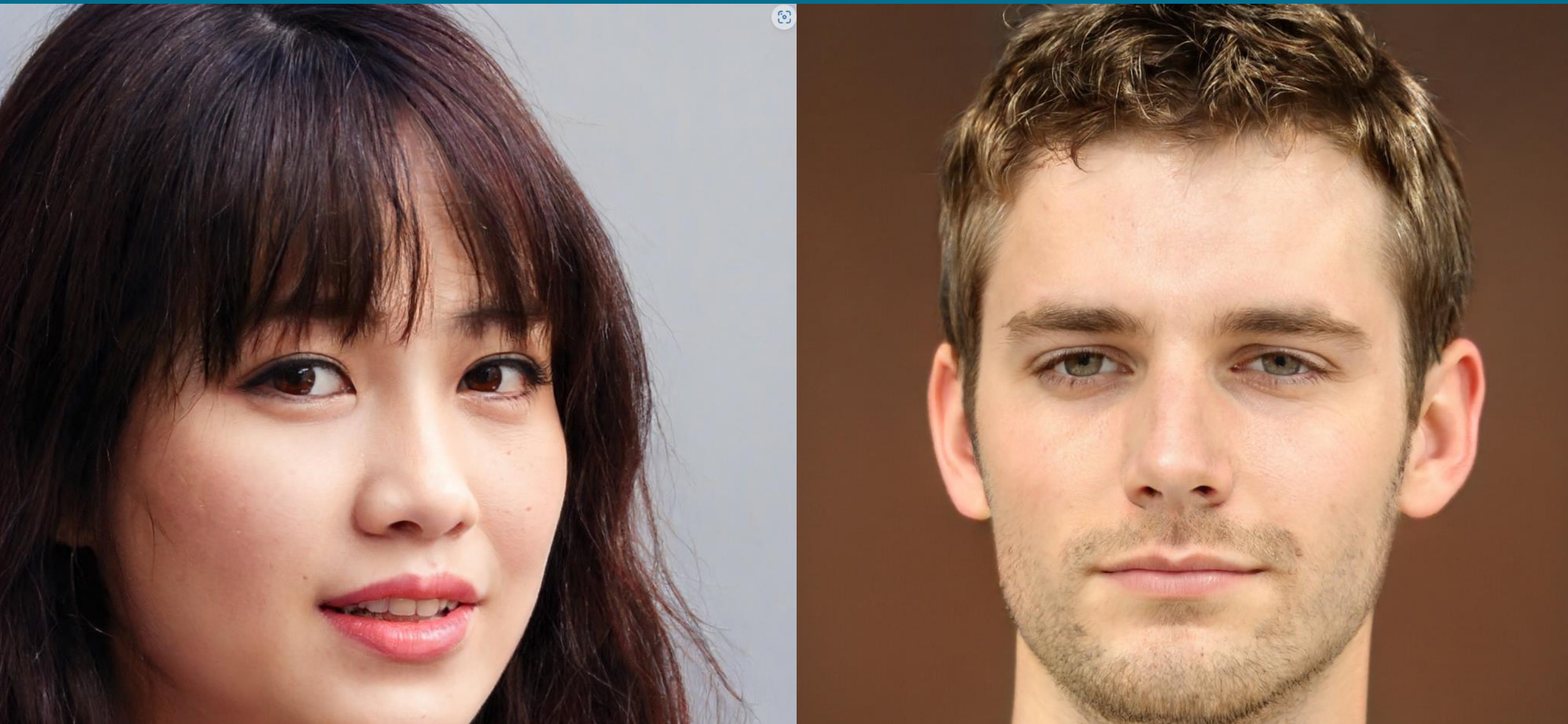
$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{n}\left[1 + \gamma_i - \exp\left(\gamma_i\right) - {\mu_i}^2\right]$$

```
model = Model(encoder_input, decoder(encoder(encoder_input)))
model.compile(loss='mse', optimizer='adam')
# Add a KL loss function to the model's loss computations
z_loss = -0.5 * tf.reduce_sum(1 + z_log_var - tf.exp(z_log_var) -
                              tf.square(z_mean), axis=-1)
model.add_loss(tf.reduce_mean(z_loss) / (28 * 28))
model.fit(x_train, x_train, epochs=10, validation_data=(x_test, x_test))
```

