Deep Learning Project

Pseudo-Label: The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks

O. Guedj, T. Marcoux Pépin, HD. Nguyen

Contents

List of Figures

1 Introduction

Le but de ce projet est de comprendre et de reproduire les résultats de l'article de Dong-Hyun Lee "Pseudo-Label: The Simple and Efficient Semi-Supervised Learning Method for DeepNeural Networks" publié en juillet 2013.

Dans ce rapport nous commençons par détailler les méthodes utilisées puis nous présentons les résultats que nous avons pu obtenir sur un échantillon de 100 observations de la base de données de chiffres manuscrits MNIST.

Principe général de la méthode: pseudo-labelling

Le pseudo-labelling est une méthode dite "semi-supervisée" car elle combine l'utilisation de données "labellisées" et de données "non labelisées".

L'idée est d'entraîner un réseau de neurones grâce aux données dont on connaît le label puis de prédire le label des données non-labélisées grâce à ce réseau de neurones.

Les labels obtenus sont appelés "pseudo-labels". Le réseau de neurones est alors ré-entraîné mais cette fois sur toutes les données puisqu'à présent elles sont toutes labelisées.

Avec la méthode présentée dans cet article, le training sur les données labellisées et la labelisation des données non labellées se fait en même temps. C'est à dire à chaque mini batch le réseau ajuste ses poids sur les données labellées puis prédit les données non labellées et ensuite la fonction de coût est calculée: elle intègre la perte occasionnée par les données labellées et également celle occasionnée par les données non labellées. Enfin on applique la backprobagation.

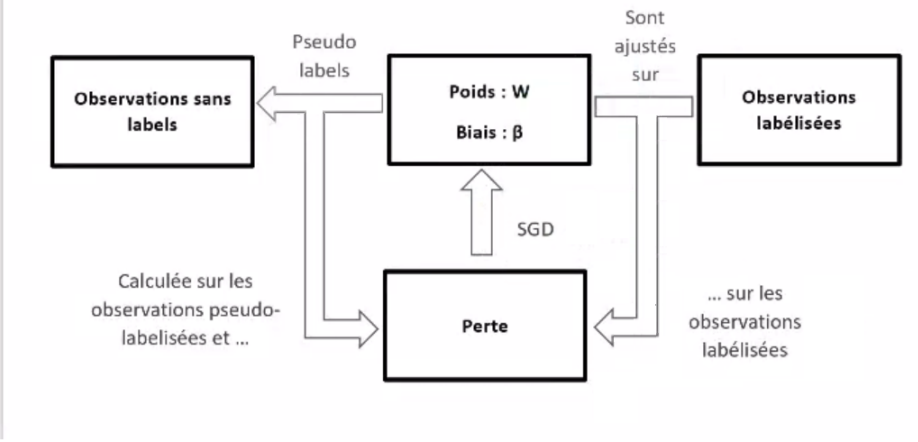


Figure 1: Graphe explicatif d'un réseau de neurones utilisant le pseudo labelling

Cette méthode est très avantageuse car le travail d'étiquetage est long et

onéreux: elle rend possible l'exploitation de données non labélisées. On observe même une amélioration de la précision entre un réseau de neurones supervisé et un réseau de neurones semi-supervisé.

2 Notions mathématiques

Cette partie vise à expliquer précisément la logique mathématique derrière la méthode Pseudo-Label.

2.1 Réseau de neurones profond

Dans la suite, on considérera un réseau de neurone multicouches à M couches de neurones cachés:

$$h_i^k = s^k \left(\sum_{j=1}^{d^k} W_{ij}^k h_j^{k-1} + b_k^i \right), k = 1, ..., M+1$$

Pour la k- ième couche cachée h^k, s^k est la fonction d'activation non linéaire, comme la fonction sigmoïde. Les f_i sont les sorties de couches utilisées pour prédire la classe désirée et les $x_j = h_j^0$ sont les données d'entrées.

L'entraînement du réseau repose sur la minimisation de la fonction de perte suivante:

$$\sum_{i=1}^{C} L(y_i, f_i(x))$$

où C est le nombre de classes a prédire, les y_i sont les annotations des observations, f_i est la réponse du réseau pour la prédiction de la i-ième observation et x représente les données d'entrée. Si l'on considère la fonction sigmoïde comme fonction d'activation, nous pouvons choisir la *Cross Entropy* comme fonction de perte:

$$L(y_i, f_i(x)) = -y_i \log f_i - (1 - y_i) \log(1 - f_i)$$

2.2 Denoising Auto-Encoder

Un auto encodeur est un réseau de neuronnes non supervisé dont le but est d'apprendre une représenation des données puis de reconstruire les données initiales à partir de cette représentation.

La particularté de ce réseau de neuronnes est que la taille de l'output est égale (et doit absolument l'être) à la taille de l'input.

