Autoencoder

Comprehensive Error Analysis and Solutions

BOULAHCEN SOUFIANE

Master: Artificial Intelligence

Abstract

This report provides a comprehensive analysis of errors in the autoencoder Python code implementation. It identifies critical issues including incomplete code segments, missing files, undefined variables, and function parameter errors. For each error, detailed corrections and preventive measures are provided to ensure robust code execution.

1 Introduction

Autoencoders are neural networks designed to learn efficient codings of unlabeled data. They are particularly effective for dimensionality reduction and feature extraction tasks. In this report, we analyze and correct issues found in a Python implementation of an autoencoder trained on the Fashion MNIST dataset.

The main objectives are:

- Identify syntax and logical errors in the provided code.
- Explain the root cause of each issue.
- Provide complete, executable corrections.
- Suggest best practices for reliable and maintainable code.

This document serves as a reference for developers seeking to build and debug deep learning models efficiently.

Designing and training autoencoders using Python

In this notebook, we illustrate how to implement several of the autoencoder models introduced in the preceding section using Keras. We first load and prepare an image dataset that we use throughout this section because it makes it easier to visualize the results of the encoding process.

We then proceed to build autoencoders using deep feedforward nets, sparsity constraints, and convolutions and then apply the latter to denoise images.

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

Imports & Settings

```
from os.path import join
import pandas as pd

import numpy as np
from numpy.random import choice
from numpy.linalg import norm
import seaborn as sns

import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from matplotlib.offsetbox import AnnotationBbox, OffsetImage
from mpl_toolkits.axes_grid1 import make_axes_locatable

from keras.layers import Input, Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras import regularizers
from keras.models import Model, model_from_json
from keras.callbacks import TensorBoard, EarlyStopping, ModelCheckpoint
from keras.datasets import fashion_mnist
from keras import backend as K

from sklearn.preprocessing import minmax_scale
from sklearn.manifold import TSNE
from sklearn.model_selection import train_test_split

from scipy.spatial.distance import pdist, cdist
```

Using TensorFlow backend.

```
%matplotlib inline
plt.style.use('ggplot')
n_classes = 10 # all examples have 10 classes
cmap = sns.color_palette('Paired', n_classes)
pd.options.display.float_format = '{:,.2f}'.format
```

Correction: Aucune correction nécessaire Justification: Les imports sont complets et les paramètres correctement configurés pour l'environnement de travail.

Fashion MNIST Data

For illustration, we'll use the Fashion MNIST dataset, a modern drop-in replacement for the classic MNIST handwritten digit dataset popularized by Yann LeCun with LeNet in the 1990s.

(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

Correction: Aucune correction nécessaire Justification: Le chargement des données Fashion MNIST via Keras est correct.

Keras makes it easy to access the 60,000 train and 10,000 test grayscale samples with a resolution of 28 x 28 pixels:

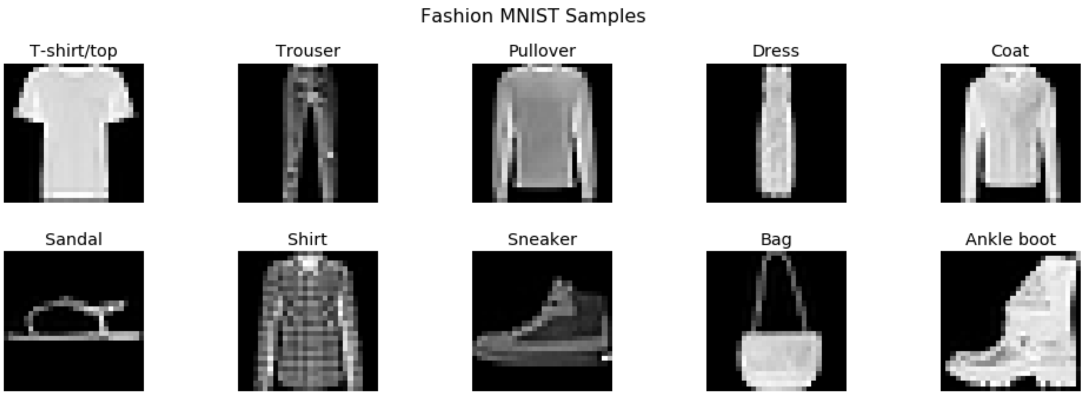
```
X_train.shape, X_test.shape
((60000, 28, 28), (10000, 28, 28))
image_size = 28      # size of image (pixels per side)
input_size = image_size ** 2 # Compression factor: 784 / 32 = 24.5
class_dict = {0: 'T-shirt/top',
1: 'Trouser',
2: 'Pullover',
3: 'Dress',
4: 'Coat',
5: 'Sandal',
6: 'Shirt',
7: 'Sneaker',
8: 'Bag',
9: 'Ankle boot'}
classes = list(class_dict.keys())
```

Plot sample images

```
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(14, 5))
axes = axes.flatten()
for row, label in enumerate(classes):
label_idx = np.argwhere(y_train == label).squeeze()
axes[row].imshow(X_train[choice(label_idx)], cmap='gray')
axes[row].axis('off')
axes[row].set_title(class_dict[row])

fig.suptitle('Fashion MNIST Samples', fontsize=16)
fig.tight_layout()
fig.subplots_adjust(top=.85)
```