Demo

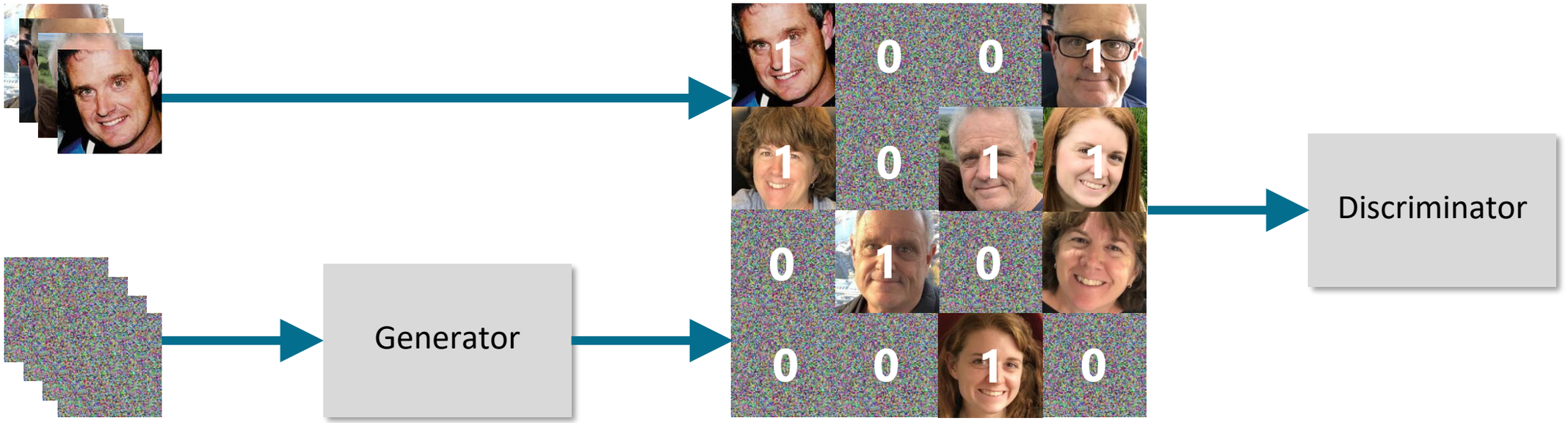
Variational Autoencoders

thispersondoesnotexist.com



Generative Adversarial Networks (GANs)

- Use one model (the *discriminator*) to train another model (the *generator*) to produce images from random inputs
- After training, use the generator to produce realistic images



Building and Training GANs

- Architect model to avoid mode collapse and instability
  - Use strided convolutions (strides=2) rather than pooling and upsampling layers in both the generator and the discriminator
  - Use batch normalization after all layers except for the generator's output layer and the discriminator's input layer
  - Use ReLU activation in the generator, except for the last layer, which should use **tanh** activation instead (requires preconditioning of training data)
  - Use leaky ReLU activation in the discriminator
- Implement custom training loop to train generator and discriminator separately in each batch of each epoch

Implementing a Discriminator

```
discriminator = Sequential()
discriminator.add(Conv2D(16, (3, 3), activation=LeakyReLU(0.2), strides=2,
                         padding='same'))
discriminator.add(BatchNormalization()) # Or Dropout
discriminator.add(Conv2D(32, (3, 3), activation=LeakyReLU(0.2), strides=2,
                         padding='same'))
discriminator.add(BatchNormalization()) # Or Dropout
discriminator.add(Flatten()) # Or GlobalAveragePooling2D
discriminator.add(Dense(1, activation='sigmoid'))
discriminator.compile(loss='binary_crossentropy', optimizer='adam')
discriminator.trainable = False
```

Implementing a Generator

```
generator = Sequential()
generator.add(Dense(7 * 7 * 16))
generator.add(Reshape([7, 7, 16]))
generator.add(BatchNormalization())
generator.add(Conv2DTranspose(32, (3, 3), strides=2, activation='relu', padding='same'))
generator.add(BatchNormalization())
generator.add(Conv2DTranspose(16, (3, 3), strides=2, activation='tanh', padding='same'))

# Form a GAN from the generator and the discriminator
model = Sequential([generator, discriminator])
model.compile(loss='binary_crossentropy', optimizer='adam')

# Don't call fit(); model requires a custom training loop
```

