FACULTÉ DES SCIENCES DHAR EL MAHRAZ
UNIVERSITÉ SIDI MOHAMED BEN ABDELLAH

Université Sorbonne Paris Nord

RAPPORT DES TRAVAUX PRATIQUES DE MODULE : DEEP LEARNING

Master: WISD & EID2

Module: Deep Learning

11/03/2022

Réalisé par :

Mohamed LAMGARRAJ

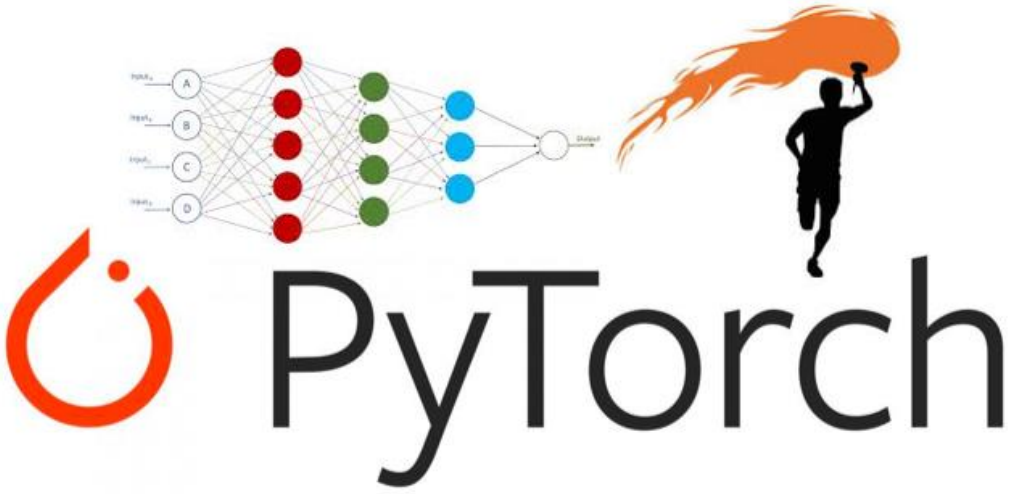
Yazid TAGNAOUTI MOUMNANI

Responsable de module :

Pr. Younès BENNANI

Responsable des TPs:

Pr. Guénaël CABANES

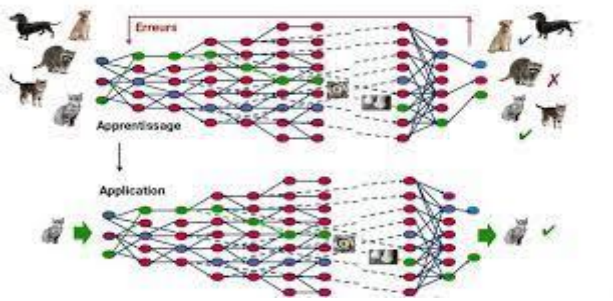


ANNEE UNIVERSITAIRE 2021/2022

Table des matières

1. Introduction

Deep Learning ou L'apprentissage profond en français est un type d'apprentissage automatique et d'intelligence artificielle (IA) qui imite la façon dont les humains acquièrent certains types de connaissances. L'apprentissage en profondeur est un élément important de la science des données, qui comprend les statistiques et la modélisation prédictive. Il est extrêmement bénéfique pour les scientifiques des données qui sont chargés de collecter, d'analyser et d'interpréter de grandes quantités de données, l'apprentissage en profondeur rend ce processus plus rapide et plus facile.



PyTorch est une bibliothèque Python open source pour l'apprentissage en profondeur développée et maintenue par Facebook.Le projet a démarré en 2016 et est rapidement devenu un framework populaire parmi les développeurs et les chercheurs.

Torch (Torch7) est un projet open source d'apprentissage profond écrit en C et généralement utilisé via l'interface Lua. C'était un projet précurseur de PyTorch et n'est plus activement développé. PyTorch inclut "Torch" dans le nom, reconnaissant la bibliothèque torche précédente avec le préfixe "Py" indiquant le focus Python du nouveau projet.

La flexibilité de PyTorch se fait au détriment de la facilité d'utilisation, en particulier pour les débutants, par rapport à des interfaces plus simples comme Keras. Le choix d'utiliser PyTorch au lieu de Keras donne une certaine facilité d'utilisation, une courbe d'apprentissage légèrement plus rapide et plus de code pour plus de flexibilité, et peut-être une communauté universitaire plus dynamique.

2.TP1 : Tensors avec PyTorch

PyTorch est une bibliothèque d'apprentissage automatique open source basée sur la bibliothèque Torch, utilisée pour des applications telles que la vision par ordinateur et le traitement du langage naturel, principalement développée par le laboratoire de recherche sur l'IA de Facebook (FAIR). Il s'agit d'un logiciel gratuit et open source publiée sous la licence BSD modifiée. Bien que l'interface Python soit plus raffinée et soit l'objectif principal du développement, PyTorch possède également une interface C++.

Un certain nombre de logiciels d'apprentissage en profondeur sont construits sur PyTorch, notamment Tesla Autopilot, Uber's Pyro, HuggingFace's Transformers, PyTorch Lightning et Catalyst.

Les tenseurs sont une structure de données spécialisée très similaire aux tableaux et aux matrices. Dans PyTorch, nous utilisons des tenseurs pour encoder les entrées et les sorties d'un modèle, ainsi que les paramètres du modèle.

Les tenseurs sont similaires aux arrays de NumPy, sauf que les tenseurs peuvent fonctionner sur des GPU ou d'autres matériels spécialisés pour accélérer le calcul.

2.1 Tensors

1. Vectors (1D Tensors)

Vectors (1D Tensors)

```
#Create a vector (one-dimensional tensor)
t1 = torch.tensor([1., 2, 3, 4])
t1
print(t1.shape)
```

```
torch.Size([4])
```

```
# Print number of dimensions (1D) and size of tensor
print(f'dim: {t1.dim()}, size: {t1.size()[0]}')
```

```
dim: 1, size: 4
```

```
t2 = torch.Tensor([1, 0, 2, 0])
t2
```

```
tensor([1., 0., 2., 0.])
```

```
# Element-wise multiplication
t1 * t2
```

```
tensor([1., 0., 6., 0.])
```

```
# Scalar product
t1 @ t2
```

```
tensor(7.)
```

```
# In-place replacement of random number from 0 to 20
x = torch.Tensor(7).random_(20)
x
```

:

2. Matrices (2D Tensors)

▼ Matrices (2D Tensors)

```
[ ] # Create a matrix (two-dimensional tensor) 2x4 tensor
    m = torch.Tensor([[2, 5, 3, 7],
                      [4, 2, 1, 9]])
    m
```

```
tensor([[2., 5., 3., 7.],
        [4., 2., 1., 9.]])
```

```
m.dim()
```

```
2
```

```
[ ] print(m.size(0), m.size(1), m.size(), sep=' -- ')
```

```
2 -- 4 -- torch.Size([2, 4])
```

```
[ ] # Returns the total number of elements, hence num-el (number of elements)
    m.numel()
```

```
8
```

```
[ ] # Indexing row 0, column 2 (0-indexed)
    m[0][2]
```

```
tensor(3.)
```

```
[ ] # Indexing row 0, column 2 (0-indexed)
    m[0,2]
```

```
tensor(3.)
```

3. 3D Tensors

▼ (3D Tensors)

Tensors can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the .shape property of a tensor.

```
[ ] #Create a 3-dimensional tensor
    t3 = torch.tensor([[[11,12, 13,4],[13,14,15,58],[13,14,15,58]],[[15,16,17,56],[17,18,19.,0],[13, 14, 15,58]]])
    t3
```

```
tensor([[[11., 12., 13.,  4.],
         [13., 14., 15., 58.],
         [13., 14., 15., 58.]],

        [[15., 16., 17., 56.],
         [17., 18., 19.,  0.],
         [13., 14., 15., 58.]]])
```

```
[ ] t3.dim()
```

```
3
```

```
[ ] t3.shape
```

```
torch.Size([2, 3, 4])
```

Note that it's not possible to create tensors with an improper shape.

```
[ ] # Matrix (2-dimensional tensor)
    t4 = torch.tensor([[5., 6,11],
                       [7, 8],
                       [9, 10]])
    t4
```

2.2 Operations et gradients Tensor

4.Tensor operations and gradients

On peut combiner les tenseurs avec les opérations arithmétiques usuelles. Regardons un exemple :

```
# Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
x, w, b
```

Nous avons créé trois tenseurs : x, w et b, tous des nombres. W et b ont un paramètre supplémentaire requirements_grad défini sur True. Nous verrons ce qu'il fait dans un instant. Créons un nouveau tenseur y en combinant ces tenseurs.

```
# Arithmetic operations
y = w * x + b
y
```

Pour calculer les dérivées, nous pouvons invoquer la méthode. **backward** sur notre résultat y.

```
# Compute derivatives
y.backward()
```

The derivatives of y with respect to the input tensors are stored in the .grad property of the respective tensors.

```
# Display gradients
print('dy/dx:', x.grad)
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

```
dy/dx: None
dy/dw: tensor(3.)
dy/db: tensor(1.)
```

5. Des fonction sur les Tensors

Outre que les opérations arithmétiques, le module torche contient également de nombreuses fonctions de création et de manipulation de tenseurs. Regardons quelques exemples.

```
[ ] # Create a tensor with a fixed value for every element
    t6 = torch.full((3, 2), 42)
    t6
```

```
tensor([[42, 42],
        [42, 42],
        [42, 42]])
```

```
[ ] # Create a tensor filled with the scalar value 0, with the
    t10=torch.zeros(2, 3)
    t10
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

```
[ ] # Concatenate two tensors with compatible shapes
    t7 = torch.cat((t2, t6))
    t7
```

```
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.],
        [42., 42.],
        [42., 42.],
        [42., 42.]])
```

```
[ ] # Compute the sin of each element
    t8 = torch.sin(t7)
    t8
```

```
tensor([[-0.9589, -0.2794],
        [ 0.6570,  0.9894],
        [ 0.4121, -0.5440],
        [-0.9165, -0.9165],
        [-0.9165, -0.9165],
        [-0.9165, -0.9165]])
```

```
[ ] # Change the shape of a tensor
    t9 = t8.reshape(3, 2, 2)
    t9
```

```
tensor([[[-0.9589, -0.2794],
         [ 0.6570,  0.9894]],

        [[ 0.4121, -0.5440],
         [-0.9165, -0.9165]],

        [[-0.9165, -0.9165],
         [-0.9165, -0.9165]]])
```

```
[ ] # Create a a 2-D tensor with ones on
    t11=torch.eye(3)
    t11
```

```
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

```
[ ] #Splits the tensor into chunks. Each
    torch.split(t11,1,1)
```

```
(tensor([[1.],
         [0.],
         [0.]]),
 tensor([[0.],
         [1.],
         [0.]]),
 tensor([[0.],
         [0.],
         [1.]]))
```

2.3 Interoperability avec Numpy

6.Interopérabilité avec Numpy

Numpy est une bibliothèque open source populaire utilisée pour le calcul mathématique et scientifique en Python. Il permet des opérations efficaces sur de grands tableaux multidimensionnels et dispose d'un vaste écosystème de bibliothèques de support, notamment :

- Pandas pour les E/S de fichiers et l'analyse de données
- Matplotlib pour le traçage et la visualisation
- OpenCV pour le traitement d'images et de vidéos

Au lieu de réinventer la roue, PyTorch interagit bien avec Numpy pour tirer parti de son écosystème existant d'outils et de bibliothèques.

Voici comment nous créons un tableau dans Numpy :

```
[ ]  import numpy as np

     x = np.array([[1, 2], [3, 4.]])
     x

     array([[1., 2.],
            [3., 4.]])
```

We can convert a Numpy array to a PyTorch tensor using torch.from_numpy.

```
[ ]  y = torch.from_numpy(x)
     y

     tensor([[1., 2.],
             [3., 4.]], dtype=torch.float64)
```

Let's verify that the numpy array and torch tensor have similar data types.

```
[ ]  x.dtype, y.dtype

     (dtype('float64'), torch.float64)
```

We can convert a PyTorch tensor to a Numpy array using the .numpy method of a tensor.

```
[ ]  # Convert a torch tensor to a numpy array
     z = y.numpy()
     z

     array([[1., 2.],
            [3., 4.]])
```

L'interopérabilité entre PyTorch et Numpy est essentielle car la plupart des ensembles de données avec lesquels vous travaillerez seront probablement lus et prétraités en tant que tableaux Numpy.

La capacité à calculer automatiquement les gradients pour les opérations tensorielles est essentielle pour la formation de modèles d'apprentissage en profondeur. Prise en charge GPU : tout en travaillant avec des ensembles de données volumineux et de grands modèles, les opérations de tenseur PyTorch peuvent être effectuées efficacement à l'aide d'une unité de traitement graphique (GPU). Les calculs qui prennent généralement des heures peuvent être effectués en quelques minutes à l'aide de GPU. Nous tirerL'interopérabilité entre PyTorch et Numpy est essentielle car la plupart des ensembles de données avec lesquels vous travaillerez seront probablement lus et prétraités en tant que tableaux Numpy.

L'intégralité du code de cet article est disponible dans le notebook **google-colabe** suivant si vous souhaitez le tourner vous-même.

https://drive.google.com/file/d/1zpbRsMubsz2J5A4gsHzewWBHAfRCw6c5/view?usp=sharing

3.TP2 : Transformation et visualisation des Tensors

PyTorch peut fonctionner à la fois sur le CPU et le GPU d'un ordinateur. Le CPU est utile pour les tâches séquentielles, tandis que le GPU est utile pour les tâches parallèles. Avant d'exécuter sur l'appareil désiré, nous devons d'abord nous assurer que nos tenseurs et modèles sont transférés dans la mémoire de l'appareil. Cela peut être fait avec les deux lignes de code suivantes :

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
X = torch.randn(n_points, 2).to(device)
```

La première ligne crée une variable, appelée `device`, qui est assignée au GPU s'il y en a un de disponible, sinon elle est assignée par défaut au CPU. À la ligne suivante, un tenseur est créé et envoyé à la mémoire du périphérique en appelant `.to(device) .`

Importing necessary libraries

```
import torch
import torch.nn as nn
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.pyplot import plot, title, axis
```

Nous allons besoin de ces 3 fonctions pour ce travail :

- **set_default** pour personnaliser la disposition des figures.
- **show_scatterplot** dessine un nuage de points X en 2D, chaque point de données a une couleur spécifique.
- **plot_bases** affiche une flèche le long de l'axe des x en rouge et une flèche le long de l'axe des y en vert.

```
def set_default(figsize=(10, 10), dpi=100):
    plt.style.use(['dark_background', 'bmh'])
    plt.rc('axes', facecolor='k')
    plt.rc('figure', facecolor='k')
    plt.rc('figure', figsize=figsize, dpi=dpi)

def show_scatterplot(X, title=''):
    plt.figure()
    x_min=-1.5
    x_max=1.5
    colors = (X-x_min)/(x_max-x_min)
    colors = (colors * 511).short().numpy()
    colors = np.clip(colors, 0, 511)
    colors= colors[:, 0]
    plt.scatter(X[:, 0], X[:, 1], c=colors, s=30)
    plt.title(title)
    plt.axis('off')

def plot_bases(bases, width=0.04):
    plt.arrow(*bases[0], *bases[2], width=width, color=(1,0,0), zorder=10, alpha=1., length_includes_head=True)
    plt.arrow(*bases[1], *bases[3], width=width, color=(0,1,0), zorder=10, alpha=1., length_includes_head=True)
```

```
[ ] set_default()
```

PyTorch peut fonctionner à la fois sur le CPU et le GPU d'un ordinateur.

Le CPU est utile pour les tâches séquentielles, tandis que le GPU est utile pour les tâches parallèles. Les tenseurs et modèles sont transférés dans la mémoire de l'appareil. Cela peut être fait avec la ligne de code suivante :

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

1.Create a tensor X of 1000 instances in 2-D and sent it to the device's memory.