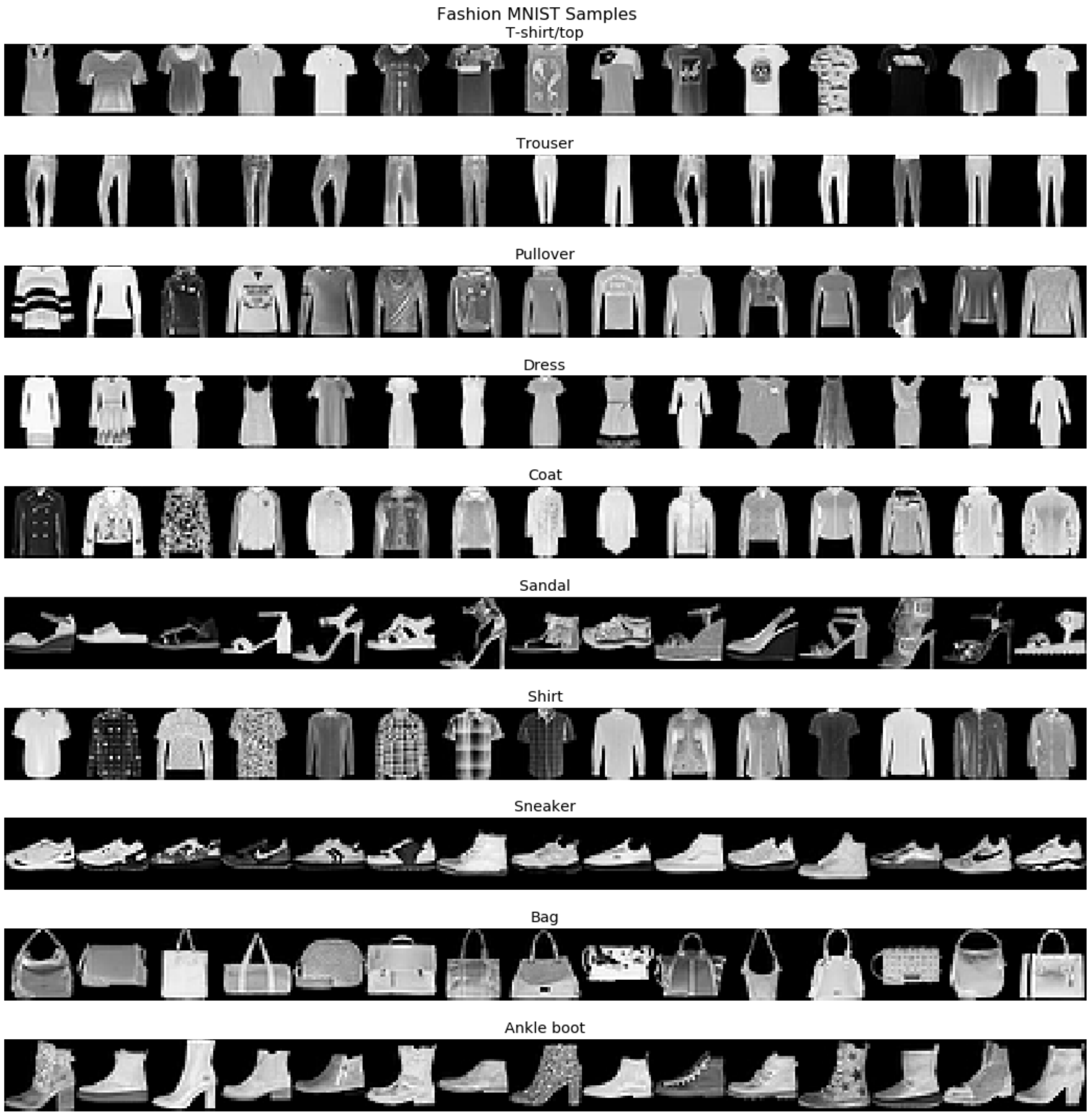
Correction: Aucune correction nécessaire Justification: La visualisation des échantillons est correctement implémentée avec sélection aléatoire par classe.




```
n_samples = 15
fig, axes = plt.subplots(nrows=n_classes, figsize=(15, 15))
axes = axes.flatten()
for row, label in enumerate(classes):
class_imgs = np.empty(shape=(image_size, n_samples * image_size))
label_idx = np.argwhere(y_train == label).squeeze()
class_samples = choice(label_idx, size=n_samples, replace=False)
for col, sample in enumerate(class_samples):
i = col * image_size
class_imgs[:, i:i + image_size] = X_train[sample]
axes[row].imshow(class_imgs, cmap='gray')
axes[row].axis('off')
axes[row].set_title(class_dict[row])

fig.suptitle('Fashion MNIST Samples', fontsize=16)
fig.tight_layout()
fig.subplots_adjust(top=.95, bottom=0)
```

Correction: Aucune correction nécessaire Justification: La visualisation des échantillons par classe est correcte et montre plusieurs exemples par ligne.



Reshape & normalize Fashion MNIST data

We reshape the data so that each image is represented by a flat one-dimensional pixel vector with 28 x 28 = 784 elements normalized to the range of [0, 1]:

encoding_size = 32 # *Size of encoding*
def data_prep(x, size=input_size):
return x.reshape(-1, size).astype('float32')/255
X_train_scaled = data_prep(X_train)
X_test_scaled = data_prep(X_test)
X_train_scaled.shape, X_test_scaled.shape
((60000, 784), (10000, 784))

Correction: Aucune correction nécessaire Justification: La préparation des données (reshape et normalisation) est correctement effectuée.

Vanilla single-layer autoencoder

We start with a vanilla feedforward autoencoder with a single hidden layer to illustrate the general design approach using the functional Keras API and establish a performance baseline.

