SAISON 22/23

Formation

Introduction au Deep Learning

Séquence 7

Un petit détour par PyTorch

FIDLE

SAISON 22/23

Cette session va être enregistrée.
Retrouvez-nous sur notre chaine YouTube :-)

This session will be recorded.
Find us on our YouTube channel :-)

https://fidle.cnrs.fr/youtube

SAISON 22/23

Formation

Introduction au Deep Learning

Séquence 7

Un petit détour par PyTorch

FIDLE

Ressources

https://fidle.cnrs.fr

Course materials (pdf)

Practical work environment*

Corrected notebooks

Videos (YouTube)

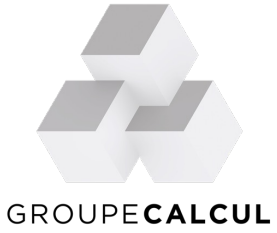
(*) Procedure via Docket or pip
Remember to get the latest version !

List and networks

You can also subscribe to :

http://fidle.cnrs.fr/listeinfo
Fidle information list

https://listes.services.cnrs.fr/wws/info/devlog
List of ESR* « calcul » group



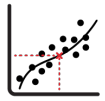
https://listes.math.cnrs.fr/wws/info/calcul
List of ESR* « calcul » group

(*) ESR is Enseignement Supérieur et Recherche, french universities and public academic research organizations

Program

1

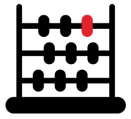
History, Fundamental Concepts


Basic Regression
DNN


Basic Classification
DNN

2 3

Hight Dimensionnal Data
CNN

4

Demystify **mathematics** for neural networks.

5

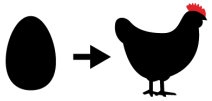
Training strategies
Evaluation


Sparse data (text)
Embedding

6

Sequences data
RNN

7
PyTorch
A small detour with **PyTorch**.

8

«Attention is All You Need»
Transformers

9
New !

Graph Neural Network
GNN

10

Autoencoder networks
AE

11

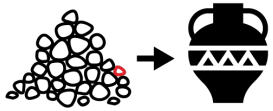
Variational Antoencoder
VAE

12

Project session
«My project in 180 s»

13

Generative Adversarial Networks
GAN

14
New !

Diffusion Model
Text to image

15

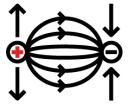
AI, Law, Society and **Ethics**

16
New !

Model and training optimization
Resource efficiency

17

Jean-Zay
GPU acceleration

18
New !

Physics-Informed Neural Networks
PINNS

19

Deep Reinforcement Learning
RL

20

What will be tomorrow's AI
Review & perspectives !

20
Séquences
du 17
novembre
au 14 mai
2023

FIDLE

SAISON
22/23

Roadmap

7

PyTorch

A small detour with **PyTorch**.

7.1 Comprendre PyTorch

- Objet pytorch.tensor
- Différentiation automatique
- Calcul GPU

7.2 Etude d'un exemple

7.3 Aller plus loin…

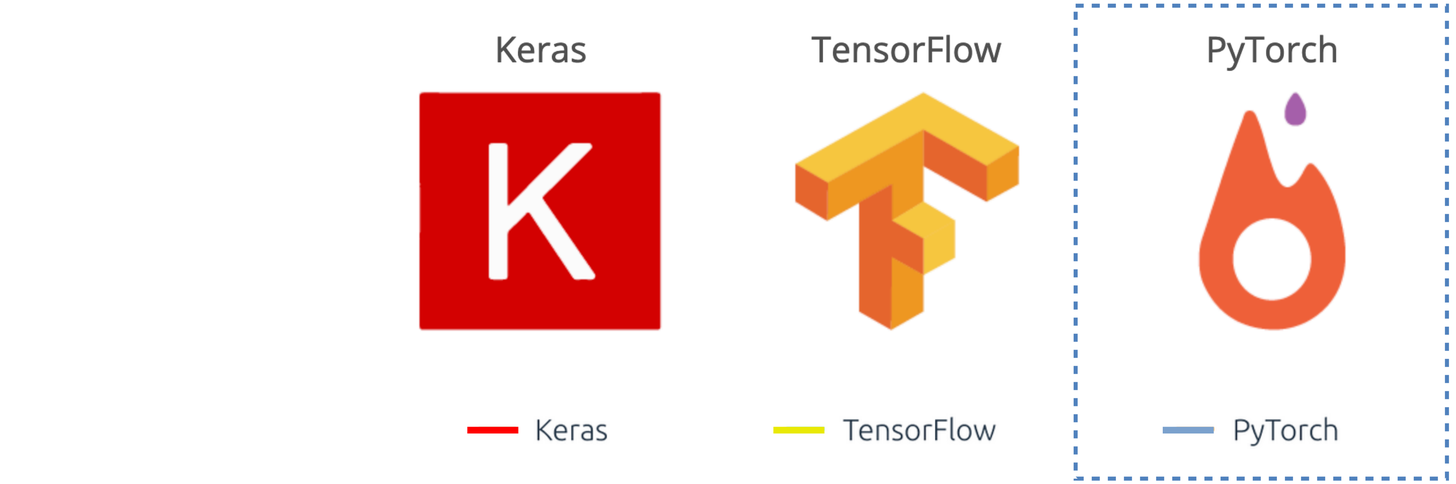
- Charger et adapter une architecture
- Récupérer les paramètres du réseau
- N'apprendre qu'une sous-partie du réseau
- Définir ses propres couches
- Récupérer les données modifiées dans le réseau

Laurent Risser (CNRS – Institut de Mathématiques de Toulouse – 3IA ANITI)

