Apprentissage profond avec PyTorch

Partie 2 : Réseaux convolutifs / réseaux récurrents

1 Deep Learning – Apprentissage profond

1.1 Introduction

Dans les parties précédentes, nous nous sommes intéressés à la forme la plus courante d'un réseau de neurone. Néanmoins, malgré sa simplicité structurelle, l'apprentissage d'un tel réseau reste une procédure sensible, dont le succès dépend largement du type d'application, des données disponibles et du choix des paramètres.

1.2 Pourquoi le Deep Learning

L'inconvénient des réseaux de neurones historiques provient du faible nombre de couches cachées, une ou deux en pratique. Cela rend difficile l'utilisation de ces réseaux sur des applications complexes comme la reconnaissance de caractères, ou d'images assez simples.

Autrement dit, les applications opérationnelles nécessitent des structures de réseau avec un nombre de couches plus conséquent. Malheureusement, plus le nombre de couches augmente et plus les problèmes numériques deviennent rédhibitoires, sans parler des temps de calcul qui explosent.

Le Deep Learning correspond aux réseaux de neurones comprenant plusieurs couches internes et potentiellement un très grand nombre. On utilise le terme "Deep" (profond) pour évoquer la "profondeur" du réseau, c'est-à-dire son nombre de couches. Les premiers papiers de recherche sur Deep Learning datent de 1986. Mais ce n'est qu'à partir de 2006 que des progrès ont réellement été réalisés mais les choses ont vraiment changé depuis les années 2010.

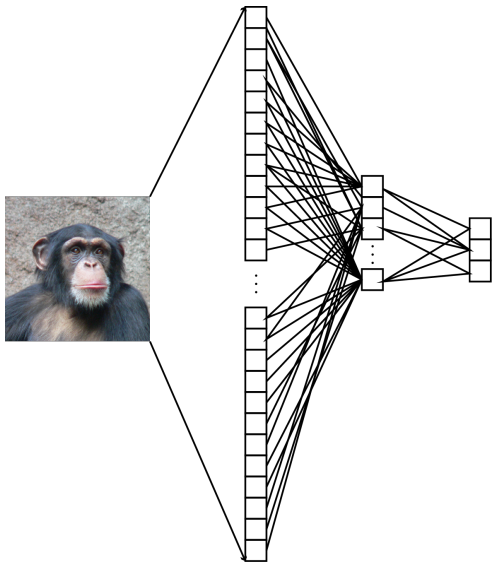
Tout a commencé par une publication de l'équipe de Geoffrey Hinton (http://www.cs.toronto.edu/~hinton/). Cette publication décrivait comment pré-entraîner un réseau de plusieurs couches avec une approche non supervisée et progressive, c'est-à-dire en progressant couche par couche, puis en terminant l'apprentissage par une procédure de rétropropagation, supervisée cette fois.

Ensuite, à partir de 2011, les progrès ont été constants et rapides. D'une part, il est montré qu'il vaut mieux entraîner les réseaux sur de très grands jeux de données plutôt que de pratiquer des prétraitements sur des jeux de données plus réduits. D'autre part, les problèmes de puissance de calcul nécessaire pour la phase d'apprentissage étaient résolus par l'utilisation astucieuse de GPU (Graphical Process Unit).

Rapidement, les grandes entreprises du WEB se sont rendu compte du potentiel du Deep Learning et elles ont commencé à investir de manière très importante. En particulier, Google a embauché Geoffrey Hinton, et Facebook le français Yann LeCun, les deux chercheurs précurseurs et parmi les plus réputés du domaine.

1.3 Comment ça marche

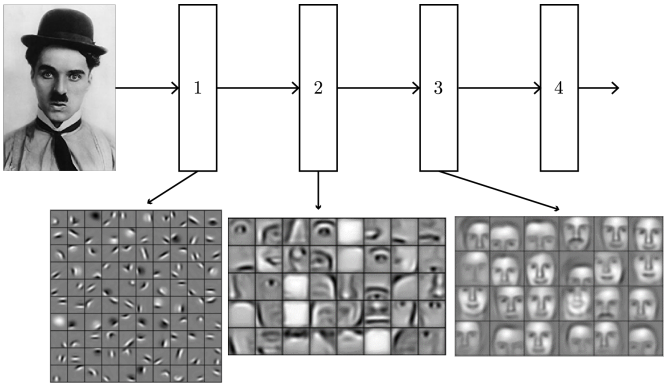
Imaginons que nous souhaitions réaliser une application de reconnaissance d'image. Par exemple, une petite image de 512*512 pixels, c'est-à-dire, 262 544 pixels. Si nous connectons chaque pixel à un neurone d'une couche d'entrée et que la couche interne comporte 512 neurones, cela donne 512*262 144, soient 134217728 poids synaptiques pour une seule couche...On voit que cette approche n'est pas la bonne.



Pour résoudre ce problème, il faut être capable d'extraire de l'image des caractéristiques pertinentes pour les présenter à un réseau plus à même de les reconnaître et de les classer.

1.4 Les réseaux convolutifs

L'idée centrale des réseaux convolutifs (CNN) est de concevoir une architecture comprenant des couches dédiées qui vont apprendre ces prétraitements au lieu de les coder, afin d'extraire les caractéristiques de l'image. Celles-ci sont ensuite transmises à un réseau plus classique qui effectue la phase de reconnaissance. Une autre façon de décrire cette approche est de la voir comme une décomposition hiérarchique du processus de reconnaissance, où chaque couche participe à la création de représentation de plus en plus abstraites et conceptuelles.



Une autre avancée importante a été de se rendre compte qu'il valait mieux augmenter le nombre des exemples d'apprentissage, plutôt que vouloir effectuer des pré-traitements pour résoudre les problèmes d'invariance d'échelle, de translation ou de rotation. En effet, un réseau de neurones ne sait pas reconnaître des formes dans une image qui sont transformées par translation, rotation ou changement d'échelle. Autrement dit, un chat à droite ou à gauche de l'image, plus loin ou plus près, ne sera pas reconnu comme la même forme, même si c'est exactement le même chat.

