Université de Paris

Master MLDS: Machine Learning for Data Science

Projet Apprentissage Profond

Prédiction des données MNIST avec uniquement 100 labels

Avec un algorithme d'apprentissage semi-supervisé basé sur les pseudo-labels

Réalisé par :

Arab Asma

Boulfoul Rayane

Et-tali Mouad

Promotion : 2021-2022

Table des matières

I. Introduction

Ces dernières années, le Deep Learning a eu un impact considérable sur de nombreux domaines, de la médecine à la sécurité en passant par la reconnaissance de caractères et la compression d'images. Les domaines d'utilisation des réseaux de neurones s'élargissent car les performances des algorithmes sont nettement supérieures à celles de nombreuses autres méthodes et techniques **[2]**.

L'apprentissage profond est un ensemble de méthodes d'apprentissage qui tentent de modéliser les données avec des architectures complexes combinant différentes transformations non linéaires. Les briques élémentaires du Deep Learning sont les neurones, qui sont combinés pour former les réseaux de neurones profonds **[3]**.

L'objectif principal de ce projet est de mettre en place un réseau de neurones capable d'effectuer une prédiction des données MNIST avec un dataset d'apprentissage composé d'uniquement 100 images labelisés, en se basant sur la méthode d'apprentissage semi-supervisé proposée dans l'article scientifique : **"Pseudo-label : The simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks" [1]**. Ce projet a été réalisé en utilisant la même approche que celle proposée dans l'article. À savoir, les pseudo-labels, qui sont considérées comme les vraies labels dans le processus d'apprentissage, pour obtenir la prédiction maximale et effectuer la classification des données non étiquetées. Cette méthode a été considérée comme un préalable à la technique d'apprentissage semi-supervisé puisque l'algorithme utilise simultanément les données étiquetées et non étiquetées afin d'entraîner le réseau neuronal et permet d'obtenir une plus grande précision lors de la phase de test.

Dans ce rapport, nous allons d'abord expliquer les principales idées et méthodes proposés dans l'article, par la suite nous exposerons l'algorithme mis en place. Enfin nous présenterons les résultats obtenus avec notre implémentation et on les comparera avec un réseau de neurones classique (baseline). Nous inclurons le code commenté de notre implémentation en Annexe.

II. La méthode proposée par l'article :

L'article propose une méthode simple et efficace pour faire un apprentissage semi-supervisé. Elle consiste à entrainer le réseau de neurones de manière supervisé en utilisant simultanément les données labélisées et non labélisées, ces derniers auront des pseudo-labels (le label sera celui de la classe ayant la plus grande probabilité lors de la prédiction) qui seront considérés comment vrai. Cette technique peut être appliquée avec différentes architectures de réseaux de neurones.

1. Les réseaux de neurones profonds :

Les réseaux de neurones considérés sont les réseaux multicouches avec M couches. La fonction d'activation pour les **hidden layers** est la fonction ReLU (Rectified Linear Unit), et celle de la **output layer** est la fonction Sigmoid,. La fonction de cout utilisée est la Cross Entropy.

2. Dropout :

Cette technique est appliquée lors l'apprentissage des réseaux de neurones, elle consiste à ignorer un certain nombre de neurones avec une probabilité 1-p, et ceci en omettant ces neurones lors du forward ou du backward pass **[5]**. L'utilisation du dropout permet de réduire l'overfitting. L'article propose d'utiliser une probabilité de dropout égale à 0.5 pour les neurones cachés et un dropout de 0.2 pour les neurones visibles du réseau multicouches. Pour le bon fonctionnement de l'algorithme SGD (stochastic gradient descent) avec le dropout, les auteurs du papier proposent d'utiliser un pas dégressif d'apprentissage et un momentum.