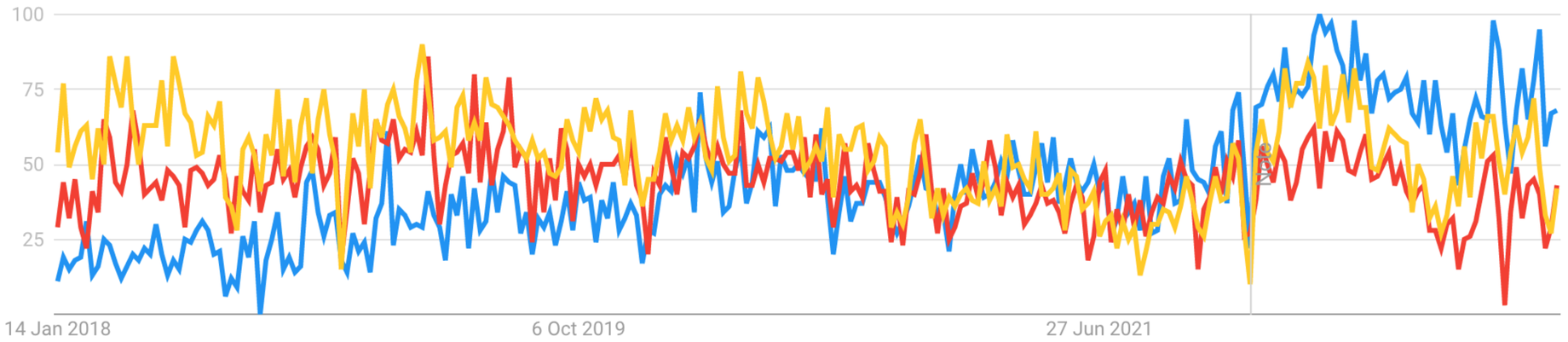
Introduction

FIDLE

Keras TensorFlow PyTorch



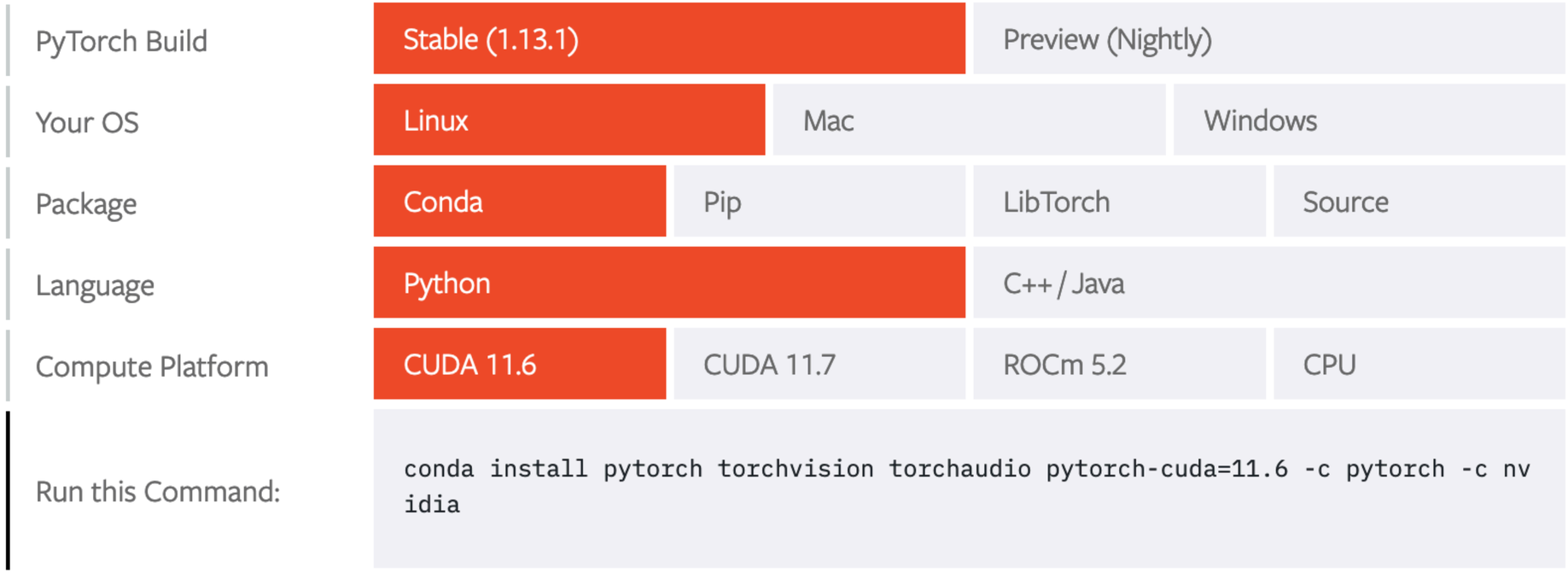
Google Trends (France)



11 Jan 2023

→ Complexité intermédiaire entre TensorFlow et Keras
→ Rendu open-source par Facebook's AI sur GitHub en 2017
→ Relativement simple si les concepts principaux des réseaux de neurones sont assimilés
→ Très flexible
→ Plus apprécié des *méthodologistes* que des personnes visant une mise en production rapide