En pratique, un tel réseau peut être vu comme un jeu de Lego, mais avec des couches de neurones à la place des briques. Les briques les plus utilisées étant les suivantes :

- Les couches de *convolution* qui traitent les données provenant d'une image.
    - L'opération de convolution consiste à appliquer un filtre sur une image. Le filtre est une petite matrice de taille variable (*3*x*3*, *5*x*5*, *9*x*9*, …) appelée « *kernel* » (ou « *kernel de convolution* »). Voici un exemple de noyau 2x2 :

-1	0
1	1

Maintenant supposons que notre image est donnée par le tableau qui suit :

127	59	198
97	150	230
54	138	111

- On suppose que chaque case du tableau correspond à un pixel codé en niveau de gris sur l'échelle [0,255]. Avant de réaliser le produit de convolution, il y a deux hyper paramètres à ajuster, le padding et le stride. Le padding indique combien de rangées de 0 on ajoute autour de l'entrée. Le stride indique de combien de pas on se déplace entre deux calculs de la convolution. La convolution "classique" a une sortie plus petite que l'entrée à cause de la taille du filtre que l'on doit placer à l'intérieur de la feature map d'entrée. Ajouter du padding permet par exemple de retrouver la taille initiale. Ajouter un stride permet de sauter des valeurs, cela correspond à faire du sous-échantillonnage de la sortie.
- Nous allons faire le calcul avec padding de 0 et un stride de 1 :
  - (**-1** x 127) + (**0** x 59) + (**1** x 97) + (**1** x 150) = 120
  - (**-1** x 59) + (**0** x 198) + (**1** x 150) + (**1** x 230) = 321, comme l'échelle de valeur est [0,255] nous avons 255
  - …
- Finalement nous obtenons :

120	255
95	99

- Les deux figures qui suivent montrent des variantes du produit de convolution

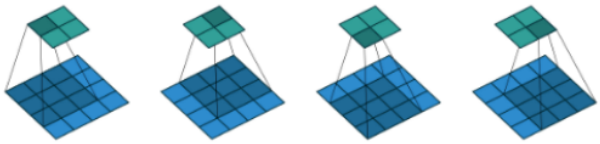


Figure 1 : convolution simple, pas de padding, stride de 1, kernel 2x2

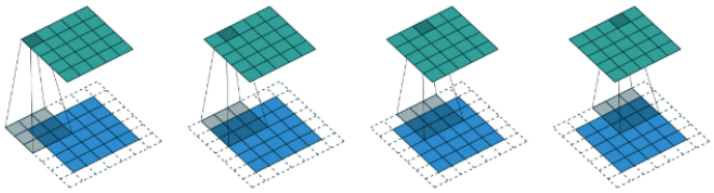


Figure 2 : convolution avec padding de 1, stride de 1, kernel 3x3

- Les couches de *pooling* qui compressent l'information en réduisant la taille de l'image intermédiaire par sous-échantillonnage (subsampling).
  - En particulier, les types de pooling les plus populaires sont le max et l'average pooling, où les valeurs maximales et moyennes sont prises, respectivement.

Figure 3 : Exemple de max pooling, pas de padding, stride de 2, kernel de taille 2x2

- Les couches de correction *relu* (ReLU pour Unité de Rectification Linéaire) qui applique une fonction mathématique sur chaque signal de sortie d'une couche, généralement y = max(0,x). Il s'agit en fait, ni plus, ni moins que d'une fonction d'activation.
- Les couches *fully connected* qui correspondent à des couches classiques de neurones formels totalement connectées.

- Le *Dropout* est une technique de régularisation utilisée afin de réduire l'over-fitting dans le réseau de neurones. Habituellement, on utilise le dropout sur les couches fully connected, mais il également possible d'utiliser le dropout après les couches de max-pooling, créant ainsi une augmentation du bruit.
- Une couche *loss* (perte) qui est la dernière du réseau. Elle spécifie comment l'entraînement du réseau pénalise l'écart entre le signal de sortie prévu et réel. Une des plus utilisées est la fonction *softmax* pour prédire une seule classe parmi plusieurs classes mutuellement exclusives. D'autres permettent de prédire plusieurs valeurs de probabilité indépendante, ou bien encore de régresser vers des valeurs réelles.

Pour résumer, dans un réseau convolutif, l'opération de convolution est suivie d'une couche de fonctions de transfert, généralement ReLU. A la sortie de la convolution, avant la fonction ReLU, nous obtenons une *feature map (ou activation map)*. Elle(s) possède(nt) des valeurs positives et négatives, parce que les poids peuvent être négatifs. Quand elle passe par la couche de ReLU, la ReLU met les valeurs négatives à zéro, et laisse les valeurs positives inchangées. Les tableaux de sortie de la couche ReLU s'appellent aussi des feature maps. Cette opération non linéaire permet au système de détecter des motifs sur l'image. Imaginons l'image d'une personne en train de travailler sur son ordinateur dans une pièce où le papier peint est rayé gris sur beige. Cette image compte des contours francs, comme le bord de l'écran par rapport au mur du fond, mais d'autres moins nets comme le contraste entre les lignes et la tapisserie du mur. Si une convolution détecte les contours verticaux, sa sortie est grande sur le contour de l'écran d'ordinateur (fort contraste) mais plus petite pour les barres du papier peint (faible contraste). Quand on passe cette feature map par une ReLU, les contours nets apparaissent, les autres sont mis à 0. Cela permet au système de détecter les motifs importants. Mais comme chacun des 60 neurones regarde le même endroit, il se peut qu'un des 59 autres soit ajusté pour détecter les contours plus flous.