3. Denoising Auto-Encoder (DAE) :

Le DAE est un réseau de neurone utilisé pour faire de l'apprentissage non supervisé. Ce type d'auto-encodeur permet de rendre l'apprentissage plus robuste à la corruption des données, et ceci en rajoutant volontairement du bruits aux entrées. Le réseaux a comme entrées les données bruitées, qu'il essaye d'approximer aux données originales en sortie. Ce type de réseau est aussi utilisé pour initialiser d'autres réseaux de neurones. Pour une tâche de classification par exemple, la couche du décodeur est omise après l'apprentissage, puis remplacé par une couche de sortie dédiée à la classification.

Dans la méthode proposée dans le papier, le DAE a été utilisé pour initialiser le réseau de neurones multicouches, dans une étape appelé « unsupervised pre-training ». Les données

ont été bruitées avec du bruit gaussien avec une probabilité de 0.5, un dropout a été appliqué aux couches cachées avec une probabilité égale à 0 .5, l'algorithme d'optimisation utilisé pour l'optimisation est le momentum avec un pas dégressif d'apprentissage.

4. Pseudo-Label

Ce sont les classes cibles pour les données non labélisées considéré comme les vrais labels, ils sont choisis selon la probabilité maximale prédite pour chaque échantillon non labelisé. Les pseudo-label sont utilisé dans l'étape « Fine-Tuning » avec le Dropout.

Comme les données labélisées sont en nombre réduit comparé aux données labélisées, il est nécessaire d'équilibrer entre eux pour obtenir un réseau de neurones avec de bonne performance. Pour cela on utilise la fonction de cout suivante lors du fine-tuning :

$$L = \frac{1}{n}\sum_{m=1}^{n}\sum_{i=1}^{C} L(y_i^m, f_i^m) + \alpha(t)\frac{1}{n'}\sum_{m=1}^{n'}\sum_{i=1}^{C} L(y_i'^m, f_i'^m),$$

- *n* est le nombre de mini-batch des données labelisés pour le SGD et *n'* pour les données non labélisées,
- f_i^m c'est les neurones output des m instances des données labélisées, y_i^m est le label
- f'^m_i c'est pour les données non labélisées, y'^m_i représentent les pseudo-labels
- $\alpha(t)$ est le coefficient permettant de balancer entre les labels et les pseudos labels sa formule est comme suit :

$$\alpha(t) = \begin{cases} 0 & t < T_1 \\ \frac{t-T_1}{T_2-T_1}\alpha_f & T_1 \le t < T_2 \\ \alpha_f & T_2 \le t \end{cases}$$

Avec $\alpha_f = 3$. La valeur de T_1 et de T_2 dépend si on fait ou non un pre-training, dans le cas où on l'effectue alors $T_1 = 200$ et $T_2 = 800$ sinon $T_1 = 100$ et $T_2 = 600$.

III. Pseudo algorithme :

L'algorithme que nous proposons s'appuie sur l'ensemble des concepts vu précédemment, il est comme suit :

1. **Initialisation du réseau de neurones (Pré-entrainement non supervisé) :** on initialise l'ensemble des poids et des biais en effectuant un pré-entrainement non supervisé

avec le DAE sur toutes les données de l'ensemble d'entrainement, un dropout est appliqué aux couche cachées avec un pourcentage de 50 %.

2. **Entrainement avec les données labélisées :** L'étape précédente nous a permis d'obtenir un réseau de neurones initialisé, nous l'entraînons par la suite d'une manière supervisée avec 100 images labelisé. Nous utilisons le dropout avec une probabilité de 0.5 pour les nœuds cachés et 0.2 pour les nœuds visibles. La fonction de coût utilisée est softmax cross entropy.
3. **Prédiction pour les données non labélisées :** L'objectif de cette étape est l'obtention des pseudo-labels, et ceci en faisant passer l'ensemble des images non labelisées via le réseau de neurones entrainé dans l'étape précédente.
4. **Fine-tuning avec les pseudos labels :** on crée un nouveau dataset composé des 100 images labélisées et des images avec des pseudo-labels pour entrainer à nouveau le réseau de neurones. A cette étape nous utilisons la fonction de coût qui permet d'atteindre l'équilibre entre label et pseudo-label.