→ Trouver la bonne ligne de commande : https://pytorch.org/get-started/locally/



Introduction : Comparaison rapide avec Keras

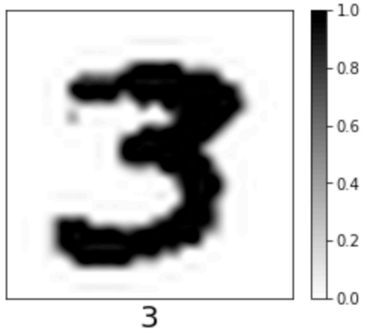
fidle/MNIST/01-DNN-MNIST.ipynb

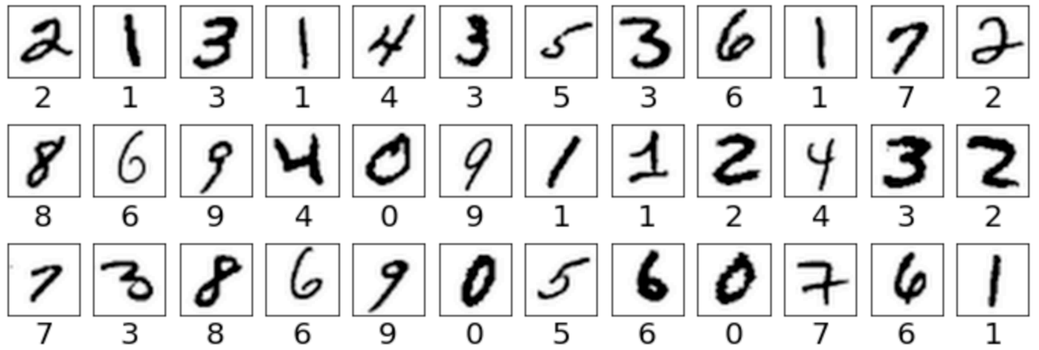
```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

xmax=x_train.max()
x_train = x_train / xmax
x_test  = x_test  / xmax

pwk.plot_images(x_train, y_train, [27],  x_size=5,y_size=5, colorbar=True, save_as='01-one-digit')
pwk.plot_images(x_train, y_train, range(5,41), columns=12, save_as='02-many-digits')
```

fidle/MNIST_PyTorch/01-DNN-MNIST_PyTorch.ipynb

```
mnist_trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=None)
x_train=mnist_trainset.data.type(torch.DoubleTensor)
y_train=mnist_trainset.targets
xmax=x_train.max()
x_train = x_train / xmax
x_test  = x_test  / xmax
np_x_train=x_train.numpy().astype(np.float64)
np_y_train=y_train.numpy().astype(np.uint8)
ooo.plot_images(np_x_train,np_y_train , [27],  x_size=5,y_size=5, colorbar=True)
ooo.plot_images(np_x_train,np_y_train, range(5,41), columns=12)
```

Assez similaire… mais utilise autre chose que des numpy.arrays





fidle/MNIST/01-DNN-MNIST.ipynb

```
hidden1     = 100
hidden2     = 100

model = keras.Sequential([
    keras.layers.Input((28,28)),
    keras.layers.Flatten(),
    keras.layers.Dense( hidden1, activation='relu'),
    keras.layers.Dense( hidden2, activation='relu'),
    keras.layers.Dense( 10,      activation='softmax')
])

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

Plus verbeux mais potentiellement plus flexible

fidle/MNIST_PyTorch/01-DNN-MNIST_PyTorch.ipynb

```
class MyModel(nn.Module):
    """
    Basic fully connected neural-network
    """
    def __init__(self):
        hidden1     = 100
        hidden2     = 100
        super(MyModel, self).__init__()
        self.hidden1 = nn.Linear(784, hidden1)
        self.hidden2 = nn.Linear(hidden1, hidden2)
        self.hidden3 = nn.Linear(hidden2, 10)

    def forward(self, x):
        x = x.view(-1,784)
        x = self.hidden1(x)
        x = F.relu(x)
        x = self.hidden2(x)
        x = F.relu(x)
        x = self.hidden3(x)
        x = F.softmax(x, dim=0)
        return x

model = MyModel()
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=1e-3)
```

Introduction : Comparaison rapide avec Keras

fidle/MNIST/01-DNN-MNIST.ipynb

```
batch_size  = 512
epochs      =  16

history = model.fit(  x_train, y_train,
                      batch_size      = batch_size,
                      epochs          = epochs,
                      verbose         = 1,
                      validation_data = (x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
```

fidle/MNIST_PyTorch/01-DNN-MNIST_PyTorch.ipynb

```
def fit(model,X_train,Y_train,X_test,Y_test, EPOCHS = 5, BATCH_SIZE = 32):

    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(),lr=1e-3) #lr is the learning rate
    model.train()

    history=convergence_history_CrossEntropyLoss()

    history.update(model,X_train,Y_train,X_test,Y_test)

    n=X_train.shape[0] #number of observations in the training data

    #stochastic gradient descent
    for epoch in range(EPOCHS):

        batch_start=0
        epoch_shuffler=np.arange(n)
        np.random.shuffle(epoch_shuffler) #remark that 'utilsData.DataLoader' could be used instead

        while batch_start+BATCH_SIZE < n:
            #get mini-batch observation
            mini_batch_observations = epoch_shuffler[batch_start:batch_start+BATCH_SIZE]
            var_X_batch = Variable(X_train[mini_batch_observations,:,:]).float() #the input image is flattened
            var_Y_batch = Variable(Y_train[mini_batch_observations])

            #gradient descent step
            optimizer.zero_grad()               #set the parameters gradients to 0
            Y_pred_batch = model(var_X_batch)   #predict y with the current NN parameters

            curr_loss = loss(Y_pred_batch, var_Y_batch)  #compute the current loss
            curr_loss.backward()                         #compute the loss gradient w.r.t. all NN parameters
            optimizer.step()                             #update the NN parameters