Training the Model

```
for epoch in range(epochs):
    np.random.shuffle(x_train) # Shuffle images at the start of each epoch

    for step in steps_per_epoch:
        # Train the discriminator to differentiate between real and generated images
        # TODO: Fetch a batch of real images and combine with a batch of fake images (x)
        # TODO: Generate labels for images where 0 == fake and 1 == real (y)
        discriminator.train_on_batch(x, y)

        # Train the model (trains the generator but not the discriminator)
        x = np.random.normal(size=(batch_size, input_size))
        y = np.array([1] * batch_size)
        model.train_on_batch(x, y)
```

Demo

Generative Adversarial Networks

Diffusion Models

- Trained to turn random noise into images using diffusion process
- Commercial examples include Stable Diffusion, ImageGen, and DALL·E
- Slower at inference time because decoding (like training) is progressive

Training

During training, images have **random noise** added in stages. The model is trained to **remove the noise** one step at a time, reversing the progression.

Inference (Generation)

At inference time, an image **filled with random pixel values** is input to the model. The model repeatedly denoises the image to produce a final image.

Text-Guided Diffusion

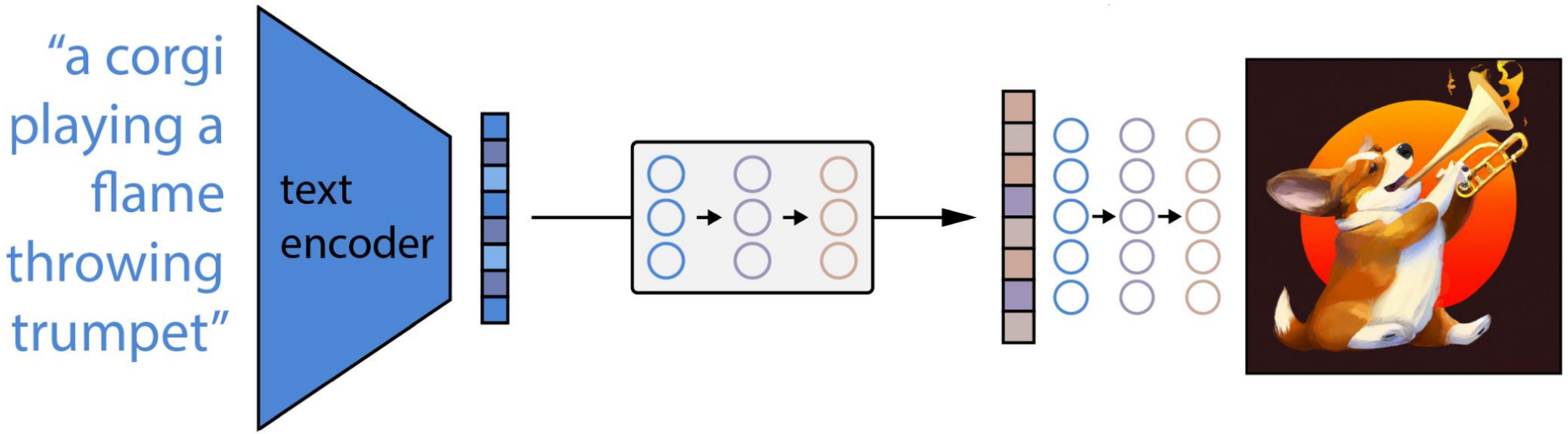
Prompt

A hyperrealistic photograph of ancient Tokyo/London/Paris architectural ruins in a flooded apocalypse landscape of dead skyscrapers, lens flares, cinematic, hdri, matte painting, concept art, celestial, soft render, highly detailed, cgsociety, octane render, trending on artstation, architectural

DALL·E

- Text-guided diffusion model from OpenAI available by REST API
- DALL·E 2 supports image generation, image variations, inpainting, and outpainting
  - 3.5 billion parameters
  - 650 million text-image pairs
- DALL·E 3 supports image generation with additional features on the way

How DALL·E 2 Works



Contrastive Language-Image Pretraining (CLIP) model encodes the prompt, **generating a text embedding** in latent space

"Prior" model generates an **image embedding** from the text embedding and random noise

Decoder uses **reverse diffusion** to **generate a 64x64 image** from the image embedding. CNNs **upsample the image** to 256x256 and then 1,024x1,024 to produce the final image.