Un Autoencoder a trois composantes: l'encoder (qui apprend la représentation des images), le code (la représentation apprise) et le decoder (qui se sert de la

représentation apprise pour recréer l'image)[1].

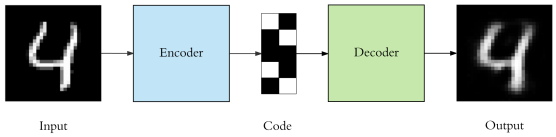


Figure 2: Schématisation d'un Autoencoder

L'utilité d'une telle méthode est la réduction de dimension: quel est le minimum d'information nécessaire qui permet de recréer une image le plus fidèlement possible. Ainsi, l'idée générale d'un autoencoder est proche de celle d'une Analyse en Composante Principale.

Le Denoising Auto Encoder est un autoencodeur qui agit sur des images bruitées. Il essaye d'apprendre une représentation de l'image sans tenir compte du bruit pour ensuite recréer des images "plus nettes".[2]

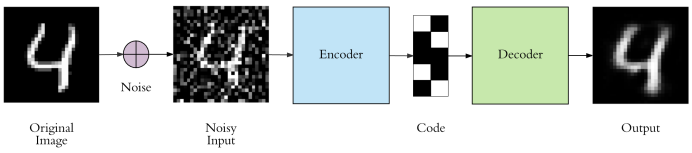


Figure 3: Schématisation d'un Denoising Auto-Encoder

Dans notre cas, son utilisation lors de la phase de pré-apprentissage permet de rendre le choix des pseudo-labels plus robustes.

Ainsi on considère:

$$h_i = s\left(\sum_{j=1}^{d_v} W_{ij}\widetilde{x}_j + b_i\right)$$

$$\hat{x}_j = s\left(\sum_{i=1}^{d_h} W_{ij}h_i + a_j\right)$$

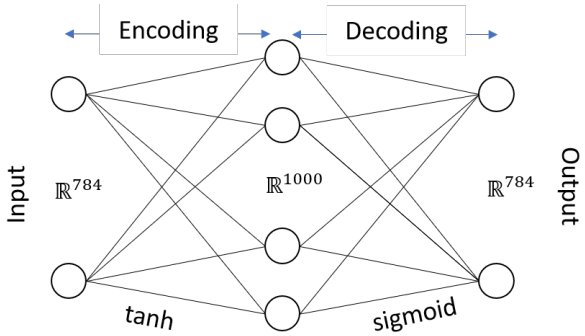
où $\widetilde{x}$ est la version corrompue de la j-ième observation et où $\hat{x}_j$ en est la valeur corrigée. Entraîner un "autoencoder" consiste a minimiser l'erreur de reconstruction entre x_j et $\hat{x}_j$. Dans le cas de l'utilisation sur les données MNIST, cette erreur est à nouveau minimisée par la *Cross Entropy*:

$$L(x, \hat{x}) = \sum_{j=1}^{d_v} -x_j \log\hat{x} - (1 - x_j)\log(1 - \hat{x})$$

[1] https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[2] id.

Pour imager le DEA nous avons entraîné un DAE (uniquement) sur les données MNIST avec une couche cachée de 1000 *units*. Les images étant de taille 28 par 28 la couche d'entrée a 784 unités. Puisque les images sont en noir et blanc alors les inputs sont compris entre 0 et 1. On peut donc utiliser la *Cross Entropy* comme fonction de perte. Ce réseau est exposé dans le figure suivant :



En entraînant ce réseau de DAE, on obtient le résultat :

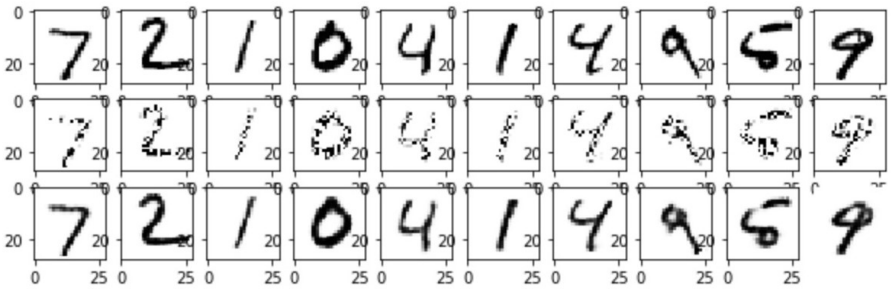


Figure 4: Résultat de DAE en appliquant l'architecture dans Figure **??** sur les données MNIST: données initiales, données bruitées, données reconstituées

2.3 Dropout

Le dropout est une technique de régularisation qui vise à réduire le problème d'overfitting souvent observé lors de l'apprentissage d'un réseau de neurones à beaucoup de couches. Le principe est que, durant la phase d'entraînement et à chaque itération de la backpropagation, une certaine fraction des neurones est ignoréé par le réseau de neurones. Le choix p des neurones à abandonner (drop) est aléatoire, on n'en fixe que la proportion. Il a été montré que le choix idéal de p est 0.5. Le Dropout peut-être appliqué à la partie supervisée de l'apprentissage

de réseaux de neurones comme illustré dans le graphique ci-dessous [3]