            #prepare the next mini-batch of the epoch
            batch_start+=BATCH_SIZE

        history.update(model,X_train,Y_train,X_test,Y_test)

    return history


batch_size  = 512
epochs      = 128


history=fit(model,x_train,y_train,x_test,y_test,EPOCHS=epochs,BATCH_SIZE = batch_size)


var_x_test = Variable(x_test[:,:,:]).float()
var_y_test = Variable(y_test[:])
y_pred = model(var_x_test)

loss = nn.CrossEntropyLoss()
curr_loss = loss(y_pred, var_y_test)

val_loss = curr_loss.item()
val_accuracy  = float( (torch.argmax(y_pred, dim= 1) == var_y_test).float().mean() )
```

Définitivement plus verbeux… mais :

- Permet de facilement rentrer dans des détails algorithmique
- Donne un bon accès aux données intermédiaires

1 : Comprendre PyTorch

FIDLE

PyTorch tensors

- Visiblement très proche d'un numpy array, mais dans PyTorch
- Peut être utilisé pour :
  - Rétro-propager le calcul d'un gradient (voir partie 1.2)
  - Transférer facilement des données sur un GPU (voir partie 1.3)

Instantiation :

```
import torch
torch.tensor([[1,-2],[3,4]])
```

```
tensor([[ 1, -2],
        [ 3,  4]])
```

```
import numpy as np
torch.tensor(np.array([[1,-2],[3,4]]))
```

```
tensor([[ 1, -2],
        [ 3,  4]])
```

```
torch.zeros([2,4],dtype=torch.float32)
```

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

Accès à une valeur :

```
A=torch.tensor([[1,-2],[3,4]])
print(A[0,1])
```

```
tensor(-2)
```

```
print(A[0,1].item())
```

```
-2
```

Conversion en numpy :

```
print(A.numpy())
```

```
[[ 1 -2]
 [ 3  4]]
```

Echanger deux axes (transposée) :

```
A=torch.randn([10,20,30])
print(A.shape)
```

```
torch.Size([10, 20, 30])
```

```
A_t=torch.transpose(A,0,2)
print(A.shape)
print(A_t.shape)
```

```
torch.Size([10, 20, 30])
torch.Size([30, 20, 10])
```

Equivalent du « Reshape » :

```
A=torch.randint(0,10,[4,4])
print(A)
```

```
tensor([[7, 6, 2, 9],
        [4, 1, 0, 9],
        [9, 3, 9, 6],
        [3, 2, 2, 5]])
```

```
print(A.view([2,8]))
```

```
tensor([[7, 6, 2, 9, 4, 1, 0, 9],
        [9, 3, 9, 6, 3, 2, 2, 5]])
```

```
print(A.view([-1,2,2]))
```

```
tensor([[[7, 6],
         [2, 9]],

        [[4, 1],
         [0, 9]],

        [[9, 3],
         [9, 6]],

        [[3, 2],
         [2, 5]]])
```

```
print(A.flatten())
```

```
tensor([7, 6, 2, 9, 4, 1, 0, 9, 9, 3, 9, 6, 3,
2, 2, 5])
```

Changer le type des données (cast) :

```
A=torch.randn([2,2],dtype=torch.float32)
```

```
print(A.type())
```

```
torch.FloatTensor
```

```
B=A.type(torch.DoubleTensor)
print(A.type())
print(B.type())
```

```
torch.FloatTensor
torch.DoubleTensor
```

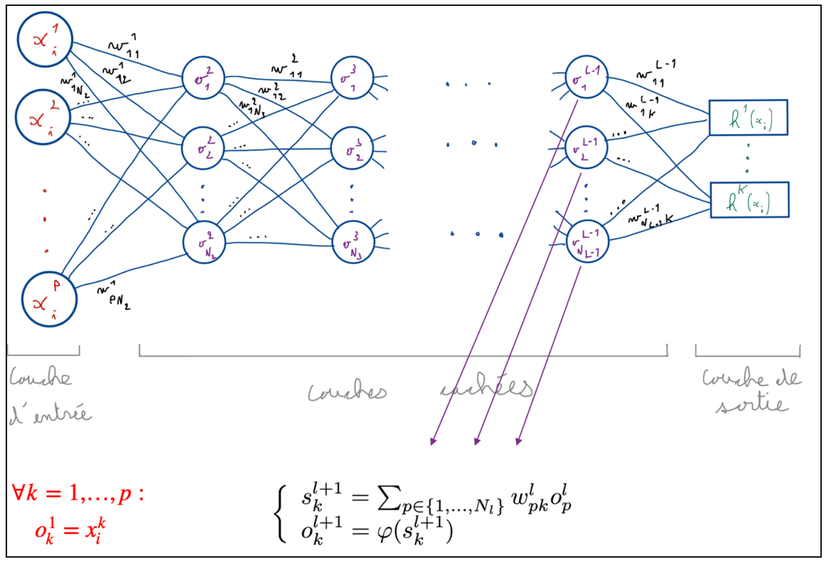
```
C=A.int()
print(A.type())
print(C.type())
```

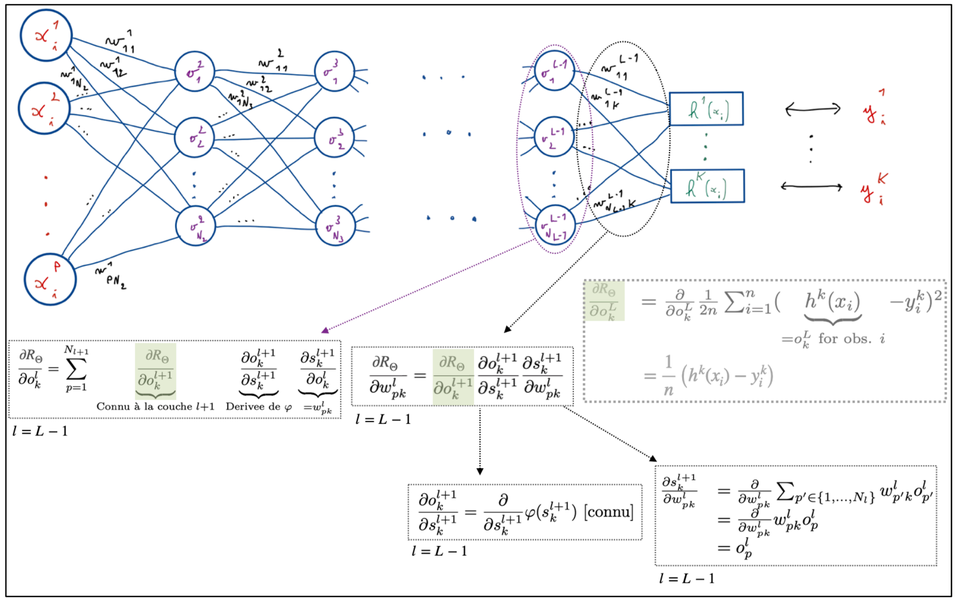
```
torch.FloatTensor
torch.IntTensor
```

Pour aller plus loin : https://pytorch.org/docs/stable/tensors.html