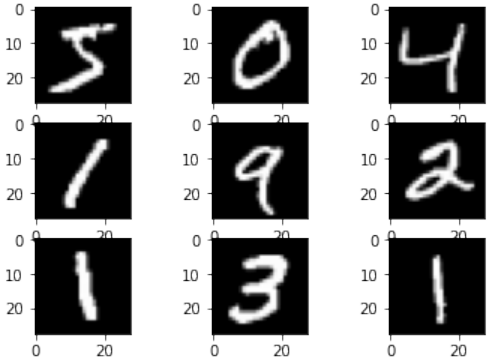
La couche ReLU est généralement suivie d'une couche de *pooling*. Une feature map en sortie de la couche ReLU est divisée en fenêtres, ou plutôt en tuiles, par exemple de taille 4x4, qui ne se chevauchent pas. Si la feature map a une taille de 1000x1000, il y aura 250x250 tuilles de tailles 4x4. Chaque neurone de la couche pooling prend une de ces fenêtres et en calcule la valeur maximale. En d'autres termes, une fenêtre comporte 16 nombres, et le neurone produit le plus grand de ces 16 nombres sur sa sortie. C'est ce que l'on appelle le max pooling. Il y a un neurone de ce type pour chaque fenêtre dans toutes les feature maps. Au total, la sortie de la couche comportera 60x250x250 neurones.

En fait, le pooling sert à produire une représentation invariante par rapport à de petits déplacements des motifs dans l'image d'entrée. Le max pooling produit la valeur la plus grande dans son entrée, qui correspond au motif le plus marqué dans son champ récepteur. Si ce motif se déplace d'un pixel ou deux, tout en restant dans la même fenêtre du neurone de pooling, la sortie de ce neurone sera inchangée. C'est pour ces raisons qu'un réseau de neurones convolutif est constitué d'un empilement de couches de convolutions, de ReLU et de pooling.

2. Mise en œuvre d'un CNN avec PyTorch

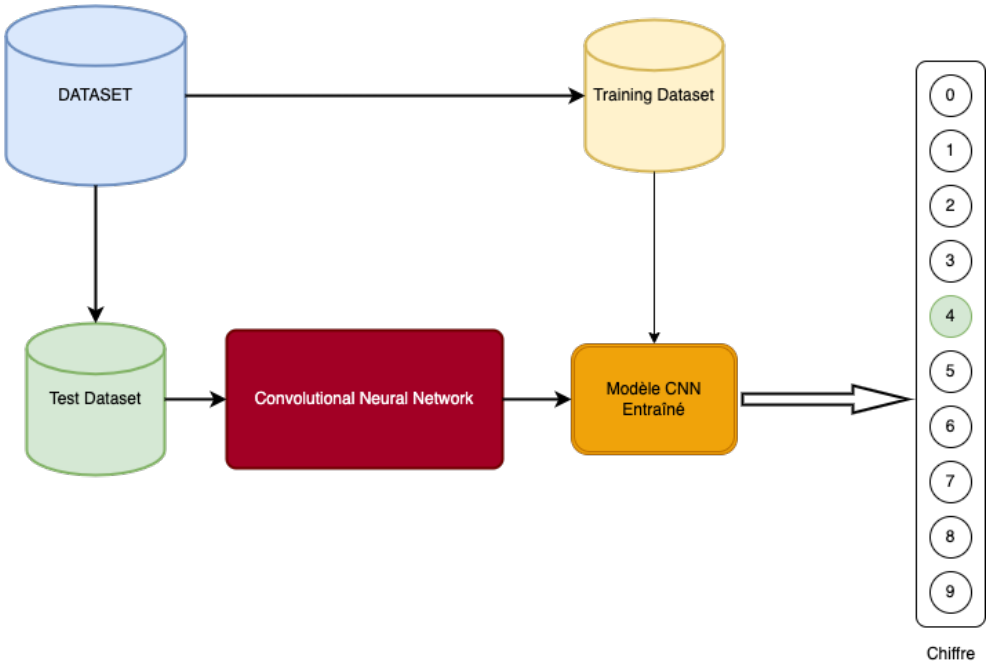
2.1 Introduction

Comme exemple de CNN nous allons reproduire les résultats historiques obtenus par Yann Lecun et son équipe sur la base données MNIST (http://yann.lecun.com/exdb/mnist/). Bien que l'ensemble MNIST soit résolu, c'est un bon point de départ pour développer et pratiquer une méthodologie pour résoudre des tâches de classification d'images à l'aide des réseaux de neurones convolutifs (CNN). MNIST est une sorte de « Hello world » pour les RN. Il s'agit d'un problème de reconnaissance de caractères : les chiffres de 0 à 9. Voici un extrait de la base de données MNIST :



2.2 Architecture des données

Pour rappel, voici comme est utilisé le jeu de données :



2.3 Exploration de la BDD MNist

La base de données est disponible sur la page internet de Yann Lecun : http://yann.lecun.com/exdb/mnist/ . Nous pouvons télécharger le jeu de données à partir du site Web du MNIST, mais il est plus pratique d'utiliser l'API torchvision fournit par PyTorch qui récupère directement les données MNIST. Pour cela, nous devons inclure la ligne **`import torchvision.datasets as datasets`** (conda prompt : **`pip install torchvision`**) en entête du script Python **`CNNMnist.py`**.

Le code qui suit (fichier **`CNNMnist.py`**) permet de télécharger automatiquement les images de la BDD Mnist :

```
import torch
import torch.nn as nn
import torchvision.datasets as datasets
import torchvision.transforms as transforms

train_data = datasets.MNIST(root = './data', train = True,
                            transform = transforms.ToTensor(), download = True)

test_data = datasets.MNIST(root = './data', train = False,
                           transform = transforms.ToTensor())
```

```
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
…
Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw
```

On peut afficher les caractéristiques de la BDD MNIST :

```
print(train_data)
```

```
Dataset MNIST
    Number of datapoints: 60000
```

```
    Root location: data
    Split: Train
    StandardTransform
Transform: ToTensor()
```

```
print(test_data)
```

```
Dataset MNIST
    Number of datapoints: 10000
    Root location: data
    Split: Test
    StandardTransform
Transform: ToTensor()
```

La base d'entrainement est composée de 6000 images de 28x28 pixels :

```
print(train_data.data.size())
```

```
torch.Size([60000, 28, 28])
```

Les labels correspondent à un vecteur de 6000 éléments qui contient pour chacune des images un chiffre entre 0 et 9 :

```
print(train_data.targets.size())
```

```
torch.Size([60000])
```

