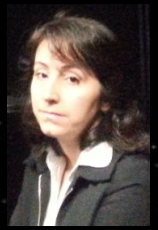
XII ESCOLA DO CBPF

22 de julho a 02 de agosto de 2019

Inteligência Artificial Utilizando *Deep Learning* e Aplicações em Física

Márcio Portes de Albuquerque (CBPF)

Clécio R. De Bom (CBPF/CEFET-RJ)



Elisangela L. Faria (CBPF)

Uma introdução conceitual de aprendizado de máquina e uma abordagem prática em aplicações de redes neurais profundas com foco em aplicações científicas e tecnológicas.

CBPF 70 ANOS

1949 - 2019

XII ESCOLA DO CBPF

22 de julho a 02 de agosto de 2019

Ementa

AULA 1: Apresentações iniciais, Redes Neurais Artificiais, Aprendizado de máquina, Algoritmos/Termos; Redes Perceptron de Várias Camadas.

AULA 2: Rede Neural Convolucional: Treinamento, Maxpooling, Dropout

AULA 3: Overfitting, funções de ativação. Redes VGG e AlexNet, Resnet e Inception

AULA 4: Autoencoders e aprendizagem não supervisionada.
- Redes Geradoras Adversariais (GAN).

AULA 5: Introdução a Redes Neurais Bayesianas, Redes Neurais Convolucionais Baseadas em Região (R-CNN). Introdução a Reinforcement Learning

Exemplos / Demonstrações

Pré-requisitos:
Recomenda-se o domínio da linguagem de programação Python
Porém é possível o uso de C ou MATLAB.

XII ESCOLA DO CBPF

22 de julho a 02 de agosto de 2019

You will need:

Python (anaconda3)
Keras
Tensorflow
matplotlib
numpy

For the examples you will also need:
astropy (Aulas Clécio)
Google Colab (on-line)

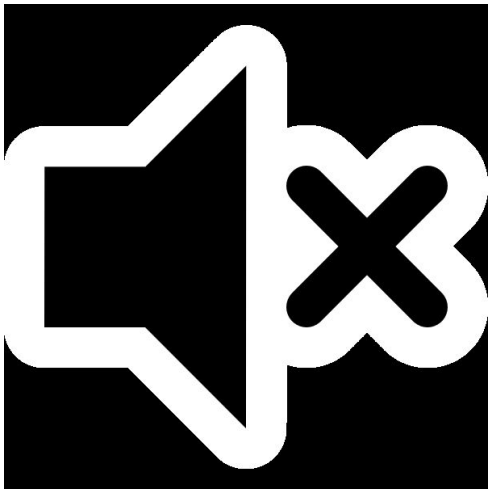
We will have several examples in Google Colabs or in Python notebooks. The examples and datasets required can be downloaded in

Clearnightsrthebest.com

debom@cbpf.br

XII ESCOLA DO CBPF

22 de julho a 02 de agosto de 2019



Para um melhor aproveitamento do conteúdo ministrado, recomendamos à audiência que mantenha os aparelhos eletrônicos (celulares, laptops) Desligados ou mudos durante as aulas.

XII ESCOLA DO CBPF

22 de julho a 02 de agosto de 2019

login: XII-Escola-CBPF

senha: escolacbpf2019

Laboratório de Redes e Sistemas (LARS)

GEFORCE RTX

NVIDIA.
GPU Computing

CBPF
SciMining
6.RTX2080TI

36x RTX 2080Ti
6x Quadro K6000
6x 1080Ti
13x 1050Ti

- 32 CPU cores, 128 GB RAM (expansível até 1TB),
- 26112 CUDA cores, 84 Teraflops, 66GB VRAM,
- 4xWaterCoolers - completo silêncio,
- 1632 Tmus unid. de texturas e 528 ROps unid. para renderização
- 4TB HD/backup Enterprise e 500 GB SSD/OS, 2 fontes 1600W

NVIDIA GPU Computing

CBPF SciMining 6.RTX2080TI

GPGPU – General Purpose GPU

(processador da placa de vídeo)

As GPU foram originalmente usadas para renderização de imagens em jogos 3D.

Essa capacidade de processamento está sendo aproveitada de forma mais ampla para acelerar o processamento computacional em áreas como pesquisa científica, processamento de sinais/imagens, algebra linear, estatística, reconstrução 3D, modelagem financeira, exploração de petróleo e gás....

SciMIning x Santos Dumont (2) PERFORMANCE

FP32 (float)

HPC	Modelo	TFLOPS	Nós	GPU/ Nós	Total TFLOPS		Percentual
SDU	K40	5,046	198	2	1998,22		100%
SciMIning	RTX2080Ti	13,45	1	6	80,70		4%
SciMIning 7	RTX2080Ti	13,45	7	6	564,90		28%

https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305

Fermilab (FP32 (float) performance)			
K20	2	2,87	5,74
K40	2	4,29	8,58
P100	10	9,526	95,26
V100	4	14,13	56,52
Total			166,1

https://wilsonweb.fnal.gov/phigpu.shtml

SciMIning x Wilson(3) - RENDER CONFIG

TENSOR CORES

HPC	Modelo	TENSOR CORES	Nós	GPU/ Nós	Total TENSOR CORES		Percentual
Wilson	Varios	2560			2560		100%
SciMIning	RTX2080Ti	544	1	6	3264		128%
SciMIning 7	RTX2080Ti	544	7	6	22848		700%

https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305

NVIDIA GPU Computing

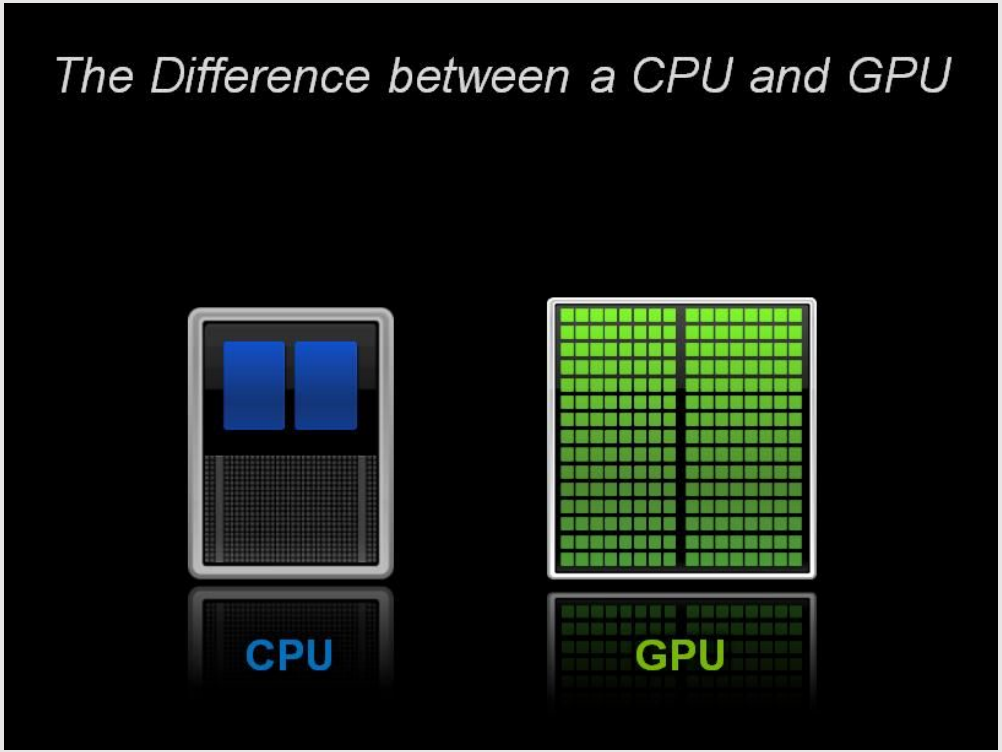
CBPF SciMining 6.RTX2080TI

GPU – Unidade de Processamento Gráfico

(processador da placa de vídeo)

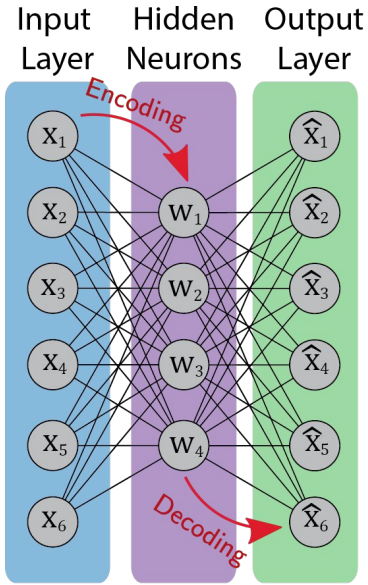
A GPU é responsável pelos cálculos de processamento de vetores, texturas, projeções das imagens em um sistema de vídeo do computador.

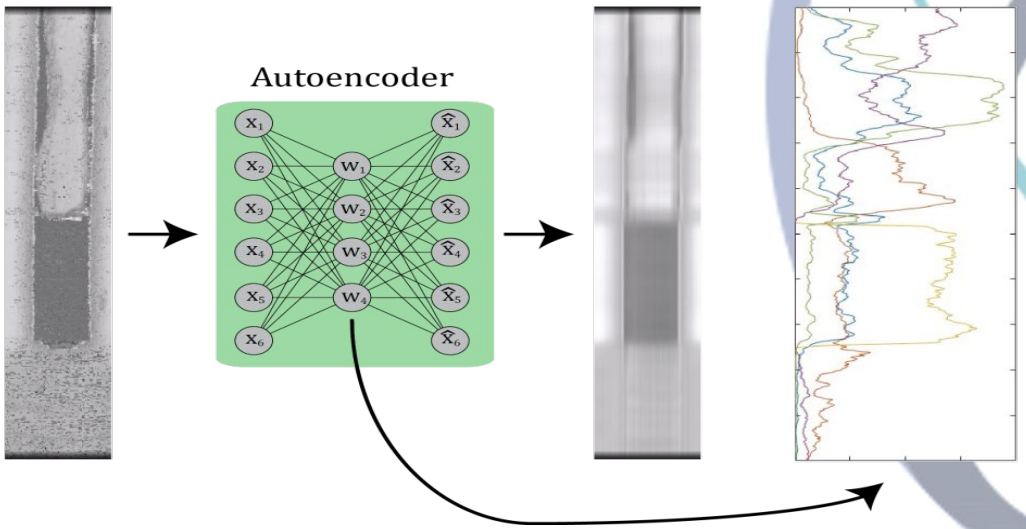
Libera a CPU para fazer outras tarefas (de outra complexidade).



What is an AutoEncoder?