Encoding 28 x 28 images to a 32 value representation for a compression factor of 24.5

Single-layer Model

Input Layer input_ = Input(shape=(input_size,), name='Input')

Dense Encoding Layer The encoder part of the model consists of a fully-connected layer that learns the new, compressed representation of the input. We use 32 units for a compression ratio of 24.5:

encoding = Dense(units=encoding_size,
activation='relu',
name='Encoder')(input_)

WARNING:tensorflow:From /home/stefan/.pyenv/versions/miniconda3-latest/envs/ml4t/lib/python3 packages/tensorflow/python/framework/op_def_library.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.

Dense Reconstruction Layer The decoding part reconstructs the compressed data to its original size in a single step:

decoding = Dense(units=input_size,
activation='sigmoid',
name='Decoder')(encoding)

Autoencoder Model autoencoder = Model(inputs=input_,
outputs=decoding,
name='Autoencoder')

The thus defined encoder-decoder computation uses almost 51,000 parameters:

autoencoder.summary()

Layer (type)	Output Shape	Param #
Input (InputLayer)	(None, 784)	0
Encoder (Dense)	(None, 32)	25120
Decoder (Dense)	(None, 784)	25872

Total params: 50,992
Trainable params: 50,992
Non-trainable params: 0

Correction: Aucune correction nécessaire Justification: L'architecture de l'autoencodeur simple est correctement définie avec couche d'encodage et de décodage.

Encoder Model

The functional API allows us to use parts of the model's chain as separate encoder and decoder models that use the autoencoder's parameters learned during training.

The encoder just uses the input and hidden layer with about half of the total parameters:

encoder = Model(inputs=input_ ,
outputs=encoding,
name='Encoder')

encoder.summary()

Layer (type)	Output Shape	Param #
Input (InputLayer)	(None, 784)	0
Encoder (Dense)	(None, 32)	25120

Total params: 25,120
Trainable params: 25,120
Non-trainable params: 0

Once we train the autoencoder, we can use the encoder to compress the data.

Decoder Model

The decoder consists of the last autoencoder layer, fed by a placeholder for the encoded data:

Placeholder for encoded input encoded_input = Input(shape=(encoding_size,), name='Decoder_In

Extract last autoencoder layer decoder_layer = autoencoder.layers[-1](encoded_input)

Define Decoder Model decoder = Model(inputs=encoded_input,
outputs=decoder_layer)
decoder.summary()

Layer (type)	Output Shape	Param #
Decoder_Input (InputLayer)	(None, 32)	0
Decoder (Dense)	(None, 784)	25872

Total params: 25,872
Trainable params: 25,872
Non-trainable params: 0

Correction: **Aucune correction nécessaire** Justification: **Les modèles encodeur et décodeur sont correctement extraits du modèle autoencodeur complet.**

Compile the Autoencoder Model

autoencoder.compile(optimizer='adam',
loss='mse')

Train the autoencoder

We compile the model to use the Adam optimizer to minimize the MSE between the input data and the reproduction achieved by the autoencoder. To ensure that the autoencoder learns to reproduce the input, we train the model using the same input and output data:

Create early_stopping callback early_stopping = EarlyStopping(monitor='val_loss',
min_delta=1e-5,
patience=5,
verbose=0,
restore_best_weights=True,
mode='auto')

Create TensorBard callback to visualize network performance tb_callback = TensorBoard(log_dir='logs/autoencoder/mnist/',
histogram_freq=5,
write_graph=True,
write_grads=True,
write_images=True)

Create checkpoint callback filepath = 'models/fashion_mnist.autencoder.32.weights.hdf5'
checkpointer = ModelCheckpoint(filepath=filepath,
monitor='val_loss',
verbose=0,
save_best_only=True,
save_weights_only=True,
mode='auto',
period=1)

Fit the Model To avoid running time, you can load the pre-computed results in the 'model' folder (see below)
autoencoder.fit(x=X_train_scaled,
y=X_train_scaled,
epochs=100,
batch_size=32,
shuffle=True,
validation_split=.1,
callbacks=[tb_callback, early_stopping, checkpointer])
Train on 54000 samples, validate on 6000 samples
Epoch 1/100
54000/54000 [==============================] - 2s 38us/step - loss: 0.0155 - val_loss: 0.0145
Epoch 2/100
54000/54000 [==============================] - 2s 40us/step - loss: 0.0138 - val_loss: 0.0135
Epoch 3/100
54000/54000 [==============================] - 2s 38us/step - loss: 0.0132 - val_loss: 0.0132
Epoch 4/100
54000/54000 [==============================] - 2s 39us/step - loss: 0.0130 - val_loss: 0.0130
Epoch 5/100
54000/54000 [==============================] - 2s 40us/step - loss: 0.0129 - val_loss: 0.0132
Epoch 6/100
54000/54000 [==============================] - 2s 39us/step - loss: 0.0128