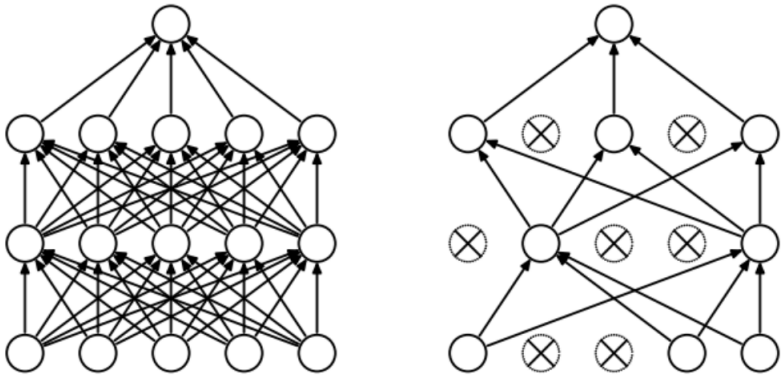


Figure 5: Schématisation de la méthode de régularisation dropout

L'équation des couches cachées de notre réseau de neurones sera donc de la forme:

$$h_i^k = drop\left(s^k\left(\sum_{j=1}^{d^k} W_{ij}^k h_j^{k-1} + b_i^k\right)\right)$$

tel que

$$drop(x) = 0$$

avec une probabilité de 0.5, sinon

$$drop(x) = x$$

.

2.4 Pseudo-Label

On appelle *Pseudo-Label* les classes cibles utilisées pour les données non annotées comme si elles étaient les vraies annotations. La classe avec la plus haute probabilité prédite est choisie pour chaque observation non annotée.

$$y_i' = \begin{cases} 1 & \text{si } i = \text{argmax}_{i'} f_{i'}(x) \\ 0 & \text{sinon.} \end{cases}$$

Le Pseudo-Label est utilisée dans une étape de *fine-tuning* avec dropout. Le réseau est pré-entraîné avec des données annotées et non annotées simultanément.

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

Comme le nombre total de données annotées et non annotées est différent et que leur répartition est importante pour les performances du réseau, la fonction de perte utilisée est :

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

où n est le nombre de batches annotés pour la descente de gradient stochastique, n' est le nombre de batches non annotés, f_i^m est la sortie de m observations annotées, y_i^m est la classe de ces observations, $f_i'^m$ est la sortie de m observations non annotées, $y_i'^m$ est le pseudo-label des observations non annotées défini plus haut et $\alpha(t)$ est un coefficient permet d'équilibrer les répartitions.

Le choix de $\alpha(t)$ adéquat est important pour garantir de bonnes performances du réseau. Si $\alpha(t)$ est choisi trop grand, l'apprentissage est perturbé même pour les données annotées. Alors que si $\alpha(t)$ est trop petit, le réseau ne tire pas toute l'information des données non annotées. Donc, $\alpha(t)$ est défini pour augmenter afin de perfectionner le processus d'optimisation et ainsi permettre aux pseudo-labels de devenir aussi proche que possible de la réalité.

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

où $\alpha_f = 3$, $T_1 = 100$, $T_2 = 600$ sans pré-apprentissage et $T_1 = 200$, $T_2 = 800$ avec Denoising Auto-encoder.

3 Etude de cas : données MNIST

3.1 Architecture des modèles

Notre étude est effectuée avec l'aide de types de réseau :

- Perceptron multicouche (MLP) et réseau de neurones convolutionnel. Ce type de réseau est testé dans l'article de Dong-Hyun Lee
- Réseau de neurones convolutionnel qui est connu comme un bon outil pour le traitement d'image

L'architecture de chaque modèle est précisée comme suit.