- ANN aim to reproduce input data.
- **Encode/Decode** strategy.
- **IDENTITY FUNCTION → Hidden Features**





Auto Encoders Example

```
from keras.layers import Input, Dense
from keras.models import Model

# this is the size of our encoded representations
encoding_dim = 32
 # 32 floats -> compression of factor 24.5, assuming the input is 784
floats

# this is our input placeholder
input_img = Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)
```

Example from: https://blog.keras.io/building-autoencoders-in-keras.html

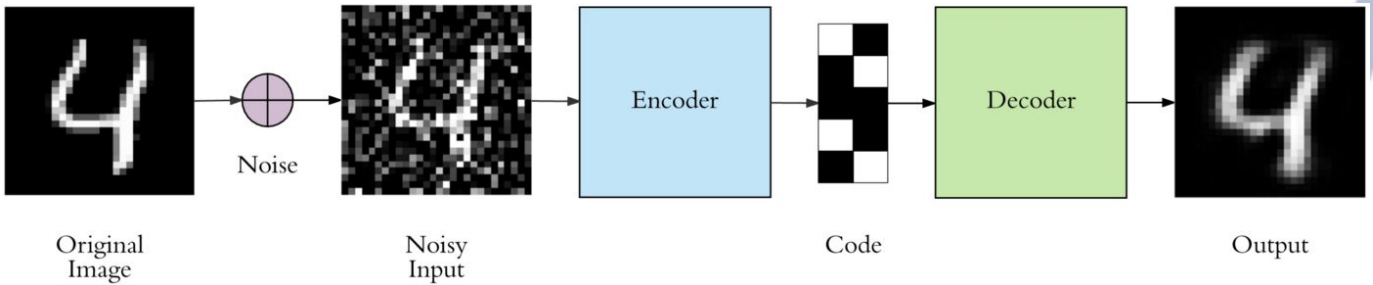
Auto Encoders – A simple example

```
# this model maps an input to its encoded representation

encoder = Model(input_img, encoded)

# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# create the decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

Warning: The autoencoders do overfit a lot if they are too complex.

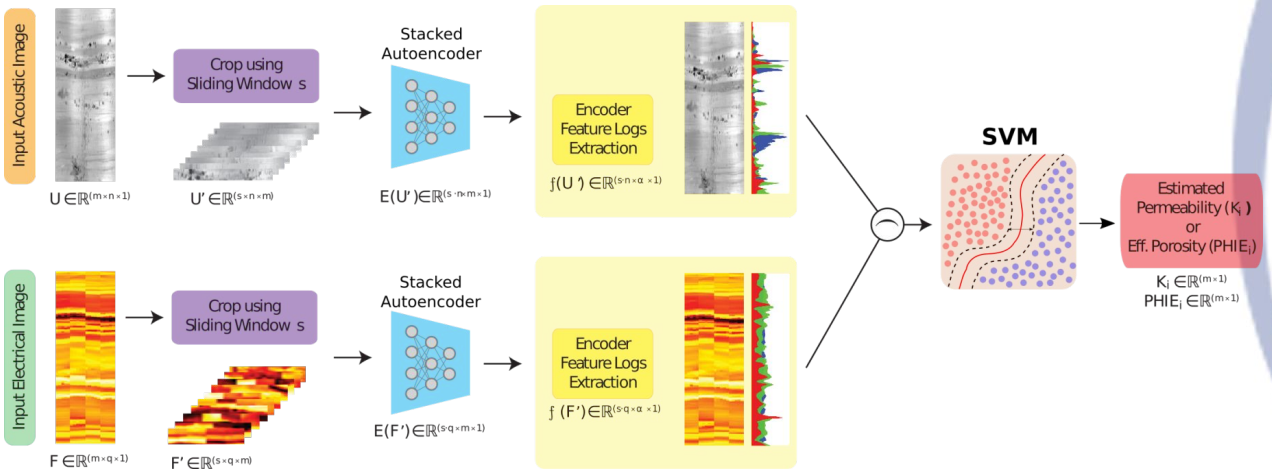
Small number of parameters forces the autoencoder to learn an intelligent representation of the data.

Denoising


Noise
Encoder
Decoder
Original Image
Noisy Input
Code
Output

Regression with AE? But Why?

- Small datasets
- “Empty” datasets



Variational AE

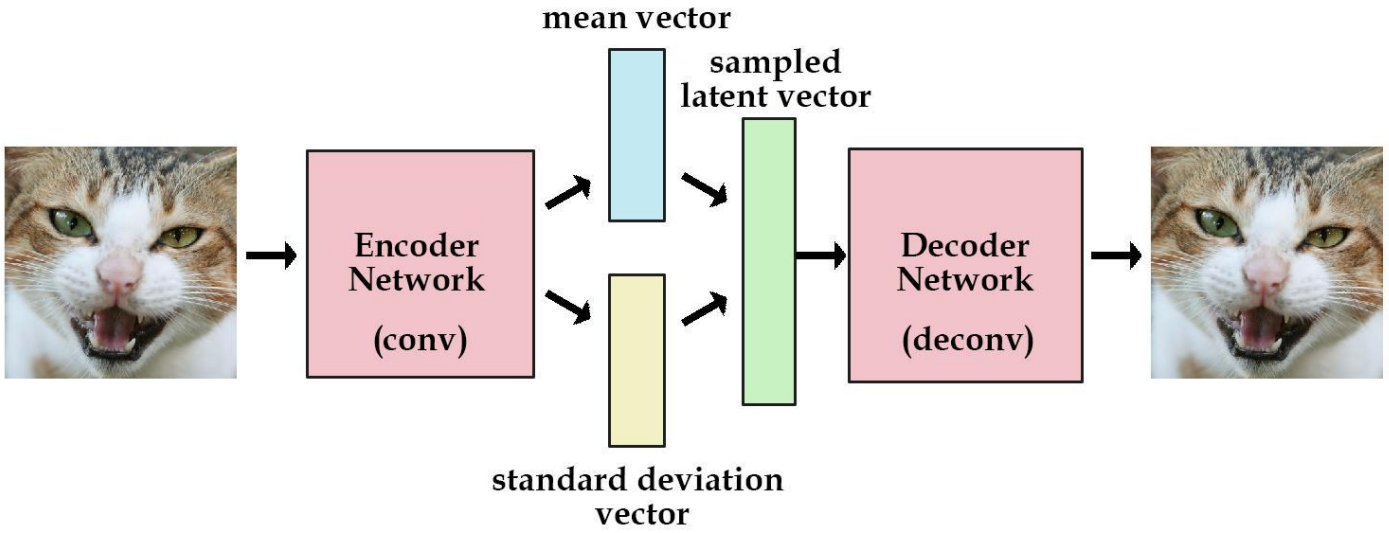


Image Credit http://kvfrans.com/variational-autoencoders-explained/

Variational AE – Generative Model

An autoencoder that learns a latent variable model for its input data. So instead of letting your neural network learn an arbitrary function, you are learning the parameters of a probability distribution modeling your data. If you sample points from this distribution, you can generate new input data samples: a VAE is a "generative model".

$$q(Z|X) \qquad p(X|Z)$$

Encode Decode

```
x = Input(batch_shape=(batch_size, original_dim))
h = Dense(intermediate_dim, activation='relu')(x)
z_mean = Dense(latent_dim)(h)
z_log_sigma = Dense(latent_dim)(h)
```

Variational AE – Generative Model

Latent normal distribution that is assumed to generate the data, via z = z_mean + exp(z_log_sigma) * epsilon, where epsilon is a random normal tensor.

A decoder network maps these latent space points back to the original input data.

```
def sampling(args):
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=(batch_size,
latent_dim),
                              mean=0.,
std=epsilon_std)
    return z_mean + K.exp(z_log_sigma) * epsilon

z = Lambda(sampling,
output_shape=(latent_dim,))([z_mean, z_log_sigma])
```

$$q_\phi(\mathbf{z}_n|\mathbf{x}_n) = \mathcal{N}(\mathbf{z}_n|\mu_\phi(\mathbf{x}_n), \mathrm{diag}(\sigma^2_\phi(\mathbf{x}_n))),$$

$$\mathbf{z} = g_\phi(\mathbf{x}, \epsilon) = \mu_\phi(\mathbf{x}) + \sigma_\phi(\mathbf{x}) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$

Variational AE – Generative Model

Latent normal distribution that is assumed to generate the data, via z = z_mean + exp(z_log_sigma) * epsilon, where epsilon is a random normal tensor.

A decoder network maps these latent space points back to the original input data.

```
decoder_h = Dense(intermediate_dim, activation='relu')
decoder_mean = Dense(original_dim, activation='sigmoid')
h_decoded = decoder_h(z)
x_decoded_mean = decoder_mean(h_decoded)
```

https://blog.keras.io/building-autoencoders-in-keras.html

Variational AE – Generative Model

Latent normal distribution that is assumed to generate the data, via z = z_mean + exp(z_log_sigma) * epsilon, where epsilon is a random normal tensor.

A decoder network maps these latent space points back to the original input data.

```
# end-to-end autoencoder
vae = Model(x, x_decoded_mean)

# encoder, from inputs to latent space
encoder = Model(x, z_mean)

# generator, from latent space to reconstructed inputs
decoder_input = Input(shape=(latent_dim,))
_h_decoded = decoder_h(decoder_input)
_x_decoded_mean = decoder_mean(_h_decoded)
generator = Model(decoder_input, _x_decoded_mean)
```

https://blog.keras.io/building-autoencoders-in-keras.html

Variational AE – Generative Model

We add an extra term to the Loss

$$D_{\mathrm{KL}}(Q\|P) = \sum_i Q(i) \ln\left(\frac{Q(i)}{P(i)}\right)$$

Consider an observed variable that is connected to the latent variable z.
As we are dealing with gaussians we may write

$$\mathrm{KL}[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})] = -\frac{1}{2}\sum_{k=1}^{K}\{1 + \log\sigma_k^2 - \mu_k^2 - \sigma_k^2\}$$

D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," in Proceedings of the 2nd International Conference on Learning Representations (ICLR), 2014.

http://louistiao.me/posts/implementing-variational-autoencoders-in-keras-beyond-the-quickstart-tutorial/#kingma2014

Variational AE – Generative Model

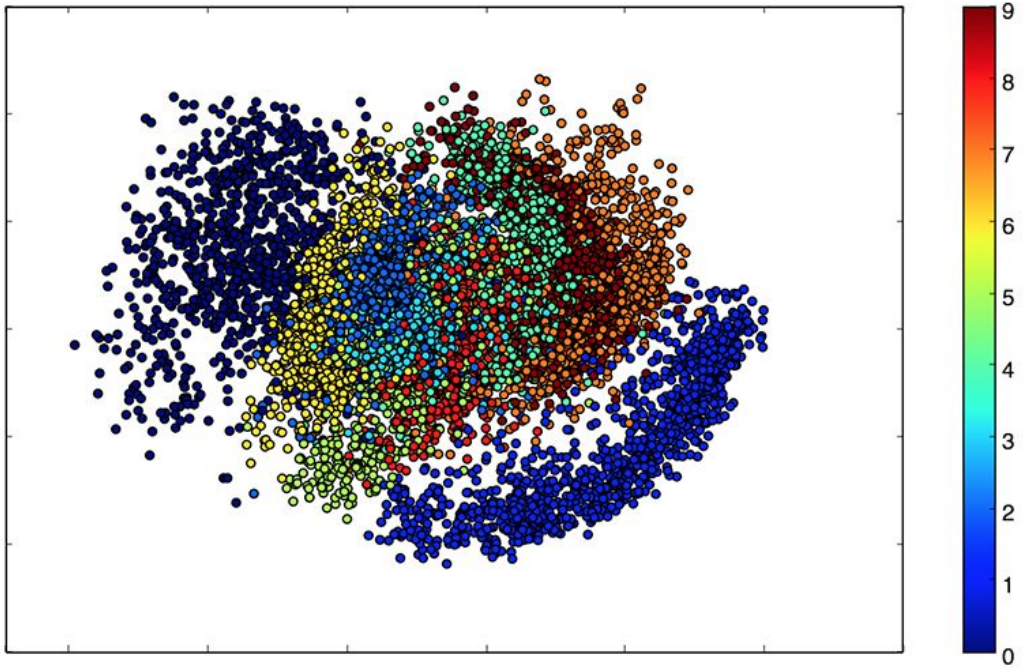
```
def vae_loss(x, x_decoded_mean):
    xent_loss = objectives.binary_crossentropy(x,
x_decoded_mean)
    kl_loss = - 0.5 * K.mean(1 + z_log_sigma -
K.square(z_mean) - K.exp(z_log_sigma), axis=-1)
    return xent_loss + kl_loss
```

Example adapted from: https://blog.keras.io/building-autoencoders-in-keras.html

Variational AE – Generative Model

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train),
np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

vae.fit(x_train, x_train,
        shuffle=True,
        epochs=epochs,
        batch_size=batch_size,
        validation_data=(x_test, x_test))
```

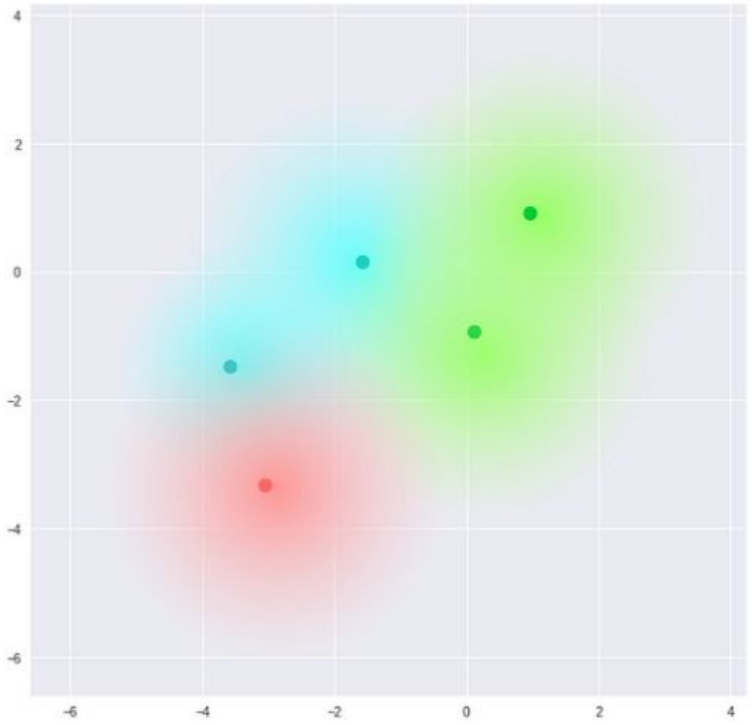
```
x_test_encoded = encoder.predict(x_test, batch_size=batch_size)
plt.figure(figsize=(6, 6))
plt.scatter(x_test_encoded[:, 0], x_test_encoded[:, 1], c=y_test)
plt.colorbar()
plt.show()
```