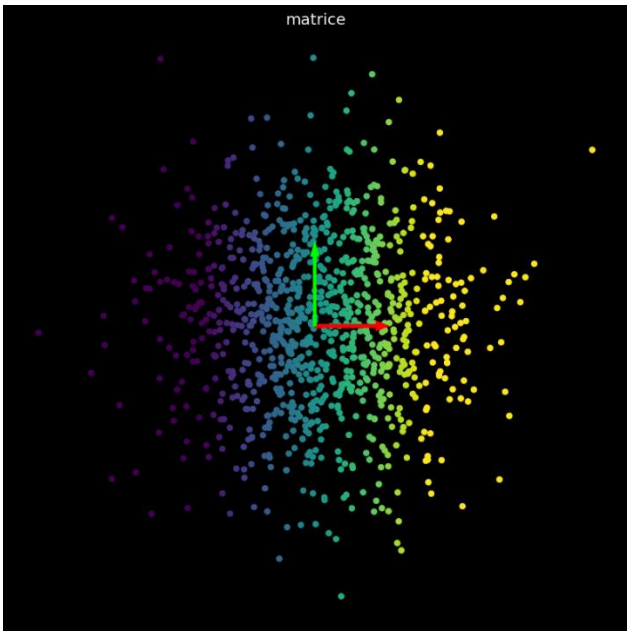
```
[ ] n_points = 1000
    X = torch.randn(n_points, 2).to(device)
```

```
[ ] X
```

```
tensor([[-0.6141,  1.1355],
        [ 0.9358, -0.2010],
        [ 0.3731, -0.3877],
        ...,
        [ 0.1133,  1.2811],
        [ 0.8480,  0.1644],
        [-1.5269,  0.5113]])
```

2.Visualise the original point cloud X using the two functions above.

```
[ ] show_scatterplot(x,'first plot')
    repére = torch.cat((torch.zeros(2, 2), torch.eye(2))).to(device)
    plot_bases(repére)
```



3.Expliquer brièvement la décomposition en valeurs singulières (SVD)

Une transformation linéaire peut être représentée sous forme de matrice. En utilisant la décomposition en valeur singulière, nous pouvons décomposer cette matrice en trois matrices composantes, chacune représentant une transformation linéaire différente.

$$W = U \begin{bmatrix} s_1 & 0 \\ 0 & s_2 \end{bmatrix} V^\top$$

Les matrices U et le transposé de V sont orthogonales et représentent les transformations de rotation et de réflexion. La matrice du milieu est diagonale et représente une transformation d'échelle.

4.Expliquer la relation entre la rotation et les matrices orthogonales

Une matrice orthogonale est une matrice dont la transposée (conjuguée) est son inverse.

Une matrice de rotation est une matrice qui prend un vecteur **V** et le mappe sur un autre vecteur V' où, **(V*V')/(|V|x|V'|) = constante**.

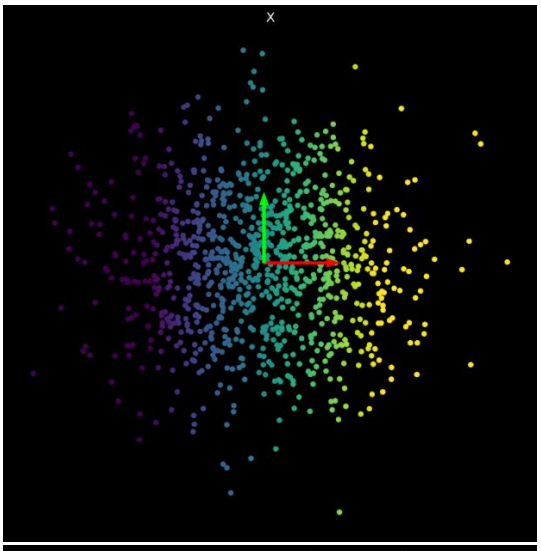
5.Expliquer la relation entre la mise à l'échelle et les matrices diagonales

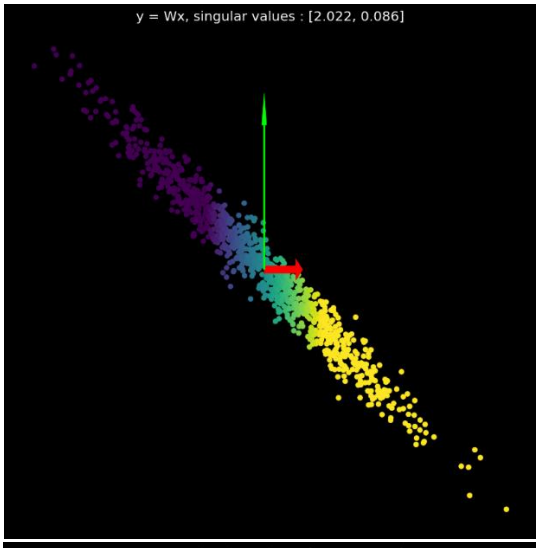
Une matrice diagonale est parfois appelée matrice de mise à l'échelle, car la multiplication de la matrice avec elle entraîne un changement d'échelle (taille). Son déterminant est le produit de ses valeurs diagonales.

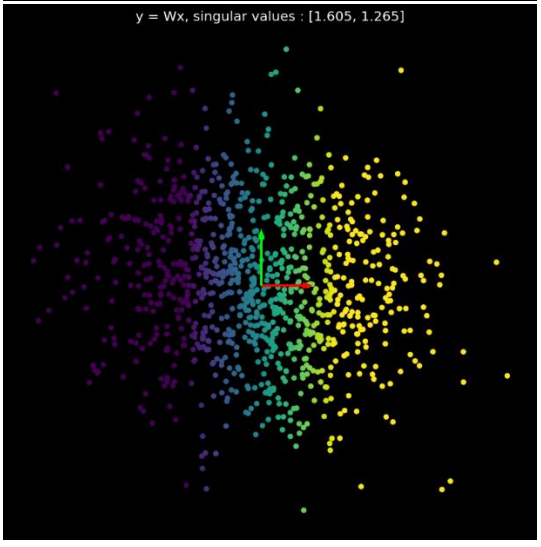
6.Calculer 10 transformations linéaires en utilisant plusieurs matrices aléatoires W de taille (2,2)

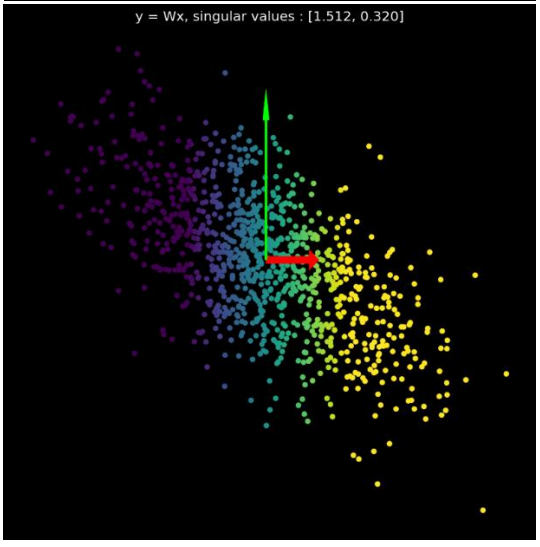
Pour chaque transformation, nous calculons la SVD de W et visualisons le nuage de points transformé Y avec ses valeurs singulières correspondantes (s1,s2) .

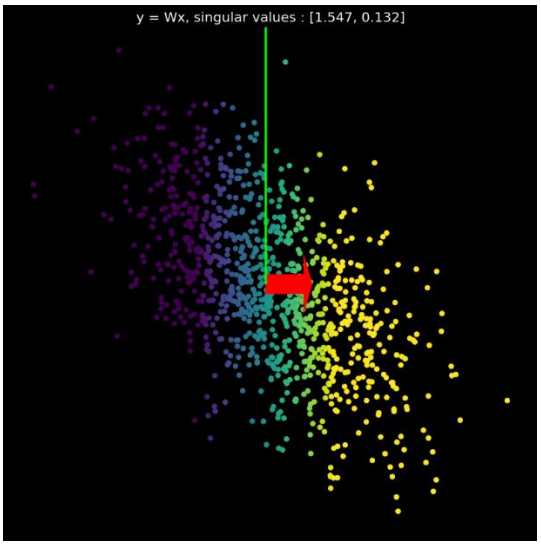
```
[ ]  show_scatterplot(X, 'X')
     plot_bases(repére)

     for i in range(10):
         # create a random matrix
         W = torch.randn(2, 2).to(device)
         # transform points
         Y = X @ W.t()
         # compute singular values
         U, S, V = torch.svd(W)
         # plot transformed points
         show_scatterplot(Y, 'y = Wx, singular values : [{:.3f}, {:.3f}]'.format(S[0], S[1]))
         # transform the basis
         new_OI = repére @ W.t()
         # plot old and new basis
         plot_bases(repére)
     #     plot_bases(new_OI)
```

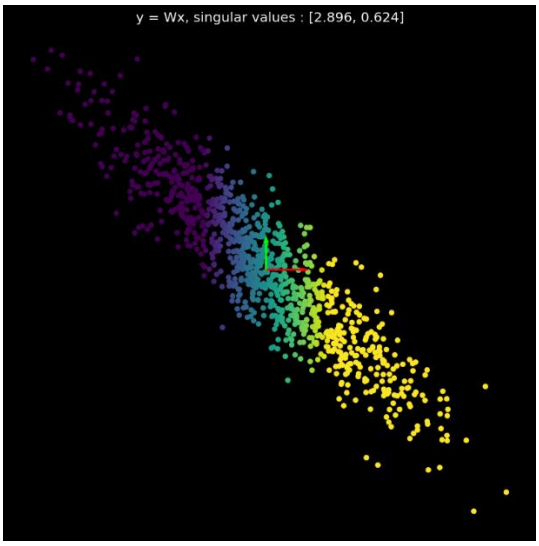








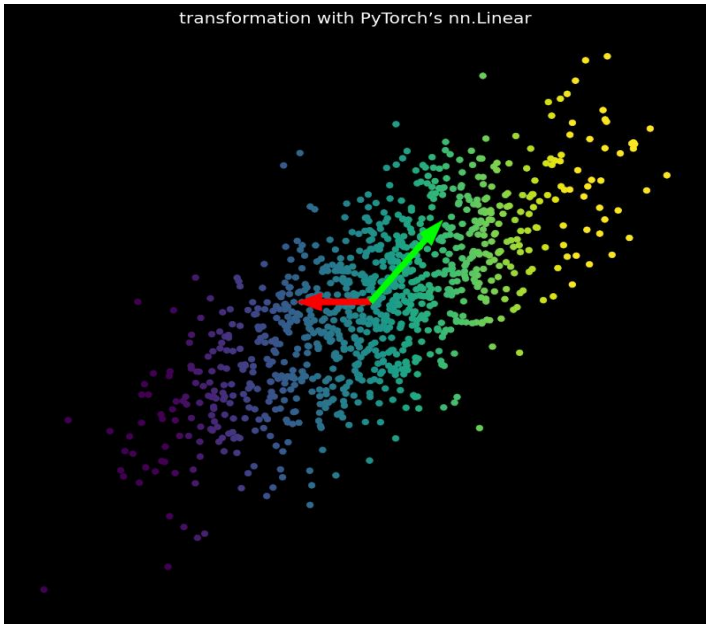




D'après la visualisons des transformations linéaires de plusieurs matrices aléatoires on peut voir l'effet des valeurs singulières sur les transformations résultantes, plus que les valeurs singulières grand de 1, plus que les points sont bien distribués selon l'axe qui convient, et plus que les valeurs singulières près de 0, plus que les points sont plus étalés vers le centre.

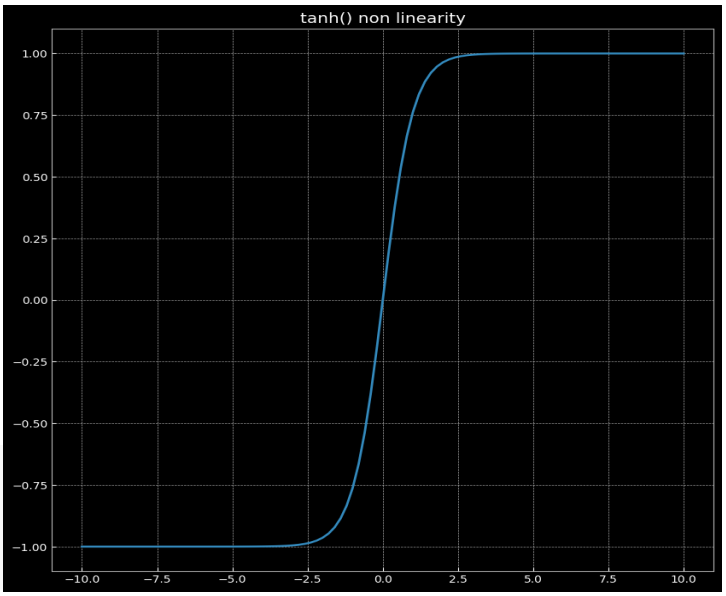
8.Calculez une transformation linéaire similaire avec la classe PyTorch nn.Linear et visualisez le nuage de points transformé Y

```
[ ]  model = nn.Sequential(
            nn.Linear(2, 2, bias=False)
     )
     model.to(device)
     with torch.no_grad():
         Y = model(X)
         show_scatterplot(Y, 'transformation with PyTorch's nn.Linear')
         plot_bases(model(repére))
```



9.Tracer le graphique de la fonction tangente hyperbolique (tanh).

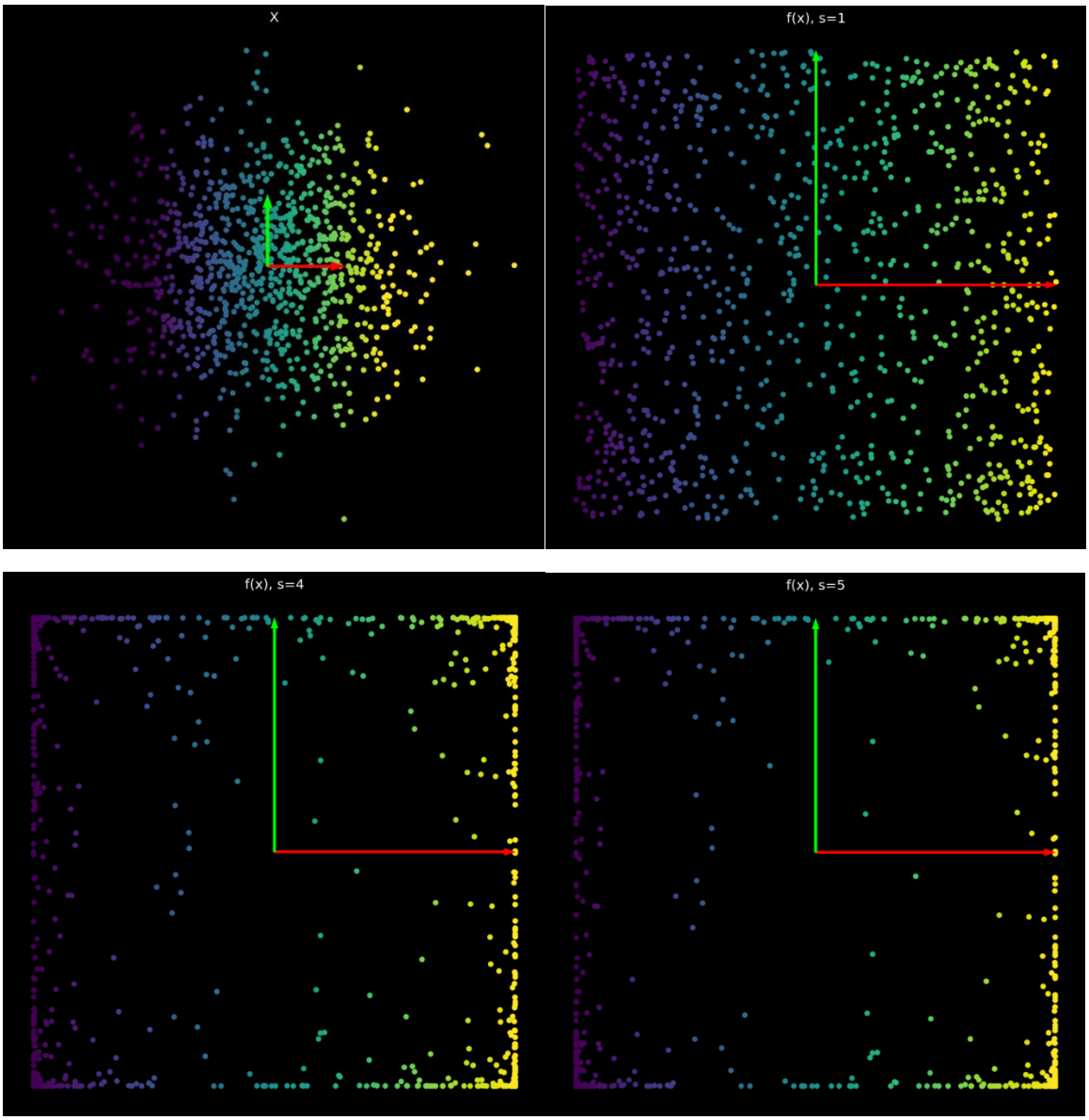
```
z = torch.linspace(-10, 10, 101)
s = torch.tanh(z)
plot(z.numpy(), s.numpy())
title('tanh() non linearity');
```



10.Pour s=1,...,5. Calculez une transformation linéaire avec la classe nn.Linear de PyTorch

La matrice de transformation doit être une matrice diagonale 2x2, avec des éléments diagonaux tous égaux à s), suivie d'une transformation non linéaire à l'aide de *tanh*. Visualisez les résultats.

