ISPR - Midterm 3 - Assignment 1 - Sergio Latrofa - 640584

Hand-wirtten digits denoising using autoencoders

Assignment

Train a denoising or a contractive autoencoder on the MNIST dataset: try out different architectures for the autoencoder, including a single layer autoencoder, a deep autoencoder with only layerwise pretraining and a deep autoencoder with fine tuning. It is up to you to decide how many neurons in each layer and how many layers you want in the deep autoencoder. Show an accuracy comparison between the different configurations.

given the encoding z1 of image x1 and z2 of image x2, a latent space interpolation is an encoding that obtained with the linear interpolation z* = a*z1 + (1 - a)*z2, with a in [0, 1]. Perform a latent space interpolation and visualize the results using:

- z1 and z2 from the same class
- z1 and z2 from different classes Plot the results, for example by showing the image reconstructions for a=0.0, 0.1, 0.2, ..., 1.0. Are the resulting images plausible digits?

Try out what happens if you feed one of the autoencoders with a random noise image and then you apply the iterative gradient ascent process described in the lecture to see if the reconstruction converges to the data manifold.

How the task has been structured

Experimentation has been carried on considering "classical" AE (train and target data identical), and DAE (Denoising Autoencoders). The three versions (flatten, deep, and fine tuend) have been built and trained for both the two families. MSE has been evaluated and compared for the six models along with a series of plotted example of denoise. Linear interpolation has also been tried with all the models, producing overall similar results. At the end, gradient ascent has been setted up for the **Fine Tuned Deep DAE, * confirming the mainfold assumption. Trial with the other AEs have been plotted, but the optimal parameter configuration has not been searched.

Notebook Organization

The notebook is divided in sections (to be executed sequentially). The order of operation is described in the above section. In the various sections you ou will find:

1. Design and Training of the AEs
2. Empirical evaluation on Test Data
3. Linear Interpolation plots
4. Manifold convergence proof
5. Conclusions and final comments

Autoencoder Design and Training

MINST Import

Import MNIST dataset, along with library functions for NN implementation and training, data sampling and digits plot.

```
import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

from random import seed, sample, choice

from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split

from keras import layers, losses
from keras.models import Model, clone_model

seed(123)
```

```
print("TensorFlow version:", tf.__version__)
```

```
TensorFlow version: 2.8.0
```

```
from keras.datasets import mnist

(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.

print (x_train.shape)
print (x_test.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
```

Further split training data into Training Set and Validation Set,the Test Set remains the one provided.

```
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:])))
```

```
print (x_train.shape)
print (x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

```
x_val = x_train[50000:60000]
y_val = y_train[50000:60000]

x_train = x_train[0:50000]
y_x_train = y_train[0:50000]
```

```
FLATTEN_INPUT_DIM = 784
```

Pick *n* random digits from the test set (Useful for examples plots).

```
def random_digits(n = 10, same_digit=False, noise=False):
  mnist_sample = []
  if noise:
    mnist_sample = x_test_noise[ np.where(y_test == same_digit ) ] if same_digit else x_te
  else:
    mnist_sample = x_test[ np.where(y_test == same_digit ) ] if same_digit else x_test

  return np.array(sample(list(mnist_sample), n))
```

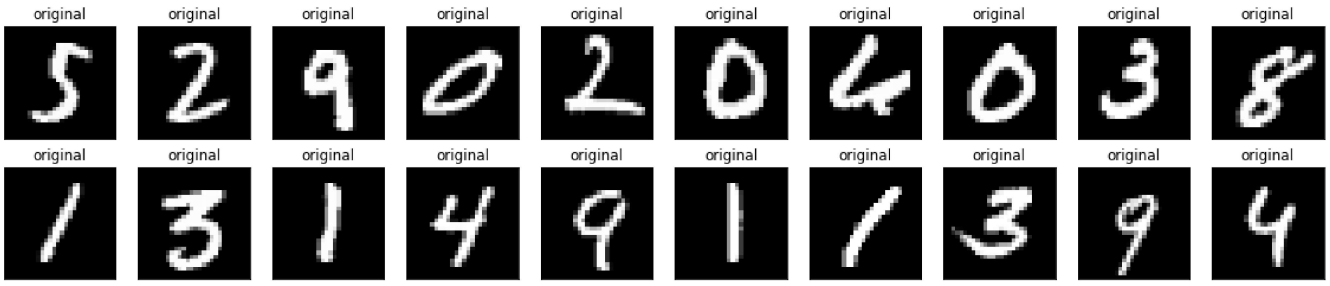
```
def all_digits(digit, dset = (x_test, y_test)):
  xs, ys = dset
  return xs[np.where(ys == digit)]
```

Plotting Functions

Visually compare *n* original digits and the their reconstruction.