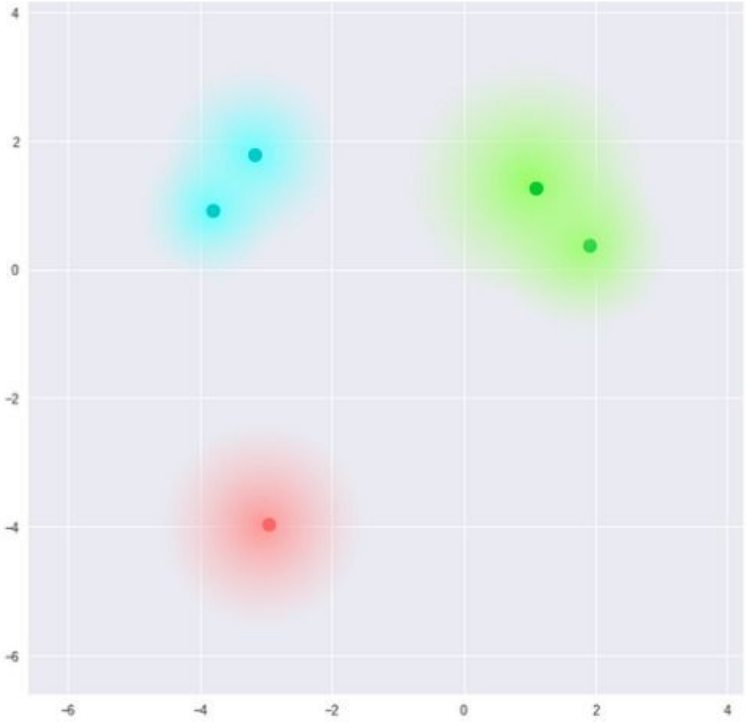
Variational AE – Generative Model

```
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
# we will sample n points within [-15, 15] standard deviations
grid_x = np.linspace(-15, 15, n)
grid_y = np.linspace(-15, 15, n)

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]]) * epsilon_std
        x_decoded = generator.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit
plt.show()
```

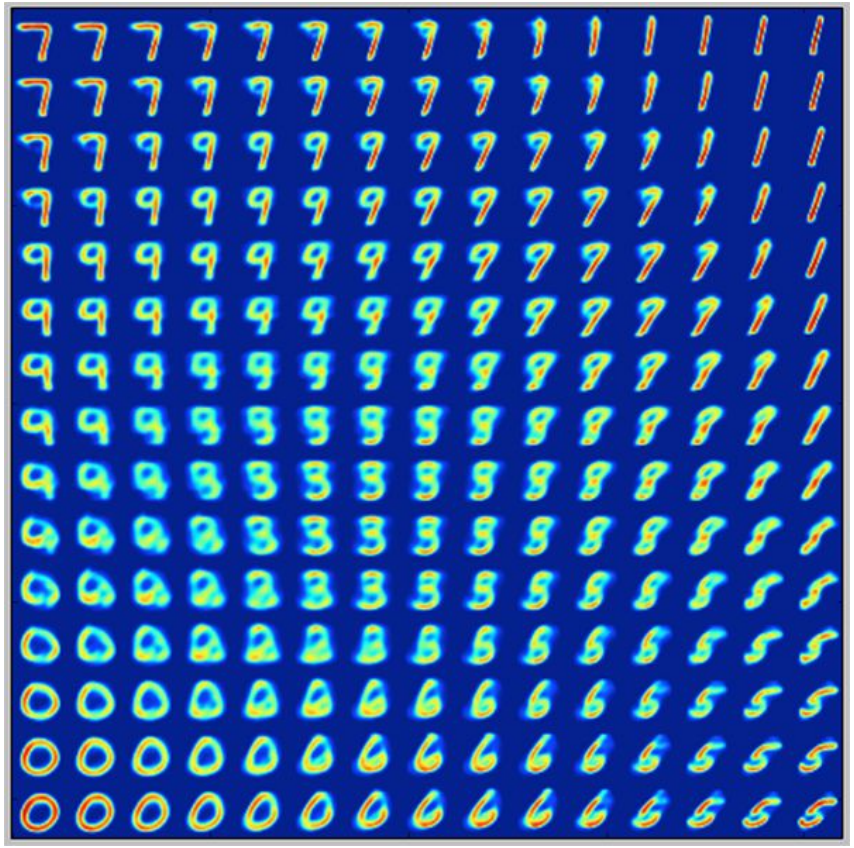
Variational AE – Generative Model



What we require



What we may inadvertently end up with



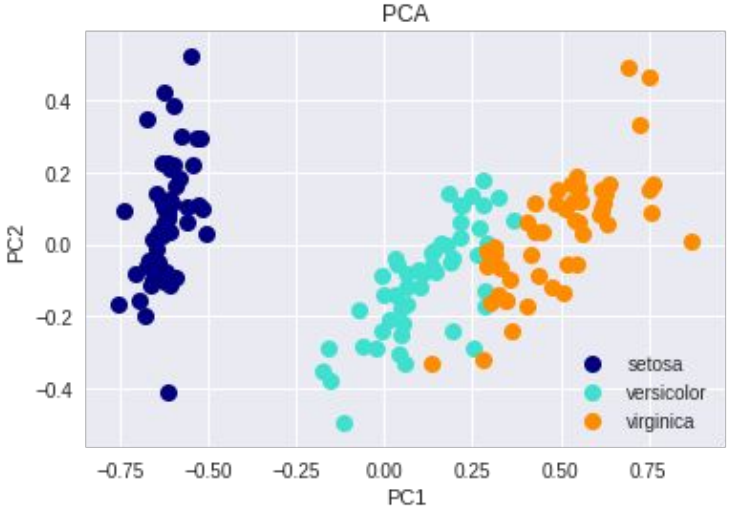
PCA vs AE

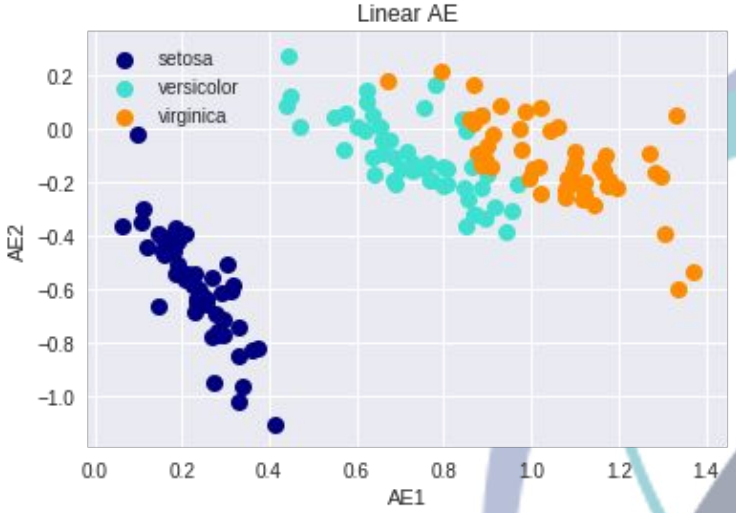
Iris Dataset

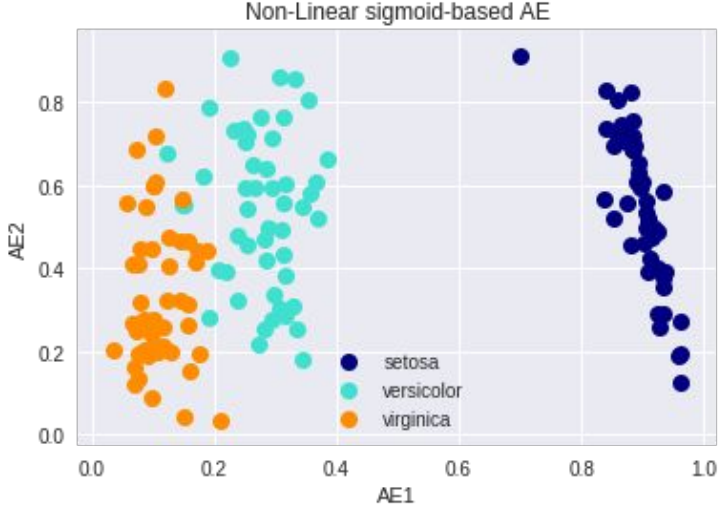
Attribute Information:
1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
-- Iris Setosa
-- Iris Versicolour
-- Iris Virginica

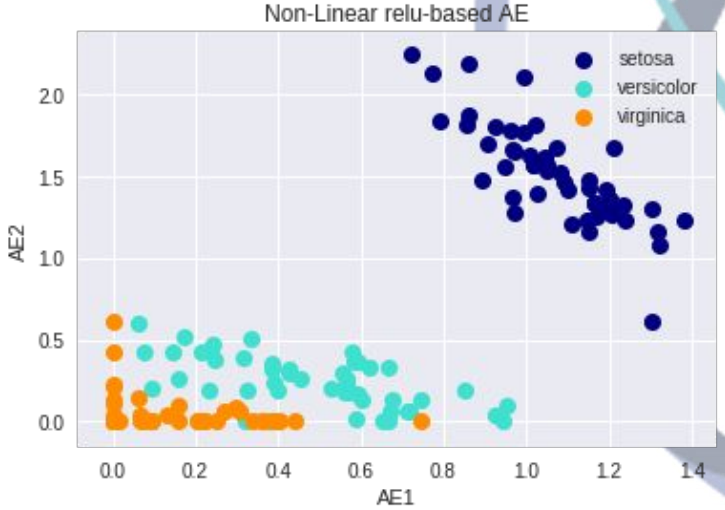
```
iris = datasets.load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names


pca = decomposition.PCA()
pca_transformed = pca.fit_transform(X_scaled)
plot3clusters(pca_transformed[:,:2], 'PCA', 'PC')
```

Example adapted from: https://towardsdatascience.com/pca-vs-autoencoders-1ba08362f450


PCA
PC2
PC1
setosa
versicolor
virginica


Linear AE
AE2
AE1
setosa
versicolor
virginica


Non-Linear sigmoid-based AE
AE2
AE1
setosa
versicolor
virginica

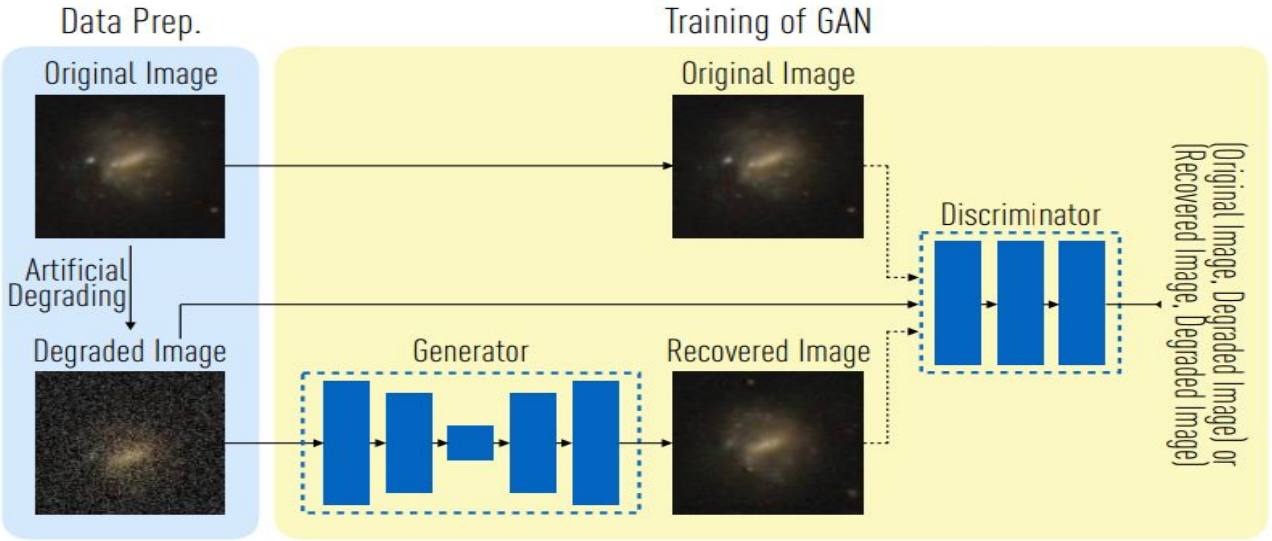

Non-Linear relu-based AE
AE2
AE1
setosa
versicolor
virginica

Generative Adversarial Neural Networks

First proposed by Goodfellow et al. 2014 arXiv:1406.2661

The Main idea is to make two networks to be adversaries.