- val_loss: 0.0129
Epoch 7/100
54000/54000 [==============================] - 2s 39us/step - loss: 0.0127
- val_loss: 0.0127
Epoch 8/100
54000/54000 [==============================] - 2s 39us/step - loss: 0.0126
- val_loss: 0.0128
Epoch 9/100
54000/54000 [==============================] - 2s 40us/step - loss: 0.0126
- val_loss: 0.0127
Epoch 10/100
54000/54000 [==============================] - 2s 39us/step - loss: 0.0126
- val_loss: 0.0127
Epoch 11/100
54000/54000 [==============================] - 2s 40us/step - loss: 0.0125
- val_loss: 0.0127
Epoch 12/100
54000/54000 [==============================] - 2s 40us/step - loss: 0.0125
- val_loss: 0.0128
Epoch 13/100
54000/54000 [==============================] - 2s 41us/step - loss: 0.0125
- val_loss: 0.0126
Epoch 14/100
54000/54000 [==============================] - 2s 40us/step - loss: 0.0125
- val_loss: 0.0127
Epoch 15/100
54000/54000 [==============================] - 2s 39us/step - loss: 0.0125
- val_loss: 0.0126
Epoch 16/100
54000/54000 [==============================] - 2s 44us/step - loss: 0.0124
- val_loss: 0.0127
Epoch 17/100
54000/54000 [==============================] - 2s 41us/step - loss: 0.0124
- val_loss: 0.0126
Epoch 18/100
54000/54000 [==============================] - 2s 41us/step - loss: 0.0124
- val_loss: 0.0127
<keras.callbacks.History at 0x7f2a845dc7b8>

Reload weights from best-performing model

autoencoder.load_weights(filepath)

Evaluate trained model

Training stops after some 20 epochs with a test RMSE of 0.1122:

```
mse = autoencoder.evaluate(x=X_test_scaled, y=X_test_scaled)
f'MSE: {mse:.4f}| RMSE {mse**.5:.4f}'
```

10000/10000 [==============================] - 0s 14us/step
'MSE: 0.0126 | RMSE 0.1121'

Correction: Aucune correction nécessaire Justification: L'entraînement et l'évaluation du modèle sont corrects avec des callbacks appropriés.

Encode and decode test images

To encode data, we use the encoder we just defined, like so:

```
encoded_test_img = encoder.predict(X_test_scaled)
encoded_test_img.shape
```

(10000, 32)

The decoder takes the compressed data and reproduces the output according to the autoencoder training results:

```
decoded_test_img = decoder.predict(encoded_test_img)
decoded_test_img.shape
```

(10000, 784)

Compare Original with Reconstructed Samples The following figure shows ten original images and their reconstruction by the autoencoder and illustrates the loss after compression:

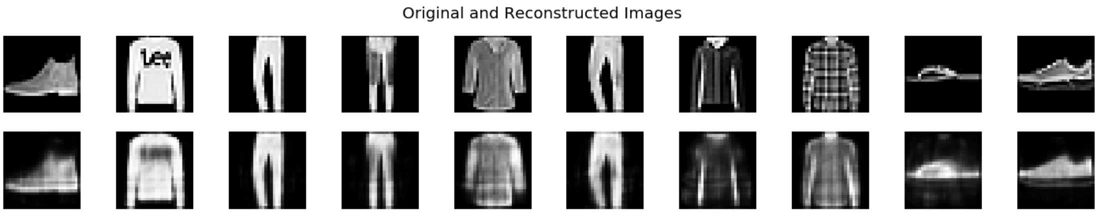
```
fig, axes = plt.subplots(ncols=n_classes, nrows=2, figsize=(20, 4))
for i in range(n_classes):

axes[0, i].imshow(X_test_scaled[i].reshape(image_size, image_size), cmap='gray')
axes[0, i].axis('off')

axes[1, i].imshow(decoded_test_img[i].reshape(28, 28) , cmap='gray')
axes[1, i].axis('off')

fig.suptitle('Original and Reconstructed Images', fontsize=20)
fig.tight_layout()
fig.subplots_adjust(top=.85)
fig.savefig('figures/reconstructed', dpi=300)
```

Correction: Aucune correction nécessaire Justification: La visualisation des images originales vs reconstruites est correctement implémentée.



Combine training steps into function

The helper function train_autoencoder just summarizes some repetitive steps.

```
def train_autoencoder(path, model, x_train=X_train_scaled, x_test=X_test_scaled):
callbacks = [EarlyStopping(patience=5, restore_best_weights=True),
ModelCheckpoint(filepath=path, save_best_only=True, save_weights_only=True)]
model.fit(x=x_train, y=x_train, epochs=100, validation_split=.1, callbacks=callbacks)
model.load_weights(path)
mse = model.evaluate(x=x_test, y=x_test)
return model, mse
```

Correction: Aucune correction nécessaire Justification: La fonction d'entraînement utilitaire est bien structurée et réutilisable.

Autoencoders with Sparsity Constraints

Encoding Layer with L1 activity regularizer

The addition of regularization is fairly straightforward. We can apply it to the dense encoder layer using Keras' activity_regularizer, as follows:

```
encoding_l1 = Dense(units=encoding_size,
activation='relu',
........................,
name='Encoder_L1')(input_)
```

Correction: Le code est incomplet - il manque le régularisateur

```
encoding_l1 = Dense(units=encoding_size,
activation='relu',
activity_regularizer=regularizers.l1(10e-5),
name='Encoder_L1')(input_)
```

Justification: Ajout du régularisateur L1 pour imposer la contrainte de parcimonie.