Vous pouvez facilement obtenir un aperçu d'un échantillon de la BDD :

```
import matplotlib.pyplot as plt

figure = plt.figure(figsize=(10, 8))
cols, rows = 5, 5
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(train_data), size=(1,)).item()
    img, label = train_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(label)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

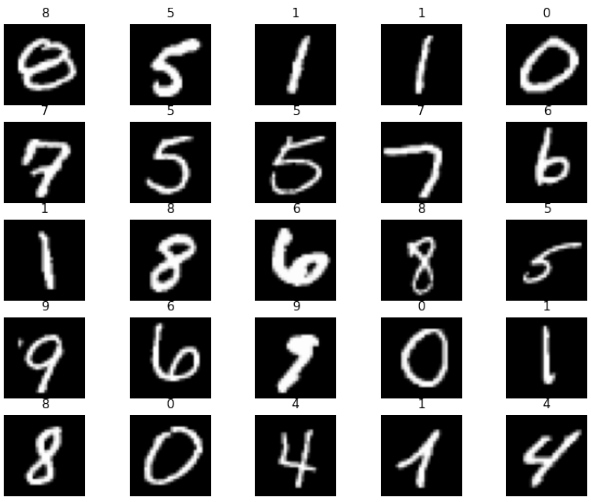


Figure 4: Extrait de la BDD MNist

2.4 Préparation des données : **`DataLoaders`**

Remarquons que si nous itérons sur l'ensemble des données **`train_data[i]`** nous n'avons accès qu'à une image et à son label. Lors de l'entraînement d'un modèle il est classique de transmettre « des échantillons » de la base sous forme de mini-batch de même taille mais composé aléatoirement. Ceci permet de réduire le surajustement du modèle et de traiter plusieurs entrées en même temps afin d'accélérer l'entraînement.

PyTorch fournit l'API `DataLoader` dédiée à cette tâche :

```
batch_size = 100

train_loader = torch.utils.data.DataLoader(dataset = train_data,
                                           batch_size = batch_size,
                                           shuffle = True)

test_loader = torch.utils.data.DataLoader(dataset =  test_data ,
                                          batch_size = batch_size,
                                          shuffle = False)
```

Ci-dessous, la description des arguments de la fonction DataLoader :

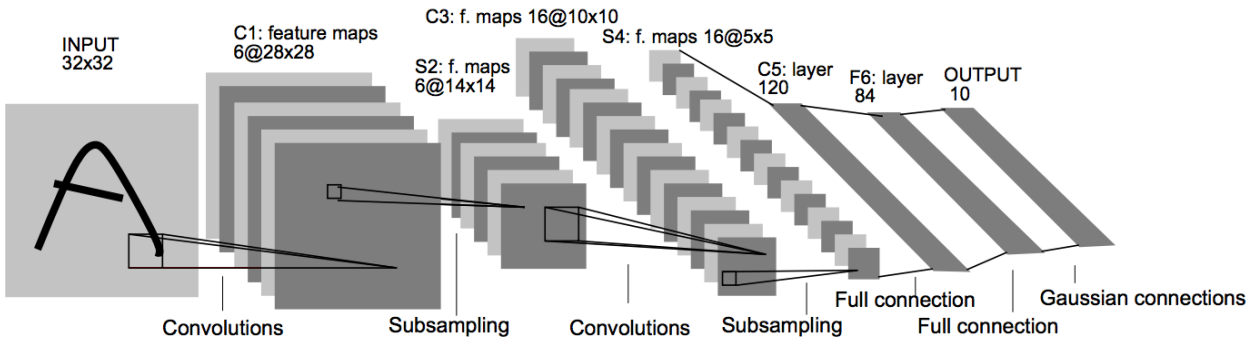
1. **Dataset** : Le premier argument de la classe DataLoader est le dataset (le jeu de données). C'est à partir de celui-ci que nous chargeons les données.

2. Mise en lots des données : **batch_size** fait référence au nombre d'échantillons d'entraînement utilisés dans une itération. En général, nous divisons nos données en ensembles d'entrainement et de test, et nous pouvons avoir des tailles de lot différentes pour chacun.

3. Mélange des données : **shuffle** prend une valeur booléenne (True/False). Si le paramètre shuffle est défini sur True, tous les échantillons sont mélangés avant d'être mis en lots.

2.5 Architecture du CNN

Pour programmer notre premier CNN, nous allons nous inspirer de l'architecture du CNN (appelée LeNet-5) proposée par Yann Lecun dans le célèbre papier[1] : http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf :



Avec une différence :

- les fonctions d'activations seront de type ReLU alors qu'initialement dans le papier de LeCun et al. il s'agissait de fonction de type tangente hyperbolique.

[1] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

Voici un extrait du code de l'architecture du réseau :

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )        # couches entièrement connectées, sortie : 10 classes
        self.out = nn.Linear(32 * 7 * 7, 10)


    def forward(self, x):

        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1) #couche d'aplatissement flatten pour aplatir les
sorties de la couche précédente conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x
```

Voici le détail de l'architecture :

1. Une couche de convolution 2d avec 16 filtres de convolution ainsi qu'un noyaux d'un pas de c(5,5) et une fonction d'activation relu.
   - Cette couche de convolution correspond à un filtre (taille 5x5x1) que l'on va balayer sur l'ensemble de notre image d'entrée. En sortie on va obtenir 16 images de taille (24x24x1). Ces images sont appelées « feature map or activation map ». Ces 16 images constituent notre nouvelle « image ».
   - Enfin, en sortie de cette convolution, on applique la fonction d'activation ReLU : f(x) = max(0,x). Cette opération non linéaire permet en général au système de détecter des motifs sur l'image.
   - Il existe différents types de convolution : https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md
2. Une couche de max-pooling 2d avec un noyau de pooling d'un pas de c(2,2).
   - Cette couche permet de réduire la taille des « images ». Comme la convolution, on applique un filtre qu'on fait glisser sur l'image et dans notre cas on garde la valeur max sur chaque fenêtre. Il s'agit d'un sous échantillonnage. En effet, on obtient un ensemble de 16 images de dimension 12x12.
3. Une couche de convolution 2d avec 16 filtres de convolution ainsi qu'un noyaux d'un pas de c(5,5) et une fonction d'activation relu.
   - Cette couche de convolution correspond à un filtre de taille (5x5). Elle prend en entrée 16 images de 12x12 et elle renvoie 32 images de taille 7x7.
   - En sortie de cette couche on applique la fonction ReLU.
4. Une couche de max-pooling 2d avec un noyau de pooling d'un pas de c(2,2).