```
def plot_rec(x_test, x_rec, n=10, rec=True):
  plt.figure(figsize=(20, 4))
  for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.title("original")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(x_rec[i].reshape(28, 28))
    plt.title("reconstructed" if rec else "original")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
  plt.show()
```

```
plot_rec(random_digits(), random_digits(), rec=False)
```

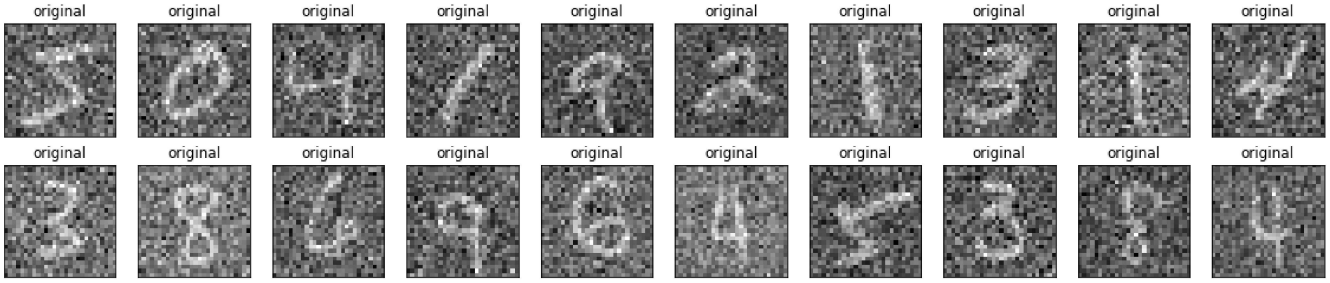


Create a noised version of the MNIST

Use Gausssian Noise level from Keras to create a "corrupted" version of the dataset, and train or

```
noiser = layers.GaussianNoise(0.4, 0)
x_train_noise = noiser(x_train, training=True).numpy()
x_val_noise = noiser(x_val, training=True).numpy()
x_test_noise = noiser(x_test,  training=True).numpy()
```

```
plot_rec(x_train_noise, x_val_noise, rec=False)
```



Fai doppio clic (o premi Invio) per modificare

One level AE

Let's begin implementing a linear autoencoder, with one internal layer only.

```
LATENT_DIM = 128
```

```
def design_AE(latent_dim, input_size):

  class Autoencoder(Model):
    def __init__(self, latent_dim):
      super(Autoencoder, self).__init__()
      self.latent_dim = latent_dim

      self.encoder = tf.keras.Sequential([
        layers.Dense(latent_dim, activation='relu'),
      ])

      self.decoder = tf.keras.Sequential([
        layers.Dense(input_size, activation='sigmoid')
      ])

```

```
    def call(self, x):
      encoded = self.encoder(x)
      decoded = self.decoder(encoded)
      return decoded

  return Autoencoder(latent_dim)
```

```
AE = design_AE(LATENT_DIM, FLATTEN_INPUT_DIM )
```

```
AE.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

```
AE.fit(x_train, x_train,
               epochs=5,
               shuffle=True,
               validation_data=(x_val, x_val)
               )
```

```
Epoch 1/5
1563/1563 [==============================] - 9s 5ms/step - loss: 0.0199 - val_loss: 0.0
Epoch 2/5
1563/1563 [==============================] - 7s 5ms/step - loss: 0.0047 - val_loss: 0.0
Epoch 3/5
1563/1563 [==============================] - 7s 5ms/step - loss: 0.0031 - val_loss: 0.0
Epoch 4/5
1563/1563 [==============================] - 7s 5ms/step - loss: 0.0026 - val_loss: 0.0
Epoch 5/5
1563/1563 [==============================] - 7s 5ms/step - loss: 0.0023 - val_loss: 0.0
<keras.callbacks.History at 0x7f1c15af6790>
```

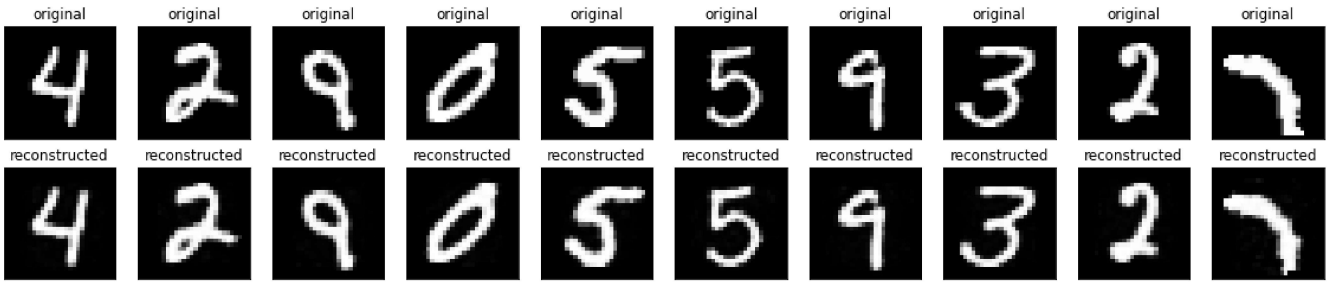
Check dimensionality of the encoding.

```
r_digits = random_digits()
r_digits.shape
```

```
(10, 784)
```