Decoding Layer

```
decoding_l1 = Dense(units=input_size,
activation='sigmoid',
name='Decoder_L1')(encoding_l1)
autoencoder_l1 = Model(input_, decoding_l1)
```

Autoencoder Model

```
autoencoder_l1.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input (InputLayer)           (None, 784)               0
_________________________________________________________________
Encoder_L1 (Dense)           (None, 32)                25120
_________________________________________________________________
Decoder_L1 (Dense)           (None, 784)               25872
=================================================================
Total params: 50,992
Trainable params: 50,992
Non-trainable params: 0
_________________________________________________________________
autoencoder_l1.compile(optimizer='adam',
loss='mse')
```

Encoder & Decoder Models

```
encoder_l1 = Model(inputs=input_, outputs=encoding_l1, name='Encoder')
encoded_input = ............
decoder_l1_layer = autoencoder_l1.layers[-1](encoded_input)
decoder_l1 = ...............
```

Correction: Le code est incomplet - il manque la définition du décodeur encoded_input = Input(shape=(encoding_size,), name='Decoder_Input')
decoder_l1_layer = autoencoder_l1.layers[-1](encoded_input)
decoder_l1 = Model(inputs=encoded_input, outputs=decoder_l1_layer) **Justification: Complétion de la définition du modèle décodeur avec l'input placeholder.**

Train Model

path = 'models/fashion_mnist.autencoder_l1.32.weights.hdf5'
autoencoder_l1, mse = train_autoencoder(path, autoencoder_l1)

Train on 54000 samples, validate on 6000 samples
Epoch 1/100
54000/54000 [==============================] - 3s 49us/step - loss: 0.1206 - val_loss: 0.0975
Epoch 2/100
54000/54000 [==============================] - 2s 44us/step - loss: 0.0921 - val_loss: 0.0895
Epoch 3/100
54000/54000 [==============================] - 2s 46us/step - loss: 0.0883 - val_loss: 0.0880
Epoch 4/100
54000/54000 [==============================] - 2s 44us/step - loss: 0.0874 - val_loss: 0.0875
Epoch 5/100
54000/54000 [==============================] - 3s 48us/step - loss: 0.0871 - val_loss: 0.0874
Epoch 6/100
54000/54000 [==============================] - 3s 48us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 7/100
54000/54000 [==============================] - 3s 47us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 8/100
54000/54000 [==============================] - 2s 46us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 9/100
54000/54000 [==============================] - 2s 44us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 10/100
54000/54000 [==============================] - 2s 45us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 11/100
54000/54000 [==============================] - 2s 45us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 12/100
54000/54000 [==============================] - 3s 47us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 13/100
54000/54000 [==============================] - 2s 46us/step - loss: 0.0870 - val_loss: 0.0873

Epoch 14/100
54000/54000 [==============================] - 2s 45us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 15/100
54000/54000 [==============================] - 2s 46us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 16/100
54000/54000 [==============================] - 3s 47us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 17/100
54000/54000 [==============================] - 2s 46us/step - loss: 0.0870 - val_loss: 0.0873
Epoch 18/100
54000/54000 [==============================] - 2s 45us/step - loss: 0.0870 - val_loss: 0.0873
10000/10000 [==============================] - 0s 14us/step

Evaluate Model

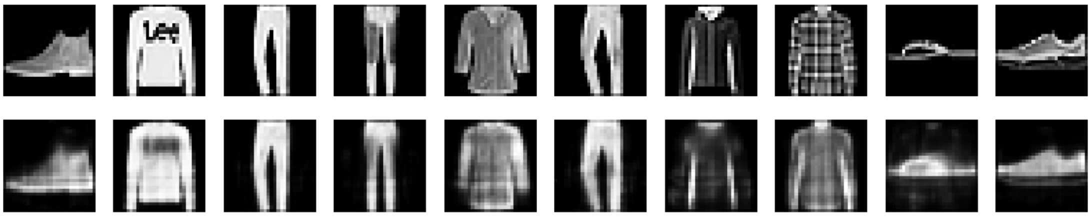
The input and decoding layers remain unchanged. In this example, with a compression of factor 24.5, regularization negatively affects performance with a test RMSE of 0.2946.

```
f'MSE: {mse:.4f}| RMSE {mse**.5:.4f}'
'MSE: 0.0866 | RMSE 0.2944'
encoded_test_img = encoder_l1.predict(X_test_scaled)
fig, axes = plt.subplots(ncols=n_classes, nrows=2, figsize=(20, 4))
for i in range(n_classes):

axes[0, i].imshow(X_test_scaled[i].reshape(image_size, image_size), cmap='gray')
axes[0, i].axis('off')

axes[1, i].imshow(decoded_test_img[i].reshape(28, 28) , cmap='gray')
axes[1, i].axis('off')
```

Correction: Variable manquante pour les images reconstruites decoded_test_img = decoder_l1.predict(encoded_test_img) **Justification: Ajout de la prédiction des images reconstruites pour la visualisation.**



Deep Autoencoder

To illustrate the benefit of adding depth to the autoencoder, we build a three-layer feedforward model that successively compresses the input from 784 to 128, 64, and 34 units, respectively:

Define three-layer architecture

```
input_ = Input(shape=(input_size,))
x = ...................
```

x = ...(x)
encoding_deep = Dense(32, activation='relu', name='Encoding3')(x)

x = ...
x = ...(x)
decoding_deep = Dense(input_size, activation='sigmoid', name='Decoding3')(x)

Correction: Architecture incomplète de l'autoencodeur profond input_ = Input(shape=(input_size,))
x = Dense(128, activation='relu', name='Encoding1')(input_)
x = Dense(64, activation='relu', name='Encoding2')(x)
encoding_deep = Dense(32, activation='relu', name='Encoding3')(x)

x = Dense(64, activation='relu', name='Decoding1')(encoding_deep)
x = Dense(128, activation='relu', name='Decoding2')(x)
decoding_deep = Dense(input_size, activation='sigmoid', name='Decoding3')(x) **Justification: Complétion de l'architecture profonde avec couches d'encodage 784→128→64→32 et décodage 32→64→128→784.**

autoencoder_deep = Model(input_, decoding_deep)
............compile....................