3.1.1 Réseau de Perceptron Multicouche

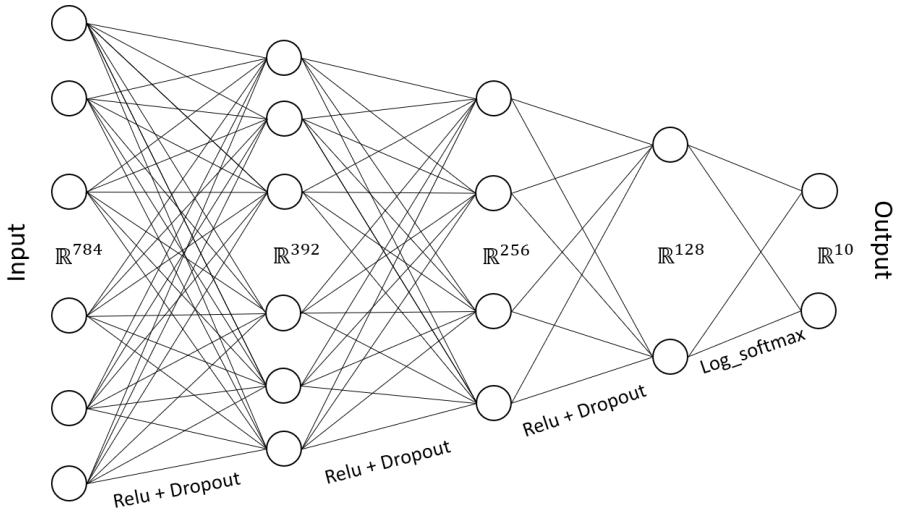


Figure 6: Architecture du Perceptron multicouche

Inspiré par le principe d'apprentissage profond qui utilise plusieurs transformation linéaires (correspondant au calcul avant d'adapter la fonction d'activation) et divers type de fonction d'activation afin de capturer les distributions complexes (suivant non linéaire) des données, nous construisons donc un réseau qui comprend 3 *hidden layers*, à la place d'un réseau d'une seule couche cachée avec un grand nombre des unités dans l'article de Lee. L'architecture de ce réseau est donc illustrée dans le Figure 6.

3.1.2 Réseau de neurones convolutionnel

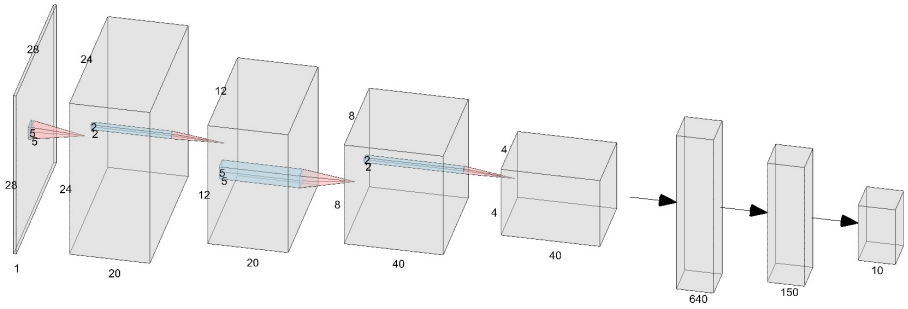


Figure 7: Architecture du Perceptron multicouche
Couches : Convolution 5x5 - Pooling 2x2 - Convolution 5x5 - Pooling 2x2 - Dropout2d - Flatten - Dense 640x150 - Dense 150x10

Pour ce réseau, on transforme les données initiales tout d'abord par une couche convolutionnelle de noyau 5x5x20, puis une couche pooling qui réduit la taille de 2D par moitié. La combinaison de couche convolitionnelle et pooling est réalisée encore une fois avant le dropout2D.

La différence principale entre dropout et dropout2d est que le dropout fonctionne sur des inputs de n'importe quelle dimension. Alors que le dropout2d est conçu pour être appliqué à des objets 4D tel que des images ou les sorties de couches de convolution. Dans ce cas, des features "proches" peuvent être très correlées et un dropout classique ne pourra pas correctement régulariser le réseau. Le dropout2d, également appelé dropout spatial, permet de s'assurer que des pixels adjacents sont soit tous nuls soit tous actifs.

Puis, les unités sont mises à plat et les dimensions de l'objet sont réduites avec l'aide de deux couche *dense* afin de correspondre à la taille de output.

3.2 Choix des hyperparamètres

Après différents essais sur notre problème particulier, à savoir réaliser une prédiction du jeu de données MNIST à partir de 100 observations annotées, nous avons choisis ces différents hyperparamètres:

- taille de batch n pour observations annotées : $n = 32$
- taille de batch n' pour observations non annotées : $n' = 256$
- nombre d'epoch pour le réseau avec seulement les observations annotées : 200
- nombre d'epoch pour le réseau utilisant *Pseudo Label* : 500
- *Stochastic Gradient Descent* avec *learning rate* est égal à 0.1

Les données MNIST possèdent au total 60.000 observations dans l'échantillon d'apprentissage. Lorsque l'on choisi 100 observations labelées avec les tailles de batch n et n' ci dessus, on obtient un nombre de batch sur les données non labelées très grand devant le nombre de batch sur les données labelées: $230 \gg 3$.

Théoriquement, on doit mettre à jour le gradient en considérant le conjoint de chaque type de batch. Un tel algorithme est coûteux en temps, alors pour faire face à ce problème nous considérons l'idée de Anirudh Shenoy [4] : Pour chaque epoch, on va faire tourner tous les mini-batchs correspondant aux observations non annotées. Puis, pour chaque 50 batchs, nous apprendrons les trois batchs correspondant aux observations annotées.