Check if the reconstruction works as expected.

```
plot_rec(r_digits, AE(r_digits).numpy())

```



One level deonsing AE

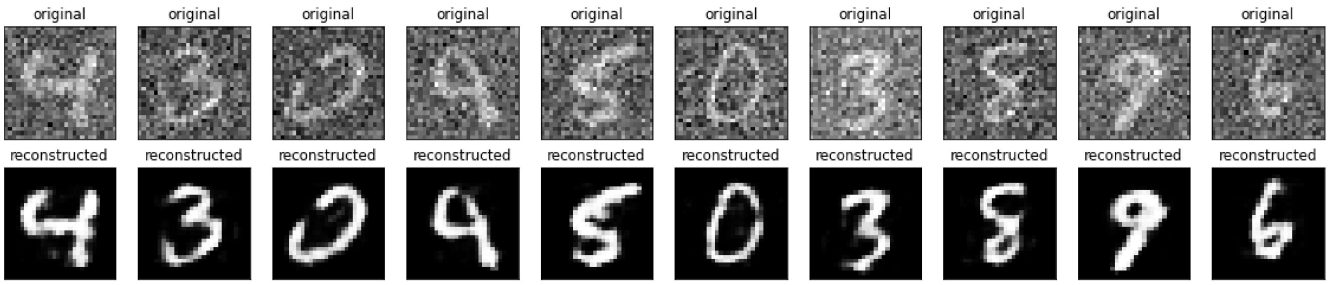
To make our flatten AE a Denoisng one, jus train it on the "currupted" version of the training data. Target will be their clean version.

```
denoising_AE = design_AE(LATENT_DIM, FLATTEN_INPUT_DIM )
denoising_AE.compile(optimizer='adam', loss=losses.MeanSquaredError())
denoising_AE.fit(x_train_noise, x_train,
                 epochs=10,
                 shuffle=True,
                 validation_data=(x_val_noise, x_val)
                 )
```

```
Epoch 1/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.0264 - val_loss: 0.0
Epoch 2/10
1563/1563 [==============================] - 7s 5ms/step - loss: 0.0133 - val_loss: 0.0
Epoch 3/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.0120 - val_loss: 0.0
Epoch 4/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.0115 - val_loss: 0.0
Epoch 5/10
1563/1563 [==============================] - 7s 5ms/step - loss: 0.0113 - val_loss: 0.0
Epoch 6/10
1563/1563 [==============================] - 7s 5ms/step - loss: 0.0111 - val_loss: 0.0
Epoch 7/10
1563/1563 [==============================] - 7s 5ms/step - loss: 0.0110 - val_loss: 0.0
Epoch 8/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.0109 - val_loss: 0.0
Epoch 9/10
1563/1563 [==============================] - 10s 6ms/step - loss: 0.0108 - val_loss: 0.
Epoch 10/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.0107 - val_loss: 0.0
<keras.callbacks.History at 0x7f1c12f4b650>
```

Check the digit reconstruction:

```
1 r_digits = random_digits(noise=True)
2 plot_rec(r_digits, denoising_AE(r_digits).numpy())
```



Deep AE - Layerwise pre-training

Define here the number of units for each layer (correpsponding to the encoding dimensionality).

Trials have demonstrated that adding more than two hidden layers does not produce good result for the MNIST task.

```
1 LATENT_DIMS = [ 512, 128 ]
```

Set a fixed (low) number of epochs, equal for each layer, depending on the time you want to spend waiting the training pahse to end.

```
1 EPOCHS_NUM = 5
```

Train each laywer as an independe auotoencoder, fed with the data encoded at the previous level. An array of AEs will be produced.

```
1 def layerwise_train(x_train,  latent_dims, initial_shape, x_val):
2   levels = []
3   last_enc = x_train
4
5   for i in range(len(latent_dims)):
6     input_size = initial_shape if i == 0 else latent_dims[i-1]
7     autoencoder =  design_AE(latent_dims[i], input_size)
8
9     #Train autoencoder on last encoding.
10    autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