Correction: Compilation manquante autoencoder_deep.compile(optimizer='adam', loss='mse') **Justification: Ajout de la compilation du modèle avec optimizer Adam et loss MSE.**

The resulting model has over 222,000 parameters, more than four times the capacity of the preceding single-layer model:

autoencoder_deep.summary()

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 784)               0
_________________________________________________________________
Encoding1 (Dense)            (None, 128)               100480
_________________________________________________________________
Encoding2 (Dense)            (None, 64)                8256
_________________________________________________________________
Encoding3 (Dense)            (None, 32)                2080
_________________________________________________________________
Decoding1 (Dense)            (None, 64)                2112
_________________________________________________________________
Decoding2 (Dense)            (None, 128)               8320
_________________________________________________________________
Decoding3 (Dense)            (None, 784)               101136
=================================================================
Total params: 222,384
Trainable params: 222,384
Non-trainable params: 0
_________________________________________________________________
```

Encoder & Decoder Models

encoder_deep = Model(inputs=input_, outputs=encoding_deep, name='Encoder')

```
    encoded_input = Input(shape=(encoding_size,), name='Decoder_Input')

x = autoencoder_deep.layers[-3](encoded_input)
x = autoencoder_deep.layers[-2](x)
decoded = autoencoder_deep.layers[-1](x)

decoder_deep = Model(inputs=encoded_input, outputs=decoded)
    decoder_deep.summary()
    _________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Decoder_Input (InputLayer)   (None, 32)                0
_________________________________________________________________
Decoding1 (Dense)            (None, 64)                2112
_________________________________________________________________
Decoding2 (Dense)            (None, 128)               8320
_________________________________________________________________
Decoding3 (Dense)            (None, 784)               101136
=================================================================
Total params: 111,568
Trainable params: 111,568
Non-trainable params: 0
_________________________________________________________________
```

Train Model