IV. Les résultats :

Nos expérimentations ont couvert plusieurs scénarios, ce qui va nous permettre de mieux faire les comparaisons :

- **NN** : le Baseline qui est un réseau de neurone classique, entrainé avec 100 images labelisées.
- **DropNN** : C'est le réseau NN avec lequel on a rajouté le dropout, entrainé avec 100 images labelisées.
- **DropNN+PL** : C'est le réseau DropNN entrainé avec 100 images labelisé et les pseudos labels des images non labelisées.
- **DropNN+PL+DAE** : c'est le réseau DropNN+PL initialisé avec le DAE, entrainé avec 100 images labelisé et les pseudos labels des images non labelisées.

	Résultat de l'article	**Résultat de notre méthode**
NN	74,19	70,03
DropNN	78,11	71,20
DropNN + PL	83,85	72,28
DropNN + PL + DAE	89,51	76,95

V. Conclusion :

L'objectif principal de ce projet est de réaliser une prédiction des images avec uniquement 100 images labelisés et la méthode implémentée est inspiré de l'article scientifique.

Nous pouvons constater à partir des résultats de l'expérimentation que :

1. En rajoutant de dropout au NN Baseline on obtient de meilleures résultats pour le DropNN.
2. L'utilisation des pseudo label lors du fine-tuning du DropNN+PL permet d'améliorer les résultats par rapport au DropNN.
3. L'initialisation avec DAE lors de la phase d'apprentissage non supervisé permet aussi d'améliorer les résultats de la prédiction.

On peut alors dire que la méthode d'apprentissage semi supervisé proposée dans le papier et l'ensemble des techniques utilisées sont efficace pour faire la prédiction sur un dataset avec un petit nombre de données labelisées.

VI. Références :

[1] Lee,D.H. Pseudo-Label: The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks p. (1-6)
[2]https://cs.stanford.edu/people/eroberts/courses/soco/projects/neuralnetworks/Applications/index.htm

[3] https://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-m-hdstat-rnn-deep-learning.pdf

[4] https://towardsdatascience.com/denoising-autoencoders-explained-dbb82467fc2

[5] https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

VII. Annexe :

```
#Les packages
import numpy as np
import matplotlib.pyplot as plt
import random
import keras
from keras.models import Sequential
from keras.layers import Dense
from tensorflow import keras
import tensorflow as tf
from keras.layers import Dropout
from keras.constraints import maxnorm
from keras.models import Model
from tensorflow.keras.datasets import mnist
from sklearn.utils import shuffle
```

Importation du dataset et pre-processiong

```
#Importer les données
(x_train, y_train), (x_test, y_test) = mnist.load_data()

#Normalisation
x_train, x_test =  x_train/255, x_test/255

#On doit changer le format des images en une couche entiérement
connctée ie. un 1D array(28*28)
num_pixels = 784
x_train = x_train.reshape(x_train.shape[0], num_pixels)
x_test = x_test.reshape(x_test.shape[0],num_pixels)

#Permet de créer le dataset de 100 images avec 10 images de chaque
classes
#La fonction retourne un tuple de 2 listes : les images et leurs
classes
def split_by_label(x_train, y_train, num_per_label):
    # pick out the same size label from data set
    counter = np.zeros(10)  # for 10 classes
    new_dataset_x = []
    new_dataset_y = []
    for x, y in zip(x_train, y_train):
        #x, y = i
        if type(y) == np.ndarray:
            y = np.argmax(y)
        if y == 0 and counter[0] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
            counter[0] += 1
            continue
        if y == 1 and counter[1] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
```

```
            counter[1] += 1
            continue
        if y == 2 and counter[2] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
            counter[2] += 1
            continue
        if y == 3 and counter[3] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
            counter[3] += 1
            continue
        if y == 4 and counter[4] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
            counter[4] += 1
            continue
        if y == 5 and counter[5] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
            counter[5] += 1
            continue

        if y == 6 and counter[6] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
            counter[6] += 1
            continue

        if y == 7 and counter[7] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
            counter[7] += 1
            continue

        if y == 8 and counter[8] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
            counter[8] += 1

            continue

        if y == 9 and counter[9] < num_per_label:
            new_dataset_x.append(x)
            new_dataset_y.append(y)
            counter[9] += 1
            continue

    print(counter)
    return new_dataset_x, new_dataset_y
```

```
 #training_small_data contient les 100 images et leur labels
 num_per_label = 10
 training_data_small = split_by_label(x_train, y_train, num_per_label)
```

```
[10. 10. 10. 10. 10. 10. 10. 10. 10. 10.]
```

```
#Récupérer les images et leurs labels
x_train_small = training_data_small[0]
y_train_small = training_data_small[1]
#Les transformer en des numpy array
x_train_small_array = np.array(x_train_small)
y_train_small_array = np.array(y_train_small)

#Mélanger les données pour éviter que l'ordre est un impacte sur
l'entrainement
tf.random.set_seed(42)
x_train_small_array, y_train_small_array =
shuffle(x_train_small_array, y_train_small_array)
```