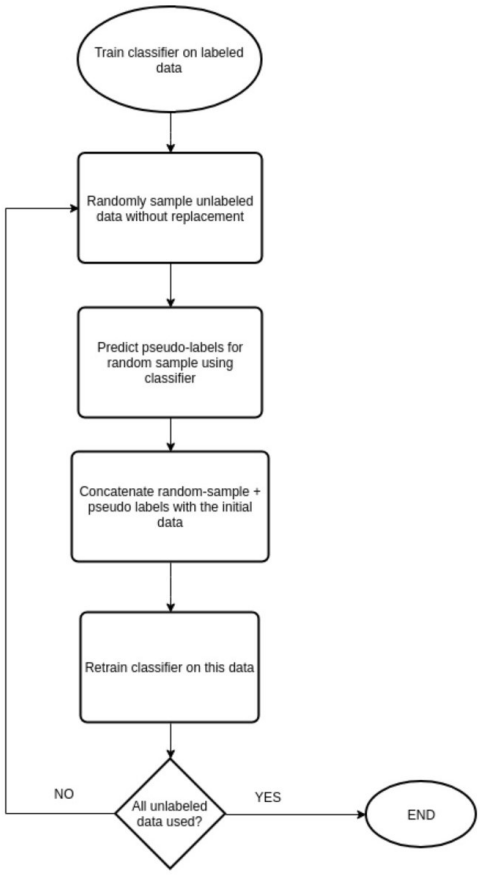


Figure 8: Schéma d'entraînement des réseaux de neurones

[4] https://towardsdatascience.com/pseudo-labeling-to-deal-with-small-datasets-what-why-how-fd6f903213af

3.3 Résultats

Les résultats présentés par la suite sont ceux obtenus dans le cas où 100 observations annotées de la base de données MNIST sont utilisées comme base d'apprentissage.

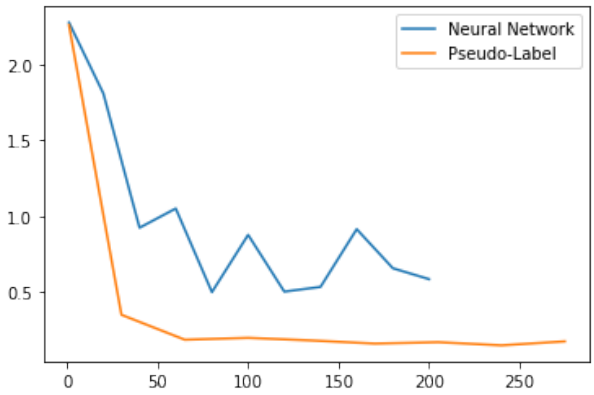


Figure 9: Valeur du coût en fonction du nombre d'epoch

On observe que, grâce à l'utilisation du Pseudo-Label, la valeur du coût en fonction du nombre d'epoch est nettement inférieure à celle obtenue pour une réseau sans PL.

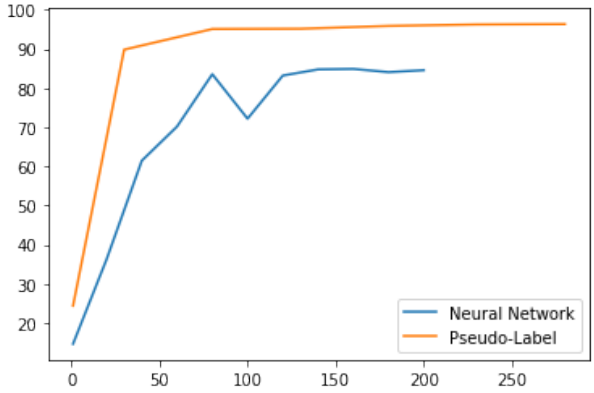


Figure 10: Précision des modèles en fonction du nombre d'epoch

De la même manière, la précision obtenue avec un modèle utilisant les Pseudo-Labels est supérieure que celle obtenue avec des modèles classiques.

	Computation	Accuracy	Accuracy Lee-article
MLP	69	77.93	78.11
+ PL	1432	83.70	83.85
+ PL + DAE	7481	84.24	89.51
CNN	76	85.80	
+ PL	1790	94.44	
+ PL + DAE	6338	**96.50**	

Table 1: Table de comparaison des résultats

Les scores de précisions présentés sont ceux obtenus pour 100 epoch, il n'y a pas de différence marquante au delà de cette valeur. On remarque que les résultats que nous avons obtenus sont relativement proches de ceux obtenus par Lee dans son article.