```python
class MyModel(nn.Module):
    """
    Basic fully connected neural-network
    """
    def __init__(self):
        hidden1      = 100
        hidden2      = 100
        super(MyModel, self).__init__()
        self.hidden1 = nn.Linear(784, hidden1)
        self.hidden2 = nn.Linear(hidden1, hidden2)
        self.hidden3 = nn.Linear(hidden2, 10)

    def forward(self, x):
        x = x.view(-1,784)
        x = self.hidden1(x)
        x = F.relu(x)
        x = self.hidden2(x)
        x = F.relu(x)
        x = self.hidden3(x)
        x = F.softmax(x, dim=0)
        return x
```

Modèle « forward » bien défini





Calcul du gradient du loss par rapport aux paramètres du modèle grâce à la différentiation automatique !

Exemple élémentaire (régression linéaire)

- Estimation $\hat{y}$ de la note à un futur concours en fonction de notes déjà obtenues en Français (X^1) et en Mathématiques (X^2).
- On suppose que $\hat{y}$ est la moyenne pondérée de X^1 et $X^2 \rightarrow \ \hat{y} = f_\Theta(X) = w_1 X^1 + w_2 X^2$.
- Les poids w_1 et w_2 sont appris sur un jeu de données $\{X_i, y_i\}_{i=1,\ldots,n}$ contenant les notes obtenues l'an dernier.

Apprentissage des paramètres w_1 et w_2

- On minimise par descente de gradient $R(w_1, w_2) = \sum_{i=1}^{n} \left(\hat{y}_i - y_i\right)^2 = \sum_{i=1}^{n} \left((w_1 X_i^1 + w_2 X_i^2) - y\right)^2$
- Calcul du gradient $\nabla R(w_1, w_2) = \left(\frac{\partial R}{\partial w_1}, \frac{\partial R}{\partial w_2}\right)^\intercal$ par différentiation automatique

```
import torch
```

```
X = torch.randn([5,2], requires_grad=False)
w = torch.randn([2], requires_grad=True)

y_pred=w[0]*X[:,0]+w[1]*X[:,1]
#y_pred.retain_grad()

print('X :',X)
print('w :',w)
print('y_pred :',y_pred)


print('X.grad :',X.grad)
print('w.grad :',w.grad)
print('y_pred.grad :',y_pred.grad)
```

```
X : tensor([[ 0.2590,  1.1822],
        [ 1.8838,  0.7159],
        [ 0.3180, -0.9601],
        [ 0.4899,  2.0838],
        [ 1.2439,  0.1527]])
w : tensor([0.3187, 1.3714], requires_grad=True)
y_pred : tensor([ 1.7039,  1.5821, -1.2153,  3.013
9,  0.6058], grad_fn=<AddBackward0>)
X.grad : None
w.grad : None
y_pred.grad : None
```

```
y_true=torch.zeros([5], requires_grad=False)

print('y_true :',y_true)
```

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

```
Risk=(torch.pow(y_pred-y_true,2.)).mean()
```

```
Risk.backward()
```

```
print('X.grad :',X.grad)
```

```
X.grad : None
```

```
print('y_pred.grad :',y_pred.grad)
```

```
y_pred.grad : None
```

```
print('w.grad :',w.grad)
```

```
w.grad : tensor([2.1062, 4.2746])
```

```
import torch
```

```
X = torch.randn([5,2], requires_grad=False)
w = torch.randn([2], requires_grad=True)

y_pred=w[0]*X[:,0]+w[1]*X[:,1]
y_pred.retain_grad()

print('X :',X)
print('w :',w)
print('y_pred :',y_pred)


print('X.grad :',X.grad)
print('w.grad :',w.grad)
print('y_pred.grad :',y_pred.grad)
```

```
X : tensor([[ 0.2590,  1.1822],
        [ 1.8838,  0.7159],
        [ 0.3180, -0.9601],
        [ 0.4899,  2.0838],
        [ 1.2439,  0.1527]])
w : tensor([0.3187, 1.3714], requires_grad=True)
y_pred : tensor([ 1.7039,  1.5821, -1.2153,  3.013
9,  0.6058], grad_fn=<AddBackward0>)
X.grad : None
w.grad : None
y_pred.grad : None
```

```
y_true=torch.zeros([5], requires_grad=False)

print('y_true :',y_true)
```

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

```
Risk=(torch.pow(y_pred-y_true,2.)).mean()
```

```
Risk.backward()
```

```
print('X.grad :',X.grad)
```

```
X.grad : None
```

```
print('y_pred.grad :',y_pred.grad)
```

```
y_pred.grad : tensor([ 0.6815,  0.6328, -0.4861,
1.2056,  0.2423])
```

```
print('w.grad :',w.grad)
```

```
w.grad : tensor([2.1062, 4.2746])
```

Utilisation de detach() : pour qu'un tenseur ne soit plus qu'une constante (on ne calcule plus son gradient)