Baseline : Réseau de neurones pur

```
# réseau de neurone avec une seule couche cachée de 5000 neurones
model_NN = Sequential()
model_NN.add(Dense(5000, input_dim=784, activation='relu'))
model_NN.add(Dense(10, activation='sigmoid'))

model_NN.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_20 (Dense)            (None, 5000)              3925000

 dense_21 (Dense)            (None, 10)                50010

=================================================================
Total params: 3,975,010
Trainable params: 3,975,010
Non-trainable params: 0
_________________________________________________________________
```

```
#configuration du modèle
model_NN.compile(optimizer= 'adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

#les batchs size utilisé pour l'apprentissage
batchSize = 32
```

```
#apprentissage du modèle avec les 100 images labélisés uniquement
history = model_NN.fit(x_train_small_array, y_train_small_array,
epochs= 20, batch_size= batchSize)

#Evaluation du modèle pour connaitre son accuracy
model_NN.evaluate(x_test,y_test)
```

```
313/313 [==============================] - 5s 14ms/step - loss: 1.2794
- accuracy: 0.7031

[1.279363989830017, 0.7031000256538391]
```

DropNN

```
del model_dropNN

del history_dropNN
del lr_schedule

# DropNN : c'est un réseau de neurones avec un Dropout de 0.5 pour les
layers cachés et 0.2 pour layer des inputs
model_dropNN = Sequential()
model_dropNN.add(Dropout(0.2, input_shape=(784,)))
model_dropNN.add(Dense(5000,  activation='relu'))
model_dropNN.add(Dropout(0.5))
model_dropNN.add(Dense(10, activation='sigmoid'))

#configuration du optimizer
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate= 1.5,
    decay_steps= 10,
    decay_rate=0.88)
optimizer = keras.optimizers.SGD(learning_rate=lr_schedule)

#configuration du modèle DropNN
model_dropNN.compile(optimizer= optimizer,
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

#Apprentissage avec les 100 images uniquement
history_dropNN = model_dropNN.fit(x_train_small_array,
y_train_small_array, epochs= 500, batch_size= batchSize)

#Evaluation des performances du modèle
model_dropNN.evaluate(x_test,y_test)
```

```
313/313 [==============================] - 4s 14ms/step - loss: 5.0459
- accuracy: 0.7120

[5.045924663543701, 0.7120000123977661]
```