- A nouveau, on récupère la sortie de la couche de convolution précédente et on applique un sous-échantillonnage qui permet de réduire la taille des 32 images à 4x4.

5. Une couche d'aplatissement flatten pour aplatir les sorties de la couche précédente.
   - L'objectif de cette couche est d'aplatir les 32 images de la couche précédente afin d'obtenir un vecteur que l'on pourra passer en entrée du réseau de neurones dense qui suit.
6. Une couche cachée composée de 1568 LTU (Linear Threshold Unit = nombre de neurones) et une fonction d'activation relu.
   - Couche de réseau de neurones complétement connectes pour la classification.
7. La dernière couche de sortie composée de 10 neurones.

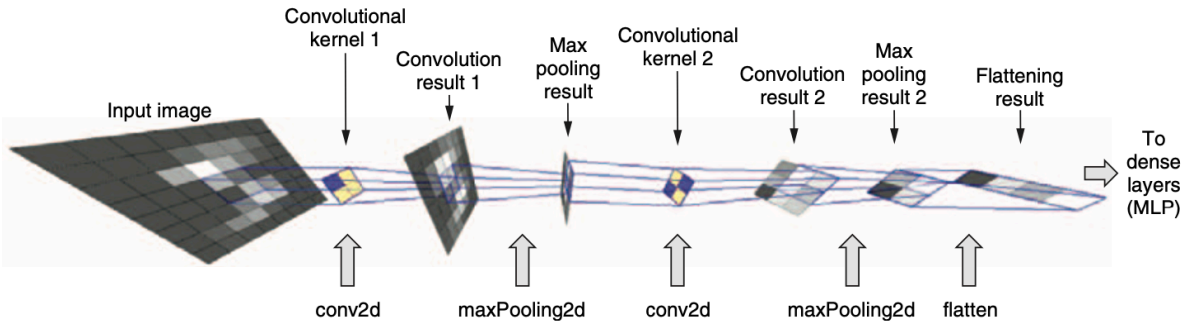


Figure 5: Aperçu de l'architecture du CNN simple du type de celui qui est construit par le code ci-dessus. Notez que ce diagramme montre un seul canal dans chaque tenseur intermédiaire, alors que les tenseurs intermédiaires dans le modèle réel possèdent plusieurs canaux (source : Deep Learning avec Javascript, Edition Manning)

Pour utiliser le modèle, il suffit de l'instancier :

```
cnn = CNN()
print(cnn)
```

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
```

2.6 Entraînement du CNN

L'étape qui suit consiste à définir une fonction de perte :

```
loss_func = nn.CrossEntropyLoss()
```

Contrairement à l'exemple précèdent dans lequel nous traitions un problème de régression, nous faisons le choix d'utiliser une fonction de perte de type Entropie croisée catégorielle très adaptée à de la classification multi classes.

```
loss_func = nn.CrossEntropyLoss()
```

Avant de pouvoir procéder à l'entraînement de notre modèle, il faut spécifier la méthode d'optimisation du gradient :

```
optimizer = optim.Adam(cnn.parameters(), lr = 0.01)
```

Maintenant, nous pouvons entrainer le modèle :

```
num_epochs = 10
```

```
for epoch in range(num_epochs):
  for i ,(images,labels) in enumerate(train_loader):


    optimizer.zero_grad()
    output = net(images)
    loss = loss_fun(output, labels)
    loss.backward()
    optimizer.step()

    if (i+1) % batch_size == 0:
      print('Epoch [%d/%d], Step [%d/%d], Loss: %.4f'%(epoch+1, num_epochs, i+1,
len(train_data)//batch_size, loss.item()))
```

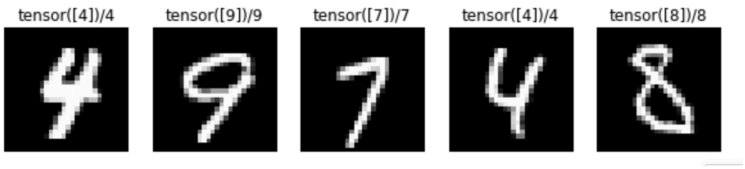
```
Epoch [1/10], Step [100/600], Loss: 0.0990
…
Epoch [1/10], Step [600/600], Loss: 0.0284
…
Epoch [10/10], Step [600/600], Loss: 0.0422
```

Le code qui suit permet d'afficher des images avec leur label et la prédiction réalisée par le réseau de neurones :

```
cols = 5
rows = 1
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(test_data), size=(1,)).item()

    img, label = test_data[sample_idx]

    img = torch.unsqueeze(img, dim=0)
    output = net(img)
    _,prediction = torch.max(output,1)
    figure.add_subplot(rows, cols, i)
    plt.title(str(prediction) + "/" + str(label))
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```



Le script **CNMNIST.py** reprend l'ensemble du code décrit précédemment. Le code source contient également une évaluation de la qualité de l'entrainement

3. Mise en œuvre d'un réseau LSTM (réseaux récurrents) avec PyTorch

3.1 Introduction

Dans cette partie, nous allons explorer comment utiliser les réseaux de neurones récurrents pour générer des données séquentielles. Nous allons utiliser la prédiction d'une série temporelle comme exemple, mais les mêmes techniques peuvent être généralisées à n'importe quel type de données séquentielles : vous pouvez l'appliquer à des séquences de notes de musique afin de générer une nouvelle musique, à la génération de texte, …

3.2 Les séries temporelles – prédiction des cas COVID par LSTM

Dans l'exemple que nous allons traiter, nous allons définir un modèle avec une couche LSTM, l'alimenter en valeurs de taille N extraites d'une série temporelle connue, et l'entraîner à prédire la valeur N+1. On peut imaginer d'essayer de prédire le cours d'une valeur boursière.