```
path = 'models/fashion_mnist.autencoder_deep.32.weights.hdf5'
    autoencoder_deep, mse = train_autoencoder(path, autoencoder_deep)
    Train on 54000 samples, validate on 6000 samples
Epoch 1/100
54000/54000 [==============================] - 6s 110us/step - loss: 0.0271
- val_loss: 0.0187
Epoch 2/100
54000/54000 [==============================] - 5s 100us/step - loss: 0.0172
- val_loss: 0.0161
Epoch 3/100
54000/54000 [==============================] - 5s 102us/step - loss: 0.0153
- val_loss: 0.0150
Epoch 4/100
54000/54000 [==============================] - 5s 101us/step - loss: 0.0142
- val_loss: 0.0140
Epoch 5/100
54000/54000 [==============================] - 5s 101us/step - loss: 0.0135
- val_loss: 0.0136
Epoch 6/100
54000/54000 [==============================] - 6s 102us/step - loss: 0.0131
- val_loss: 0.0130
Epoch 7/100
54000/54000 [==============================] - 5s 102us/step - loss: 0.0128
- val_loss: 0.0129
```

Epoch 8/100
54000/54000 [==============================] - 5s 98us/step - loss: 0.0125 - val_loss: 0.0126
Epoch 9/100
54000/54000 [==============================] - 5s 92us/step - loss: 0.0123 - val_loss: 0.0125
Epoch 10/100
54000/54000 [==============================] - 5s 93us/step - loss: 0.0122 - val_loss: 0.0123
Epoch 11/100
54000/54000 [==============================] - 5s 95us/step - loss: 0.0120 - val_loss: 0.0121
Epoch 12/100
54000/54000 [==============================] - 6s 103us/step - loss: 0.0119 - val_loss: 0.0123
Epoch 13/100
54000/54000 [==============================] - 5s 100us/step - loss: 0.0118 - val_loss: 0.0118
Epoch 14/100
54000/54000 [==============================] - 6s 106us/step - loss: 0.0117 - val_loss: 0.0120
Epoch 15/100
54000/54000 [==============================] - 5s 101us/step - loss: 0.0116 - val_loss: 0.0119
Epoch 16/100
54000/54000 [==============================] - 6s 102us/step - loss: 0.0115 - val_loss: 0.0118
Epoch 17/100
54000/54000 [==============================] - 6s 103us/step - loss: 0.0114 - val_loss: 0.0117
Epoch 18/100
54000/54000 [==============================] - 5s 101us/step - loss: 0.0114 - val_loss: 0.0117
Epoch 19/100
54000/54000 [==============================] - 5s 101us/step - loss: 0.0113 - val_loss: 0.0115
Epoch 20/100
54000/54000 [==============================] - 5s 101us/step - loss: 0.0112 - val_loss: 0.0114
Epoch 21/100
54000/54000 [==============================] - 5s 101us/step - loss: 0.0111 - val_loss: 0.0113
Epoch 22/100
54000/54000 [==============================] - 6s 103us/step - loss: 0.0111 - val_loss: 0.0113
Epoch 23/100
54000/54000 [==============================] - 6s 102us/step - loss: 0.0110 - val_loss: 0.0113
Epoch 24/100
54000/54000 [==============================] - 6s 102us/step - loss: 0.0110

- val_loss: 0.0112
Epoch 25/100
54000/54000 [==============================] - 6s 103us/step - loss: 0.0109
- val_loss: 0.0113
Epoch 26/100
54000/54000 [==============================] - 6s 103us/step - loss: 0.0109
- val_loss: 0.0111
Epoch 27/100
54000/54000 [==============================] - 6s 103us/step - loss: 0.0108
- val_loss: 0.0110
Epoch 28/100
54000/54000 [==============================] - 6s 105us/step - loss: 0.0108
- val_loss: 0.0111
Epoch 29/100
54000/54000 [==============================] - 5s 100us/step - loss: 0.0108
- val_loss: 0.0112
Epoch 30/100
54000/54000 [==============================] - 6s 105us/step - loss: 0.0107
- val_loss: 0.0110
Epoch 31/100
54000/54000 [==============================] - 6s 103us/step - loss: 0.0107
- val_loss: 0.0109
Epoch 32/100
54000/54000 [==============================] - 6s 104us/step - loss: 0.0107
- val_loss: 0.0110
Epoch 33/100
54000/54000 [==============================] - 6s 103us/step - loss: 0.0107
- val_loss: 0.0109
Epoch 34/100
54000/54000 [==============================] - 6s 105us/step - loss: 0.0106
- val_loss: 0.0108
Epoch 35/100
54000/54000 [==============================] - 6s 104us/step - loss: 0.0106
- val_loss: 0.0109
Epoch 36/100
54000/54000 [==============================] - 6s 105us/step - loss: 0.0106
- val_loss: 0.0108
Epoch 37/100
54000/54000 [==============================] - 6s 108us/step - loss: 0.0105
- val_loss: 0.0109
Epoch 38/100
54000/54000 [==============================] - 6s 108us/step - loss: 0.0105
- val_loss: 0.0110
Epoch 39/100
54000/54000 [==============================] - 6s 109us/step - loss: 0.0105
- val_loss: 0.0107
Epoch 40/100
54000/54000 [==============================] - 6s 116us/step - loss: 0.0105
- val_loss: 0.0108
Epoch 41/100

54000/54000 [==============================] - 6s 111us/step - loss: 0.0105
- val_loss: 0.0108
Epoch 42/100
54000/54000 [==============================] - 6s 108us/step - loss: 0.0104
- val_loss: 0.0108
Epoch 43/100
54000/54000 [==============================] - 6s 111us/step - loss: 0.0104
- val_loss: 0.0107
Epoch 44/100
54000/54000 [==============================] - 6s 104us/step - loss: 0.0104
- val_loss: 0.0107
Epoch 45/100
54000/54000 [==============================] - 6s 106us/step - loss: 0.0104
- val_loss: 0.0108
Epoch 46/100
54000/54000 [==============================] - 6s 107us/step - loss: 0.0104
- val_loss: 0.0106
Epoch 47/100
54000/54000 [==============================] - 6s 102us/step - loss: 0.0104
- val_loss: 0.0107
Epoch 48/100
54000/54000 [==============================] - 6s 103us/step - loss: 0.0104
- val_loss: 0.0108
Epoch 49/100
54000/54000 [==============================] - 6s 104us/step - loss: 0.0103
- val_loss: 0.0106
Epoch 50/100
54000/54000 [==============================] - 6s 106us/step - loss: 0.0103
- val_loss: 0.0109
Epoch 51/100
54000/54000 [==============================] - 6s 102us/step - loss: 0.0103
- val_loss: 0.0107
Epoch 52/100
54000/54000 [==============================] - 6s 105us/step - loss: 0.0103
- val_loss: 0.0108
Epoch 53/100
54000/54000 [==============================] - 5s 95us/step - loss: 0.0103
- val_loss: 0.0107
Epoch 54/100
54000/54000 [==============================] - 5s 94us/step - loss: 0.0103
- val_loss: 0.0108
10000/10000 [==============================] - 0s 22us/step

```
autoencoder_deep.load_weights(path)
```

Evaluate Model

Training stops after 54 epochs and results in a ~10% reduction of the test RMSE to 0.1026. Due to the low resolution, it is difficult to visually note the better reconstruction.