Generating Images with DALL·E 2

```
from openai import OpenAI

client = OpenAI(api_key='OPENAI_API_KEY')

response = client.images.generate(
    prompt='Photo of a purple unicorn',
    model='dall-e-2', # dall-e-2 (default) or dall-e-3
    size='512x512', # 256x256, 512x512, or 1024x1024 (default)
    n=1,            # Number of images to generate (default == 1)
    response_format='b64_json' # url (default) or b64_json
)

image_data = response.data[0].b64_json
image = Image.open(io.BytesIO(base64.b64decode(image_data)))
```

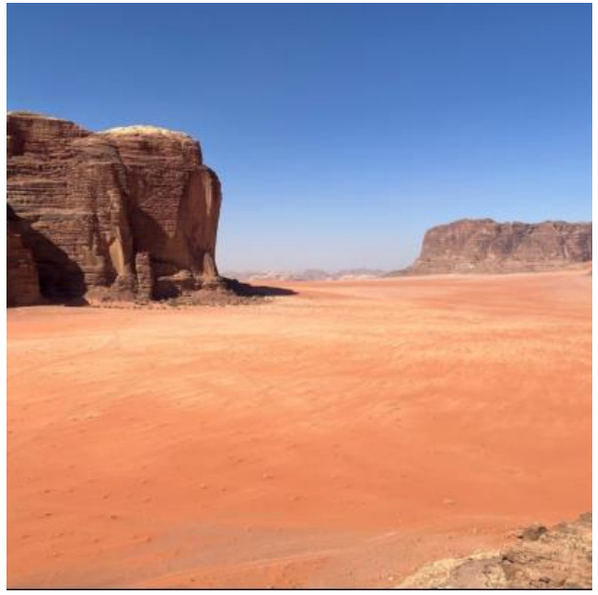
Generating Images with DALL·E 3

```
client = OpenAI(api_key='OPENAI_API_KEY')

response = client.images.generate(
    prompt='Photo of a purple unicorn',
    model='dall-e-3', # dall-e-2 (default) or dall-e-3
    size='1792x1024', # 1024x1024 (default), 1792x1024, or 1024x1792
    style='standard', # standard (default) or hd
    quality='vivid',  # vivid (default) or natural
    response_format='b64_json' # url (default) or b64_json
)

image_data = response.data[0].b64_json
image = Image.open(io.BytesIO(base64.b64decode(image_data)))
```

Creating Variations of Existing Images

```
response = client.images.create_variation(
    image=open('PATH_TO_IMAGE', 'rb'),
    response_format='b64_json', size='512x512'
)
```

Original image



Variation created by DALL·E 2

Inpainting

```
response = client.images.edit(
    image=open('PATH_TO_IMAGE', 'rb'), # Path to original image
    mask=open('PATH_TO_IMAGE_MASK', 'rb'), # Path to same image with transparent pixels
    prompt='Photograph of two people standing on a cliff overlooking the beach',
    response_format='b64_json', size='512x512'
)
```

Original image with fence in background

Image mask with transparent pixels denoting regions to be inpainted

Outpainting

```
response = client.images.edit(
    image=open('PATH_TO_IMAGE', 'rb'),
    mask=open('PATH_TO_IMAGE', 'rb'), # Same image
    prompt='Painting of a girl standing in a kitchen',
    response_format='b64_json', size='512x512'
)
```