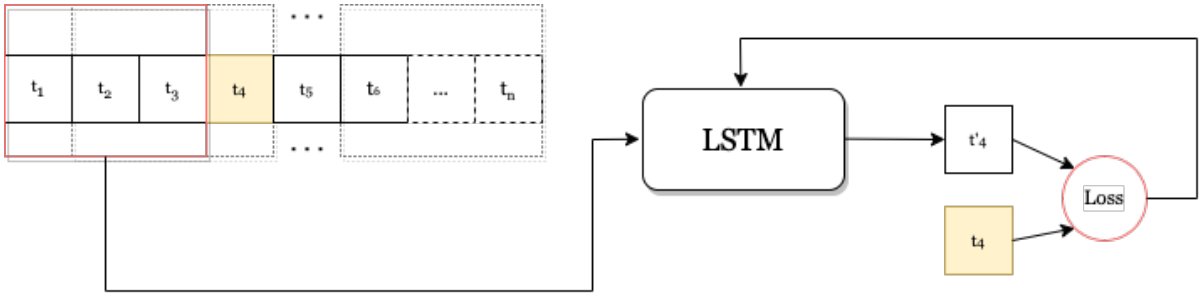


Figure 6 : schéma de principe de la prédiction de série temporelle par LSTM. Dans ce schéma on suppose que l'on connait 3 valeurs de la série temporelle, t1,t2 et t3, et on cherche à prédire t4. Ensuite, on déplace la fenêtre glissante afin de prédire le terme t5 et ainsi de suite.

Nous allons prendre un thème original pour illustrer cette partie…une série temporelle qui représente l'évolution de l'épidémie de COVID… Les données que nous allons traiter sont issues du site du Centre Européen de prévention et de contrôle des maladies :

https://opendata.ecdc.europa.eu/covid19/casedistribution/csv/data.csv

Tout le code décrit ci-dessous se trouve dans le fichier **`covid_lstm.py`**

Les données ne sont pas les plus récentes, mais vous pouvez, évidemment, traiter des données plus à jour. Le fichier CSV contient des statistiques sur le COVID en termes de nombre de cas et de décès dans chaque pays. Les détails sont spécifiés par date - d'où la possibilité de modéliser et d'apprendre à prédire l'évolution du nombre de cas/décès.

Le fichier contient 61900 lignes. Pour cet exercice, nous allons uniquement conserver les données relatives à la France et uniquement deux colonnes : la date et le nombre de cas. On précise également que la colonne **`dateRep`** contient un type date et on précise le bon format. Ce nouveau jeu de donnée contient 350 lignes (31/12/2019 -> 14/12/2020) :

```
covid_data = pd.read_csv("data.csv")
data = covid_data[covid_data['countryterritoryCode']=='FRA'][['dateRep', 'cases']]
data['dateRep'] = pd.to_datetime(data['dateRep'], format="%d/%m/%Y")
```

dateRep	cases
14/12/2020	11533
13/12/2020	13947
12/12/2020	13406
11/12/2020	13750
10/12/2020	14595
...	...
04/01/2020	0
03/01/2020	0
02/01/2020	0
01/01/2020	0
31/12/2019	0

Figure 7: Extrait de la BDD COVID

Les données ne contiennent maintenant que deux colonnes et uniquement les données de la France. L'objet de données peut être trié par ordre croissant de la date, de sorte que nous avons les dates de début en haut où les cas étaient principalement à 0. Nous allons également convertir la colonne de date en index.

```
data = data.sort_values(by="dateRep", key=pd.to_datetime)
data = data.set_index('dateRep')
```

	cases
dateRep	
31/12/2019	0
01/01/2020	0
01/02/2020	0
01/03/2020	43
01/04/2020	7578
...	...

Visualisons rapidement comment les cas se sont développés en France en 2020 :

```
fig = plt.gcf().set_size_inches(12,8)
plt.plot(data, label = 'Cas en France de COVID 19')
plt.show()
```

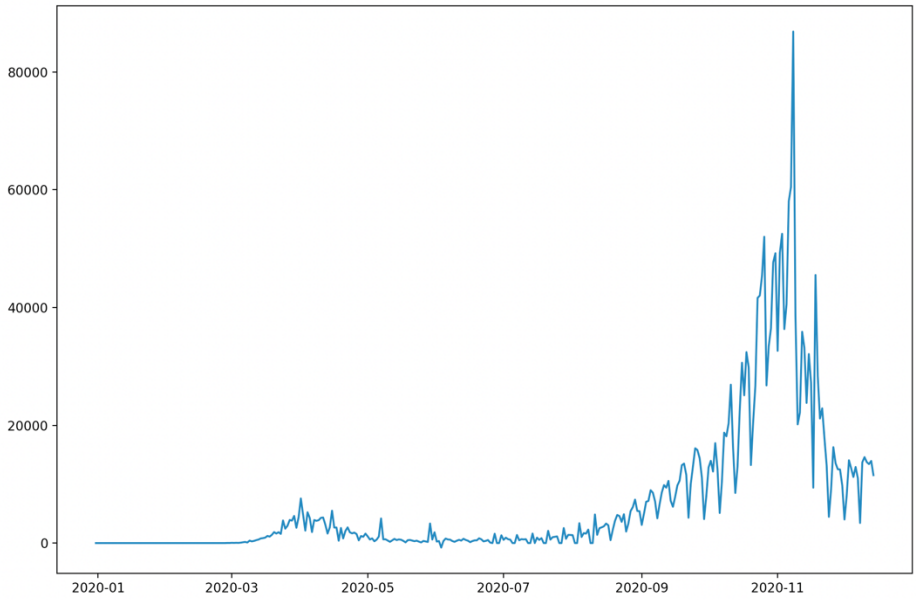


Figure 8: Cas COVID en France sur l'année 2020

Afin de réaliser la prédiction, nous allons devoir extraire des morceaux de séquences de la série temporelle. Par exemple, si nous avons la suite [1,2,3,4,5,6] et que nous avons une fenêtre glissante de 3j, nous souhaitons obtenir les séquences [1,2,3], [2,3,4], [3,4,5] et [4,5,6]. De plus, l'objectif est de prédire le jour qui suit la séquence, c'est-à-dire que la séquence [1,2,3] devra être suivie de 4.