```
    autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())

    autoencoder.fit(last_enc, last_enc, #not sure if last dec or last enc
                    epochs=EPOCHS_NUM,
                    shuffle=True
                    )

    last_enc = autoencoder.encoder(last_enc).numpy()
    levels.append(autoencoder)

  return levels
```

```
LEVELS = layerwise_train(x_train, LATENT_DIMS, FLATTEN_INPUT_DIM, x_val)
```

```
Epoch 1/5
1563/1563 [==============================] - 17s 11ms/step - loss: 0.0109
Epoch 2/5
1563/1563 [==============================] - 15s 10ms/step - loss: 0.0023
Epoch 3/5
1563/1563 [==============================] - 15s 9ms/step - loss: 0.0016
Epoch 4/5
1563/1563 [==============================] - 15s 9ms/step - loss: 0.0014
Epoch 5/5
1563/1563 [==============================] - 15s 9ms/step - loss: 0.0012
Epoch 1/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.6023
Epoch 2/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.5617
Epoch 3/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.5561
Epoch 4/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.5546
Epoch 5/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.5536
```

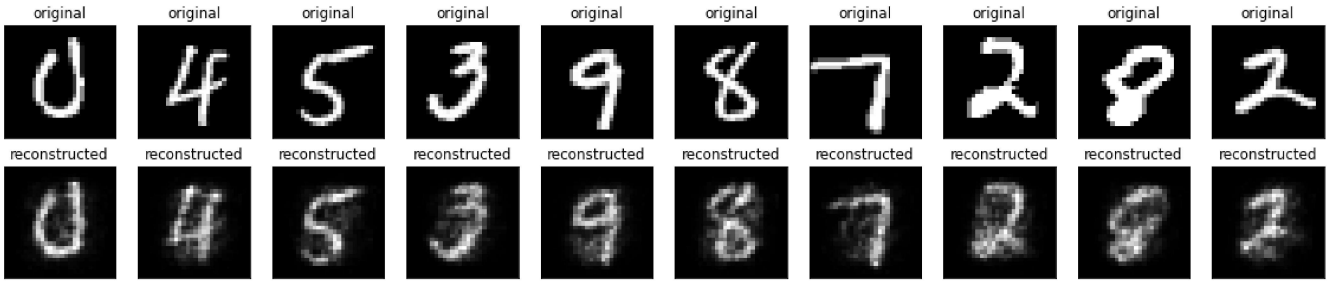
Stack the different AEs one on top of each other, actually obtaining a single **deep** model.

```
def design_deep_AE(levels, latent_dims, input_size):
    class DeepAutoencoder(Model):
      def __init__(self, levels, latent_dims, input_size ):
        super(DeepAutoencoder, self).__init__()
        self.latent_dims = latent_dims

        self.encoder = tf.keras.Sequential()
        self.encoder.add(tf.keras.Input(shape=(input_size,)))
        for i in range(len(levels)):
          enc_layer = layers.Dense(latent_dims[i], activation='relu')
          self.encoder.add(enc_layer)

        self.decoder = tf.keras.Sequential()
        self.decoder.add(tf.keras.Input(shape=(latent_dims[len(latent_dims)-1],)))
        for i in reversed(range(len(latent_dims)-1)):
```

```
        for i in reversed(range(len(latent_dims) - 1)):
          dec_layer = layers.Dense(latent_dims[i], activation='sigmoid')
          self.decoder.add(dec_layer)

        self.decoder.add(layers.Dense(input_size, activation='sigmoid'))

      def call(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

    dae = DeepAutoencoder(levels, latent_dims, input_size)
    dae.compile(optimizer='adam', loss=losses.MeanSquaredError())

    #Transfer weights from previous pretained layers
    for i in range(len(levels)):
      dae.encoder.get_layer(index=i).set_weights(levels[i].encoder.get_weights())
      dae.decoder.get_layer(index=i).set_weights(levels[len(levels) -1 - i].decoder.get_we

    return dae
```

```
deep_AE = design_deep_AE(LEVELS, LATENT_DIMS, FLATTEN_INPUT_DIM)
```

```
r_digits = random_digits()
plot_rec(r_digits, deep_AE(r_digits).numpy())
```



Deep Denoisng AE - Layerwise pre-training

To layerwise pretrain the DAE, Gaussian noise (at a certain std) will be added to each new input encoding and so one.

Input training data have been provided in their noised version too, as an experimentation choiche.

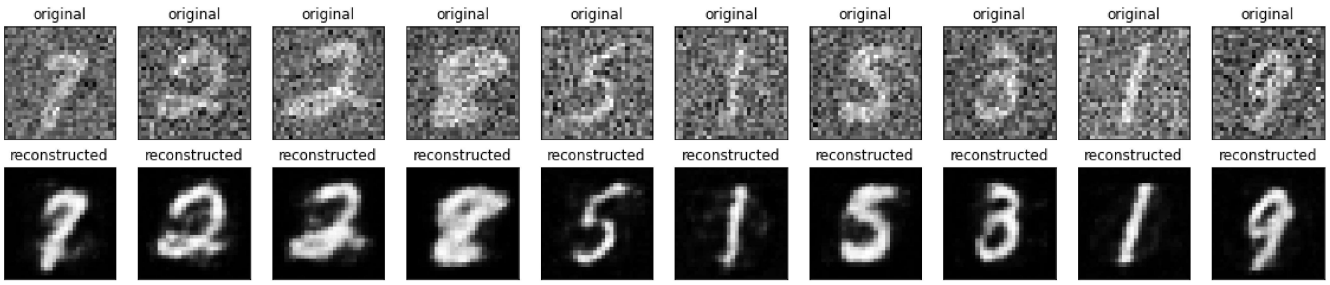
```
def layerwise_train_noise(x_noise, x_clean,  latent_dims, initial_shape, noise_std = 0.2):
  levels = []