```
x=torch.ones(10, requires_grad=True)

y=x**2
z=x**3

r=(y+z).sum()
r.backward()
print(x.grad)
```

```
tensor([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.])
```

```
x=torch.ones(10, requires_grad=True)

y=x**2
z=x.detach()**3

r=(y+z).sum()
r.backward()

print(x.grad)
```

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

Utilisation de torch.no_grad() : pour ne calculer aucun gradient et limiter la mémoire nécessaire

```
x = torch.ones(3, requires_grad=True)
y = x**2
z = x**3
r = (y+z).sum()
r.backward()
print('r :',r)
print('x.grad :',x.grad)
```

```
r : tensor(6., grad_fn=<SumBackward0>)
x.grad : tensor([5., 5., 5.])
```

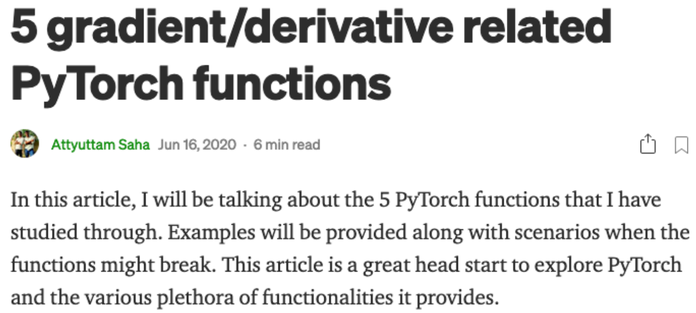
```
x = torch.ones(3, requires_grad=True)
with torch.no_grad():
    y = x**2
    z = x**3
    r = (y+z).sum()
print('r :',r)
print('x.grad :',x.grad)
```

```
r : tensor(6.)
x.grad : None
```

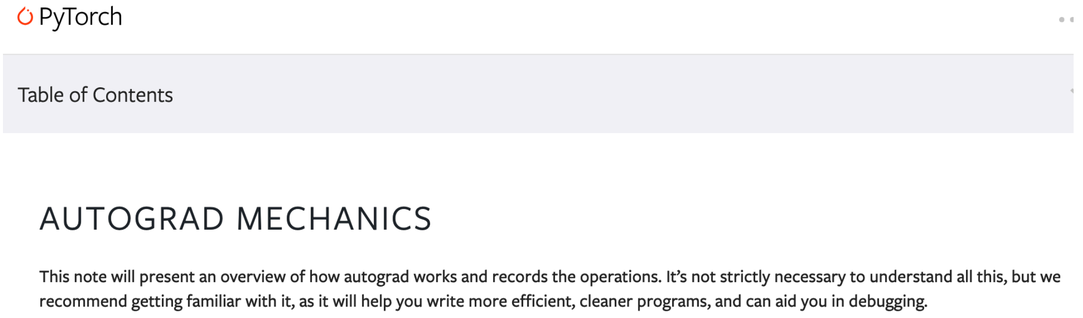
```
r.backward()
```

```
---------------------------------------------------------------------
RuntimeError                                Traceback
(most recent call last)
<ipython-input-232-d0c0300fbddc> in <module>
----> 1 r.backward()
```

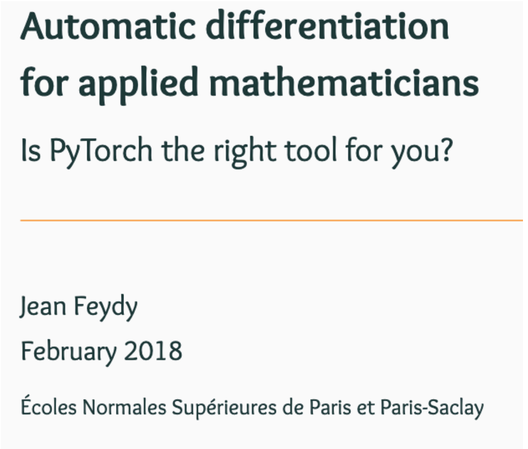
Mieux comprendre la différentiation automatique



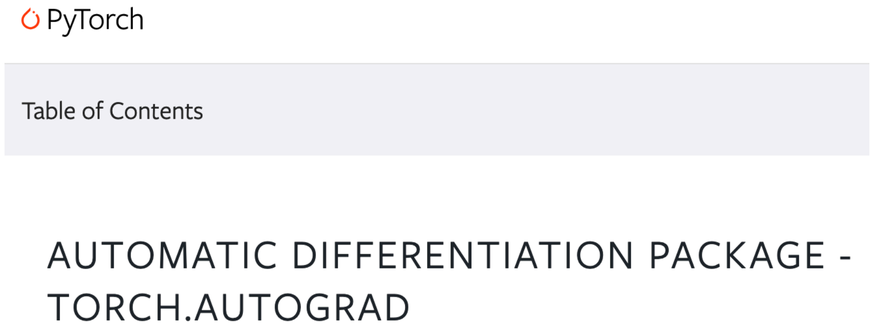
medium.com/@attyuttam/5-gradient-derivative-related-pytorch-functions-8fd0e02f13c6



pytorch.org/docs/stable/notes/autograd.html



www.jeanfeydy.com/Talks/autodiff_appliedmaths/AutoDiff_AppliedMaths.pdf



pytorch.org/docs/stable/autograd.html

Voir si une carte GPU (Nvidia) est disponible :

```
import torch

print('Cuda disponible : ',torch.cuda.is_available())
print('Nombre de devices : ',torch.cuda.device_count())
print('Device utilisé : ',torch.cuda.current_device())
print('Nom du device utilisé : ',torch.cuda.get_device_name(0))
```