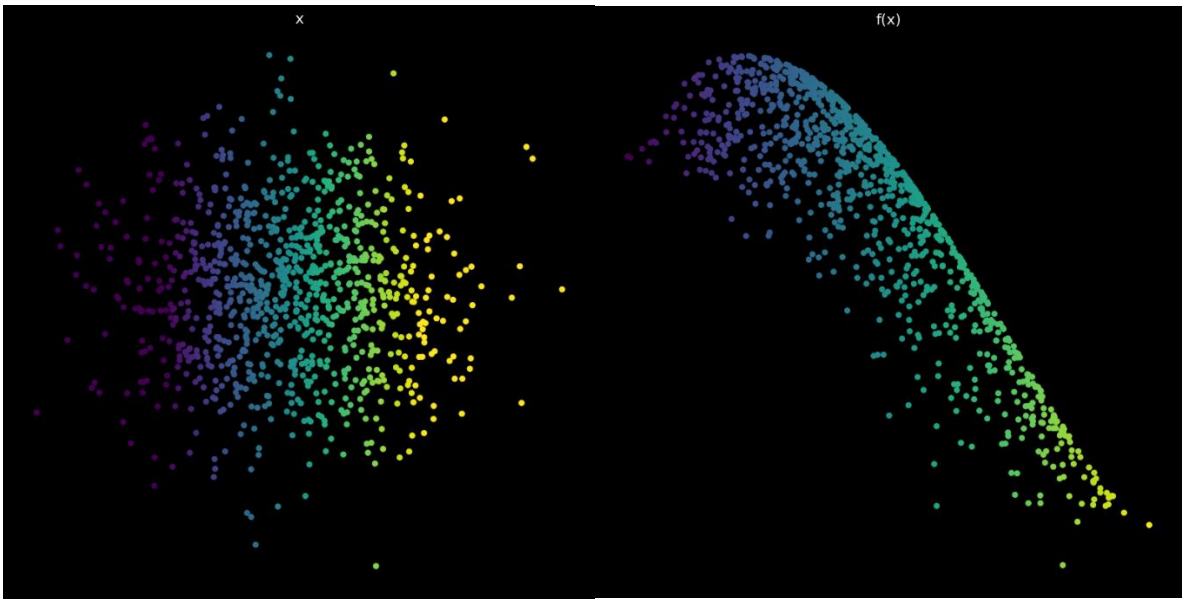
```
show_scatterplot(X, 'X')
plot_bases(repére)

model = nn.Sequential(
        nn.Linear(2, 2, bias=False),
        nn.Tanh()
)

model.to(device)

for s in range(1, 6):
    W = s * torch.eye(2)
    model[0].weight.data.copy_(W)
    Y = model(X).data
    show_scatterplot(Y, f'f(x), s={s}')
    plot_bases(repére, width=0.01)
```

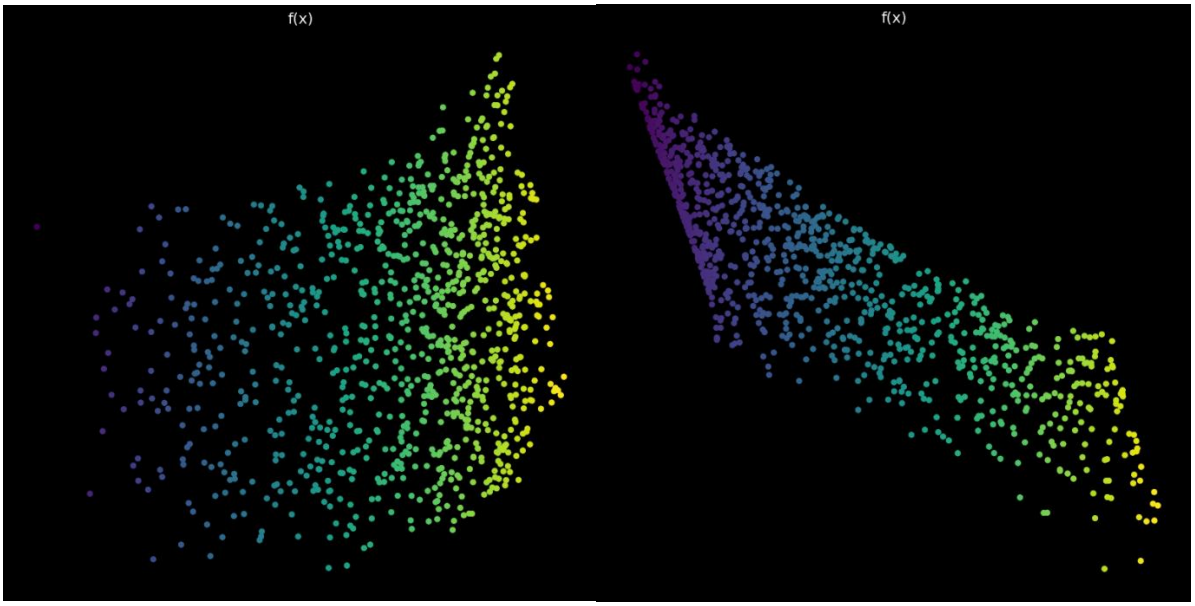


D'après les figures, nous pouvons voir que le non-linéarité a pour effet de délimiter des points entre les points de cordonnées (1,1) (1,-1) (-1,1) (-1,-1), créant ainsi un carré. Plus la valeur des S1 et S2 augmente, plus les points sont poussés vers le bord du carré, et on peut aussi voir que en forçant plus de points vers le bord, nous les étalons davantage et pouvons alors tenter de les classer.

12. la transformation Réseau constituée d'une couche linéaire, qui effectue une transformation affine, suivie d'une non-linéarité tangente hyperbolique, et enfin d'une autre couche linéaire.

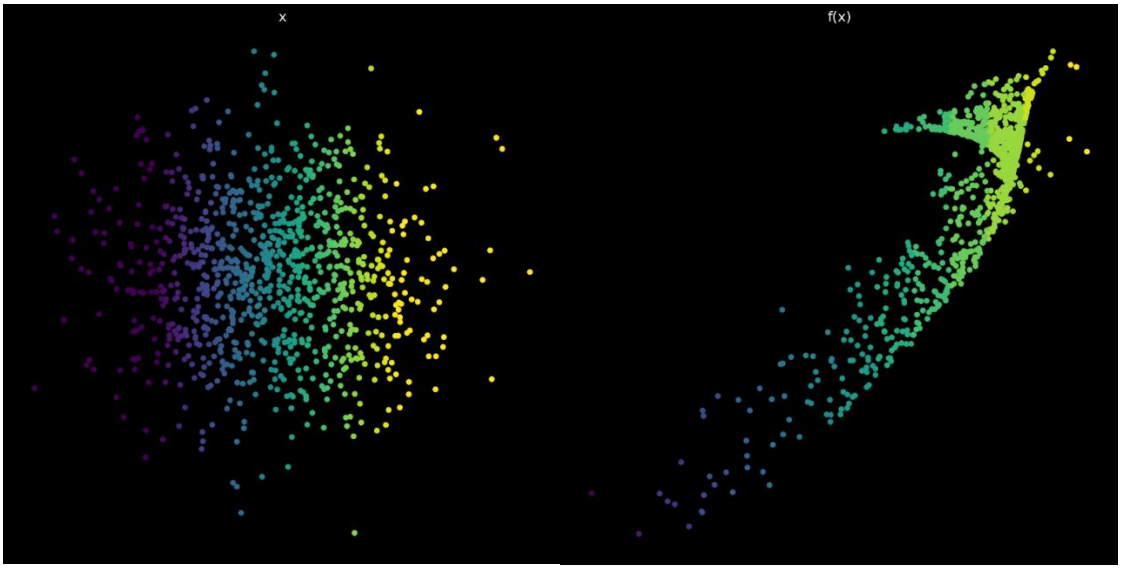
```
show_scatterplot(X, 'x')
n_hidden = 5

# NL = nn.ReLU()
NL = nn.Tanh()

for i in range(5):
    # create 1-layer neural networks with random weights
    model = nn.Sequential(
            nn.Linear(2, n_hidden),
            NL,
            nn.Linear(n_hidden, 2)
        )
    model.to(device)
    with torch.no_grad():
        Y = model(X)
    show_scatterplot(Y,'f(x)')
```

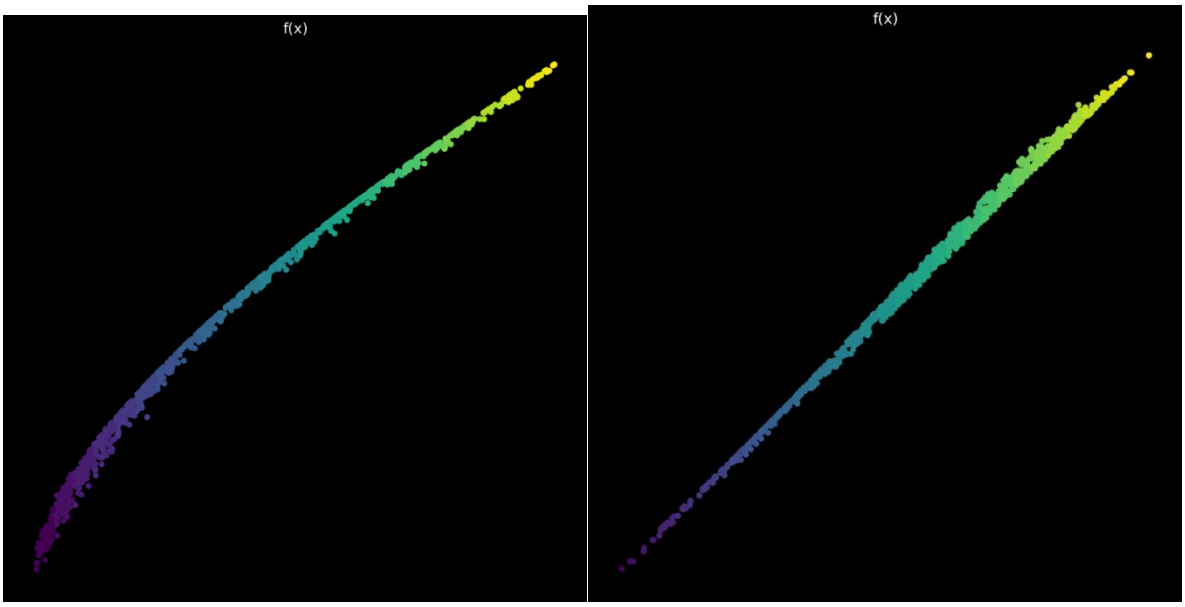




13.Deeper neural network

```
# deeper network with random weights
show_scatterplot(X,'x')
n_hidden = 5

# NL = nn.ReLU()
NL = nn.Tanh()

for i in range(5):
    model = nn.Sequential(
        nn.Linear(2, n_hidden),
        NL,
        nn.Linear(n_hidden, n_hidden),
        NL,
        nn.Linear(n_hidden, n_hidden),
        NL,
        nn.Linear(n_hidden, n_hidden),
        NL,
        nn.Linear(n_hidden, 2)
    )
    model.to(device)
    with torch.no_grad():
        Y = model(X).detach()
    show_scatterplot(Y, 'f(x)')
```





La visualisation de transformation effectuée par un simple réseau de neurones non entraîné. Le réseau est constitué d'une couche linéaire, qui effectue une transformation affine, suivie d'une tangente hyperbolique non-linéaire, et enfin d'une autre couche linéaire. Nous constatons qu'elle est différente des transformations linéaires et non linéaires vues précédemment (frontières non linéaires).

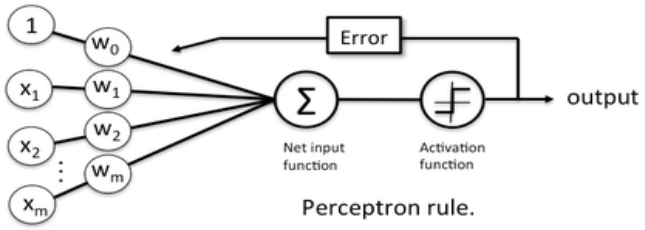
L'intégralité du code de ce TP est disponible dans le notebook **google-colabe** suivant si vous souhaitez le tourner vous-même.

https://drive.google.com/file/d/1rhkGogoKJjKkFOAb5eM5hHv-T-c-FvMo/view?usp=sharing

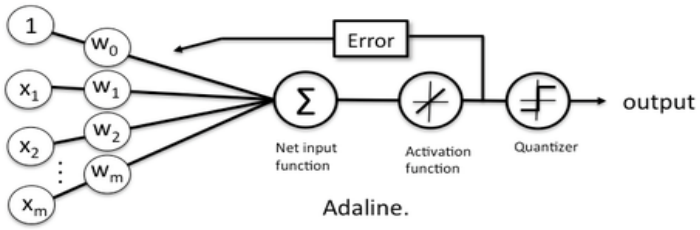
4.TP3 : Adaline, Perceptron et Perceptron multicouche

Adaline et le Perceptron sont tous deux des modèles de réseau neuronal (monocouche). Le Perceptron est l'un des algorithmes d'apprentissage les plus anciens et les plus simples, et je considérerais Adaline comme une amélioration par rapport au Perceptron.

Le Perceptron utilise les étiquettes de classe pour apprendre les coefficients du modèle. Tant que Adaline utilise des valeurs prédites continues (à partir de l'entrée nette) pour apprendre les coefficients du modèle, ce qui est plus "puissant" puisqu'il nous dit par "combien" nous avions raison ou tort,

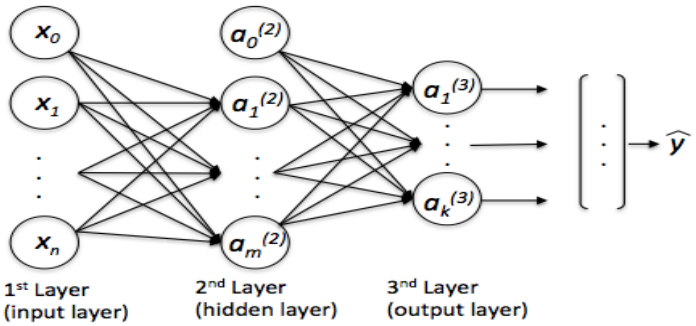


Perceptron rule.



Adaline.

Bien que vous n'ayez pas spécifiquement posé de questions sur les réseaux de neurones multicouches, permettez-moi d'ajouter quelques phrases sur l'une des architectures de réseaux de neurones multicouches les plus anciennes et les plus populaires : le Perceptron multicouche (MLP). Le terme « Perceptron » est un peu malheureux dans ce contexte, car il n'a vraiment pas grand-chose à voir avec l'algorithme Perceptron de Rosenblatt.