+PL

```
#Prédiction pour obtenir les pseudo-labels pour les données
d'entrainement non labelisées avec le drop_NN
pred = model_dropNN.predict(x_train)
y_pseudo_label = list()
for i in range(len(pred)):
    y_pseudo_label.append(np.argmax(pred[i]))

# Instantier optimizer.
optimizer_PL = keras.optimizers.Adam()
#lr_schedule_PL = keras.optimizers.schedules.ExponentialDecay(
#    initial_learning_rate= 1.5,
#    decay_steps= 10,
#    decay_rate=0.88)

# Instantier la fonction de coût.
loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
#from_logits=True

# Preparation des deux datasets d'apprentissages : 100 images
labelisés (train_dataset_label) et les images avec pseudoLabels
(train_dataset_PL)
batch_size_label = 32
batch_size_PL = 256
train_dataset_label =
tf.data.Dataset.from_tensor_slices((x_train_small_array,
y_train_small_array))
train_dataset_label =
train_dataset_label.shuffle(buffer_size=1024).batch(batch_size_label)
train_dataset_PL = tf.data.Dataset.from_tensor_slices((x_train,
y_pseudo_label))
train_dataset_PL =
train_dataset_PL.shuffle(buffer_size=1024).batch(batch_size_PL)

#cette fonction calcule loss value pour chaque batch
@tf.function
def train_step(x_label, y_label, x_PL, y_PL, epoch):
    alpha = 0.
    T1 = 100
    T2 = 600
    af = 3.
    #Ouvre un GradientTape pour enregistrer les opérations effectués
lors, ce qui permet l'auto-differentiation
    with tf.GradientTape() as tape:
            # Exécuter le forward pass de la couche
            # Les opération que la couche effectue à ses inputs
            # vont être enregistré dans le GradientTape

      logits_label = model_dropNN(x_label, training=True)
      logits_PL = model_dropNN(x_PL, training=True)
```

```
      if epoch > T1:
        alpha = ((epoch - T1) / (T2 - T1)) * af
        if epoch > T2:
          alpha = af
       # Calculer la loss value pour le minibatch.
      loss_value = loss_fn(y_label, logits_label) + alpha *
loss_fn(y_PL, logits_PL)
    # Utiliser le gradient tape pour récupéer automatiquement
    # les gradients des variables entraînables par rapport à la perte.
    grads = tape.gradient(loss_value, model_dropNN.trainable_weights)
    # Exécuter une étape de descente de gradient en mettant
    # à jour la valeur des variables pour minimiser la perte.
    optimizer_PL.apply_gradients(zip(grads,
model_dropNN.trainable_weights))
    return loss_value

#entrainement du drop NN avec PL
import time

epochs = 500
for epoch in range(epochs):
    print("\nStart of epoch %d" % (epoch,))
    start_time = time.time()
    step = 1


    # Iterer sur l'ensemble des batchs.
    for   (x_batch_label, y_batch_label), (x_batch_PL, y_batch_PL)  in
zip(train_dataset_label, train_dataset_PL):
        loss_value = train_step(x_batch_label, y_batch_label,
x_batch_PL, y_batch_PL, epoch)


    print("Time taken: %.2fs" % (time.time() - start_time))

#Evaluation
model_dropNN.evaluate(x_test,y_test)
```

```
313/313 [==============================] - 4s 13ms/step - loss: 1.1593
- accuracy: 0.7228

[1.1593446731567383, 0.7228000164031982]
```

+PL+DAE

Unsupervised pre-training: Denoising Auto-Encoder (DAE)