```
Cuda disponible :  True
Nombre de devices :  1
Device utilisé :  0
Nom du device utilisé :  Tesla K80
```

Mécanisme de transferts similaire à ce qui se fait usuellement avec CUDA :

```
import torch
import torchvision.models as models
```

```
import torch.nn as nn

h_resnet18 = models.resnet18()
h_resnet18.fc = nn.Linear(512,5)

d_resnet18 = h_resnet18.cuda()
```

```
h_simulated_data = torch.randn([1000,3,64,64])

d_mini_batch = h_simulated_data[0:10,:,:,:].cuda()
```

```
d_outputs = d_resnet18(d_mini_batch)

h_outputs = d_outputs.cpu()

print(h_outputs)
```

```
tensor([[ 0.2605, -0.4549, -0.5667,  0.8204, -0.4016],
        [ 0.5252, -0.7144, -0.4012,  0.7757, -0.8269],
        [ 0.6739, -0.2131, -1.0644,  1.0779, -0.7419],
                                ...
        [ 0.7548,  0.0197, -0.2869,  1.0721, -0.6495]],
       grad_fn=<ToCopyBackward0>)
```

Peut être rendu générique (CPU ou GPU) :

```
import torch
import torchvision.models as models

DEVICE = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(DEVICE)
```

```
cuda:0
```

```
import torch.nn as nn

h_resnet18 = models.resnet18()
h_resnet18.fc = nn.Linear(512,5)

d_resnet18 = h_resnet18.to(DEVICE)
```

```
h_simulated_data = torch.randn([1000,3,64,64])

d_mini_batch = h_simulated_data[0:10,:,:,:].to(DEVICE)
```

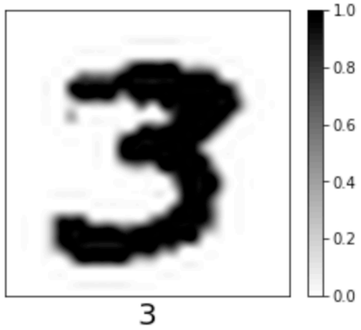
```
d_outputs = d_resnet18(d_mini_batch)

h_outputs = d_outputs.to('cpu')

print(h_outputs)
```

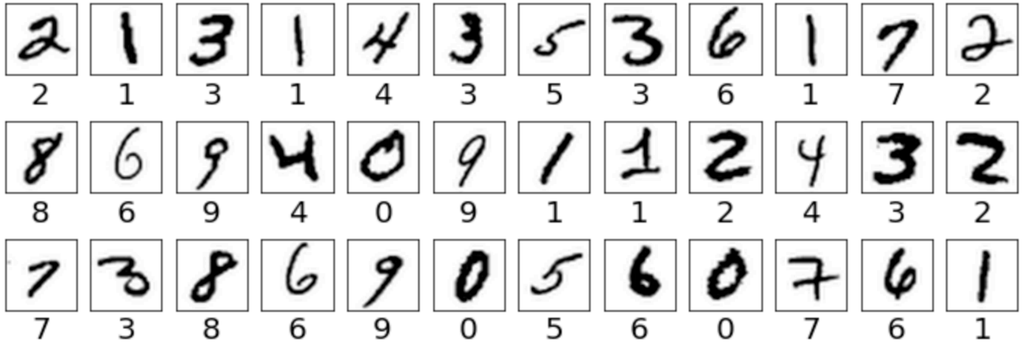
```
tensor([[ 0.2605, -0.4549, -0.5667,  0.8204, -0.4016],
        [ 0.5252, -0.7144, -0.4012,  0.7757, -0.8269],
        [ 0.6739, -0.2131, -1.0644,  1.0779, -0.7419],
                              ...
        [ 0.7548,  0.0197, -0.2869,  1.0721, -0.6495]],
       grad_fn=<ToCopyBackward0>)
```

2 : Etude d’un exemple

FIDLE

```
#get and format the training set
mnist_trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=None)
x_train=mnist_trainset.data.type(torch.DoubleTensor)
y_train=mnist_trainset.targets


#get and format the test set
mnist_testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=None)
x_test=mnist_testset.data.type(torch.DoubleTensor)
y_test=mnist_testset.targets
```





```
class MyModel(nn.Module):
    """
    Basic fully connected neural-network
    """
    def __init__(self):
        hidden1     = 100
        hidden2     = 100
        super(MyModel, self).__init__()
        self.hidden1 = nn.Linear(784, hidden1)
        self.hidden2 = nn.Linear(hidden1, hidden2)
        self.hidden3 = nn.Linear(hidden2, 10)

    def forward(self, x):
        x = x.view(-1,784)   #flatten the images before using fully-connected layers
        x = self.hidden1(x)
        x = F.relu(x)
        x = self.hidden2(x)
        x = F.relu(x)
        x = self.hidden3(x)
        x = F.softmax(x, dim=0)
        return x

model = MyModel().to(DEVICE)
```

```python
def fit(model,X_train,Y_train,X_test,Y_test, EPOCHS = 5, BATCH_SIZE = 32):
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(),lr=1e-3) #lr is the learning rate
    model.train()

    n=X_train.shape[0] #number of observations in the training data

    #stochastic gradient descent
    for epoch in range(EPOCHS):
        batch_start=0
        epoch_shuffler=np.arange(n)
        np.random.shuffle(epoch_shuffler) #remark that 'utilsData.DataLoader' could be used instead

        while batch_start+BATCH_SIZE < n:
            #get mini-batch observation
            mini_batch_observations = epoch_shuffler[batch_start:batch_start+BATCH_SIZE]
            var_X_batch = X_train[mini_batch_observations,:,:].float().to(DEVICE)
            var_Y_batch = Y_train[mini_batch_observations]