4.1 Adaline

1.Construit le modèle ADALINE en utilisant la classe nn.Module

```
[ ]  class Adaline(torch.nn.Module):
         def __init__(self, num_features):
             super(Adaline, self).__init__()
             self.linear = nn.Linear(num_features,1)
             self.linear.weight.detach().zero_()
             self.linear.bias.detach().zero_()

         def forward(self, x):
             activations = self.linear(x)
             return activations.view(-1)
```

2.Chargement des données

A l'aide de 'iris.txt', nous créons un jeu de données binaire en 2-D : Les 100 dernières instances d'iris décrites uniquement par le 2ème et 3ème caractéristique, et nous divisons l'ensemble de données en ensembles d'entraînement et de test (70 %, 30 %) enfin nous avons Normalisé le jeu de données.

```
import pandas as pd
from google.colab import drive

drive.mount('/content/drive')
path='/content/drive/My Drive/deeplearning/iris.txt'

df = pd.read_csv(path, index_col=None, header=None)    #reding iris
df.columns = ['x1', 'x2', 'x3', 'x4', 'y'] #define features names
df = df.iloc[50:150]   #taking last 100 elements that belong to 2 classes (100*2)
df['y'] = df['y'].apply(lambda x: 0 if x == 'Iris-versicolor' else 1) #coder la 2e classe par 0 la 3e par 1


# Assign features and target

X = torch.tensor(df[['x2', 'x3']].values, dtype=torch.float) #taking just last 2 features
y = torch.tensor(df['y'].values, dtype=torch.int) # y is a tensor of features

# Shuffling & train/test split

torch.manual_seed(123)
shuffle_idx = torch.randperm(y.size(0), dtype=torch.long)  #shuffle indexes
X, y = X[shuffle_idx], y[shuffle_idx]  #shuffle data (we shuffle also labels)
percent70 = int(shuffle_idx.size(0)*0.7)
X_train, X_test = X[shuffle_idx[:percent70]], X[shuffle_idx[percent70:]]  # 70% for training training
y_train, y_test = y[shuffle_idx[:percent70]], y[shuffle_idx[percent70:]]  # 30% for  testing

# Normalize (mean zero, unit variance)

mu, sigma = X_train.mean(dim=0), X_train.std(dim=0)
X_train = (X_train - mu )/sigma
X_test = (X_test - mu )/sigma
```

3.Trainer le modèle

```
def train(model, x, y, epochs, learning_rate, seed):
    cost = []
    torch.manual_seed(seed)
    optimizer = torch.optim.SGD(model.parameters(),lr=learning_rate)   #use a SGD optimizer
    c = nn.MSELoss()
    for e in range(epochs):
        yhat = model.forward(x) # calcul yhat
        loss = c(yhat,y) # calcul the loss function using MSE
        optimizer.zero_grad()    # set the gradients to zero
        loss.backward()  # calculer le gradients
        optimizer.step()  # mise a jour des poids ####
        print('epoch {} , loss {}'.format(e,loss.item()))
```

```
[ ] model = Adaline(num_features=X_train.size(1))
    train(model, X_train, y_train.float(),num_epochs=100,learning_rate=0.01,seed=100)
```

```
epoch 0 , loss 0.48571428656578064
epoch 1 , loss 0.4699629247188568
epoch 2 , loss 0.45489606261253357
epoch 3 , loss 0.4404822289943695
epoch 4 , loss 0.42669153213500977
epoch 5 , loss 0.4134954810142517
epoch 6 , loss 0.40086689591407776
epoch 7 , loss 0.3887799382209778
epoch 8 , loss 0.37721002101898193
epoch 9 , loss 0.3661336600780487
epoch 10 , loss 0.35552868247032166
```

4.Calculer la précision du modèle

```
[ ] def custom_where(cond, x_1, x_2):
        return (cond * x_1) + (torch.logical_not(cond) * x_2)
    train_pred = model.forward(X_train)
    train_acc = torch.mean(
        (custom_where(train_pred > 0.5, 1, 0).int() == y_train).float())
    test_pred = model.forward(X_test)
    test_acc = torch.mean((custom_where(test_pred > 0.5, 1, 0).int() == y_test).float())
    print('Training Accuracy:    %.2f' % (train_acc*100))
    print('Test Accuracy:    %.2f' % (test_acc*100))
    print('Weights:   ', model.linear.weight)
    print('Bias:   ', model.linear.bias)
```

```
Training Accuracy: 87.14
Test Accuracy: 76.67
Weights Parameter containing:
tensor([[0.0043, 0.3283]], requires_grad=True)
Bias Parameter containing:
tensor([0.4213], requires_grad=True)
```

4 .2 Perceptron

5.Création d'un modèle Perceptron à l'aide de la classe nn.Module

```
[ ]  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

     class Perceptron():
         def __init__(self, num_features):
             self.num_features = num_features
             self.weights = torch.zeros(num_features, 1,
                                        dtype=torch.float32, device=device)
             self.bias = torch.zeros(1, dtype=torch.float32, device=device)
             self.ones = torch.ones(1)
             self.zeros = torch.zeros(1)

         def forward(self, x):
             linear = torch.add(torch.mm(x, self.weights), self.bias)
             predictions = custom_where(linear > 0., 1, 0).float()
             return predictions

         def backward(self, x, y):
             predictions = self.forward(x)
             errors = y - predictions
             return errors


         def train(self, x, y, epochs):
             for e in range(epochs):
                 for i in range(y.shape[0]):
                     errors = self.backward(x[i].view(1, self.num_features), y[i]).view(-1)
                     self.weights += (errors * x[i]).view(self.num_features, 1)
                     self.bias += errors

         def evaluate(self, x, y):
             predictions = self.forward(x).reshape(-1)
             accuracy = torch.sum(predictions == y).float() / y.shape[0]
             return accuracy
```

6.Charger le jeu de données 'perceptron_toydata'

Diviser l'ensemble de données en ensembles d'apprentissage et de test et Normaliser les données

```
from google.colab import drive
import pandas as pd
drive.mount('/content/drive')
path='/content/drive/My Drive/deeplearning/perceptron_toydata.txt'

df = pd.read_csv(path, index_col=None, header=None , delimiter='\t')
df.columns = ['x1', 'x2', 'y']
X = torch.tensor(df[['x1', 'x2']].values, dtype=torch.float)
y = torch.tensor(df['y'].values, dtype=torch.int)
print('Class label counts:', torch.bincount(y))
print('X.shape:', X.shape)
print('y.shape:', y.shape)

# Shuffling & train/test split
shuffle_idx = torch.randperm(y.size(0), dtype=torch.long)
X, y = X[shuffle_idx], y[shuffle_idx]
percent70 = int(shuffle_idx.size(0)*0.7)
X_train, X_test = X[shuffle_idx[:percent70]], X[shuffle_idx[percent70:]]
y_train, y_test = y[shuffle_idx[:percent70]], y[shuffle_idx[percent70:]]
# Normalize (mean zero, unit variance)
mu, sigma = X_train.mean(axis=0), X_train.std(axis=0)
X_train = (X_train - mu) / sigma
X_test = (X_test - mu) / sigma
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Class label counts: tensor([50, 50])
X.shape: torch.Size([100, 2])
y.shape: torch.Size([100])
torch.Size([70, 2])
torch.Size([70])
torch.Size([30, 2])
torch.Size([30])
```

7.Train the perceptron

```
[ ]  model = Perceptron(num_features=2)

     Xtrain = torch.tensor(X_train, dtype=torch.float32, device=device)
     ytrain = torch.tensor(y_train, dtype=torch.float32, device=device)

     model.train(Xtrain, ytrain, epochs=5)

     print('Model parameters:')
     print('  Weights: %s' % model.weights)
     print('  Bias: %s' % model.bias)
```

```
Model parameters:
  Weights: tensor([[2.4768],
         [0.8986]])
  Bias: tensor([-1.])
```

8.evaluate the model (accuracy)

```
Xtest = torch.tensor(X_test, dtype=torch.float32, device=device)
ytest = torch.tensor(y_test, dtype=torch.float32, device=device)

test_acc = model.evaluate(Xtest, ytest)
print('Test set accuracy: %.2f%%' % (test_acc*100))
```

```
Test set accuracy: 100.00%
```

4.3 Multi Layer Perceptron

9.Construction d'un modèle simple de Perceptron multicouche

Avec une couche cachée. Après la couche cachée, nous utiliserons ReLU comme activation avant que les informations ne soient envoyées à la couche de sortie. En tant que fonction d'activation de sortie, nous utiliserons Sigmoid.

```
class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features,num_hidden_1):
        super(MultilayerPerceptron, self).__init__()
        self.linear_1 = torch.nn.Linear(num_features, num_hidden_1)
        self.linear_out = torch.nn.Linear(num_hidden_1, 1)



    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_out(out)
        out = F.sigmoid(out)

        return out
```

10.Création d'un jeu de données aléatoire et attribuez des étiquettes binaires {0,1}

```
def blob_label(y, label, loc): # assign labels
    target = np.copy(y)
    for l in loc:
        target[y == l] = label
    return target
x_train, y_train = make_blobs(n_samples=40, n_features=2, cluster_std=1.5, shuffle=True)
x_train = torch.FloatTensor(x_train)
y_train = torch.FloatTensor(blob_label(y_train, 0, [0]))
y_train = torch.FloatTensor(blob_label(y_train, 1, [1,2,3]))
x_test, y_test = make_blobs(n_samples=10, n_features=2, cluster_std=1.5, shuffle=True)
x_test = torch.FloatTensor(x_test)
y_test = torch.FloatTensor(blob_label(y_test, 0, [0]))
y_test = torch.FloatTensor(blob_label(y_test, 1, [1,2,3]))
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
torch.Size([40, 2])
torch.Size([40])
torch.Size([10, 2])
torch.Size([10])
```

11.Définition de modèle avec la dimension d'entrée 2 et la dimension cachée 10.

Puisque la tâche consiste à classer les étiquettes binaires, nous pouvons utiliser BCELoss (Binary Cross Entropy Loss) comme fonction de perte. L'optimiseur est SGD (Stochastic Gradient Descent) avec un taux d'apprentissage de 0,01.

```
model = MultilayerPerceptron(2, 10)
optimizer = torch.optim.SGD(model.parameters(),lr=0.01)  #use a SGD optimizer
criterion = torch.nn.MSELoss()
```

12.Vérifiez la perte de test avant et après la formation du modèle.

```
[ ] model.eval()
    y_pred = model(x_test)
    before_train = criterion(y_pred.squeeze(), y_test)
    print('Test loss before training' , before_train.item())
```

```
Test loss before training 0.23654893040657043
```

```
model.train()
epoch = 20
for epoch in range(epoch):
    optimizer.zero_grad()
    y_pred = model.forward(x_train)
    loss = criterion(y_pred,y_train)
    print('Epoch {}: train loss: {}'.format(epoch, loss.item(
    # Backward pass
    loss.backward()
    optimizer.step()
```

```
Epoch 0: train loss: 0.2857312858104706
Epoch 1: train loss: 0.28241994976997375
Epoch 2: train loss: 0.27909523248672485
Epoch 3: train loss: 0.2757806181907654
Epoch 4: train loss: 0.27250081300735474
Epoch 5: train loss: 0.26928097009658813
Epoch 6: train loss: 0.26614561676979065
Epoch 7: train loss: 0.2631177306175232
Epoch 8: train loss: 0.26021769642829895
Epoch 9: train loss: 0.2574624717235565
Epoch 10: train loss: 0.25486528873443604
Epoch 11: train loss: 0.25243523716926575
Epoch 12: train loss: 0.2501772344112396
Epoch 13: train loss: 0.24809250235557556
Epoch 14: train loss: 0.2461787760257721
Epoch 15: train loss: 0.2444310337305069
Epoch 16: train loss: 0.24284198880195618
Epoch 17: train loss: 0.24140284955501556
Epoch 18: train loss: 0.24010413885116577
Epoch 19: train loss: 0.23894067108631134
```

13.Changement des hyperparamètres

Afin d'améliorer le modèle, nous allons essayer différentes valeurs de paramètre pour les hyperparamètres (c'est-à-dire la taille de la dimension cachée, la taille de l'époque, les taux d'apprentissage). Nous également essayer de modifier la structure de modèle (c'est-à-dire ajouter plus de couches cachées) pour voir si votre mode s'améliore.

```python
class MultilayerPerceptronDifferent(torch.nn.Module):

    def __init__(self, num_features,num_hidden_1,num_hidden_2):
        super(MultilayerPerceptronDifferent, self).__init__()
        self.linear_1 = torch.nn.Linear(num_features, num_hidden_1)
        self.linear_2 = torch.nn.Linear(num_hidden_1, num_hidden_2)
        self.linear_out = torch.nn.Linear(num_hidden_2, 1)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        out = self.linear_out(out)
        out = F.sigmoid(out)
        return out


model1 = MultilayerPerceptronDifferent(2,512,256)
optimizer = torch.optim.SGD(model.parameters(),lr=0.01)
criterion = torch.nn.MSELoss()
```

```python
[ ] model1.train()
    epoch = 30
    for epoch in range(epoch):
        optimizer.zero_grad()
        y_pred = model.forward(x_train)
        loss = criterion(y_pred,y_train)
        print('Epoch {}: train loss: {}'.format(epoch, loss.item()))
        # Backward pass
        loss.backward()
        optimizer.step()

    model1.eval()
    y_pred = model1(x_test)
    after_train = criterion(y_pred.squeeze(), y_test)
    print('Test loss after Training' , after_train.item())
```

```
loss: 0.22951915860176086
loss: 0.22944670915603638
loss: 0.2293809950351715
loss: 0.22932127118110657
loss: 0.22926704585552216
loss: 0.22921770811080933
```

Nous remarquons qu'après la modification de quelques paramètres, les résultats sont parfois plus mieux que le modèle donné au départ, mais parfois sont tout loin de ce qui est demandé.

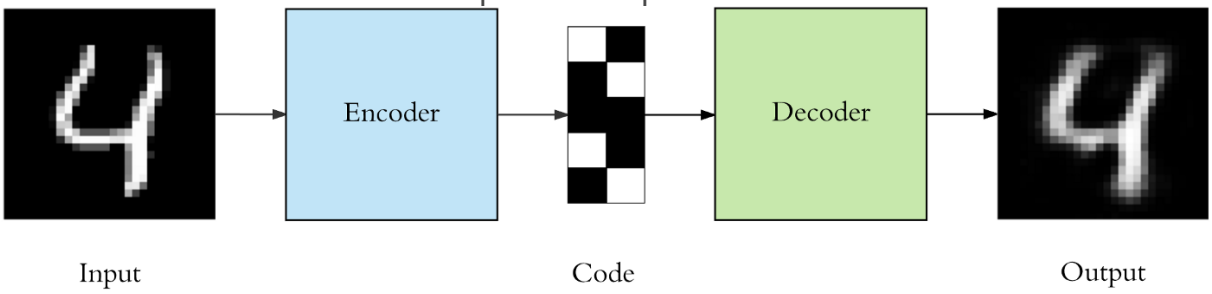
L'avantage du MLP sur les classiques Perceptron et Adaline ,En connectant les neurones artificiels de ce réseau par des fonctions d'activation non linéaires, nous pouvons créer des limites de décision complexes et non linéaires qui nous permettent de résoudre des problèmes où les différentes classes ne sont pas linéairement séparables.

L'intégralité du code de ce TP est disponible dans le notebook **google-colabe** suivant si vous souhaitez le tourner vous-même.

https://drive.google.com/file/d/1y9IFCJYIN4uN0Tz27VVYa9x22Yns_Kgm/view?usp=sharing

5.TP4 : Les Auto-encodeurs avec PyTorch

Les auto-encodeurs sont un type spécifique de réseaux de neurones à anticipation où l'entrée est la même que la sortie. Ils compriment l'entrée dans un code de dimension inférieure, puis reconstruisent la sortie à partir de cette représentation. Le code est un "résumé" compact ou une "compression" de l'entrée, également appelée représentation de l'espace latent. Un auto-encodeur se compose de 3 composants : encodeur, code et décodeur. L'encodeur comprime l'entrée et produit le code, le décodeur reconstruit ensuite l'entrée uniquement à partir de ce code.