  noisy_enc = x_noise
  clean_enc = x_clean

  noiser = layers.GaussianNoise(noise_std, 0)

  for i in range(len(latent_dims)):
    input_size = initial_shape if i == 0 else latent_dims[i-1]
    autoencoder =  design_AE(latent_dims[i], input_size)

    #Train autoencoder on last encoding.
    autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())

    autoencoder.fit(noisy_enc, clean_enc,
                    epochs=EPOCHS_NUM,
                    shuffle=True
                    )
    # We wish to learn the clean encoding.
    clean_enc = autoencoder.encoder(clean_enc).numpy()
    # From "corrupted" training examples
    noisy_enc = noiser(clean_enc, training=True)

    levels.append(autoencoder)
  return levels
```

```
DENOISING_LEVELS = layerwise_train_noise(x_train_noise, x_train, LATENT_DIMS, FLATTEN_INPU
```

```
Epoch 1/5
1563/1563 [==============================] - 15s 9ms/step - loss: 0.0190
Epoch 2/5
1563/1563 [==============================] - 14s 9ms/step - loss: 0.0110
Epoch 3/5
1563/1563 [==============================] - 15s 9ms/step - loss: 0.0099
Epoch 4/5
1563/1563 [==============================] - 15s 9ms/step - loss: 0.0093
Epoch 5/5
1563/1563 [==============================] - 15s 9ms/step - loss: 0.0089
Epoch 1/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.3598
Epoch 2/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.3252
Epoch 3/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.3215
Epoch 4/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.3201
Epoch 5/5
1563/1563 [==============================] - 5s 3ms/step - loss: 0.3190
```

```
deep_denosing_AE = design_deep_AE(DENOISING_LEVELS, LATENT_DIMS, FLATTEN_INPUT_DIM)
```

```
1 r_digits = random_digits(noise=True)
2 plot_rec(r_digits, deep_denosing_AE(r_digits).numpy())
```



Fine Tuned Deep AE

[ ] ↳ *5 celle nascoste*

Fine Tuned Deep Denoising AE

[ ] ↳ *4 celle nascoste*

Test Set Evaluation

[ ] ↳ *25 celle nascoste*

Linear Interpolation

Utility functions

[ ] ↳ *15 celle nascoste*

Interpolation plots - Same Digits

[ ] ↳ *13 celle nascoste*

Interpolation plots - Different Digits

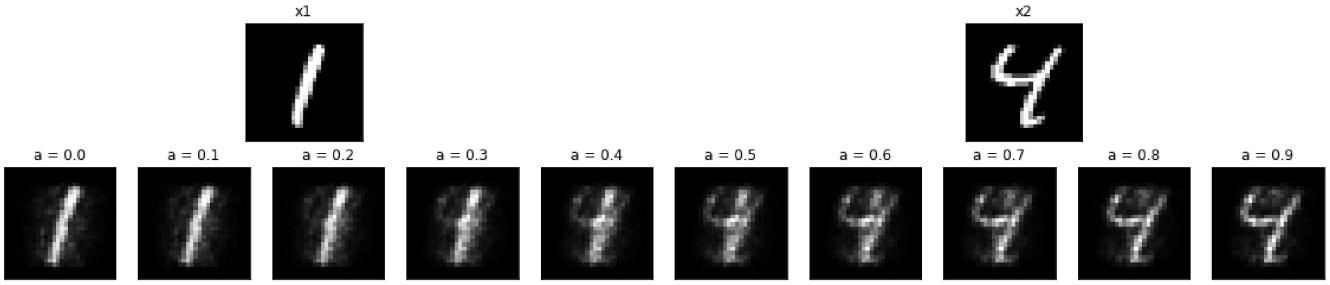
```
# No generalized random different digits function here... :(
different_digits = [x_test[5], x_test[6]]
```

Flat AE - Different Class Images

[ ] ↳ *1 cella nascosta*

Deep AE - Different Class Images