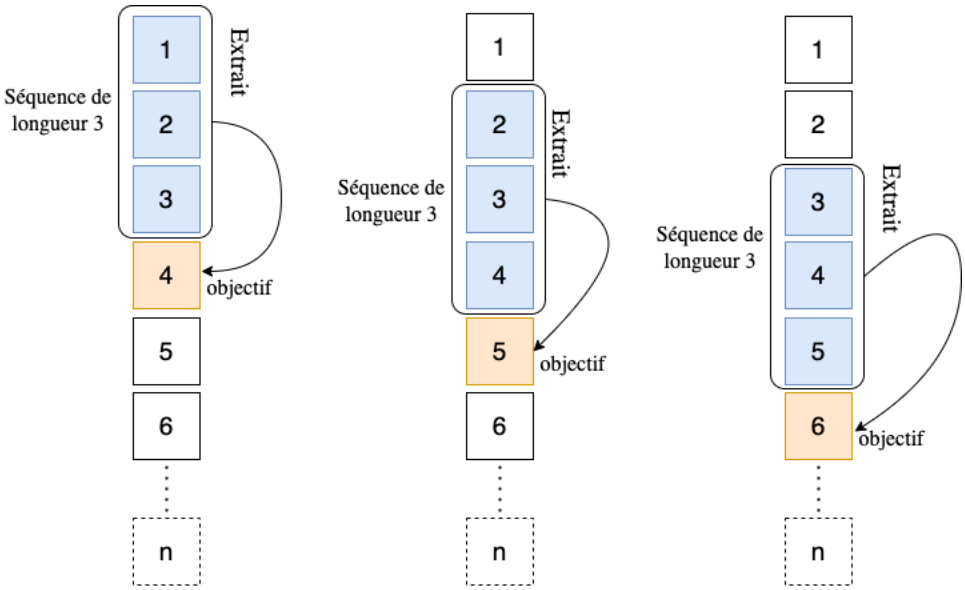


Figure 9: Fenêtre glissante ou extraits d'une longue séquence pour prédire l'élément suivant

Voici l'une des mises en œuvre possibles.

```
def extrait_seq(data, seq_length):
    extraits = []
    targets = []

    for i in range(len(data)-seq_length-1):
        extraits.append(data[i:(i+seq_length)])
        targets.append(data[i+seq_length])
    return np.array(extraits),np.array(targets)

extrait_seq([[1],[2],[3],[4],[5],[6],[7]],3)
```

```
(array([[[1], [2], [3]], [[2], [3], [4]], [[3], [4], [5]]]), array([[4], [5], [6]]))
```

Le prétraitement des données nécessite également de mettre les données à l'échelle aux bonnes valeurs. Nous pouvons utiliser **`MinMaxScaler.`** Nous utiliserons 80% des données comme ensemble d'entraînement et conserverons une longueur de séquence de 5.

```
sc = MinMaxScaler()
training_data = sc.fit_transform(data.values.copy())
seq_length = 5
x, y = extrait_seq(training_data, seq_length)
train_size = int(len(y) * 0.8)
test_size = len(y) - train_size
```

Avant d'aller plus loin, nous devons convertir les ensembles de données en tenseurs. Nous pouvons convertir les extraits et les cibles/objectifs jusqu'à l'index `train_size` pour les utiliser avec le réseau PyTorch.

```
dataX = Variable(torch.Tensor(np.array(x)))
dataY = Variable(torch.Tensor(np.array(y)))
trainX = Variable(torch.Tensor(np.array(x[0:train_size])))
trainY = Variable(torch.Tensor(np.array(y[0:train_size])))
testX = Variable(torch.Tensor(np.array(x[train_size:len(x)])))
testY = Variable(torch.Tensor(np.array(y[train_size:len(y)])))
```

Dans le réseau de neurones, nous allons définir une couche LSTM suivie d'une couche entièrement connectée. Dans PyTorch, LSTM est défini dans **`torch.nn`**. Il accepte des paramètres comme la taille de l'entrée, la taille de l'élément de l'état caché, et le nombre de couches récurrentes. Si **`num_layers`** est spécifié, le modèle empilera ce nombre de LSTMs ensemble afin que la sortie du premier LSTM soit fournie au deuxième LSTM, et ainsi de suite. Comme il s'agit d'un problème de régression, nous allons utiliser l'erreur quadratique comme critère d'optimisation.

Créons le modèle du réseau. Nous devrons définir les différentes couches. Nous allons garder une structure simple et avoir une couche LSTM, suivie d'une sortie entièrement connectée. Dans la méthode **`forward()`**, nous commencerons par créer un état caché et une mémoire cellulaire initialisée avec des zéros. La sortie du LSTM sera appliquée à la couche entièrement connectée et renvoyée comme sortie.

```
class CovidPrediction(nn.Module):
    def __init__(self, num_classes, input_size, hidden_size, num_layers):
        super(CovidPrediction, self).__init__()

        self.num_classes = num_classes
        self.num_layers = num_layers
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.seq_length = seq_length
        self.lstm = nn.LSTM(input_size=input_size,
hidden_size=hidden_size,num_layers=num_layers, batch_first=True)
        self.fully_connected = nn.Linear(hidden_size, num_classes)
```

```
    def forward(self, x):
        h0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size))
        c0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size))
        ula, (h_out, _) = self.lstm(x, (h0, c0))
        h_out = h_out.view(-1, self.hidden_size)
        out = self.fully_connected(h_out)
        return out
```