1.Importation des librairies (pyTorch pour coder les ANN)

```
import torch
import torchvision
from torch import nn
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import MNIST
from matplotlib import pyplot as plt
```

2.La fonction suivante permet de convertir un vecteur en image :

```
def to_img(x):
    x = 0.5 * (x + 1)
    x = x.view(x.size(0), 28, 28)
    return x
```

3.Une fonction utilise omshow() qui permet d'afficher les images :

```
def display_images(in_, out, n=1):
    for N in range(n):
        if in_ is not None:
            in_pic = to_img(in_.cpu().data)
            plt.figure(figsize=(18, 6))
            for i in range(4):
                plt.subplot(1,4,i+1)
                plt.imshow(in_pic[i+4*N])
                plt.axis('off')
        out_pic = to_img(out.cpu().data)
        plt.figure(figsize=(18, 6))
        for i in range(4):
            plt.subplot(1,4,i+1)
            plt.imshow(out_pic[i+4*N])
            plt.axis('off')
```

4.Définir une étape de chargement des données et charger le jeu de données MNIST :

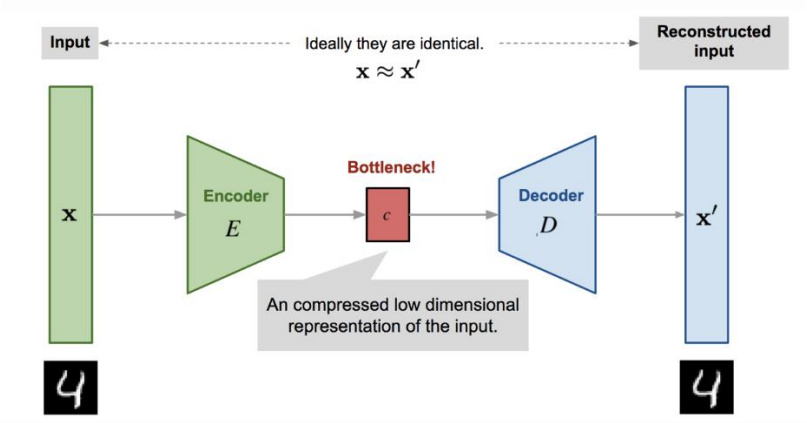
Nous pouvons importer le jeu de données à l'aide de la bibliothèque **torchvision**. Nous téléchargeons les jeux de données d'entraînement et de test et nous transformons les jeux de données d'image en **Tensor**. Nous n'avons pas besoin de normaliser les images car les jeux de données contiennent des images colorées. . Le **DataLoaderest** utilisé pour créer des chargeurs de données pour les ensembles de formation, de validation et de test, qui sont divisés en mini-lots. Le **batch-size** est le nombre d'échantillons utilisés dans une itération lors de la formation du modèle.

```
batch_size = 256

img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

dataset = MNIST('./data', transform=img_transform, download=True)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
9913344/? [00:00<00:00, 3095006.03it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
29696/? [00:00<00:00, 559285.73it/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
1649664/? [00:00<00:00, 1722371.87it/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
```

3.1 Standard Auto-encoder

L'encodeur et le décodeur sont tous deux des réseaux de neurones à anticipation entièrement connectés, essentiellement les ANN . Le code est une couche unique d'un ANN avec la dimensionnalité de notre choix. Le nombre de nœuds dans la couche de code (taille du code) est un *hyperparamètre* que nous définissons avant de former l'auto-encodeur.



L'entrée passe d'abord par l'encodeur, qui est un ANN entièrement connecté, pour produire le code. Le décodeur, qui a la structure ANN similaire, produit alors la sortie uniquement en utilisant le code. Le but est d'obtenir une sortie identique à l'entrée. Notez que l'architecture du décodeur est l'image miroir du codeur. Ce n'est pas une exigence, mais c'est généralement le cas. La seule exigence est que la dimensionnalité de l'entrée et de la sortie doit être la même. Tout ce qui se trouve au milieu peut être joué avec.

Passons maintenant à la pratique et implémentons un SAE !

6.Définir l'architecture du modèle Auto-encoder et la perte de reconstruction avec : n = 28 x 28 = 784 /Utilisez d = 30 pour AE standard (couche cachée sous-complète)

```
d=30
#d = 500 #activer ce ligne et recompiler pour le denoising autoencoder
class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, d),
            nn.Tanh(),
        )
        self.decoder = nn.Sequential(
            nn.Linear(d, 28 * 28),
            nn.Tanh(),
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

model = Autoencoder().to(device)
criterion = nn.MSELoss()
```

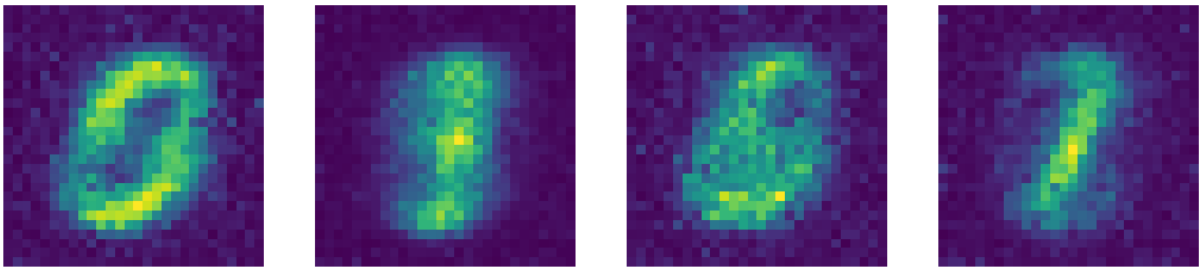
7.Configurez l'optimiseur. On utilise ici : learning_rate égal à 1e-3

```
[8] learning_rate = 1e-3

    optimizer = torch.optim.Adam(
        model.parameters(),
        lr=learning_rate,
    )
```

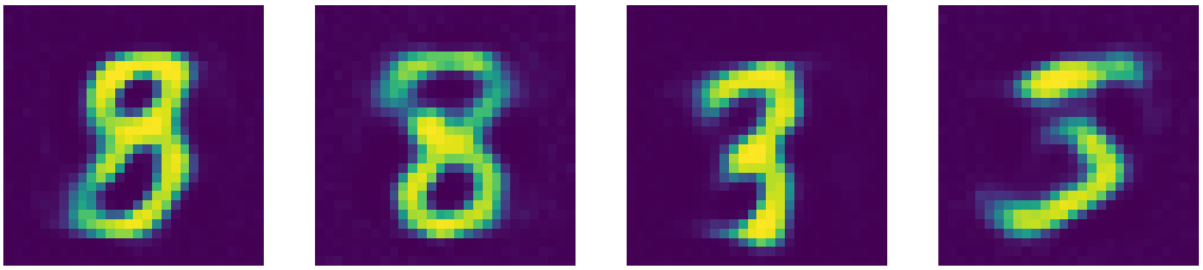
8.Entraînez l'auto-encodeur standard :

```
num_epochs = 20
for epoch in range(num_epochs):
    for data in dataloader:
        img, _ = data
        img = img.to(device)
        img = img.view(img.size(0), -1)
        # ===================forward=====================
        output = model(img)
        loss = criterion(output, img.data)
        # ===================backward====================
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    # ===================log========================
    print(f'epoch [{epoch + 1}/{num_epochs}], loss:{loss.item():.4f}')
    display_images(None, output)
```

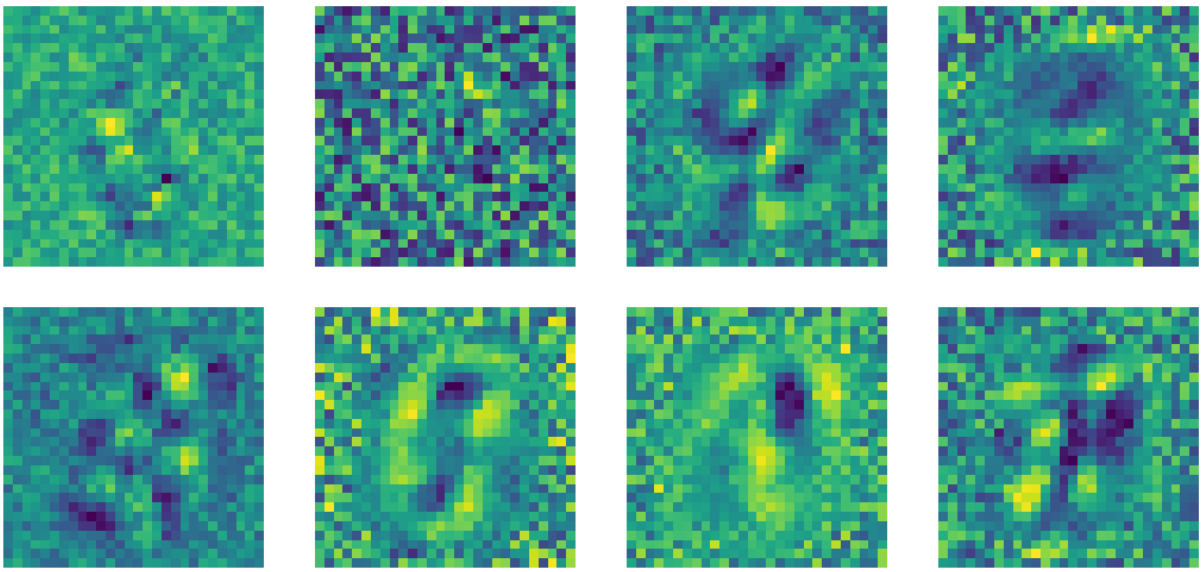
Coder input



Decoder output

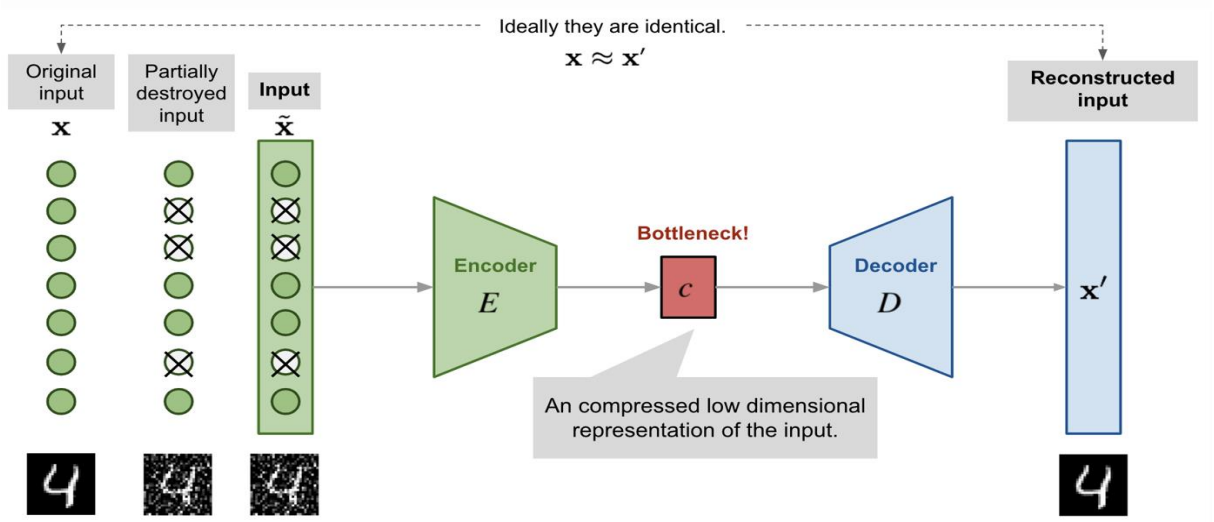


9. Visualisez quelques noyaux de l’encodeur standard:



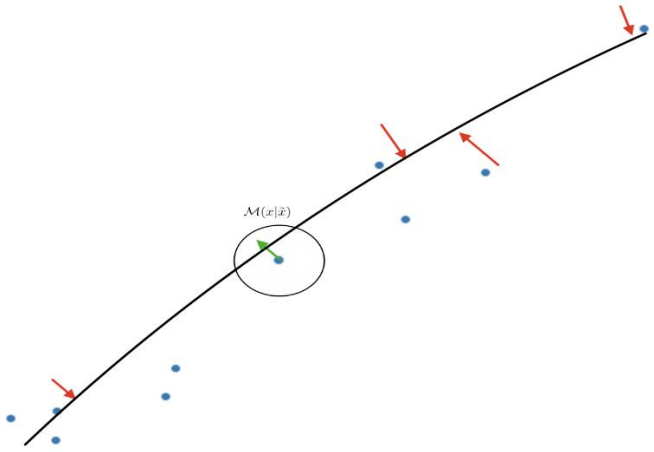
3.2 Denoising Auto-encoder

Ce type d'Auto-encoder est une alternative au concept d'Auto-encoder standard dont nous venons de parler, qui est sujet à un risque élevé de surajustement. Dans le cas d'un Denoising Auto-encoder, les données sont partiellement corrompues par des bruits ajoutés au vecteur d'entrée de manière stochastique. Ensuite, le modèle est formé pour prédire le point de données d'origine non corrompu comme sortie.



Vous pouvez voir sur l'image suivant quelques données représentées par les points bleus. Nos données corrompues resteront dans le cercle noir de la corruption équiprobable. Lors de la formation, l'objectif est de minimiser la fonction de coût log-vraisemblance négative.

Cette optimisation conduit à minimiser la distance entre l'entrée corrompue et la variété noire qui caractérise nos entrées. Ainsi, notre modèle apprend un champ de vecteurs de reconstruction D(E(x))-x, certains de ces vecteurs sont représentés par les flèches rouges.



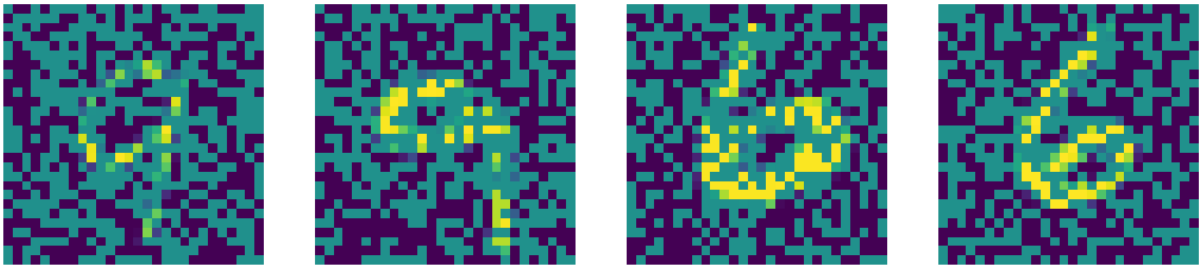
Passons maintenant à la pratique et implémentons un DAE !

Nous gardant la même architecture du modèle Auto-encoder et la perte de reconstruction avec : n = 28 x 28 = 784, utilisant cette fois d = 500 pour AE de débrouillage (couche cachée sous-complète). Voir le code dans la partie précédente.