(arXiv:1702.00403v1).



arXiv:1702.00403v1

Generative Adversarial Neural Networks

One need to set two networks: Generator and a Discriminator

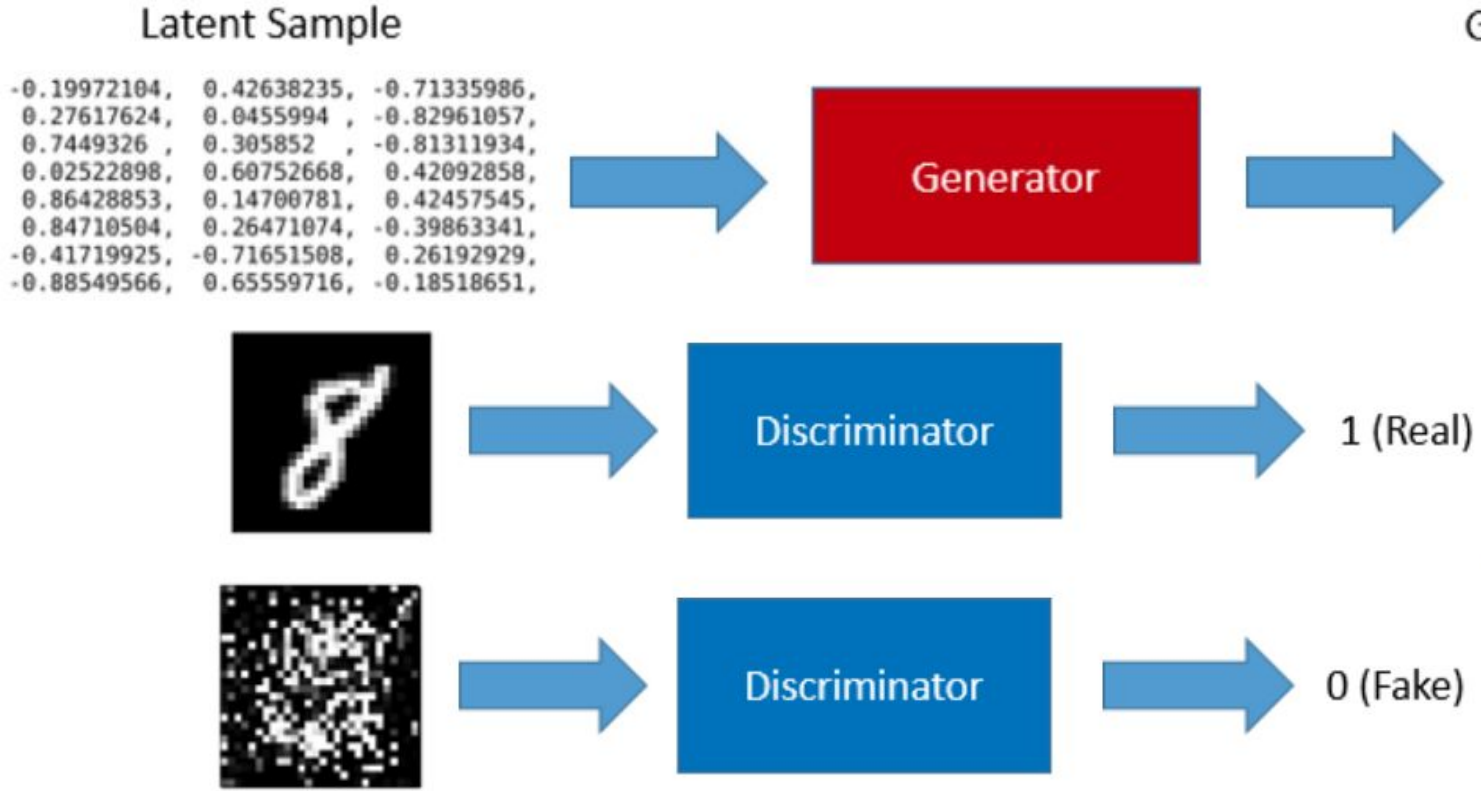


Image credit: https://towardsdatascience.com/understanding-generative-adversarial-networks-4dafc963f2ef

Generative Adversarial Neural Networks

The key here is the training …. There is a loop:
first we train the discriminator. Let the generator create some images out of noise and train **the discriminator only** to define true or false.

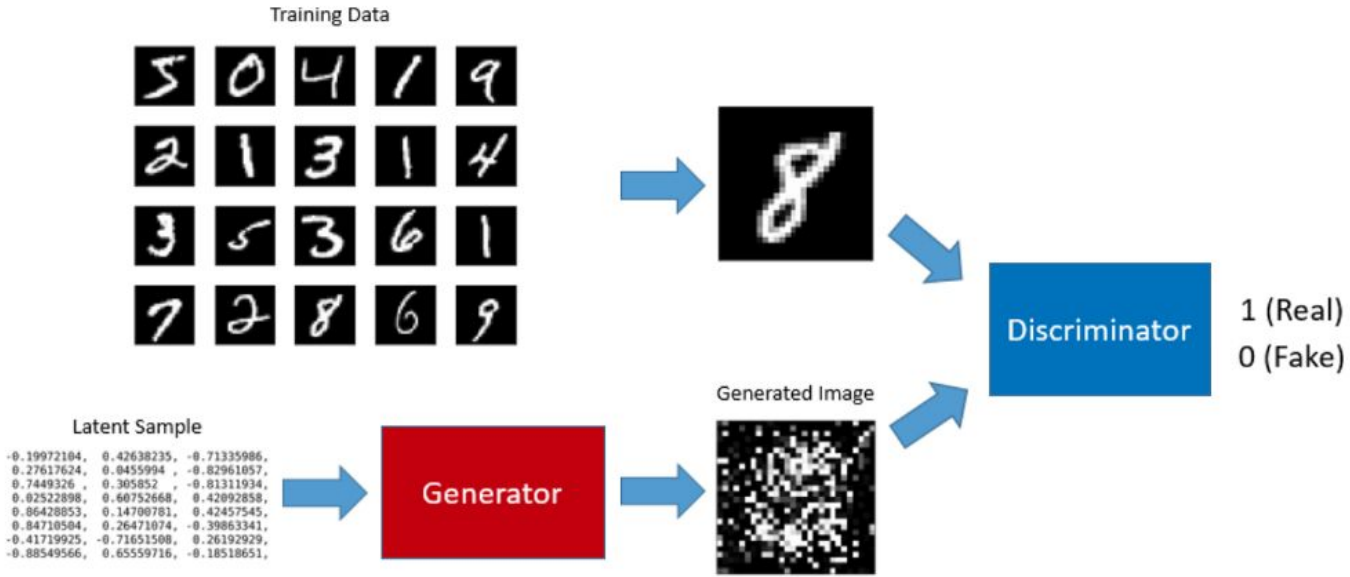


Image credit: https://towardsdatascience.com/understanding-generative-adversarial-networks-4dafc963f2ef

Generative Adversarial Neural Networks

The key here is the training …. There is a loop:
first we train the discriminator. Let the generator create some images out of noise and train **the discriminator only** to define true or false. Consider a Keras model D to be the discriminator , G the Generator and AD the Adversarial Model, i.e., the full model [G,D]

```
generated_images = G.predict(x=random_data, batch_size=batch_size)

d_loss_real = D.train_on_batch(image_real_batch, output_true)
d_loss_fake = D.train_on_batch(generated_images, output_false)
d_loss = 0.5 * np.add(d_loss_fake, d_loss_real)
```

Generative Adversarial Neural Networks

The key here is the training ….
Then we set the Discriminator to be **non-trainable**. After that we optimize the generator. We create some random noise data flows through the Network (Generator and Discriminator). The Generator weights will be updated by backpropagation from the non-trainable Discriminator to the Generator.

```
D.trainable = False
y = np.ones([batch_size, 1])
noise = np.random.uniform(-1.0, 1.0, size=[batch_size, 100])
AD_loss = AD.train_on_batch(random_data, y)
D.trainable = True
```

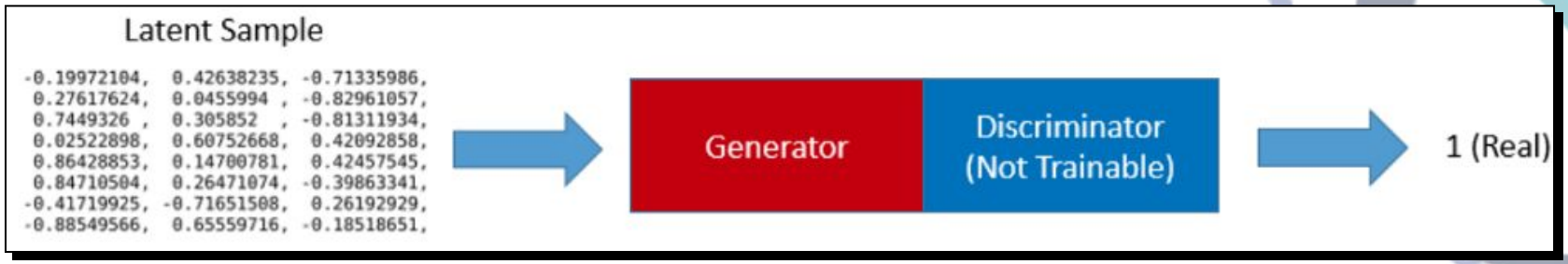


Image credit: https://towardsdatascience.com/understanding-generative-adversarial-networks-4dafc963f2ef

Generative Adversarial Neural Networks are hard to train...

Some general Advice
Gain is particulary sensible to vanishing gradients/ sparse gradients
LeakyReLU is recomend instead of ReLU.

If Discriminator loss converges rapidly and thus prevents the Generator from learning make its learning rate bigger than the Adversarial model learning rate you can also use a different training noise sample for the Generator.

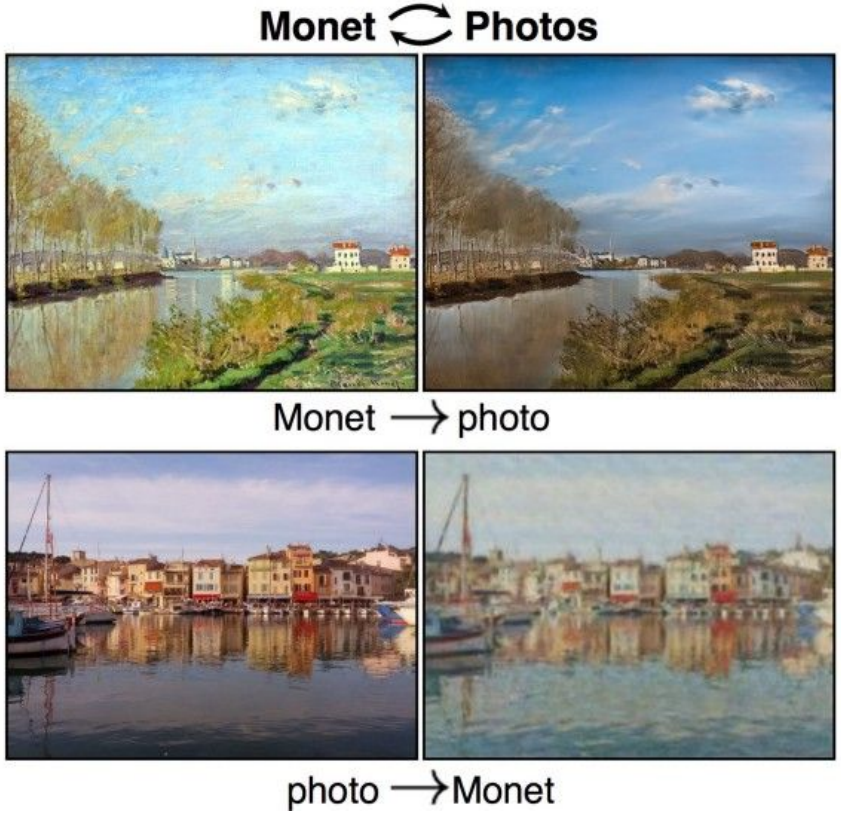
If generated images look like noise. Use dropout on both Discriminator and Generator. Low dropout values generate more realistic images.