4 Conclusion

Dans ce projet nous avons tenté de reproduire les méthodes de l'article de Lee en implémentant et appliaquant le "Pseudo-label", un Denoising Auto Encoder, un dropout à deux réseaux de neurones (un MLP et un CNN) appliqués aux données MNIST. Les modèles étant assez long à faire tourner nous n'avons pas pu prouver la robustesse des résultats. Cependant l'utilisation du dropout et la proximité de nos résultats avec ceux présentés dans l'article nous rassurent à ce propos. En guise d'ouverture il est intéressant de soulever deux problèmes pouvant se poser:

- Le premier concerne la répartition des classes dans les données labelisées. Que donnerait la phase de pseudo labélisation si une ou plusieurs classes étaient sur ou sous représentée ?
- Le second problème vient du petit nombre de données possédant un label. Cependant, la quantité de ce type de données augmentent progressivement puisque les pseudo-labels sont par la suite considérés comme les vrais labels. Ces pseudo-labels sont prédits *grâce* aux poids du réseau ajustés par les données labélisées. Ainsi, la prédiction des pseudo-labels dépend directement de la qualité des données non labelisées (et non seulement de leur quantité). Alors, au vu du faible nombre de données labelisées utilisées les prédictions seront très sensibles aux données "abberantes".

5 Annexes

Tous les programmes de nos travaux sur Python se trouve dans ce lien : `https://github.com/hoangdungnguyen/PL_deeplearning`