Transparent pixels identifying region to be outpainted

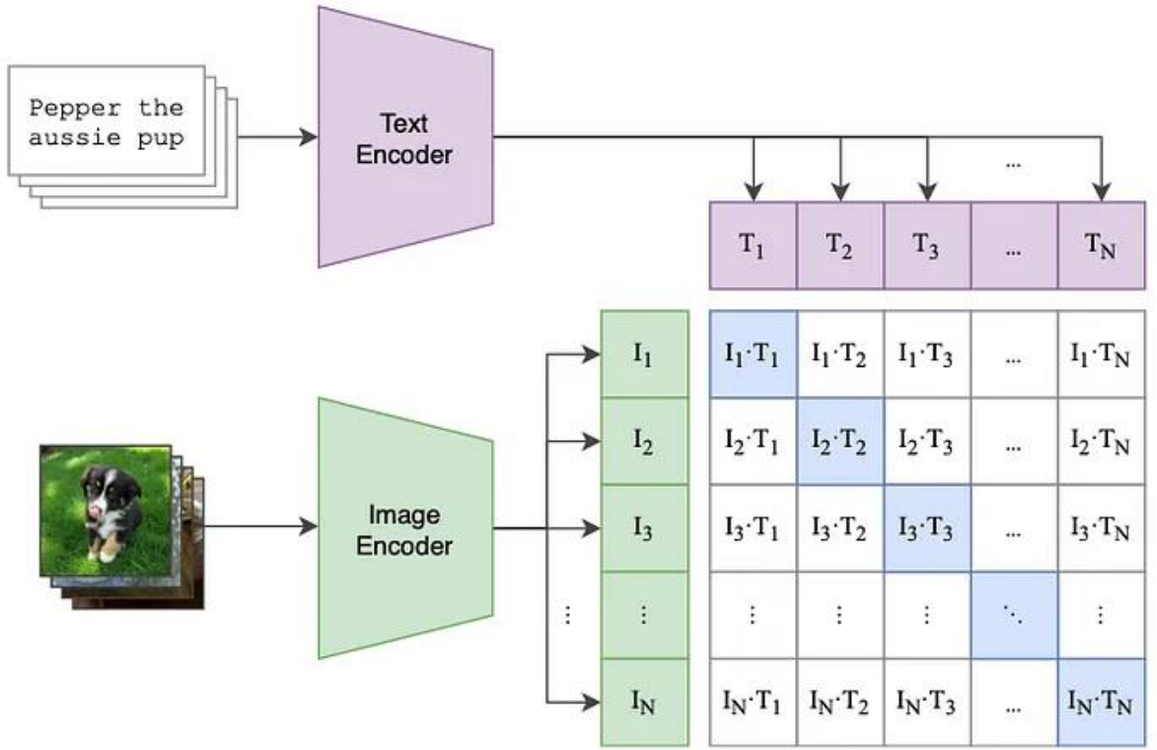
Region to be expanded via outpainting

Demo

DALL·E

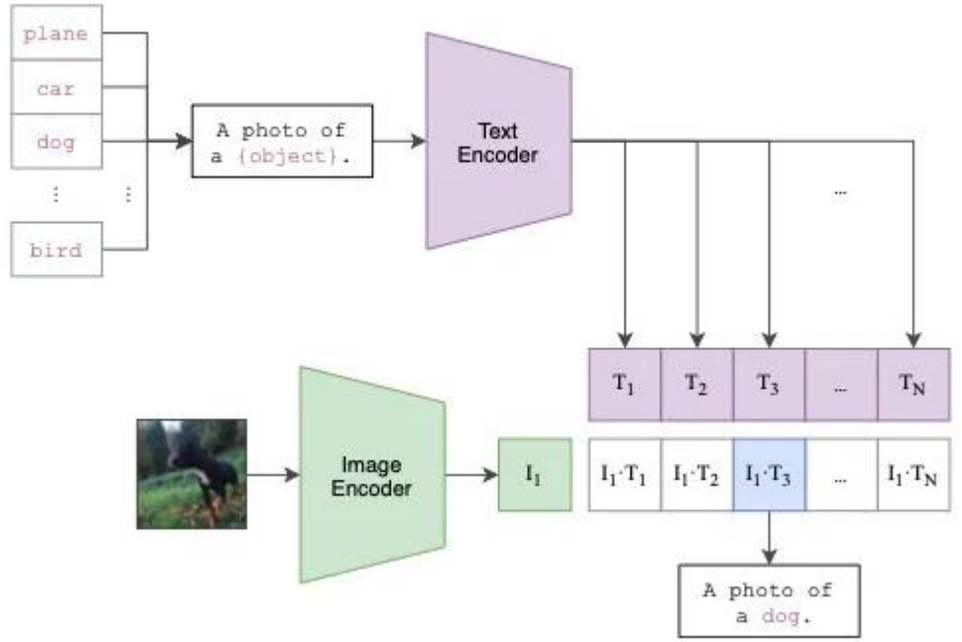
Contrastive Language-Image Pretraining (CLIP)