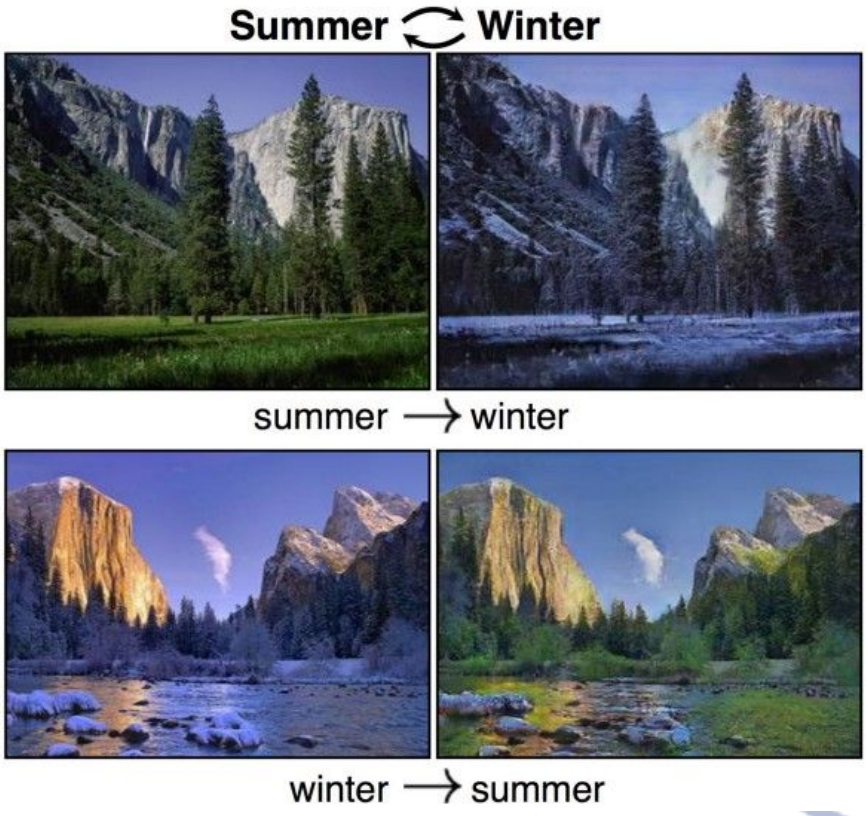
Image credit: https://towardsdatascience.com/understanding-generative-adversarial-networks-4dafc963f2ef

Some Scary smiles from DCGAN

Cycle GAN: Make your own Monet

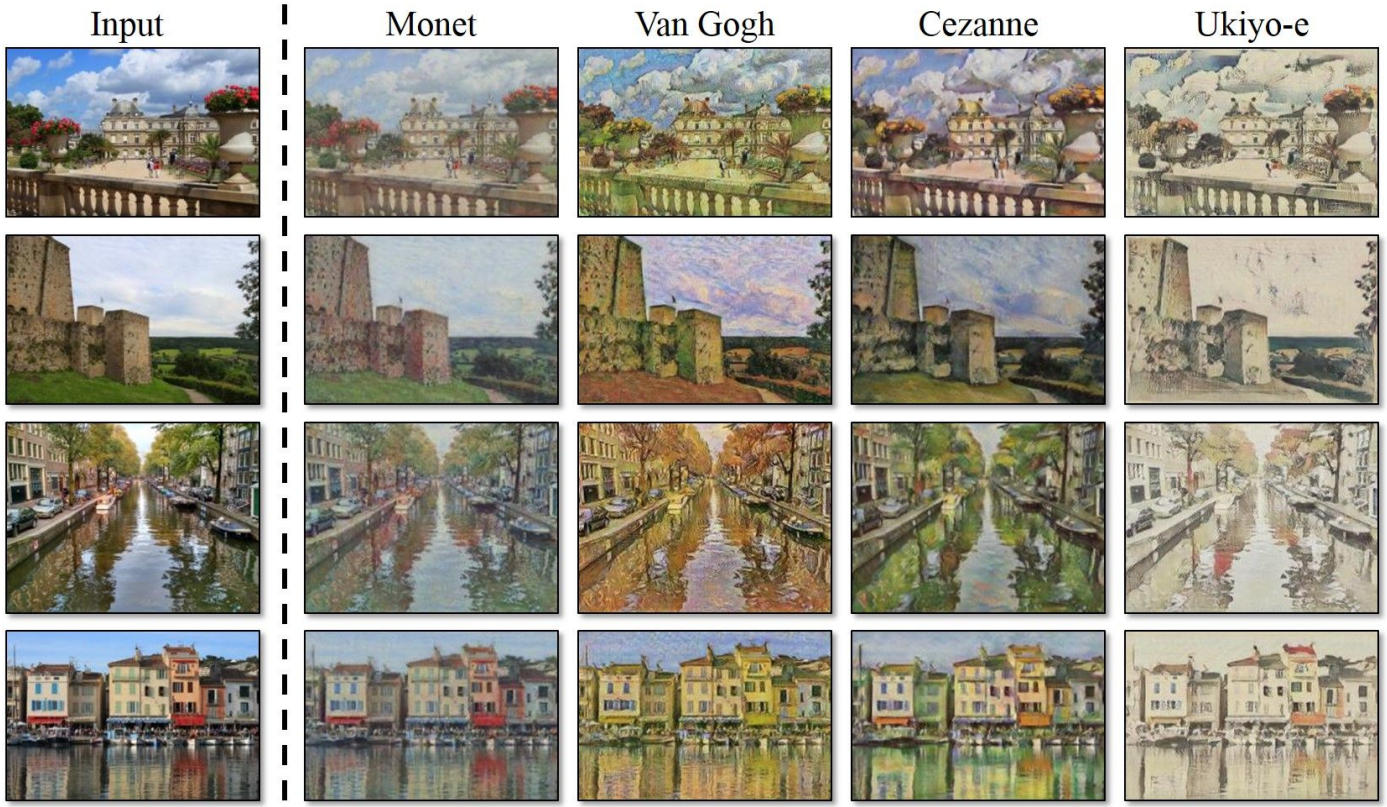
arXiv:1703.10593





Cycle GAN: Make your own Monet

arXiv:1703.10593



What people are doing with this?

DeepCMB: Lensing Reconstruction of the Cosmic Microwave Background with Deep Neural Networks

J. Caldeira[a,1,*], W. L. K. Wu[b], B. Nord[b,c,e], C. Avestruz[a,b], S. Trivedi[d], K. T. Story[f]

arXiv:1810.01483

Deblending galaxy superpositions with branched generative adversarial networks

David M. Reiman[1]⋆†, Brett E. Göhre[1]‡§
[1]Department of Physics, University of California Santa Cruz, 1156 High Street, Santa Cruz, California 95064, USA

arXiv:1810.10098

Enabling Dark Energy Science with Deep Generative Models of Galaxy Images

Siamak Ravanbakhsh[1], François Lanusse[2], Rachel Mandelbaum[2], Jeff Schneider[1], and Barnabás Póczos[1]

arXiv:1609.05796

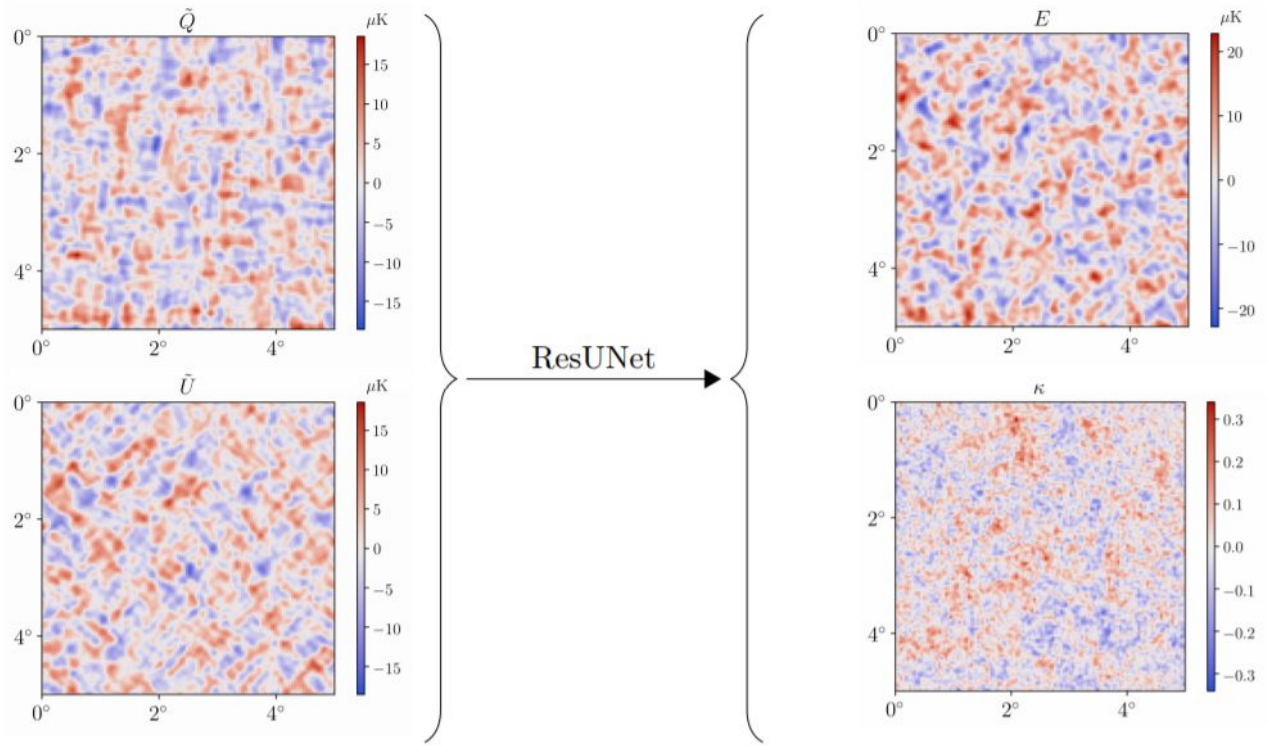
Fast Cosmic Web Simulations with Generative Adversarial Networks

A.C. Rodríguez,[a] T. Kacprzak,[b] A. Lucchi,[c] A. Amara,[b] R. Sgier,[b] J. Fluri,[b] T. Hofmann,[c] and A. Réfrégier[b]