Le code dans la suite est pour le réseau de neurones convolutionnel utilisant le *Pseudo Labels* avec DAE en pré-apprentissage.

```

#!/usr/bin/env python
# coding: utf-8
get_ipython().run_line_magic('matplotlib', 'inline')
from matplotlib import pyplot as plt
import torch
from torch import nn
import torch.nn.functional as F
import numpy as np
import pandas as pd
import time
import gc

torch.manual_seed(42)
np.random.seed(42)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

from tensorflow.examples.tutorials.mnist import input_data
from sklearn.model_selection import StratifiedShuffleSplit
from tqdm import tqdm_notebook


mnist = input_data.read_data_sets("", one_hot=True)

##### Define hyper-parameters #####

# Dropout parameters
dropoutRate_0 = 0.
dropoutRate_1 = 0.5 #ref


# iteraction parameters
num_labelled = 100
trainingEpochs = 200
PLtrainingEpochs = 500
train_BS = 32 # ref
unlabel_BS = 256 #ref

# balancing coefficient
T1 = 200 #ref
T2 = 800 #ref
a = 0. #ref
af = 3. #ref

# DAE
corruption_proba = 0.5

# loading data
x_test = mnist.test.images
y_test = mnist.test.labels
x_trainall = mnist.train.images
y_trainall = mnist.train.labels
```

```
54
55 x_trainall = torch.from_numpy(x_trainall).type(torch.FloatTensor)
56 y_trainall = torch.from_numpy(y_trainall).type(torch.FloatTensor)
57 trainall = torch.utils.data.TensorDataset(x_trainall, y_trainall)
58 trainall_loader = torch.utils.data.DataLoader(trainall,
59   batch_size = 1000, shuffle = True, num_workers = 8)
60
61 ##### Create noisy observation ####
62 def making_noise(x, prob):
63   if type(x) == np.ndarray :
64     x = torch.from_numpy(x).type(torch.FloatTensor)
65   dim1, dim2 = x.shape
66   corrupted_factor = np.concatenate([np.zeros([dim1, int(dim2*prob)
       ]),
67     np.ones([dim1, dim2- int(dim2*prob)])], axis = 1)
68   np.apply_along_axis(np.random.shuffle,1,corrupted_factor)
69   return x*torch.from_numpy(corrupted_factor).type(torch.
       FloatTensor)
70
71 class ConvDenoiser(nn.Module):
72       def __init__(self):
73         super(ConvDenoiser, self).__init__()
74         ## encode layer ##
75         self.fc1 = nn.Linear(784,1000)
76         ## decode layer
77         self.fc2 = nn.Linear(1000,784)
78
79       def forward(self, x):
80         x = F.tanh(self.fc1(x))
81         x = F.sigmoid(self.fc2(x))
82         return x
83
84
85 #
86 def DAE(model, train_loader):
87
88     optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
89     n_epochs = 100
90     begin = time.time()
91     for epoch in tqdm_notebook(range(n_epochs)):
92         # monitor training loss
93         train_loss = 0.0
94         for X_batch,_ in train_loader :
95
96             X_batch_noise = making_noise(X_batch, corruption_proba)
97
98             optimizer.zero_grad()
99             outputs = model(X_batch_noise)
100            # calculate the loss
101            # the "target" is still the original, not-noisy images
102            loss = F.binary_cross_entropy(outputs, X_batch)
103            loss.backward()
104            optimizer.step()
105            train_loss += loss.item()*X_batch.size(0)
106        if (epoch+1)% 10 == 0 or epoch ==0:
107            train_loss = train_loss/len(train_loader)
108            print('Epoch: {} : Time = {:.2f} | Train Loss = {:.3f}'
```

```
    .
                format(epoch+1, time.time() - begin,train_loss))

DAE_model = ConvDenoiser()
DAE(DAE_model, trainall_loader)

x_test_noise = making_noise(x_test[0:100], corruption_proba)
x_test_DAE = DAE_model(x_test_noise)
x_test_DAE = x_test_DAE.detach().numpy()
x_test_noise = x_test_noise.numpy()
f, a = plt.subplots(3, 10, figsize=(10, 3))
plt.axis('off')
for i in range(10):
    a[0][i].imshow(np.reshape(x_test[i], (28, 28)),
      cmap='Greys',  interpolation='nearest')
    a[1][i].imshow(x_test_noise[i].reshape(28, 28),
      cmap='Greys',  interpolation='nearest')
    a[2][i].imshow(x_test_DAE[i].reshape(28, 28),
      cmap='Greys',  interpolation='nearest')
plt.show()

def autoencoder_data(tensor_data):
  return DAE_model(making_noise(tensor_data, corruption_proba))

x_test = autoencoder_data(x_test)
y_test = mnist.test.labels
test_BS = len(x_test)// (num_labelled//train_BS)

y_test = torch.from_numpy(y_test.argmax(1)).type(torch.LongTensor)
test = torch.utils.data.TensorDataset(x_test, y_test)
test_loader = torch.utils.data.DataLoader(test,
  batch_size = test_BS, shuffle = True, num_workers = 8)

del x_test, y_test, x_trainall, y_trainall, trainall,
    trainall_loader
gc.collect()

def random_data_generator(nb_PLdata = False):
  stratSplit = StratifiedShuffleSplit(test_size=100, n_splits=1)
  stratSplit.get_n_splits(mnist.train.images,
    np.argmax(mnist.train.labels, axis = 1))

  for train_index, test_index in stratSplit.split(X = mnist.train.
    images,
    y = mnist.train.labels):
    x_train = mnist.train.images[test_index]
    y_train = mnist.train.labels[test_index]
    if nb_PLdata == False :
      x_PL = mnist.train.images[train_index]
    else:
      x_PL = mnist.train.images[train_index][0:nb_PLdata]
  return x_train, y_train, x_PL

x_train_viewtest, y_train_viewtest, x_PL_viewtest =
    random_data_generator()
print('Labeled data size :', x_train_viewtest.shape)
print('Unlabeled data size :', x_PL_viewtest.shape)
```

```
print('Proportion of class label in train data: ')

print(pd.DataFrame(np.unique(np.argmax(y_train_viewtest,1),
  return_counts = True)).to_string())

f, a = plt.subplots(1, 10, figsize=(10, 3))
for i in range(10):
    a[i].imshow(np.reshape(x_train_viewtest[i], (28, 28)),
      cmap='Greys',  interpolation='nearest')
plt.show()

print(y_train_viewtest[0:10].argmax(1))

def data_build(nb_PLdata = False):
  x_train, y_train, x_PL = random_data_generator(nb_PLdata)
  x_train = torch.from_numpy(x_train).type(torch.FloatTensor)
  y_train = torch.from_numpy(y_train.argmax(1)).type(torch.
    LongTensor)

  x_PL = torch.from_numpy(x_PL).type(torch.FloatTensor)

  train = torch.utils.data.TensorDataset(x_train, y_train)
  train_loader = torch.utils.data.DataLoader(train,
    batch_size = train_BS, shuffle = True, num_workers = 8)

  unlabeled = torch.utils.data.TensorDataset(x_PL)
  unlabeled_loader = torch.utils.data.DataLoader(unlabeled,
    batch_size = unlabel_BS, shuffle = True, num_workers = 8)