```
original, interp = incremental_interpolation(different_digits, deep_AE)
plot_interp(original, interp)
```

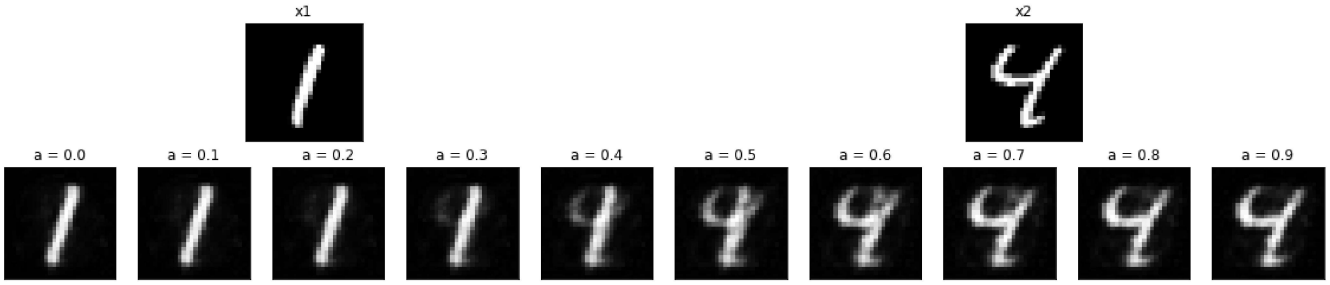


Denoising AE - Same Class Images

```
original, interp = incremental_interpolation(different_digits, denoising_AE)
plot_interp(original, interp)
```

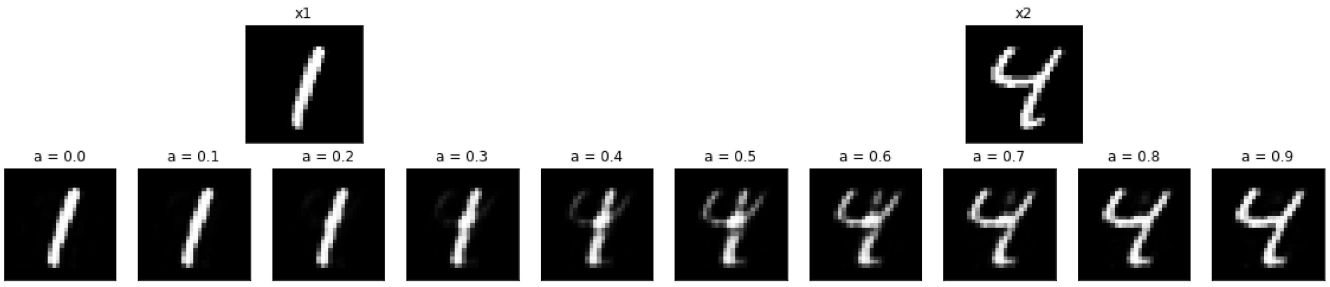
Deep Denoising AE - Different Class Images

```
1 original, interp = incremental_interpolation(different_digits,deep_denosing_AE)
2 plot_interp(original, interp)
```



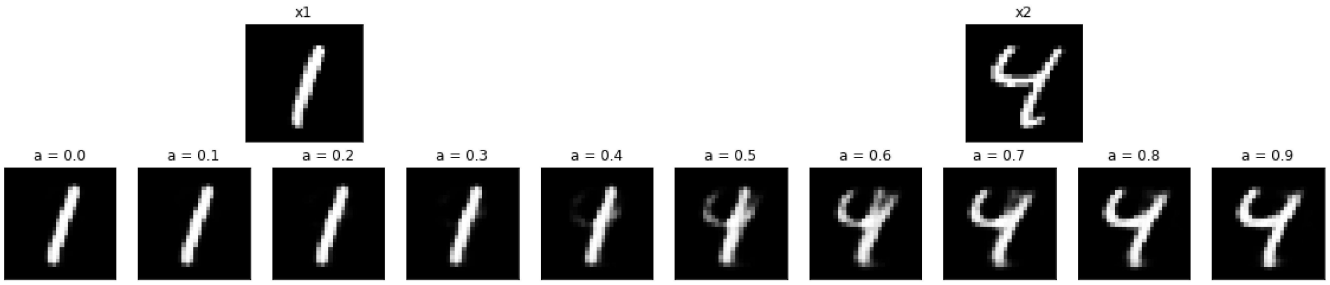
Fine Tuned AE - Different Class Images

```
1 original, interp = incremental_interpolation(different_digits, ft_AE)
2 plot_interp(original, interp)
```



Fine Tuned AE - Different Class Images

```
1 original, interp = incremental_interpolation(different_digits, denoising_ft_AE)
2 plot_interp(original, interp)
```



Gradient Ascent and Convergence to Manifold

Implementation

First, calcolate the mean over all images of a certain digit: the one that we want to "visit" the manifold of.