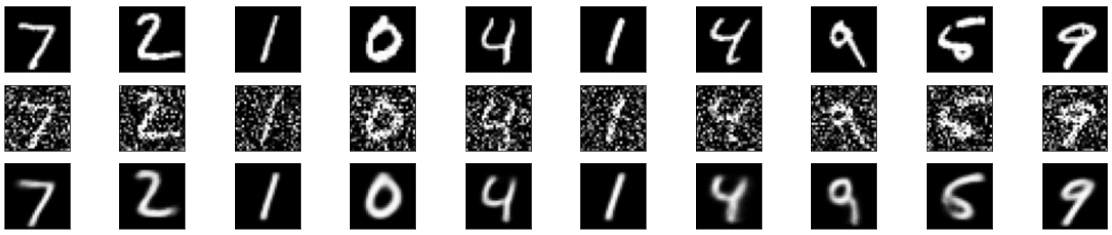
```
#Ajouter du bruit gaussien aux images avec une proba de 0.5
noise_proba=0.5
x_train_noisy = x_train + noise_proba * np.random.normal(loc=0.0,
```

```
scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_proba * np.random.normal(loc=0.0,
scale=1.0, size=x_test.shape)

#Limiter les valeurs des pixels entre 0 et 1
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)

#DAE avec Dropout
model_DAE = Sequential()
model_DAE.add(Dense(5000, input_dim=784, activation='relu'))
model_DAE.add(Dropout(0.5))
model_DAE.add(Dense(128, activation='relu'))
model_DAE.add(Dense(64, activation='relu'))
model_DAE.add(Dense(32, activation='relu'))
model_DAE.add(Dense(64, activation='relu'))
model_DAE.add(Dense(128, activation='relu'))
model_DAE.add(Dense(5000, activation='relu'))
model_DAE.add(Dropout(0.5))
model_DAE.add(Dense(784, activation='sigmoid'))

model_DAE.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 5000)              3925000

 dropout (Dropout)           (None, 5000)              0

 dense_1 (Dense)             (None, 128)               640128

 dense_2 (Dense)             (None, 64)                8256

 dense_3 (Dense)             (None, 32)                2080

 dense_4 (Dense)             (None, 64)                2112

 dense_5 (Dense)             (None, 128)               8320

 dense_6 (Dense)             (None, 5000)              645000

 dropout_1 (Dropout)         (None, 5000)              0

 dense_7 (Dense)             (None, 784)               3920784

=================================================================
Total params: 9,151,680
Trainable params: 9,151,680
```

```
Non-trainable params: 0
_________________________________________________________________
```

```
model_DAE.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

model_DAE.fit(x_train_noisy, x_train,
                epochs=10,
                batch_size=32,
                shuffle=True,
                validation_data=(x_test_noisy, x_test))

predicted = model_DAE.predict(x_test_noisy)

plt.figure(figsize=(40, 4))
for i in range(10):
    # Afficher les images originales
    ax = plt.subplot(3, 20, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Afficher les images bruitées
    ax = plt.subplot(3, 20, i + 1 + 20)
    plt.imshow(x_test_noisy[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Afficher les images reconstruites
    ax = plt.subplot(3, 20, 2*20 +i+ 1)
    plt.imshow(predicted[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)


plt.show()
```



```
#Enregistrer le DAE en format h5
model_DAE.save("DAE.h5")
```

DropNN initialisé avec DAE

```
del pretrained_DAE

#Recharger le DAE pré-entrainé
pretrained_DAE = tf.keras.models.load_model("DAE.h5")
pretrained_DAE.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 5000)              3925000

 dropout (Dropout)           (None, 5000)              0

 dense_1 (Dense)             (None, 128)               640128

 dense_2 (Dense)             (None, 64)                8256

 dense_3 (Dense)             (None, 32)                2080

 dense_4 (Dense)             (None, 64)                2112

 dense_5 (Dense)             (None, 128)               8320

 dense_6 (Dense)             (None, 5000)              645000

 dropout_1 (Dropout)         (None, 5000)              0

 dense_7 (Dense)             (None, 784)               3920784

=================================================================
Total params: 9,151,680
Trainable params: 9,151,680
Non-trainable params: 0
_________________________________________________________________
```

```
#Supprimer les couches pour garder qu'une seule couche caché de 5000
neurones
x = pretrained_DAE.layers[-9].output

#Rajouter une couche de classification avec 10 neurones et une
fonction d'activation sigmoid
predictions = Dense(10, activation='sigmoid')(x)
model_dropNN_DAE = Model(inputs=pretrained_DAE.input,
outputs=predictions)
model_dropNN_DAE.summary()
```

```
Model: "model_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 dense_input (InputLayer)    [(None, 784)]             0

 dense (Dense)               (None, 5000)              3925000

 dropout (Dropout)           (None, 5000)              0

 dense_19 (Dense)            (None, 10)                50010

=================================================================
Total params: 3,975,010
Trainable params: 3,975,010
Non-trainable params: 0
_________________________________________________________________
```

```
model_dropNN_DAE.compile(optimizer= 'adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
model_dropNN_DAE.fit(x_train_small_array, y_train_small_array, epochs=
100, batch_size= 32)
```

```
model_dropNN_DAE.evaluate(x_test,y_test)
```

```
313/313 [==============================] - 5s 14ms/step - loss: 1.1211
- accuracy: 0.7559

[1.1211119890213013, 0.7559000253677368]
```