  return train_loader, unlabeled_loader

class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.conv1 = nn.Conv2d(1, 20, kernel_size=5)
            self.conv2 = nn.Conv2d(20, 40, kernel_size=5)
            self.conv2_drop = nn.Dropout2d(dropoutRate_1)
            self.fc1 = nn.Linear(640, 150)
            self.fc2 = nn.Linear(150, 10)
            self.log_softmax = nn.LogSoftmax(dim = 1)

        def forward(self, x):
            x = x.view(-1,1,28,28)
            x = F.relu(F.max_pool2d(self.conv1(x), 2))
            x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)),
    2))
            x = x.view(-1, 640)
            x = F.relu(self.fc1(x))
            x = F.dropout(x, training=self.training)
            x = F.relu(self.fc2(x))
            x = self.log_softmax(x)
            return x


# Now let's define a function to evaluate the network
#and get loss and accuracy values.
def evaluate(model, test_loader):
```

```
    model.eval()
    correct = 0
    loss = 0
    with torch.no_grad():
        for data, labels in test_loader:
            data = data.cuda()
            output = model(data)
            predicted = torch.max(output,1)[1]
            correct += (predicted == labels.cuda()).sum()
            loss += F.nll_loss(output, labels.cuda()).item()

    return (float(correct)/len(test)) *100, (loss/len(test_loader))


# First, let's train the model on the labeled set for 300 epochs
def train_supervised(model, train_loader, test_loader, verbose_step
     = 10):
    test_acc_list = []
    test_loss_list = []
    optimizer = torch.optim.SGD( model.parameters(), lr = 0.1)
    model.train()
    begin = time.time()
    for epoch in tqdm_notebook(range(trainingEpochs)):
        correct = 0
        running_loss = 0
        for batch_idx, (X_batch, y_batch) in enumerate(train_loader
    ):
            X_batch, y_batch = autoencoder_data(X_batch).cuda(),
    y_batch.cuda()

            output = model(X_batch)
            labeled_loss = F.cross_entropy(output, y_batch)

            optimizer.zero_grad()
            labeled_loss.backward()
            optimizer.step()
            running_loss += labeled_loss.item()

        if (epoch+1) %verbose_step == 0 or epoch ==0:
            test_acc, test_loss = evaluate(model, test_loader)
            test_acc_list.append(test_acc)
            test_loss_list.append(test_loss)
            print('Epoch: {} : Time = {:.2f} | Train Loss = {:.3f}
    \
              | Test Acc = {:.3f} | Test Loss= {:.3f} '.
                format(epoch+1,time.time()-begin,
                  running_loss/(10 * num_labelled),
                  test_acc, test_loss))
            model.train()
    return test_acc_list, test_loss_list

def alpha_weight(epoch):
    if epoch < T1:
        return 0.0
    elif epoch > T2:
        return af
    else:
```

```
        return ((epoch-T1) / (T2-T1))*af

def semisup_train(model, train_loader, unlabeled_loader,
  test_loader, verbose_step = 10):
    alpha_list = []
    test_acc_list = []
    test_loss_list = []
    begin = time.time()
    optimizer = torch.optim.SGD(model.parameters(), lr = 0.1,)

    # Instead of using current epoch we use a "step" variable to
    calculate alpha_weight
    # This helps the model converge faster
    step = 0

    model.train()
    for epoch in tqdm_notebook(range(PLtrainingEpochs)):
        for batch_idx, x_unlabeled in enumerate(unlabeled_loader):


            # Forward Pass to get the pseudo labels
            x_unlabeled = autoencoder_data(x_unlabeled[0]).cuda()
            model.eval()
            output_unlabeled = model(x_unlabeled)
            _, pseudo_labeled = torch.max(output_unlabeled, 1)
            model.train()

            # Now calculate the unlabeled loss using the pseudo
    label
            output = model(x_unlabeled)
            unlabeled_loss = (alpha_weight(step) *
              F.cross_entropy(output, pseudo_labeled))

            # Backpropogate
            optimizer.zero_grad()
            unlabeled_loss.backward()
            optimizer.step()

            # For every 50 unlabeled batches train one epoch on
    labeled data
            if batch_idx % 50 == 0:
              for batch_idx, (X_batch, y_batch) in enumerate(
    train_loader):
                X_batch = autoencoder_data(X_batch).cuda()
                y_batch = y_batch.cuda()
                output = model(X_batch)
                labeled_loss = F.cross_entropy(output, y_batch)

                optimizer.zero_grad()
                labeled_loss.backward()
                optimizer.step()

              # Now we increment step by 1
              step += 1

        if (epoch+1) %verbose_step == 0 or epoch ==0:
            test_acc, test_loss =evaluate(model, test_loader)
```

```
            print('Epoch: {} : Time = {:.2f} | Alpha Weight = {:.3f} \
              | Test Acc = {:.3f} | Test Loss = {:.3f} '.
                  format(epoch+1, time.time()- begin,
                    alpha_weight(step),
                    test_acc, test_loss))

            """ LOGGING VALUES """
            alpha_list.append(alpha_weight(step))
            test_acc_list.append(test_acc)
            test_loss_list.append(test_loss)
        """ ************** """
        model.train()
    return alpha_list, test_acc_list, test_loss_list


def run_function(verbose_supervised = 20, verbose_unsupervised = 50) :
    train_loader, unlabeled_loader = data_build()
    print('===== Supervised training =====')
    net = Net().cuda()
    NN_acc, NN_loss = train_supervised(net, train_loader,
      test_loader, verbose_supervised)

    print('===== Semi-supervised training =====')

    net = Net().cuda()
    alpha_list, PL_acc, PL_loss = semisup_train(net, train_loader,
      unlabeled_loader, test_loader, verbose_unsupervised)

    print('===== Conclusion =====')
    print('Supervised accuracy = ', NN_acc[-1])
    print('+PL accuracy = ', PL_acc[-1])
    return NN_acc, NN_loss, alpha_list, PL_acc, PL_loss

NN_acc, NN_loss, alpha_list, PL_acc, PL_loss = run_function()

plt.clf()
plt.plot(np.append(1,np.arange(20,201,20)), NN_acc)
plt.plot(np.append(1,np.arange(50,501,50)), PL_acc)

plt.clf()
plt.plot(np.append(1,np.arange(20,201,20)), NN_loss)
plt.plot(np.append(1,np.arange(50,501,50)), PL_loss)
```

Listing 1: CNN + PL + DAE programme