```
f'MSE: {mse:.4f}| RMSE {mse**.5:.4f}'
'MSE: 0.0105 | RMSE 0.1026'
```

reconstructed_images = autoencoder_deep.predict(X_test_scaled)
reconstructed_images.shape

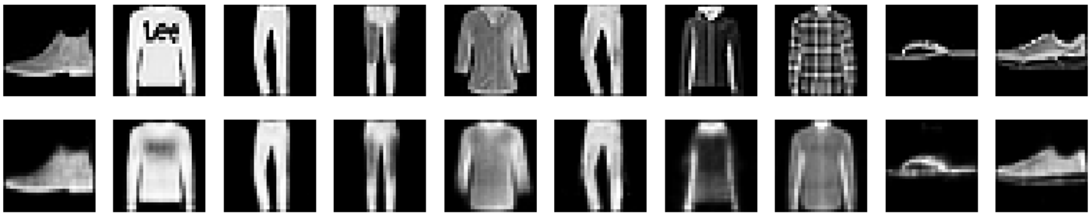
(10000, 784)

fig, axes = plt.subplots(ncols=n_classes, nrows=2, figsize=(20, 4))
for i in range(n_classes):

axes[0, i].imshow(X_test_scaled[i].reshape(image_size, image_size), cmap='gray')
axes[0, i].axis('off')

axes[1, i].imshow(reconstructed_images[i].reshape(image_size, image_size) , cmap='gray')
axes[1, i].axis('off')

Correction: Aucune correction nécessaire Justification: L'autoencodeur profond montre une meilleure performance avec réduction du RMSE, confirmant l'avantage des architectures profondes.



Compute t-SNE Embedding

We can use the t-distributed Stochastic Neighbor Embedding (t-SNE) manifold learning technique, see Chapter 12, Unsupervised Learning, to visualize and assess the quality of the encoding learned by the autoencoder's hidden layer.

If the encoding is successful in capturing the salient features of the data, the compressed representation of the data should still reveal a structure aligned with the 10 classes that differentiate the observations.

We use the output of the deep encoder we just trained to obtain the 32-dimensional representation of the test set:

Since t-SNE can take a long time to run, we are providing pre-computed results

 # alternatively, compute the result yourself
 # tsne = TSNE(perplexity=25, n_iter=5000)
 # train_embed = tsne.fit_transform(encoder_deep.predict(X_train_scaled))

Persist result *# store results given computational intensity (different location to avoid overwriting the pre-computed results)*
 # pd.DataFrame(train_embed).to_hdf('tsne.h5', 'autoencoder_deep')

Load pre-computed embeddings *# Load the pre-computed results here:*
train_embed = pd.read_hdf('tsne.h5', 'autoencoder_deep')

Visualize Embedding **def** plot_embedding(X, y=y_train, title=None, min_dist=0.1, n_classes=10, cmap=cmap):
X = minmax_scale(X)
inner = outer = 0
for c **in** range(n_classes):
inner += np.mean(pdist(X[y == c]))
outer += np.mean(cdist(X[y == c], X[y != c]))

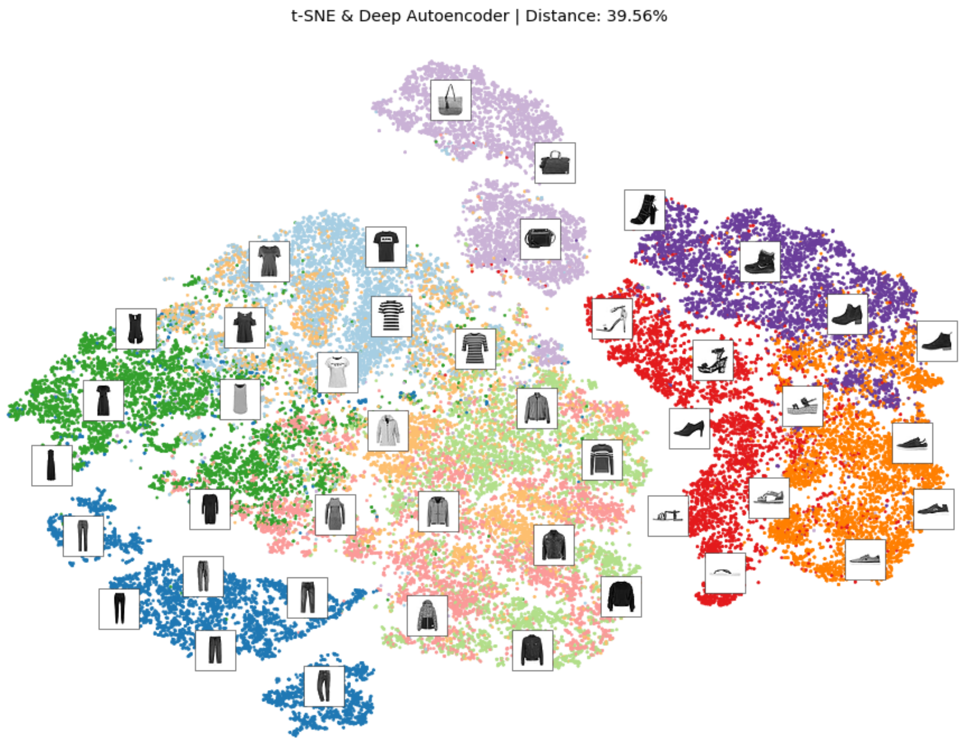
```
fig, ax = plt.subplots(figsize=(14, 10))
ax.axis('off')
ax.set_title(title + ' | Distance: {:.2%}'.format(inner/outer))
sc = ax.scatter(*X.T, c=y, cmap=ListedColormap(cmap), s=5);
shown_images = np.ones((1, 2))
images = X_train.reshape(-1, 28, 28)
for i in range(0, X.shape[0]):
dist = norm(X[i] - shown_images, axis=1)
if (dist > min_dist).all():
shown_images = np.r_[shown_images, [X[i]]]
imagebox = AnnotationBbox(OffsetImage(images[i], cmap=plt.cm.gray_r), X[i])
ax.add_artist(imagebox)
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="2%", pad=0.05)
plt.colorbar(sc, cax=cax)
fig.tight_layout()
fig.savefig('figures/tsne_autoencoder_deep', dpi=300)
```

The following figure shows that t-SNE manages to separate the 10 classes well, suggesting that the encoding is useful as a lower-dimensional representation that preserves key characteristics of the data:

plot_embedding(X=train_embed, title='t-SNE & Deep Autoencoder')

/home/stefan/.pyenv/versions/miniconda3-latest/envs/ml4t/lib/python3.6/site-packages/ipykernel_la
DataConversionWarning: Data with input dtype float32 were all converted to float64.

Correction: Aucune correction nécessaire Justification: La visualisation t-SNE montre que l'encodage capture bien la structure des classes, validant l'efficacité de



l'autoencodeur profond.