Nous pouvons définir un réseau de prédiction comme :

```
model = CovidPrediction(1, 1, 4, 1)
print(model)
```

```
CovidPrediction( (lstm): LSTM(1, 4, batch_first=True) (fully_connected):
Linear(in_features=4, out_features=1, bias=True) )
```

Le modèle est assez simple. Définissons le critère et d'optimisation ainsi que la méthode d'optimisation du gradient et exécutons la boucle d'apprentissage.

```
num_epochs = 1000
learning_rate = 0.01

model = CovidPrediction(1, 1, 4, 1)
criterion = torch.nn.MSELoss()

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

for epoch in range(num_epochs):
    outputs = model(trainX)
    loss = criterion(outputs, trainY)
    loss.backward()
    optimizer.step()
    if epoch % 100 == 0:
      print("Iteration: %d, loss:%f" % (epoch, loss.item()))
```

```
Iteration: 0, loss:0.107539
Iteration: 100, loss:0.001255
Iteration: 200, loss:0.000815
…
Iteration: 800, loss:0.000261
Iteration: 900, loss:0.000259
```

Une fois l'apprentissage terminé, vous pouvez prédire la valeur qui doit suivre chaque extrait de cinq valeurs. En d'autres termes, en regardant le nombre d'infections au COVID-19 de cinq jours consécutifs, nous allons prédire le nombre d'infections attendu le sixième jour.

Pour visualiser dans quelle mesure le modèle a prédit les cas futurs, prédisons pour tous les extraits possibles dans l'ensemble de données d'entraînement et comparons-les avec les valeurs réelles dans l'ensemble de données d'entraînement.

```
model.eval()
train_predict = model(dataX)
```

Nous devrons convertir les valeurs en tableaux NumPy afin de pouvoir afficher les courbes.

```
data_predict = train_predict.data.numpy()
data_actual = dataY.data.numpy()
```

Comme les données ont été précédemment mises à l'échelle en utilisant la transformation **MinMax**, nous devons appliquer la transformation inverse pour obtenir les valeurs réelles.

```
data_predict = sc.inverse_transform(data_predict)
data_actual = sc.inverse_transform(data_actual)
```

Visualisons et comparons. La Fig. 20 montre la sortie du code qui suit. Vous pouvez voir que dans la dernière partie, les résultats du modèle s'écartent de la réalité en raison des limites des valeurs dans les données d'apprentissage.

```
fig = plt.gcf().set_size_inches(12,8)
plt.plot(data_actual)
plt.plot(data_predict)
plt.suptitle('')
plt.legend(['Cas réels en 2020', 'Cas prédits (20% des derniers points ne sont pas dans
l\'ensemble d\'entrainement) '], loc='upper left')
plt.show()
```

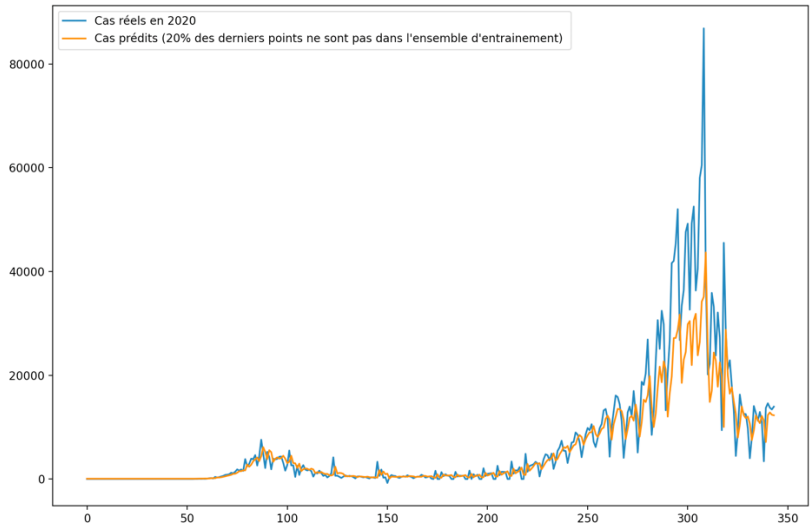


Figure 10 : Données réelles / Prédictions

Bien que le modèle semble fiable pour la plupart des données, en raison de la nature des données (les cas de COVID continuaient d'augmenter), le modèle n'est pas en mesure de capturer ces pics parce que (1) la mise à l'échelle limite « la compréhension du modèle » pour les valeurs très élevées et (2) il n'y avait pas assez de pics présents dans les données d'entraînement.

Travail à faire :
Question : Essayer d'entrainer le modèle sur un nombre d'époque plus important et voir si la prédiction est meilleure. Vous pouvez aussi augmenter la taille de l'ensemble d'entraînement.
Question : Une fois le modèle entrainé sur les données d'un pays, essayez, sans réentraîner le modèle, de prédire l'évolution des cas COVID pour un autre Pays. Par exemple, la Fig. 21 présente une prédiction sur les données de l'Inde mais l'entraînement a été réalisé sur les données USA.

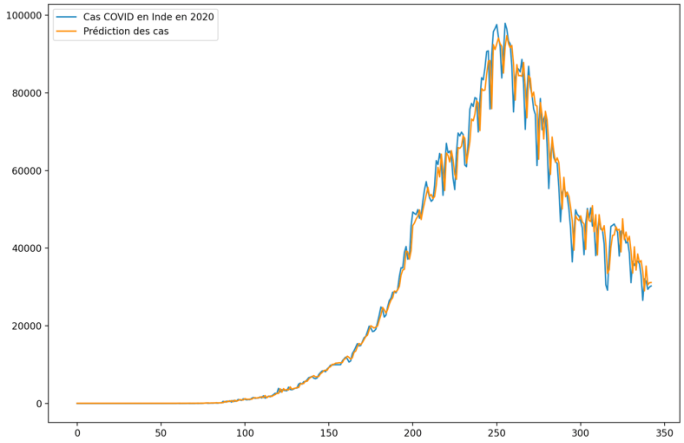


Figure 11 : Entrainement sur l'ensemble des cas USA et évaluation du modèle entrainé sur les données de l'Inde