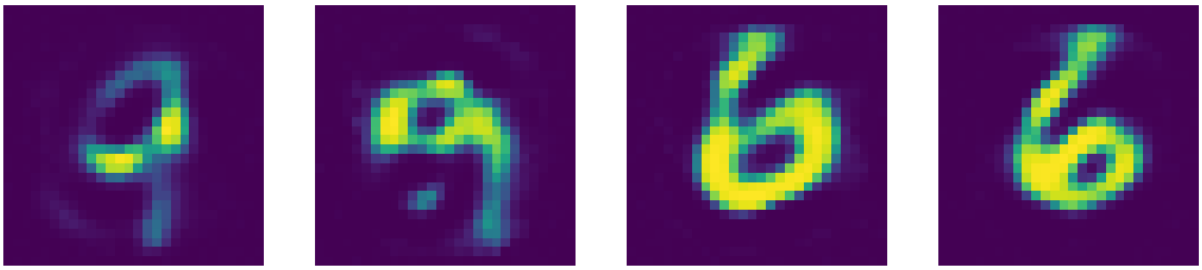
Entraînez l'auto-encodeur de débrouillage :

```
num_epochs = 20
do = nn.Dropout()
for epoch in range(num_epochs):
    for data in dataloader:
        img, _ = data
        img = img.to(device)
        img = img.view(img.size(0), -1)
        noise = do(torch.ones(img.shape)).to(device)
        img_bad = (img * noise).to(device)
        # ===================forward=====================
        output = model(img_bad)
        loss = criterion(output, img.data)
        # ===================backward====================
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    # ===================log========================
    print(f'epoch [{epoch + 1}/{num_epochs}], loss:{loss.item():.4f}')
    display_images(img_bad, output)
```

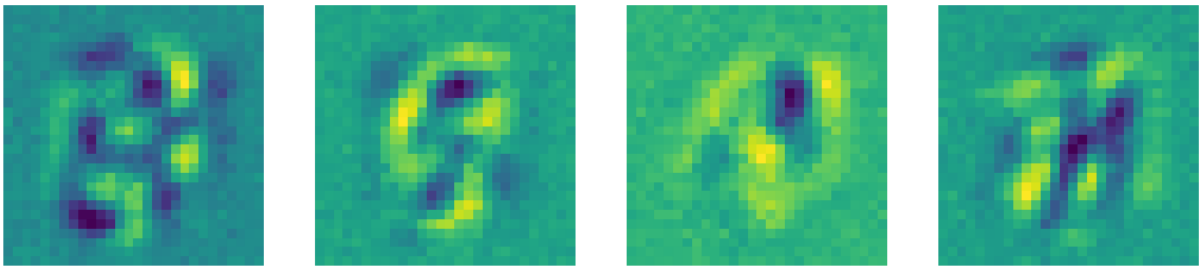
Coder input

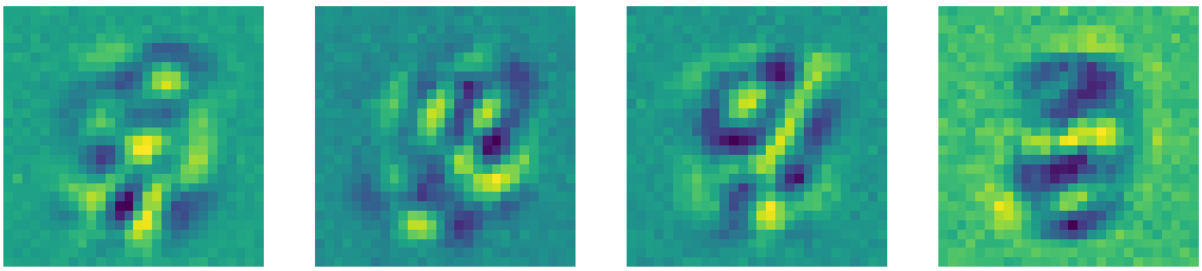


Decoder output



9. Visualisez quelques noyaux de l'encodeur de débrouillage :





10. Analyse des résultats obtenues.

Les résultats sont en effet assez similaires, mais pas exactement les mêmes. Nous avons des données bien codés sont pas trop similaire aux données entrées, et nous constatons que les résultats de l'encodeur à débrouillage sont plus complexes que les résultats de l'encodeur simple. Généralement. Notre encodeur automatique a plutôt bien fonctionné.

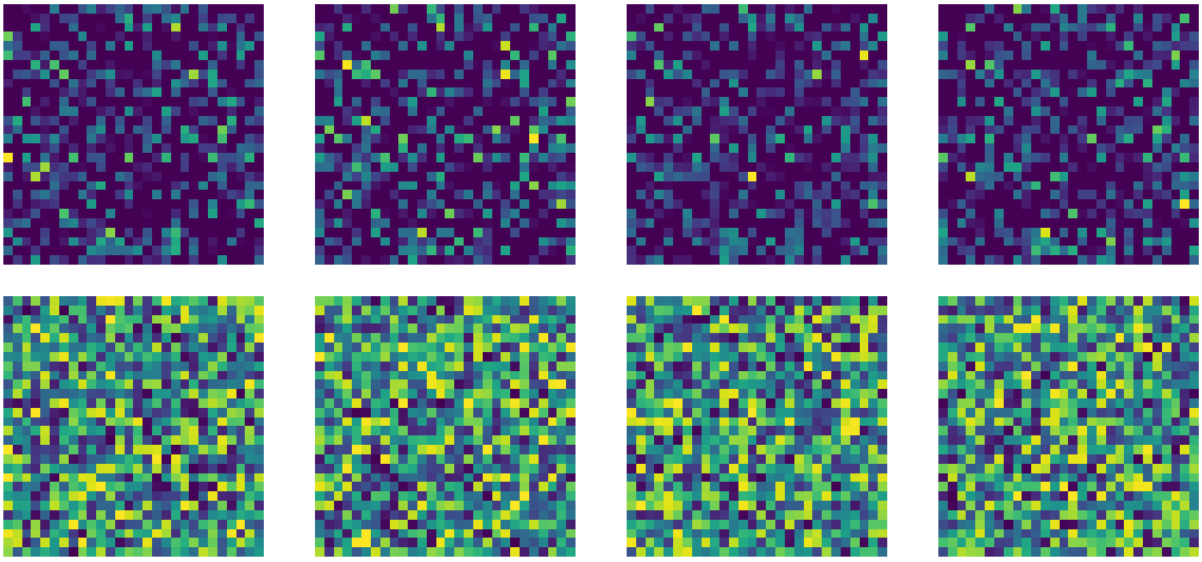
11.Modifie les paramètres de l'Auto-encoder et analyse leur impact. Conclure.

Il y a 4 hyperparamètres que nous devons définir avant de former un auto-encodeur :

- **Taille du code :** nombre de nœuds dans la couche intermédiaire. Une taille plus petite entraîne plus de compression.
- **Nombre de couches :** l'auto-encodeur peut être aussi profond que nous le souhaitons. Dans la figure ci-dessus, nous avons 2 couches dans l'encodeur et le décodeur, sans tenir compte de l'entrée et de la sortie.
- **Nombre de nœuds par couche :** l'architecture d'auto-encodeur sur laquelle nous travaillons appelée auto- encodeur empilé puisque les couches sont empilées les unes après les autres. Habituellement, les encodeurs automatiques empilés ressemblent à un "sandwitch". Le nombre de nœuds par couche diminue avec chaque couche suivante du codeur et augmente à nouveau dans le décodeur. De plus, le décodeur est symétrique au codeur en termes de structure de couche. Comme indiqué ci-dessus, cela n'est pas nécessaire et nous avons un contrôle total sur ces paramètres.
- **Fonction de perte :** nous utilisons soit l'erreur quadratique moyenne (mse) , soit l'entropie croisée binaire . Si les valeurs d'entrée sont dans la plage [0, 1], nous utilisons généralement l'entropie croisée, sinon nous utilisons l'erreur quadratique moyenne.

Dans notre cas les entré sont de taille fixe 28*28 donc nous pouvons jouer sure Fonction de perte et Taille du code :

Pour le Relu et d=100 on obtiens les résultats suivants :



Les résultats ne sont pas ce qui est demandé d'un encodeur, le décodeur ne réussite pas à donner des résultats similaires à l'entrée de l'encodeur, donc les informations sont perdues (biaisés).

Les auto-encodeurs sont une technique de réduction de dimensionnalité très utile. Ils sont très populaires comme matériel pédagogique dans les cours d'introduction à l'apprentissage en profondeur, probablement en raison de leur simplicité. Dans cet TP, nous avons essayé d'implémenter e comprendre deux types des encodeurs : les encodeurs simple ou standard et les encodeurs de débrouillage.

L'intégralité du code de ce **TP 4** est disponible dans le notebook **google-colabe** suivant si vous souhaitez le tourner vous-même.

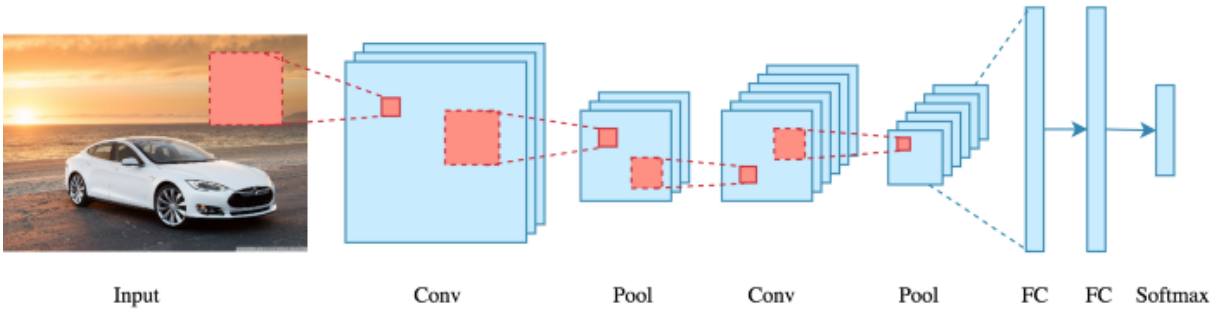
https://drive.google.com/file/d/18Tbx0PwcWrgh54_szrEktcj1-bIRd9Zh/view?usp=sharing

6.TP5 : Les Réseaux de neurones convolutifs CNN

Les réseaux de neurones convolutifs (CNN) sont partout. C'est sans doute l'architecture d'apprentissage en profondeur la plus populaire. Le récent regain d'intérêt pour l'apprentissage en profondeur est dû à l'immense popularité et à l'efficacité des CNN. L'intérêt pour CNN a commencé avec **AlexNet** en 2012 et il a connu une croissance exponentielle depuis. En seulement trois ans, les chercheurs sont passés d'AlexNet à 8 couches à **ResNet** à 152 couches.

CNN est également efficace en termes de calcul. Il utilise des opérations spéciales de convolution et de mise en commun et effectue le partage des paramètres. Cela permet aux modèles CNN de fonctionner sur n'importe quel appareil, ce qui les rend universellement attrayants.

Tous les modèles CNN suivent une architecture similaire, comme illustré dans la figure ci-dessous :

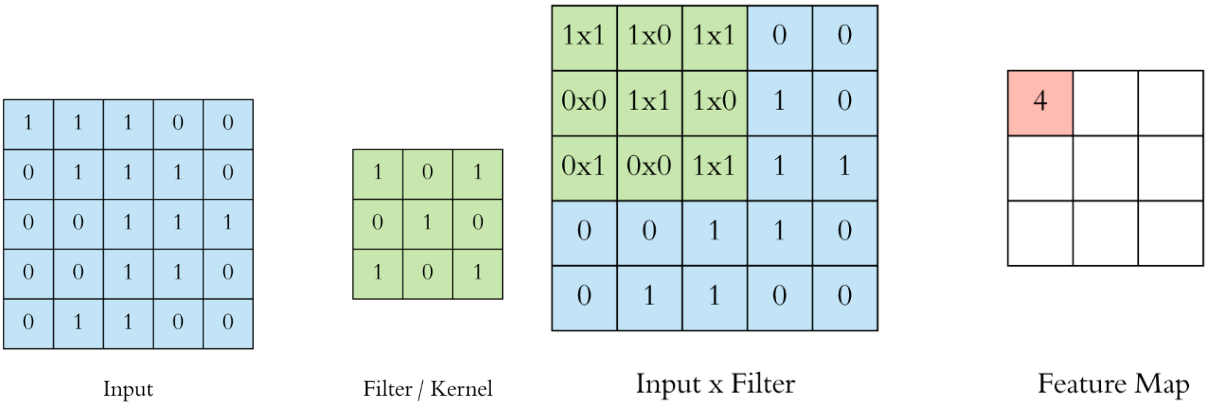


Par exemple Il y a une image d'entrée avec laquelle nous travaillons. Nous effectuons une série d'opérations de convolution + Pooling, suivies d'un certain nombre de couches entièrement connectées. Si nous effectuons une classification multiclasse, la sortie est softmax.

Convolution

Le principal élément constitutif de CNN est la couche convolutive. La convolution est une opération mathématique permettant de fusionner deux ensembles d'informations. Dans notre cas, la convolution est appliquée sur les données d'entrée à l'aide d'un filtre de convolution pour produire une carte de caractéristiques. De nombreux termes sont utilisés, alors visualisons-les un par un.

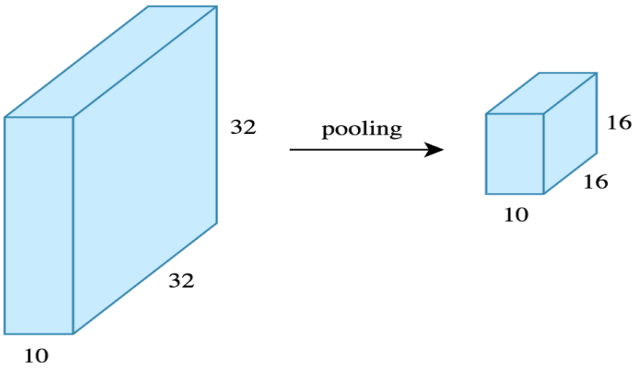
Ici, le filtre est en haut à gauche, la sortie de l'opération de convolution **"4"** est affichée dans la carte de caractéristiques résultante. Nous faisons ensuite glisser le filtre vers la droite et effectuons la même opération, en ajoutant également ce résultat à la carte des fonctionnalités.



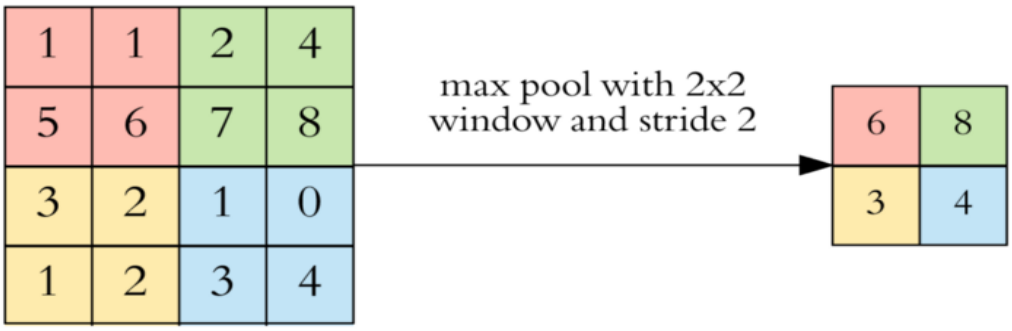
Pooling

Après une opération de convolution, nous effectuons généralement une Pooling pour réduire la dimensionnalité. Cela nous permet de réduire le nombre de paramètres, ce qui à la fois raccourcit le temps d'entraînement et combat le surapprentissage. Les couches de regroupement sous-échantillonnent chaque carte d'entités indépendamment, réduisant la hauteur et la largeur, en gardant la profondeur intacte.

Le type de Pooling le plus courant est le **MaxPooling** ou regroupement maximum qui prend simplement la valeur maximale dans la fenêtre de regroupement. Contrairement à l'opération de convolution, la pooling n'a pas de paramètres. Il fait glisser une fenêtre sur son entrée et prend simplement la valeur maximale dans la fenêtre. Semblable à une convolution, nous spécifions la taille de la fenêtre et la foulée.



Voici le résultat de le Max-PooLing en utilisant une fenêtre 2x2 et une foulée 2. Chaque couleur désigne une fenêtre différente. Étant donné que la taille de la fenêtre et la foulée sont de 2, les fenêtres ne se chevauchent pas.



Passons maintenant à la pratique et implémentons un CNN avec Pytorche !