```
1 def digit_mean(d=0):
2   all_d_digit = x_test[np.where(y_test == d)]
3   mean = np.mean(all_d_digit, axis=0)
4   plt.imshow(mean.reshape(28, 28))
5   plt.show()
6   return mean
```

```
1 mean = digit_mean(4)
```

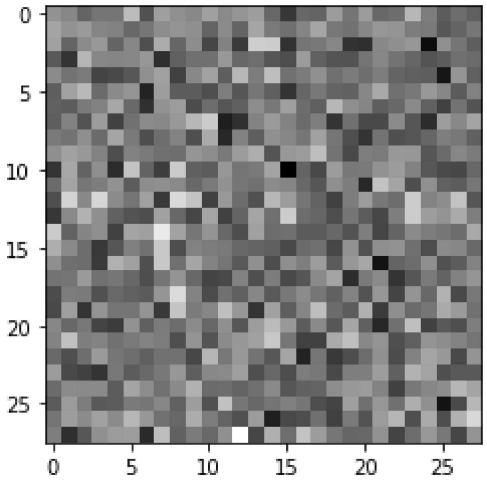
Generate a random "pure noise" image

```
1 np.random.rand(2,784)
```

```
array([[0.30994262, 0.56180932, 0.89409349, ..., 0.91739786, 0.19239294,
        0.05881813],
       [0.50095271, 0.49221619, 0.28111015, ..., 0.44367629, 0.36182941,
        0.4950016 ]])
```



```
1 pure_noise = layers.GaussianNoise(1, 0)(np.random.rand(1,784),  training=True).numpy()
2 plt.imshow(pure_noise.reshape(28, 28))
3 plt.show()
```



We will use the loss function to gradually correct our noise image, being it proportional to the gradient of the probability distribution of the digit.

$$g(\boldsymbol{h}) - \boldsymbol{x} \propto \frac{\partial \log p(\boldsymbol{x})}{\partial x}$$

```
1 def calc_loss(input, target, model):
2   return (model(input) - target)
```

During the gradient ascent iterative process, we add the gradient the noisy image in order to move

```
def ascention(input, target, model, n=10, eta=1, normalization=False, inversion=False):
  target = model(np.expand_dims(target, 0))
  imgs = []
  for i in range(n):
    #Use loss as gradient, exploitng proportionaliity
    gradient = calc_loss(input, target, model)

    input = input + gradient*eta
    # Normalize at each step to avoid float explosion
    if normalization:
      input /= np.linalg.norm(input)

    imgs.append(-(input.numpy()) if inversion else input.numpy())
  return imgs
```

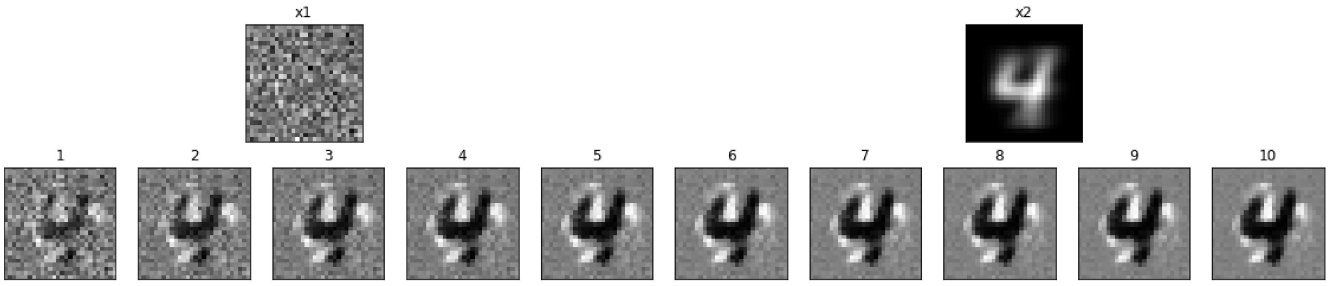
Let's test the gradient ascent iterative process...

Search of parameters (Deep Fine Tuned DAE)

In this first setting, we need to increase the loss in order to make the changing visible.

```
mnfd = ascention( input=np.expand_dims(pure_noise[0], 0), target=mean,  model=denoising_ft
```

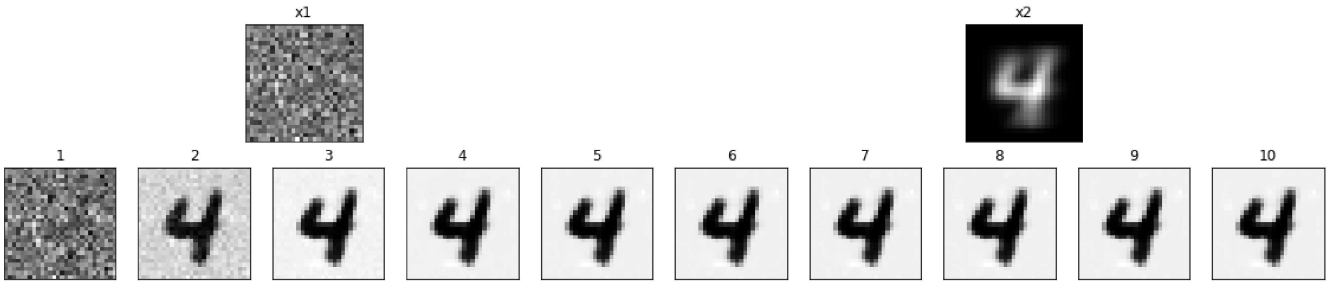
```
plot_interp([pure_noise, mean ], mnfd , step_idx=False)
```



To visaully improve the results we normalized the image at each step. Obtaining a more clear result