- Introduced in 2021 by OpenAI and trained on more than **400 million** text-image pairs
- Correlates text and images by training a model to maximize embedding similarities
- DALL·E 2 uses CLIP's text encoder to generate embeddings from text prompts



Zero-Shot Image Classification

- Classify photos without:
  - Training a CNN from scratch or using transfer learning with a pretrained CNN
  - Assembling a labeled dataset
- Present model with an image and several possible descriptions
- Model predicts which description is "correct" by computing similarity of image embedding and each text embedding



OpenCLIP

- Open-source version of CLIP
  - Hosted at https://github.com/mlfoundations/open_clip
- OpenAI shared model weights but not the training dataset
- LAION (Large-scale Artificial Intelligence Network) assembled massive text-image datasets, trained several versions of OpenCLIP with them, and published the datasets and model weights
  - Includes LAOIN-5B dataset with **5.85 billion** text-image pairs
- Many OpenCLIP models trained on these datasets available in Hugging Face's **transformers** package

Using clip-vit-large-patch14

```
from PIL import Image
from transformers import pipeline

model = pipeline(
    model='openai/clip-vit-large-patch14',
    task='zero-shot-image-classification'
)

image = Image.open('PATH_TO_IMAGE')
model(image, candidate_labels=['owl', 'giraffe', 'camel'])
```

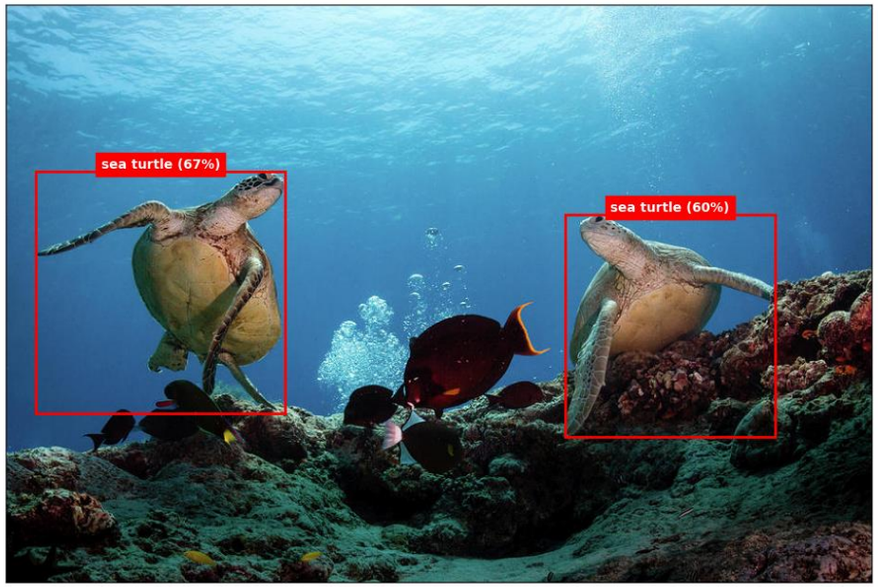
```
[{'score': 0.9978225231170654, 'label': 'giraffe'},
 {'score': 0.002148183062672615, 'label': 'camel'},
 {'score': 2.921208033512812e-05, 'label': 'owl'}]
```

Demo

Zero-Shot Image Classification

Zero-Shot Object Detection

```python
from PIL import Image
from transformers import pipeline

model = pipeline(
    model='google/owlv2-base-patch16-ensemble',
    task='zero-shot-object-detection'
)

image = Image.open('PATH_TO_IMAGE')
candidate_labels = ['parrot', 'sea turtle', 'giraffe']
predictions = model(image, candidate_labels)
```




```
[{'score': 0.6651442646980286,
  'label': 'sea turtle',
  'box': {'xmin': 31, 'ymin': 115, 'xmax': 290, 'ymax': 283}},
 {'score': 0.6040006279945374,
  'label': 'sea turtle',
  'box': {'xmin': 581, 'ymin': 145, 'xmax': 799, 'ymax': 299}}]
```

Demo

Zero-Shot Object Detection