1.Importation des outils nécessaires, PyTorche est toujours là :

```
[1] import torch
    import torch.nn as nn
    import torch.nn.functional as F
    import torch.optim as optim
    from torchvision import datasets, transforms
    import matplotlib.pyplot as plt
    import numpy
```

Les fonctions suivantes permet de visualiser les des données avant et après le passage par le modèle :

```
def set_default(figsize=(10, 10), dpi=100):
    plt.style.use(['dark_background', 'bmh'])
    plt.rc('axes', facecolor='k')
    plt.rc('figure', facecolor='k')
    plt.rc('figure', figsize=figsize, dpi=dpi)


def plot_data(X, y, d=0, auto=False, zoom=1):
    X = X.cpu()
    y = y.cpu()
    plt.scatter(X.numpy()[:, 0], X.numpy()[:, 1], c=y, s=20, cmap=plt.cm.Spectral)
    plt.axis('square')
    plt.axis(np.array((-1.1, 1.1, -1.1, 1.1)) * zoom)
    if auto is True: plt.axis('equal')
    plt.axis('off')

    _m, _c = 0, '.15'
    plt.axvline(0, ymin=_m, color=_c, lw=1, zorder=0)
    plt.axhline(0, xmin=_m, color=_c, lw=1, zorder=0)


def plot_model(X, y, model):
    model.cpu()
    mesh = np.arange(-1.1, 1.1, 0.01)
    xx, yy = np.meshgrid(mesh, mesh)
    with torch.no_grad():
        data = torch.from_numpy(np.vstack((xx.reshape(-1), yy.reshape(-1))).T).float()
        Z = model(data).detach()
    Z = np.argmax(Z, axis=1).reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.3)
    plot_data(X, y)
```

2.Chargez le Dataset (MNIST) à l'aide des utilitaires PyTorch DataLoader et visualisez quelques images :

```
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=64, shuffle=True)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False, transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=1000, shuffle=True)

plt.figure(figsize=(16, 6))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    image, _ = train_loader.dataset.__getitem__(i)
    plt.imshow(image.squeeze().numpy())
    plt.axis('off');
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ../data
9913344/? [00:00<00:00, 16200502.83it/s]
Extracting ../data/MNIST/raw/train-images-idx3-ubyte.gz to ../data/MNIST/raw
```

3.Creation des classes de model

Nous avons besoins de deux classes :

- **Une classe FC2Layers :** représente les couches totalement connectées ou bien es couches de classification.
- **Un classe CNN :** qui contient l'extraction des caractéristiques, les couches de convolution et les couche de Pooling (Max Pooling).

```
input_size  = 28*28   # size of images
output_size = 10      # number of classes
class FC2Layer(nn.Module):
    def __init__(self, input_size, n_hidden, output_size):
        super(FC2Layer, self).__init__()
        self.input_size = input_size
        self.network = nn.Sequential(
            nn.Linear(input_size, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, output_size),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, x):
        x = x.view(-1, self.input_size)
        return self.network(x)

class CNN(nn.Module):
    def __init__(self, input_size, n_feature, output_size):
        super(CNN, self).__init__()
        self.n_feature = n_feature
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=n_feature, kernel_size=5)
        self.conv2 = nn.Conv2d(n_feature, n_feature, kernel_size=5)
        self.fc1 = nn.Linear(n_feature*4*4, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x, verbose=False):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = x.view(-1, self.n_feature*4*4)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.log_softmax(x, dim=1)
        return x
```

4.Exécuter sur un GPU : chaîne de périphérique :

La commutation entre CPU et GPU dans PyTorch est contrôlée via une chaîne de périphérique, qui déterminera sans problème si le GPU est disponible, revenant au CPU sinon .

5.Trainer un petit réseau entièrement connecté :

```
[ ]  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
[ ]  def get_n_params(model):
         np=0
         for p in list(model.parameters()):
             np += p.nelement()
         return np
```

```
[ ]  n_hidden = 8 # 8 hidden units

     model_fnn = FC2Layer(input_size, n_hidden, output_size)
     model_fnn.to(device)
     optimizer = optim.SGD(model_fnn.parameters(), lr=0.01, momentum=0.5)
     print('Number of parameters: {}'.format(get_n_params(model_fnn)))

     for epoch in range(0, 1):
         train(epoch, model_fnn)
         test(model_fnn)
```

```
Number of parameters: 6442
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.261318
Train Epoch: 0 [6400/60000 (11%)]       Loss: 2.014086
Train Epoch: 0 [12800/60000 (21%)]      Loss: 1.334367
Train Epoch: 0 [19200/60000 (32%)]      Loss: 1.042423
Train Epoch: 0 [25600/60000 (43%)]      Loss: 0.641469
Train Epoch: 0 [32000/60000 (53%)]      Loss: 0.536230
Train Epoch: 0 [38400/60000 (64%)]      Loss: 0.781325
Train Epoch: 0 [44800/60000 (75%)]      Loss: 0.323836
Train Epoch: 0 [51200/60000 (85%)]      Loss: 0.650726
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.540229

Test set: Average loss: 0.4465, Accuracy: 8657/10000 (87%)
```

6.Trainer un ConvNet avec le même nombre de paramètres :

```
n_features = 6 # 6 feature maps

model_cnn = CNN(input_size, n_features, output_size)
model_cnn.to(device)
optimizer = optim.SGD(model_cnn.parameters(), lr=0.01, momentum=0.5)
print('Number of parameters: {}'.format(get_n_params(model_cnn)))

for epoch in range(0, 1):
    train(epoch, model_cnn)
    test(model_cnn)
```

```
Number of parameters: 6422
Train Epoch: 0 [0/60000 (0%)]   Loss: 2.291794
Train Epoch: 0 [6400/60000 (11%)]       Loss: 1.821252
Train Epoch: 0 [12800/60000 (21%)]      Loss: 0.873875
Train Epoch: 0 [19200/60000 (32%)]      Loss: 0.342783
Train Epoch: 0 [25600/60000 (43%)]      Loss: 0.335366
Train Epoch: 0 [32000/60000 (53%)]      Loss: 0.113160
Train Epoch: 0 [38400/60000 (64%)]      Loss: 0.341414
Train Epoch: 0 [44800/60000 (75%)]      Loss: 0.279331
Train Epoch: 0 [51200/60000 (85%)]      Loss: 0.160294
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.133932

Test set: Average loss: 0.1647, Accuracy: 9488/10000 (95%)
```

7.Changement de nombre des paramètres

```
n_hidden = 16
model_fnn = FC2Layer(input_size, n_hidden, output_size)
model_fnn.to(device)
optimizer = optim.SGD(model_fnn.parameters(), lr=0.001, momentum=0.5)
print('Number of parameters: {}'.format(get_n_params(model_fnn)))

for epoch in range(0, 1):
    train(epoch, model_fnn)
    test(model_fnn)
```


```
n_hidden = 16
model_fnn = FC2Layer(input_size, n_hidden, output_size)
model_fnn.to(device)
optimizer = optim.SGD(model_fnn.parameters(), lr=0.001, momentum=0.5)
print('Number of parameters: {}'.format(get_n_params(model_fnn)))

for epoch in range(0, 1):
    train(epoch, model_fnn)
    test(model_fnn)
```


8.Analyser les résultats et l'impact de ces paramètres.

Le ConvNet est plus performant avec le même nombre de paramètres, grâce à son utilisation des connaissances préalables sur les images. Les performances de ConvNet chutent lorsque nous permutons les pixels, mais les performances du réseau entièrement connecté restent les mêmes

- **ConvNet suppose que les pixels se trouvent sur une grille et sont stationnaires/locaux**
- **Il perd des performances lorsque cette hypothèse est fausse**
- **Le réseau entièrement connecté ne fait pas cette hypothèse**
- **Il réussit moins bien quand c'est vrai, car il ne profite pas de ces connaissances préalables.**

CNN est une technique d'apprentissage en profondeur très fondamentale. Nous avons couvert un large éventail de sujets et la section de visualisation est à mon avis la plus intéressante. Il existe très peu de ressources sur le Web qui effectuent une exploration visuelle approfondie des filtres de convolution et des cartes de caractéristiques. J'espère que cela a été utile.

L'intégralité du code de cet article est disponible dans le notebook google-colabe suivant si vous souhaitez le tourner vous-même.

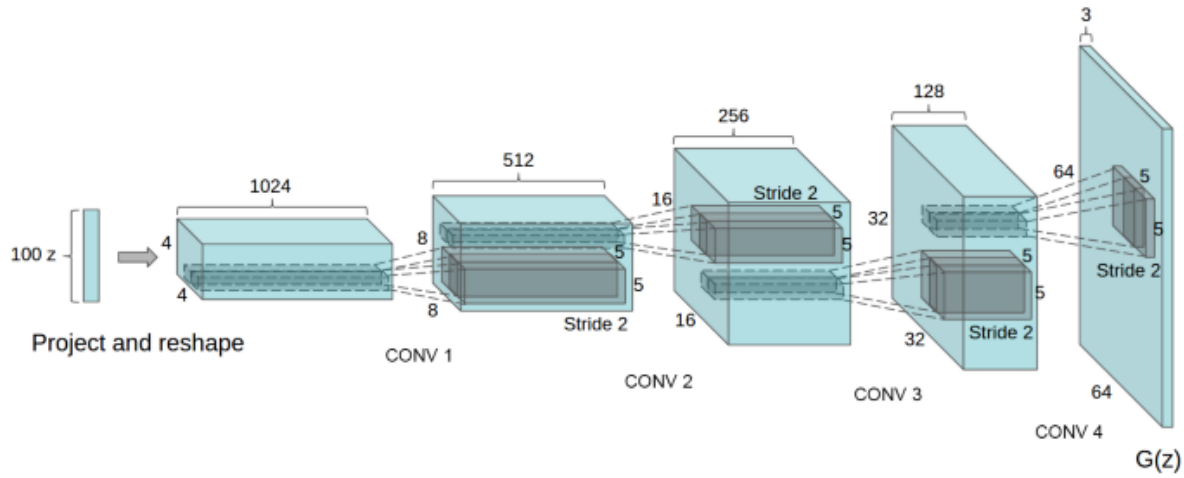
L'intégralité du code de ce **TP 5** est disponible dans le notebook **google-colabe** suivant si vous souhaitez le tourner vous-même.

https://drive.google.com/file/d/1HYt3RWcJvKbwWb1KXWFDQDU9J4DyFxl4/view?usp=sharing

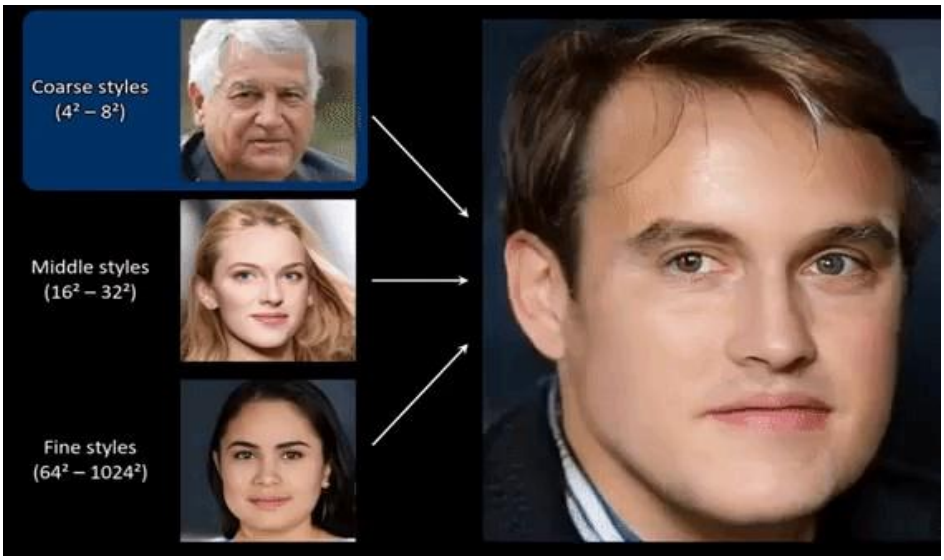
7.TP6 : Réseaux antagonistes génératifs (GAN)

Les GANs sont un type de réseau de neurones utilisé pour l'apprentissage automatique non supervisé. Ils sont composés de deux modules antagonistes : les réseaux générateurs et *coût*. Ces modules se font concurrence de telle sorte que le réseau *coût* tente de filtrer les faux exemples tandis que le *générateur* tente de tromper ce filtre en créant des exemples réalistes. Grâce à cette compétition, le générateur du modèle apprend à créer des données réalistes. Ces données peuvent être utilisées dans des tâches telles que la prédiction future ou pour générer des images spécialisées suite à un entraînement sur un jeu de données particulier.

Quel est donc ce générateur magique G ? Ce qui suit est le DCGAN qui est l'une des conceptions les plus populaires pour le réseau de générateurs. Il effectue plusieurs convolutions transposées pour suréchantillonner z afin de générer l'image x. Nous pouvons le considérer comme le classificateur d'apprentissage en profondeur dans le sens inverse.



L'objectif principal des GAN (Generative Adversarial Networks) est de générer des données à partir de zéro, principalement des images, mais d'autres domaines, y compris la musique, ont été créés. Mais le champ d'application est bien plus vaste que cela. Tout comme l'exemple ci-dessous, il génère un zèbre à partir d'un cheval. Dans l'apprentissage par renforcement, cela aide un robot à apprendre beaucoup plus rapidement.



1. Importation des librairies et de gan_tools_cuda.py

```
[ ]  import torch
     from torch import nn, optim
     from torch.autograd.variable import Variable

     import gan_tools_cuda as gt
```

2. Chargement des données d'apprentissage et création du "loader"

Le loader présentera au réseau les données par paquet de 100, dans un ordre aléatoire.

```
# Load data
data = gt.mnist_data()
# Create loader with data, so that we can iterate over it
data_loader = torch.utils.data.DataLoader(data, batch_size=100, shuffle=True)
# Num batches
num_batches = len(data_loader)
```

3. Création du discriminateur

La classe Python DiscriminatorNet décrivant la structure du réseau discriminateur.Le réseau est composé de trois couches cachées et d'une couche de sortie. Nous utiliserons le module "séquentiel" de Pytorch qui décrit l'ordre des différentes transformations à appliquer aux données dans chaque couche.

L'entrée du réseau est un vecteur avec 784 valeurs (une image 28x28), la sortie est une valeur unique, allant de 0 (fausse image) à 1 (vraie image) grâce à l'utilisation d'une fonction sigmoïde.

Les trois couches cachées utilisent une fonction LeakyReLU pour transformer leur sortie en valeurs quasi positives, avec alpha = 0,2. Elles utilisent également une fonction Dropout pour définir aléatoirement à zéro 30% des valeurs de sortie des neurones des couches cachées (il a été prouvé que cela augmentait les performances du réseau en empêchant le sur-apprentissage). Le nombre de neurones dans chaque couche cachée est respectivement de 1024, 512 et 256.

La classe possède une méthode "forward" permettant de calculer la sortie du réseau à partir d'un vecteur d'entrée (représenaant une image).

1. Il est important d'utiliser `nn.LeakyReLU` comme fonction d'activation pour éviter de tuer les gradients dans les régions négatives. Sans ces gradients, le générateur ne recevra pas de mises à jour.
2. A la fin de la séquence, le discriminateur utilise `nn.Sigmoid()` pour classer l'entrée.

```
class DiscriminatorNet(torch.nn.Module):

    def __init__(self):
        super(DiscriminatorNet, self).__init__()
        n_features = 784
        n_out = 1

        self.hidden0 = nn.Sequential(
            nn.Linear(n_features, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )

        """ AUTRES COUCHES A COMPLETER """

        self.out = nn.Sequential(
            torch.nn.Linear(256, n_out),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.hidden0(x)
        x = self.hidden1(x)
        x = self.hidden2(x)
        x = self.out(x)
        return x
```

4.Création du générateur

Création une classe Python **GeneratorNet** décrivant la structure du réseau générateurs. Le réseau est composé de trois couches cachées et d'une couche de sortie. L'entrée du réseau est un vecteur avec 100 valeurs, la sortie est une image avec 784 valeurs (utilisant nn.Tanh() au lieu de nn.Sigmoid() pour convertir les valeurs de sortie). Les trois couches cachées utilisent une fonction LeakyReLU pour transformer leur sortie en valeurs quasi positives, avec alpha = 0,2, mais

pas de Dropout ici. Le nombre de neurones dans chaque couche cachée est respectivement de 256, 512 et 1024 .

1. Le générateur suréchantillonne l'entrée en utilisant plusieurs modules `nn.ConvTranspose2d` séparés par `nn.BatchNorm2d` et `nn.ReLU`.
2. A la fin de la séquence, le réseau utilise `nn.Tanh()` pour écraser les sorties à $(-1,1)(-1,1)$.
3. Le vecteur aléatoire d'entrée est de taille $\boldsymbol{nz}$. La sortie est de taille $\boldsymbol{nc \times 64 \times 64}$, où $\mathrm{nc}nc$ est le nombre de canaux.

```python
class GeneratorNet(torch.nn.Module):

    def __init__(self):
        super(GeneratorNet, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(100, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, x):
            x = x.view(x.size(0), 100)
            out = self.model(x)
            return out
```

5.Creation du processus d'apprentissage

Initialisez les instances d'un discriminateur et d'un générateur. Créez deux optimiseurs de type Adam pour les deux réseaux. Définissez la fonction de perte : ici, nous choisissons une fonction d'entropie croisée binaire pour vérifier si le discriminateur trouve la bonne réponse.