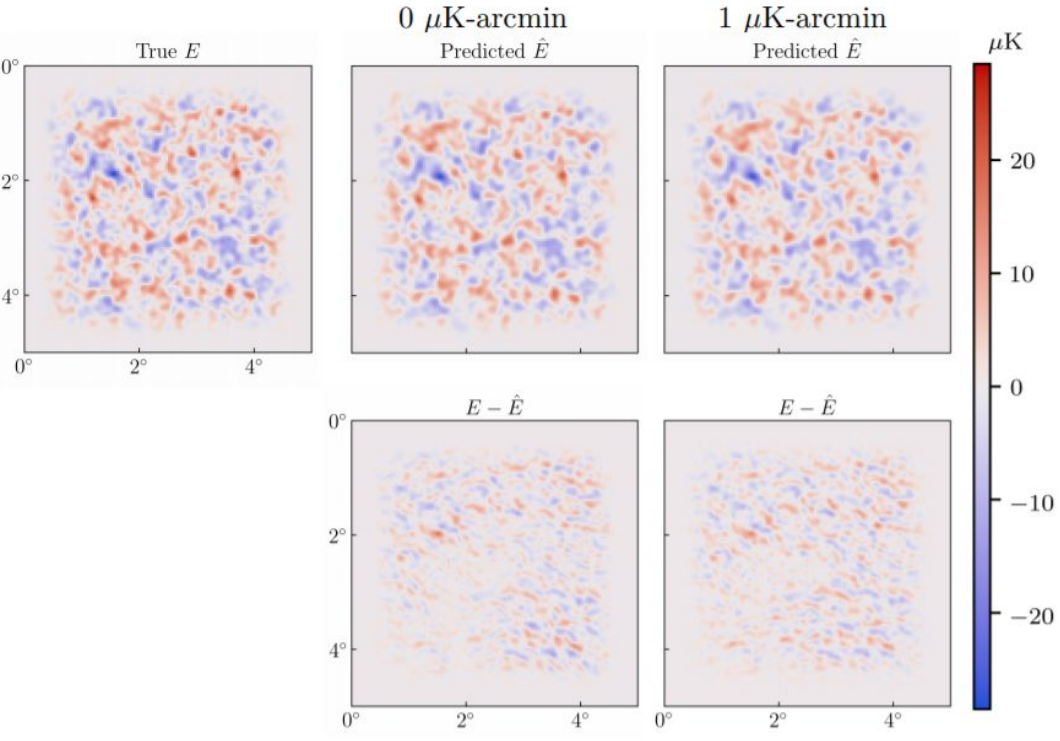
arXiv:1801.09070

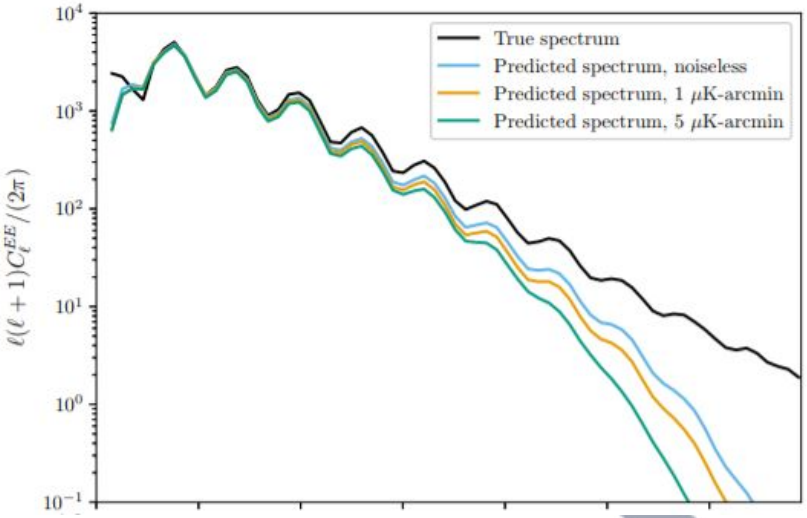
DeepCMB: Lensing Reconstruction of the Cosmic Microwave Background with Deep Neural Networks

CMB polarization maps are usually represented (Q, U) basis corresponds to Stokes parameters and is convenient for mapping onto from the CMB instruments' polarization detector coordinates.



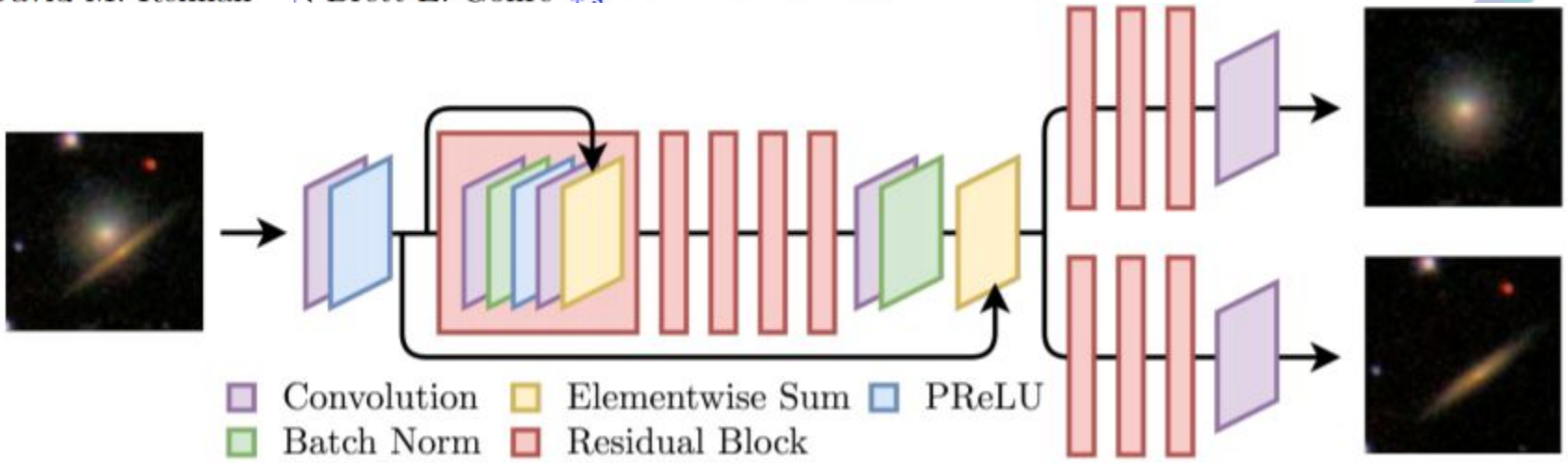
DeepCMB: Lensing Reconstruction of the Cosmic Microwave Background with Deep Neural Networks





Deblending galaxy superpositions with branched generative adversarial networks

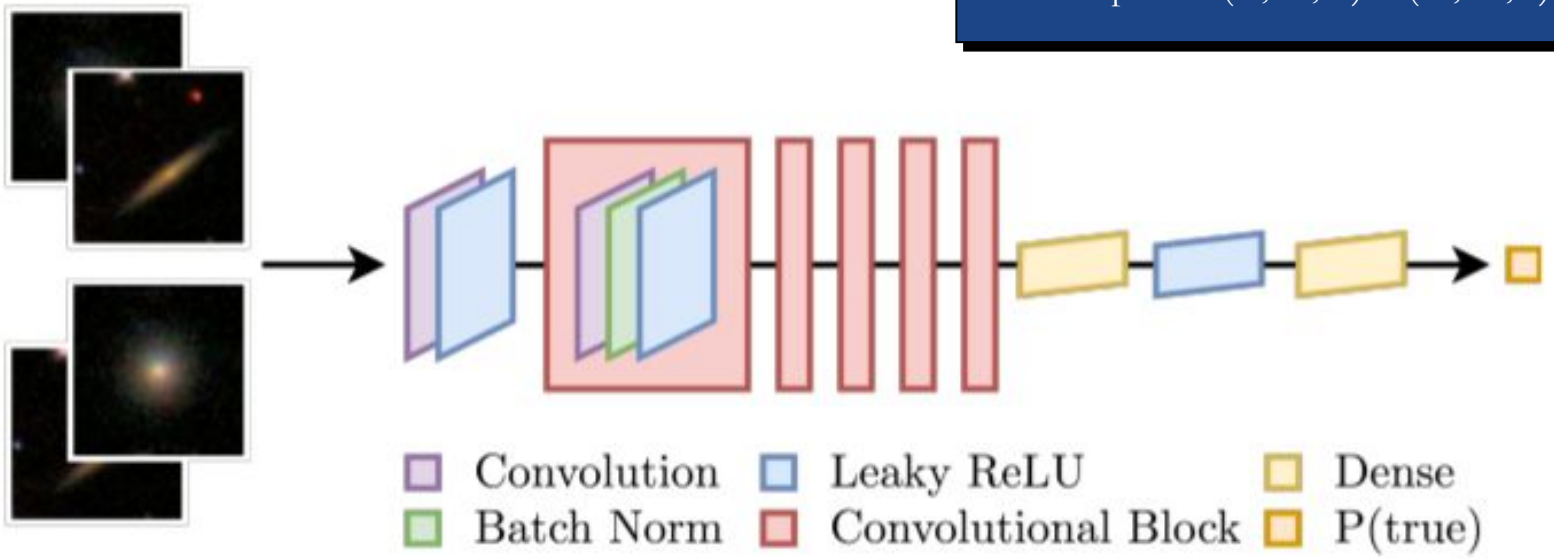
David M. Reiman[1]*†, Brett E. Göhre[1]‡§



Deblending galaxy superpositions with branched generative adversarial networks

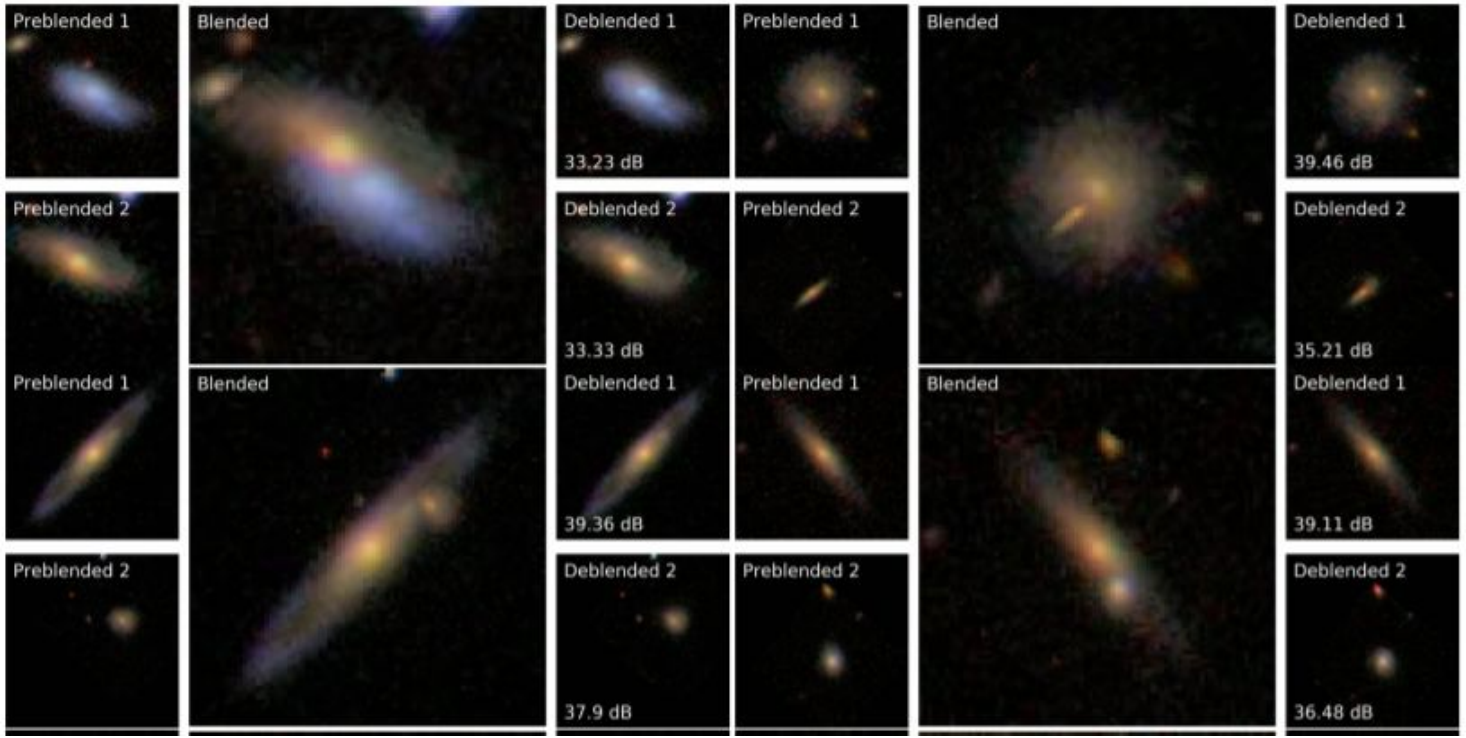
David M. Reiman[1]*†, Brett E. Göhre[1]‡§
[1]Department of Physics, University of California Santa Cruz, 1156 High Street, Santa Cruz, California 95064,

The authors used 141,553 images available from Galaxy Zoo. They used 3 bands in a JPEG format and therefore has a bit depth of 8 bits per color channel with dimension (H, W, C) = (424, 424, 3) where H, W and C are the height, width and channel dimensions, respectively and were cropped and downsampled to (H, W, C) = (80, 80, 3).