+PL+DAE

```
#Prédire les pseudo-labels pour les données non labelisées avec le
drop_NN initialisé avec DAE
pred = model_dropNN_DAE.predict(x_train)
y_pseudo_label = list()
for i in range(len(pred)):
    y_pseudo_label.append(np.argmax(pred[i]))

# Instantier optimizer.
optimizer_PL = keras.optimizers.Adam() #learning_rate=0.01
# Instantier la fonction de coût
loss_fn = keras.losses.SparseCategoricalCrossentropy()
#from_logits=True


batch_size_label = 32
batch_size_PL = 256
# Preparation des deux datasets d'apprentissages : 100 images
labelisés (train_dataset_label) et les images avec pseudoLabels
(train_dataset_PL)
train_dataset_label =
tf.data.Dataset.from_tensor_slices((x_train_small_array,
y_train_small_array))
```

```
train_dataset_label =
train_dataset_label.shuffle(buffer_size=1024).batch(batch_size_label)
train_dataset_PL = tf.data.Dataset.from_tensor_slices((x_train,
y_pseudo_label))
train_dataset_PL =
train_dataset_PL.shuffle(buffer_size=1024).batch(batch_size_PL)

@tf.function
def train_step(x_label, y_label, x_PL, y_PL, epoch):
    alpha = 0.
    T1 = 100
    T2 = 600
    af = 3.
    #Ouvre un GradientTape pour enregistrer les opérations effectués
lors, ce qui permet l'auto-differentiation
    with tf.GradientTape() as tape:
      # Exécuter le forward pass de la couche
      # Les opération que la couche effectue à ses inputs
      # vont être enregistré dans le GradientTape
      logits_label = model_dropNN_DAE(x_label, training=True)
      logits_PL = model_dropNN_DAE(x_PL, training=True)
      if epoch > T1:
        alpha = ((epoch - T1) / (T2 - T1)) * af
        if epoch > T2:
          alpha = af
      # Calculer la loss value pour le minibatch.
      loss_value = loss_fn(y_label, logits_label) + alpha *
loss_fn(y_PL, logits_PL)
    # Utiliser le gradient tape pour récupéer automatiquement
    # les gradients des variables entraînables par rapport à la perte.
    grads = tape.gradient(loss_value,
model_dropNN_DAE.trainable_weights)

    # Exécuter une étape de descente de gradient en mettant
    # à jour la valeur des variables pour minimiser la perte.
    optimizer_PL.apply_gradients(zip(grads,
model_dropNN_DAE.trainable_weights))
    return loss_value

#entrainement du drop NN DAE avec PL
import time

epochs = 500
for epoch in range(epochs):
    print("\nStart of epoch %d" % (epoch,))
    start_time = time.time()
    step = 1


    # Iterer sur l'ensemble des batchs.
```

```
    for   (x_batch_label, y_batch_label), (x_batch_PL, y_batch_PL)  in
zip(train_dataset_label, train_dataset_PL):
        loss_value = train_step(x_batch_label, y_batch_label,
x_batch_PL, y_batch_PL, epoch)



    print("Time taken: %.2fs" % (time.time() - start_time))

model_dropNN_DAE.evaluate(x_test,y_test)

313/313 [==============================] - 5s 15ms/step - loss: 4.4753
- accuracy: 0.7695

[4.475318908691406, 0.7695000171661377]
```