Chaque époque d'entraînement est divisée en deux étapes.

- **L'étape 1** consiste à mettre à jour le réseau discriminateur. Elle se fait en deux parties. Tout d'abord, on alimente le discriminateur en données réelles provenant des `dataloaders`, on calcule la perte entre la sortie et le `real_label`, puis on accumule des gradients avec la rétropropagation. Deuxièmement, nous alimentons le discriminateur en données générées par le réseau générateur en utilisant le `fixed_noise`, nous calculons la perte entre la sortie et le `fake_label`, et ensuite nous accumulons le gradient. Enfin, nous utilisons les gradients accumulés pour mettre à jour les paramètres du réseau discriminateur.

- **L'étape 2** consiste à mettre à jour le réseau générateur. Cette fois, nous alimentons le discriminateur en fausses données, mais nous calculons la perte avec le `real_label` ! Le but de cette opération est d'entraîner le générateur à faire des x^ réalistes.

```
discriminator = DiscriminatorNet()
generator = GeneratorNet()
if torch.cuda.is_available():
    discriminator.cuda()
    generator.cuda()

# Optimizers
d_optimizer = optim.Adam(discriminator.parameters(), lr=0.0002)
g_optimizer = optim.Adam(generator.parameters(), lr=0.0002)

# Loss function
loss = nn.BCELoss()
```

```
def train_discriminator(optimizer, real_data, fake_data):
    # Reset gradients
    optimizer.zero_grad()

    # 1.1 Train on Real Data
    prediction_real = discriminator(real_data)
    # Calculate error and backpropagate
    error_real = loss(prediction_real, gt.real_data_target(real_data.size(0)))
    error_real.backward()

    # 1.2 Train on Fake Data
    prediction_fake = discriminator(fake_data)
    # Calculate error and backpropagate
    error_fake = loss(prediction_fake, gt.fake_data_target(real_data.size(0)))
    error_fake.backward()

    # 1.3 Update weights with gradients
    optimizer.step()

    # Return error
    return error_real + error_fake, prediction_real, prediction_fake
```

```
def train_generator(optimizer, fake_data):
    # 2. Train Generator
    # Reset gradients
    optimizer.zero_grad()
    # Sample noise and generate fake data
    prediction = discriminator(fake_data)
    # Calculate error and backpropagate
    error = loss(prediction, gt.real_data_target(prediction.size(0)))
    error.backward()
    # Update weights with gradients
    optimizer.step()
    # Return error
    return error
```

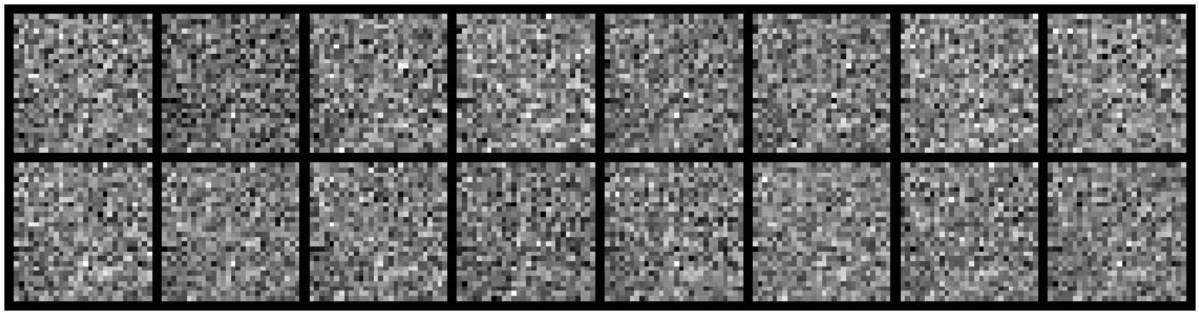
6.Apprentissage et visualisation :

Pour chaque itération et pour chaque lot ("batch") de données, les étapes d'apprentissage sont les suivantes:

- **Transformez le lot de données en variables Torch à l'aide des fonctions Variable et gt.images_to_vectors. Ce sont les vraies données pour l'étape suivante.**
- **Créez des vecteurs de bruit de taille 100 avec gt.noise (autant que de vraies images dans le lot), puis générez de fausses données avec le générateur.**
- **Former le discriminateur sur les fausses données et les données réelles, en utilisant l'optimiseur défini ci-dessus (d_optimizer).**
- **Créez d'autres vecteurs de bruit, taille 100, avec gt.noise.**
- **Entrainez le générateur sur les vecteurs de bruit.**
- **Visualisez un échantillon d'images générées avec gt.plot_gan. Cette fonction nécessite le numéro d'itération et de lot, ainsi que le nombre total de lots et le nom du générateur.**

```
[ ]  for epoch in range(20):
        for n_batch, (real_batch,_) in enumerate(data_loader):

            # 1 Train Discriminator
            # 1.1 Prepare real data
            real_data = Variable(gt.images_to_vectors(real_batch))
            if torch.cuda.is_available(): real_data = real_data.cuda()

            # 1.2 Generate fake data with the Generator
            """ A COMPLETER : Créer des vecteurs bruit """
            noise = gt.noise(real_data.size(0))

            """ A COMPLETER : Générer des images avec le générateur """
            fake_data = gt.images_to_vectors(generator(noise))
            # 1.3 Train Discriminator
            """ A COMPLETER : Entrainer le discriminateur """
            train_Disc = train_discriminator(d_optimizer,real_data,fake_data)
            # 2 Train Generator
            # 2.1 Generate noise
            """  A COMPLETER : Créer des vecteurs bruit à nouveau """
            noise = gt.noise(real_data.size(0))
            fake_data = gt.images_to_vectors(generator(noise))
            # 2.2 Train Generator
            """  A COMPLETER : Entrainer le generateur """
            train_gener =train_generator(g_optimizer,fake_data)
            # Generate images from a fixed noise input and visualize the output
            gt.plot_gan(epoch, n_batch, num_batches, generator)
```

Epoch: 0, Batch Num: [0/600]



Epoch: 19, Batch Num: [500/600]

7.Exploration et optimisation

Modifiez la structure des deux réseaux (discriminateur et générateur) en ajoutant ou enlevant des couches cachées, en modifiant le nombre de neurones dans ces couches et en jouant sur les paramètres des fonctions LeakyReLU et Dropout.

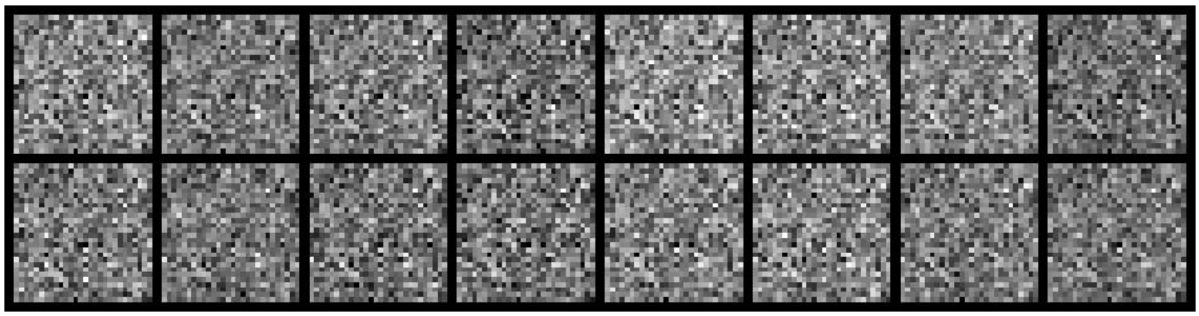
Après avoir expliqué comment évaluer la qualité des images générées par le générateur, comparez la qualité des résultats obtenus avec les modifications que vous avez testé et essayez d'obtenir le meilleur résultat possible. Une couche caché 128 … 1024 , puis dropout 0.5 et LeakyReLU 0.3 sont ajoutés.

```python
class DiscriminatorNet1(torch.nn.Module):

    def __init__(self):
        super(DiscriminatorNet1, self).__init__()
        n_features = 784
        n_out = 1
        self.hidden0 = nn.Sequential(
            nn.Linear(n_features, 1024),
            nn.LeakyReLU(0.3,inplace=True),
            nn.Dropout(0.5)
        )
        self.hidden1 = nn.Sequential(
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.3,inplace=True),
            nn.Dropout(0.5)
        )
        self.hidden2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.LeakyReLU(0.3,inplace=True),
            nn.Dropout(0.5)
        )
        self.hidden3 = nn.Sequential(
            nn.Linear(256, 128),
            nn.LeakyReLU(0.3,inplace=True),
            nn.Dropout(0.5)
        )
        self.out = nn.Sequential(
            torch.nn.Linear(128, n_out),
            torch.nn.Sigmoid()
        )
    def forward(self, x):
        x = self.hidden0(x)
        x = self.hidden1(x)
        x = self.hidden2(x)
        x = self.hidden3(x)
        x = self.out(x)
        return x
```

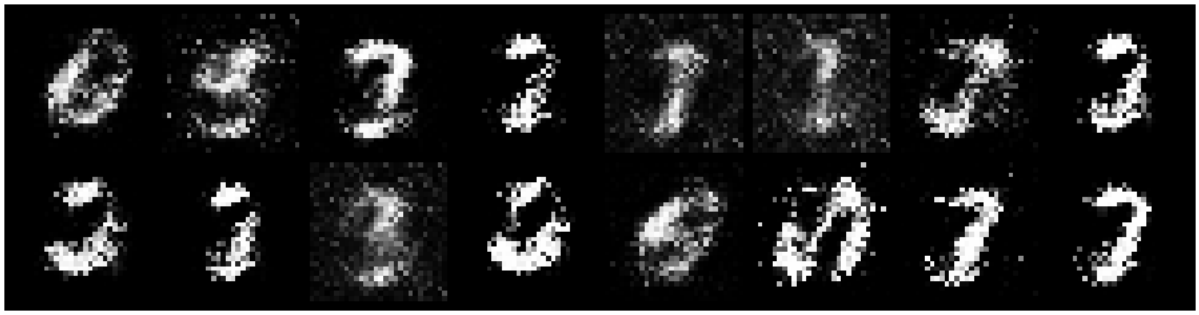
```python
[ ] class GeneratorNet1(torch.nn.Module):

        def __init__(self):
            super(GeneratorNet1, self).__init__()
            self.model = nn.Sequential(
                nn.Linear(100, 128),
                nn.LeakyReLU(0.2),
                nn.Linear(128, 256),
                nn.LeakyReLU(0.2),
                nn.Linear(256, 512),
                nn.LeakyReLU(0.2),
                nn.Linear(512, 1024),
                nn.LeakyReLU(0.2),
                nn.Linear(1024, 784),
                nn.Tanh()
            )

        def forward(self, x):
                x = x.view(x.size(0), 100)
                out = self.model(x)
                return out
```

```python
discriminator1 = DiscriminatorNet1()
generator1 = GeneratorNet1()
if torch.cuda.is_available():
    discriminator.cuda()
    generator.cuda()

# Optimizers
d_optimizer = optim.Adam(discriminator1.parameters(), lr=0.0002)
g_optimizer = optim.Adam(generator1.parameters(), lr=0.0002)

# Loss function
loss = nn.BCELoss()

def train_discriminator1(optimizer, real_data, fake_data):
    # Reset gradients
    optimizer.zero_grad()

    # 1.1 Train on Real Data
    prediction_real = discriminator1(real_data)
    # Calculate error and backpropagate
    error_real = loss(prediction_real, gt.real_data_target(real_data.size(0)))
    error_real.backward()

    # 1.2 Train on Fake Data
    prediction_fake = discriminator1(fake_data)
    # Calculate error and backpropagate
    error_fake = loss(prediction_fake, gt.fake_data_target(real_data.size(0)))
    error_fake.backward()

    # 1.3 Update weights with gradients
    optimizer.step()

    # Return error
    return error_real + error_fake, prediction_real, prediction_fake
```

```
Epoch: 0, Batch Num: [0/600]
```



```
Epoch: 19, Batch Num: [500/600]
```



Les résultats de ce réseau ne sont pas trop satisfaisants par contre le premier réseau a généré des images prête des chiffre manuscrites lisibles.

En général, le concept de génération de données nous conduit à un grand potentiel mais malheureusement à de grands dangers. Il existe de nombreux autres modèles génératifs en plus du GAN. Par exemple, le GPT-2 d'OpenAI génère des paragraphes qui peuvent ressembler à ce qu'écrit un journaliste. En effet, OpenAI décide de ne pas ouvrir son jeu de données et son modèle entraîné en raison de son utilisation abusive possible.

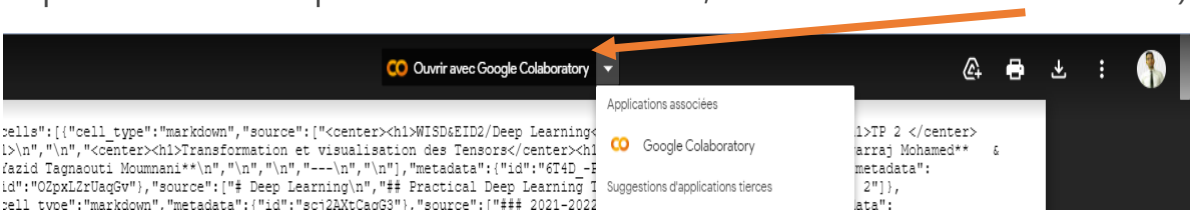
L'intégralité du code de TP **6** est disponible dans le notebook **google-colabe** suivant si vous souhaitez le tourner vous-même.

https://drive.google.com/file/d/1zxVWDX8Lso2oVIiyS_34qbhFhoLJsvR-/view?usp=sharing

8.Conclusion

Dans ce rapport nous avons cité tout ce que nous avons fait pour répondre aux travaux pratiques demandés pour le module de Deep Learning, sous la direction de monsieur Guénaël Cabanes, en exploitant ce que nous avons vu dans le cours avec monsieur Younès Bennani.

Vous trouverez le code python de chaque partie dans les fichiers **.pynb** joints avec ce rapport, ou vous pouvez les consulter à partir des liens vers les Notebooks google-colab suivants (les notebook sont créés avec **jupyter** et importés vers **colab** pour accélérer l'exécution,il faut choisir ouvrir avec colab) :



TP1:

https://drive.google.com/file/d/1zpbRsMubsz2J5A4gsHzewWBHAfRCw6c5/view?usp=sharing

TP2:

https://drive.google.com/file/d/1rhkGogoKJjKkFOAb5eM5hHv-T-c-FvMo/view?usp=sharing

TP3:

https://drive.google.com/file/d/1y9IFCJYIN4uN0Tz27VVYa9x22Yns_Kgm/view?usp=sharing

TP4:

https://drive.google.com/file/d/18Tbx0PwcWrgh54_szrEktcj1-bIRd9Zh/view?usp=sharing

TP5:

ttps://drive.google.com/file/d/1HYt3RWcJvKbwWb1KXWFDQDU9J4DyFxl4/view?usp=sharing

TP6:

https://drive.google.com/file/d/1zxVWDX8Lso2oVIiyS_34qbhFhoLJsvR-/view?usp=sharing

9.Références

D’après Yann Le Cun, le contenue est destiné à des personnes de niveau bac+4 ou bac+5.

C’est un cours porte sur les techniques de représentation et d’apprentissage profond les plus récentes. Il se concentre sur l’apprentissage supervisé, non supervisé et auto supervisé, mais aussi sur les méthodes d’enchâssement, l’apprentissage métrique et les réseaux convolutifs et récurrents. Il est illustré d’applications à la vision par ordinateur, la compréhension du langage naturel et la reconnaissance vocale. Il contient des explications utiles pour répondre aux travails demandés dans la série des TPs.

https://atcold.github.io/pytorch-Deep-Learning/fr/faq/