```
mnfd = ascention( input=np.expand_dims(pure_noise[0], 0),target=mean,  model=denoising_ft_
```

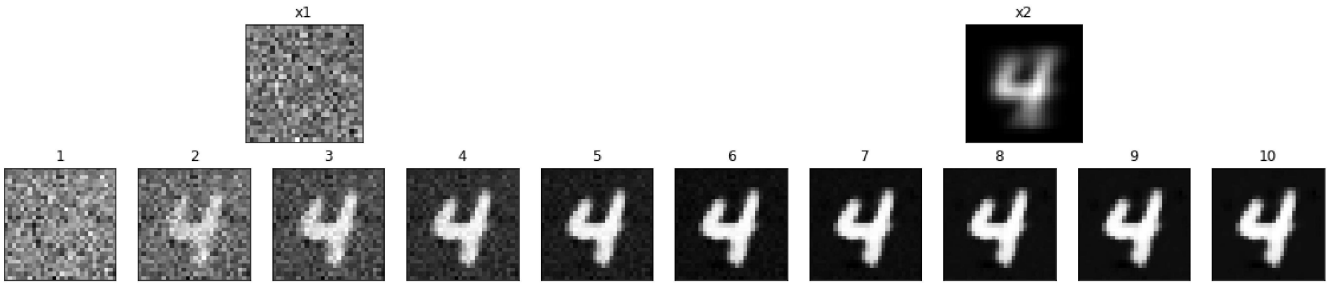
```
1 plot_interp([pure_noise, mean ], mnfd , step_idx=False)
```



Finally, inverting the grayscale colors we can obtain a "MNIST style" result.

```
1 mnfd = ascention( input=np.expand_dims(pure_noise[0], 0),target=mean,  model=denoising_ft_
```

```
1 plot_interp([pure_noise, mean ], mnfd , step_idx=False)
```



Convergence of the other models

[ ] ↳ *12 celle nascoste*

Conclusions

Results

Six different type of AEs have been trained and test. Best results in terms of error between original digit and denoised representation were obtained by the one level **DAE** and by the **Deep Fine Tuned DAE**. Given the slisgtly lower MSE and the considerable difference in training type the best AE for the MNIST task is the "flatten" one.

Comparing DAE and AE cosniderably highlights the importance of the addition of noise during the training phase, particularly in the "deep architectures" cases.

The convergence to the manifold of a digit has been proven. In particular the plotted reuslts of the gradient ascent process show how the AE can be used to approximate a distribution, revealing in some way interesting potential for **generative tasks**.

Interploation plots also allow to understand diffrences between gradient ascent and trivial "image copy" processes.

References

- **Deep Learning book - Ch. 14** - https://www.deeplearningbook.org/contents/autoencoders.html
- **Learning Dynamics of Linear Denoising Autoencoders - Arnu Pretorius, Steve Kroon, Herman Kamper** - https://arxiv.org/abs/1806.05413#:~:text=Denoising%20autoencoders%20(DAEs)%20have%20proven,noise%20influences%20learning%20in%20DAEs.
- **Deep Dreaming Tensowrflow tutorial** (Gradient Ascent in TF) - https://www.tensorflow.org/tutorials/generative/deepdream

Comments and Feedback

The assignment as a whole has been very useful to better understand differences between specifc AE types, and clarify their internal behaviour.

Mathematical bounds with matrices low rank approximation are more clear now, as far as the meaning of the manifold assumption, now understood as an optimization problem, thanks to the SGD-like approach implemented.

In particular lot of links with the NN weights optimization emerged (it's quite trivial, cosindering AE are basically NN), ending up with a larger overall personal vision.

A lot of practice with TF has also been done to accomplsih all the tasks, also solving "low level" issues with tensor shapes, acquiring AI debugging skills.

With respscet to the previous midterms this one was harder, both in terms of theoretical compexity that in programming effort, at least more than personally expectd, even though the icnrease of the complexity was announced.

I hope my feedback to be useful.

✓ 0 s data/ora di completamento: 18:56