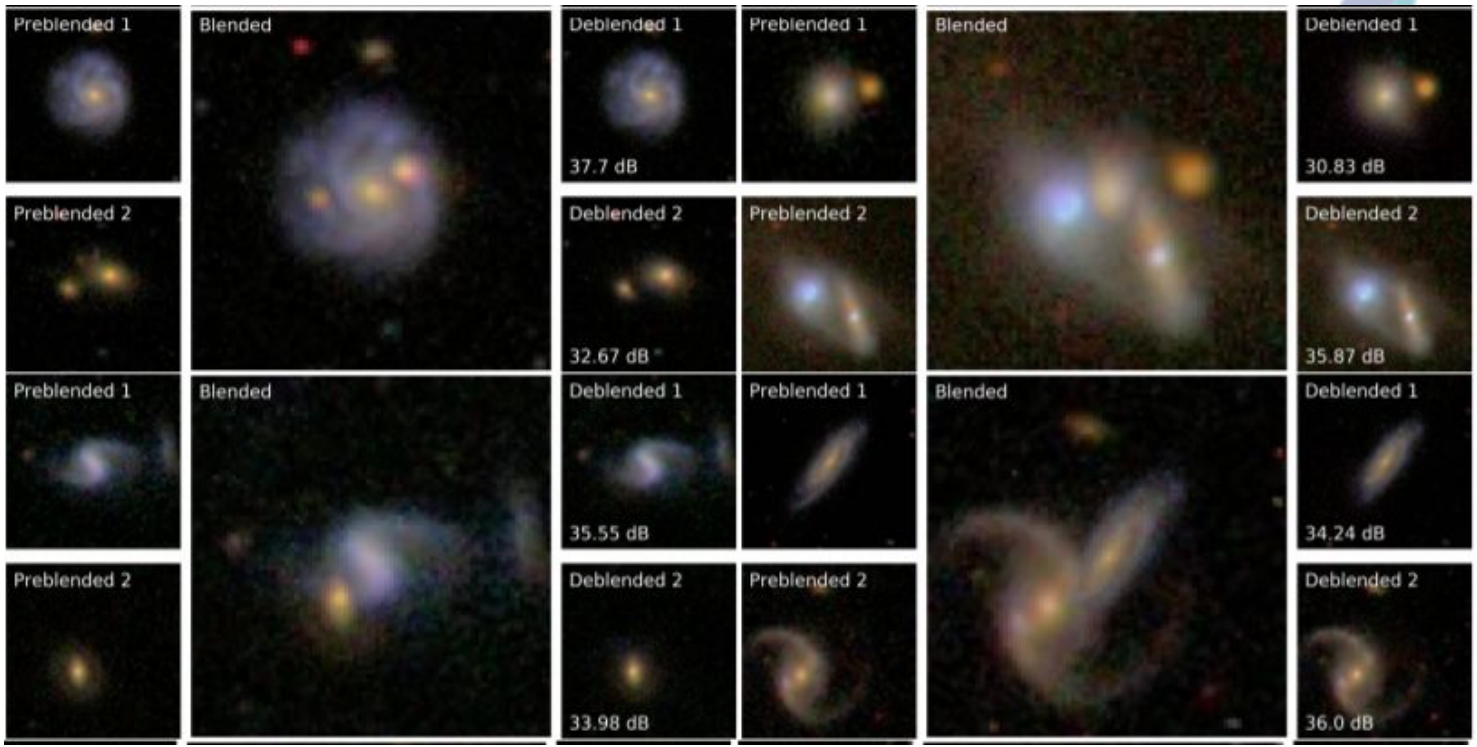
Deblending galaxy superpositions with branched generative adversarial networks

David M. Reiman[1]★†, Brett E. Göhre[1]‡§
[1]*Department of Physics, University of California Santa Cruz, 1156 High Street, Santa Cruz, California 95064, USA*



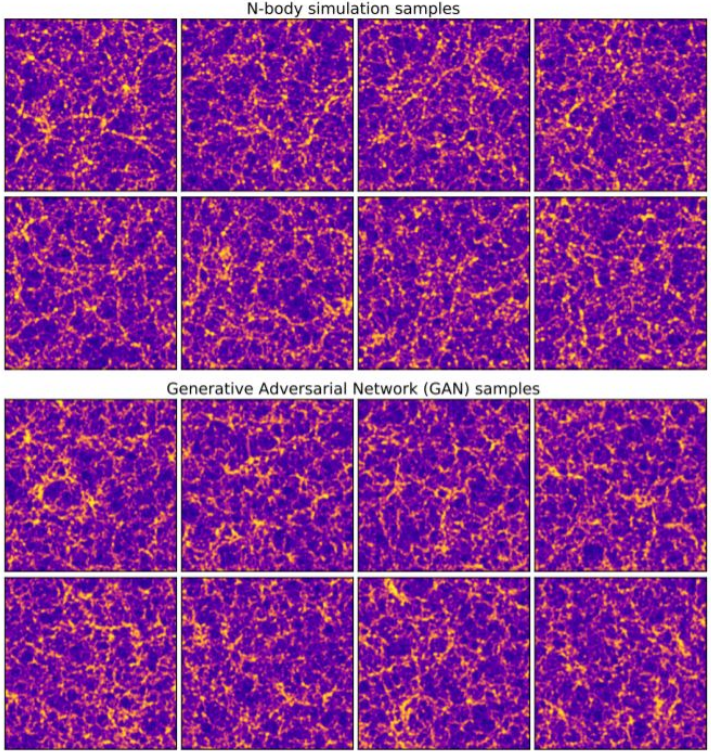
Deblending galaxy superpositions with branched generative adversarial networks

David M. Reiman[1]★†, Brett E. Göhre[1]‡§
[1]*Department of Physics, University of California Santa Cruz, 1156 High Street, Santa Cruz, California 95064, USA*



Fast Cosmic Web Simulations with Generative Adversarial Networks

A.C. Rodríguez,[a] T. Kacprzak,[b] A. Lucchi,[c] A. Amara,[b] R. Sgier,[b] J. Fluri,[b] T. Hofmann,[c] and A. Réfrégier[b]



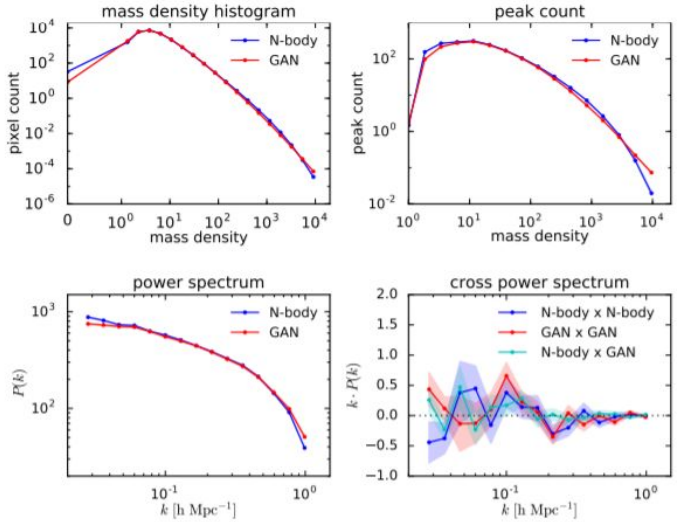
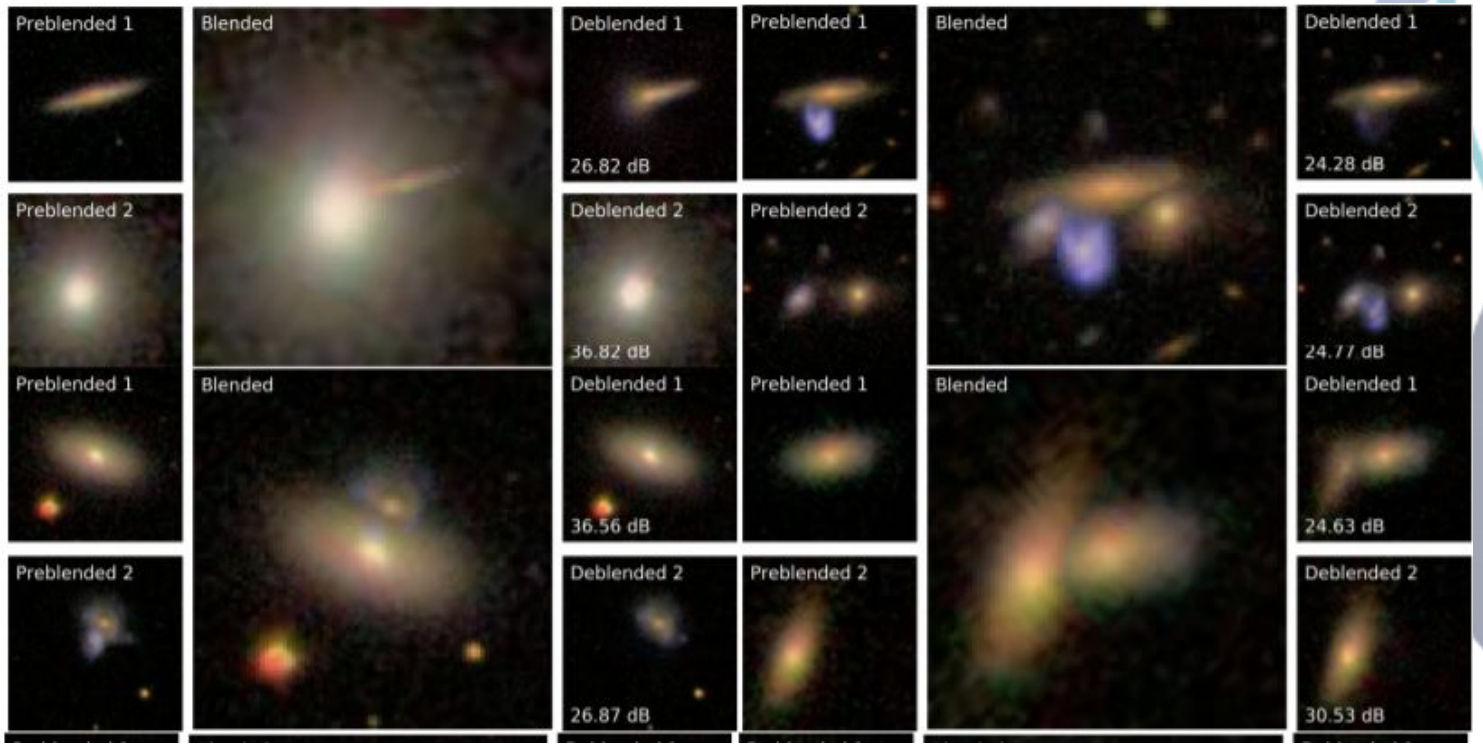


Figure 2: Comparison of summary statistics between N-body and GAN simulations, for box size of 500 Mpc. The statistics are: mass density histogram (upper left), peak count (upper right), power spectrum of 2D images (lower left) and cross power spectrum (lower right). The cross power spectrum is calculated between pairs N-body images (blue points), between pairs of GAN images (red points), and between pairs consisting of one GAN and one N-body image (cyan points). The power spectra are shown in units of h^{-1} Mpc, where $h = \mathrm{H}_0/100$ corresponds to the Hubble parameter. The standard errors on the mean of the shown with a shaded region, and are too small to be seen for the first three panels.

Deblending galaxy superpositions with branched generative adversarial networks

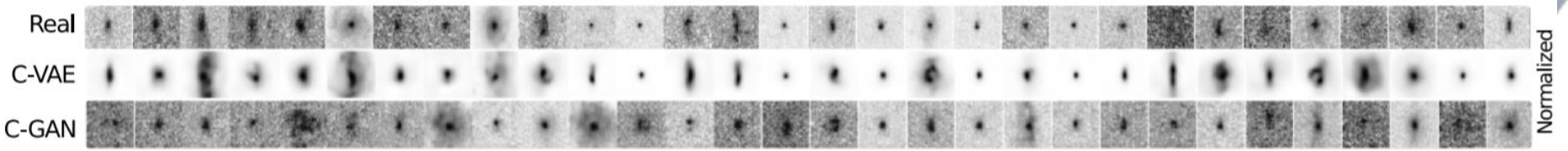
David M. Reiman[1]★†, Brett E. Göhre[1]‡§
[1]*Department of Physics, University of California Santa Cruz, 1156 High Street, Santa Cruz, California 95064, USA*

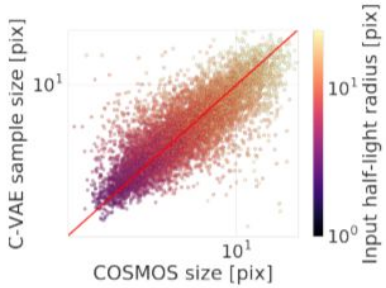
Failures



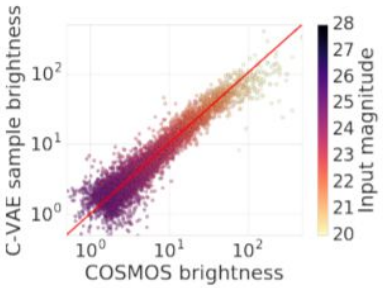
Enabling Dark Energy Science with Deep Generative Models of Galaxy Images

Siamak Ravanbakhsh[1], François Lanusse[2], Rachel Mandelbaum[2], Jeff Schneider[1], and Barnabás Póczos[1]

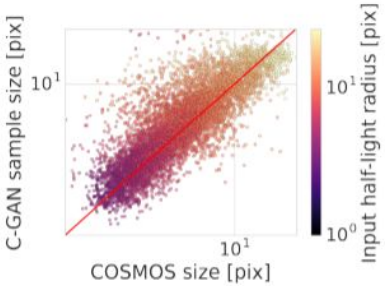




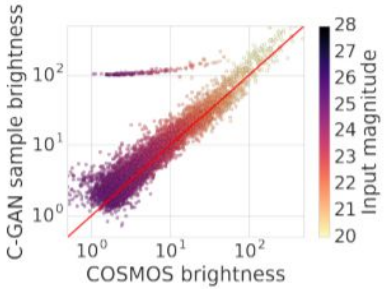
(a) Galaxy sizes



(b) Galaxy brightness

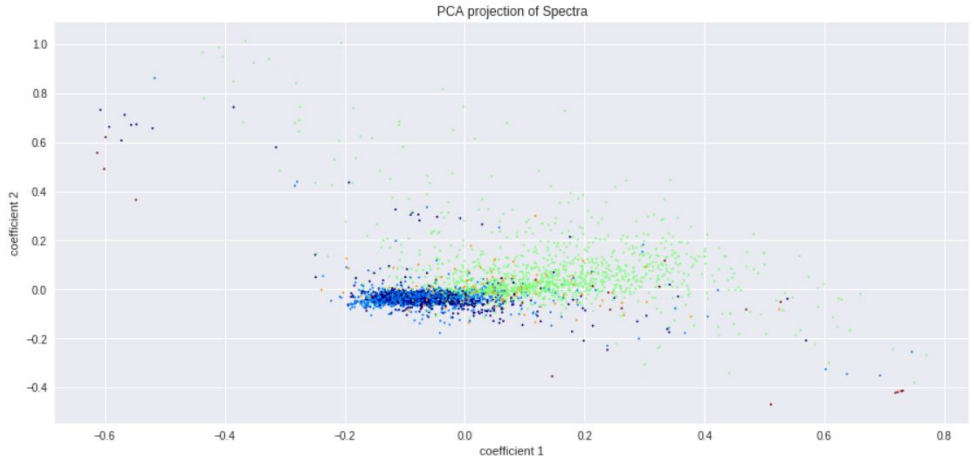


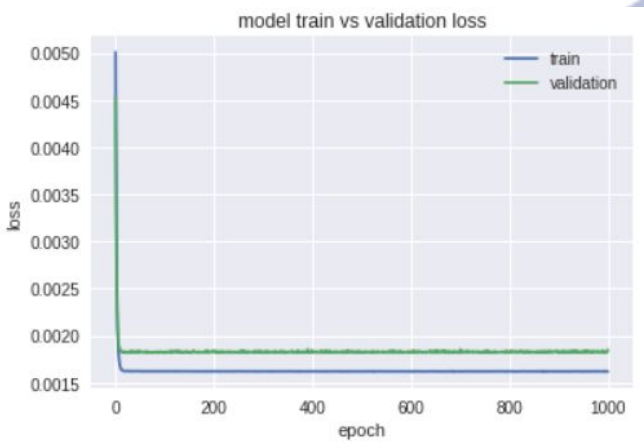
(a) Galaxy sizes



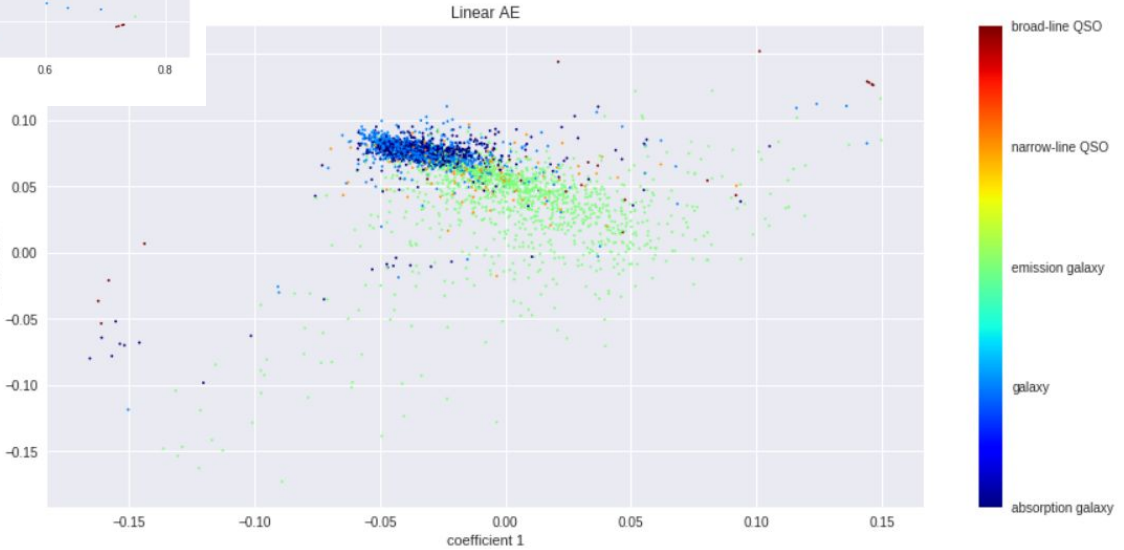
(b) Galaxy brightness

Example - PCA vs AE

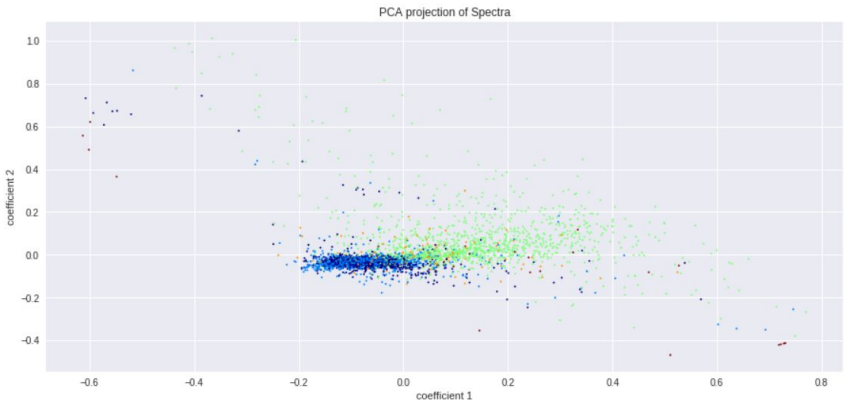


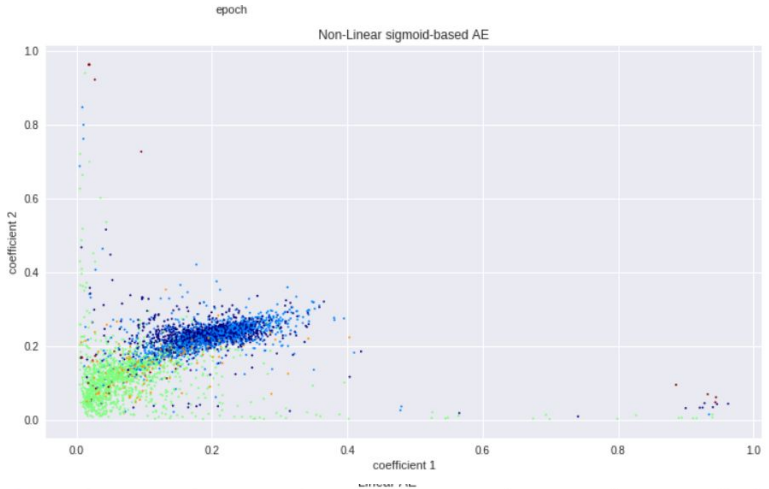


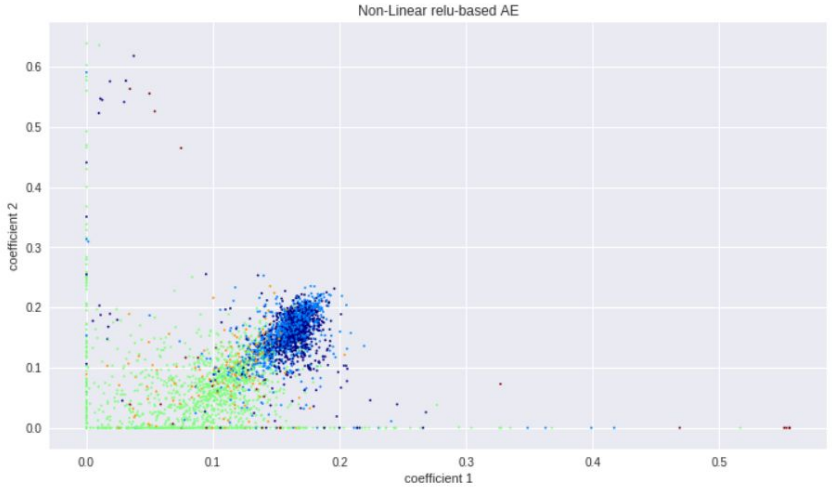
Galaxy Spectra from SDSS available from astroML example. http://www.astroml.org/book_figures/chapter7/fig_PCA_LLE.html

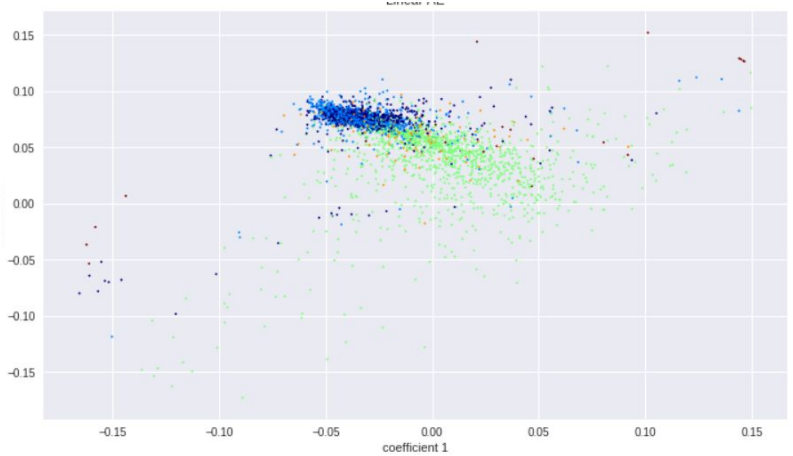


Example - PCA vs AE









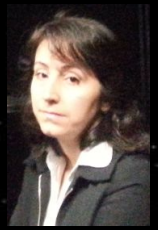
XII ESCOLA DO CBPF

22 de julho a 02 de agosto de 2019

Inteligência Artificial Utilizando *Deep Learning* e Aplicações em Física

Márcio Portes de Albuquerque (CBPF)

Clécio R. De Bom (CBPF/CEFET-RJ)



Elisangela L. Faria (CBPF)

Uma introdução conceitual de aprendizado de máquina e uma abordagem prática em aplicações de redes neurais profundas com foco em aplicações científicas e tecnológicas.

CBPF 70 ANOS

1949 - 2019