            #gradient descent step
            optimizer.zero_grad()               #set the parameters gradients to 0
            Y_pred_batch = model(var_X_batch)   #predict y with the current NN parameters
            curr_loss = loss(Y_pred_batch.to('cpu'), var_Y_batch)  #compute the current loss
            curr_loss.backward()                                   #compute the loss gradient
            optimizer.step()                                       #update the NN parameters

            #prepare the next mini-batch of the epoch
            batch_start+=BATCH_SIZE
```

```python
def fit(model,X_train,Y_train,X_test,Y_test, EPOCHS = 5, BATCH_SIZE = 32):
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(),lr=1e-3) #lr is the learning rate
    model.train()

    n=X_train.shape[0] #number of observations in the training data

    #stochastic gradient descent
    for epoch in range(EPOCHS):
        batch_start=0
        epoch_shuffler=np.arange(n)
        np.random.shuffle(epoch_shuffler) #remark that 'utilsData.DataLoader' could be used instead

        while batch_start+BATCH_SIZE < n:
            #get mini-batch observation
            mini_batch_observations = epoch_shuffler[batch_start:batch_start+BATCH_SIZE]
            var_X_batch = X_train[mini_batch_observations,:,:].float().to(DEVICE)
            var_Y_batch = Y_train[mini_batch_observations]

            #gradient descent step
            optimizer.zero_grad()               #set the parameters gradients to 0
            Y_pred_batch = model(var_X_batch)   #predict y with the current NN parameters
            curr_loss = loss(Y_pred_batch.to('cpu'), var_Y_batch)  #compute the current loss
            curr_loss.backward()                                   #compute the loss gradient
            optimizer.step()                                       #update the NN parameters

            #prepare the next mini-batch of the epoch
            batch_start+=BATCH_SIZE
```

- Définition de la fonction *loss*
- Définition de la *stratégie d'optimisation (remarque : accède aux paramètres du modèle)*

```python
def fit(model,X_train,Y_train,X_test,Y_test, EPOCHS = 5, BATCH_SIZE = 32):
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(),lr=1e-3) #lr is the learning rate
    model.train()

    n=X_train.shape[0] #number of observations in the training data

    #stochastic gradient descent
    for epoch in range(EPOCHS):
        batch_start=0
        epoch_shuffler=np.arange(n)
        np.random.shuffle(epoch_shuffler) #remark that 'utilsData.DataLoader' could be used instead

        while batch_start+BATCH_SIZE < n:
            #get mini-batch observation
            mini_batch_observations = epoch_shuffler[batch_start:batch_start+BATCH_SIZE]
            var_X_batch = X_train[mini_batch_observations,:,:].float().to(DEVICE)
            var_Y_batch = Y_train[mini_batch_observations]

            #gradient descent step
            optimizer.zero_grad()               #set the parameters gradients to 0
            Y_pred_batch = model(var_X_batch)   #predict y with the current NN parameters
            curr_loss = loss(Y_pred_batch.to('cpu'), var_Y_batch)  #compute the current loss
            curr_loss.backward()                                   #compute the loss gradient
            optimizer.step()                                       #update the NN parameters

            #prepare the next mini-batch of the epoch
            batch_start+=BATCH_SIZE
```

Précise que l'on va entrainer le modèle

- Il existe le mode train() et le mode eval()
- self.training est True par défaut

```python
def fit(model,X_train,Y_train,X_test,Y_test, EPOCHS = 5, BATCH_SIZE = 32):
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(),lr=1e-3) #lr is the learning rate
    model.train()

    n=X_train.shape[0] #number of observations in the training data

    #stochastic gradient descent
    for epoch in range(EPOCHS):
        batch_start=0
        epoch_shuffler=np.arange(n)
        np.random.shuffle(epoch_shuffler) #remark that 'utilsData.DataLoader' could be used instead

        while batch_start+BATCH_SIZE < n:
            #get mini-batch observation
            mini_batch_observations = epoch_shuffler[batch_start:batch_start+BATCH_SIZE]
            var_X_batch = X_train[mini_batch_observations,:,:].float().to(DEVICE)
            var_Y_batch = Y_train[mini_batch_observations]

            #gradient descent step
            optimizer.zero_grad()               #set the parameters gradients to 0
            Y_pred_batch = model(var_X_batch)   #predict y with the current NN parameters
            curr_loss = loss(Y_pred_batch.to('cpu'), var_Y_batch)  #compute the current loss
            curr_loss.backward()                                   #compute the loss gradient
            optimizer.step()                                       #update the NN parameters

            #prepare the next mini-batch of the epoch
            batch_start+=BATCH_SIZE
```

Gestion des *epochs* et du choix des observations dans chaque *mini-batch*

```python
def fit(model,X_train,Y_train,X_test,Y_test, EPOCHS = 5, BATCH_SIZE = 32):
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(),lr=1e-3) #lr is the learning rate
    model.train()

    n=X_train.shape[0] #number of observations in the training data

    #stochastic gradient descent
    for epoch in range(EPOCHS):
        batch_start=0
        epoch_shuffler=np.arange(n)
        np.random.shuffle(epoch_shuffler) #remark that 'utilsData.DataLoader' could be used instead

        while batch_start+BATCH_SIZE < n:
            #get mini-batch observation
            mini_batch_observations = epoch_shuffler[batch_start:batch_start+BATCH_SIZE]
            var_X_batch = X_train[mini_batch_observations,:,:].float().to(DEVICE)
            var_Y_batch = Y_train[mini_batch_observations]

            #gradient descent step
            optimizer.zero_grad()               #set the parameters gradients to 0
            Y_pred_batch = model(var_X_batch)   #predict y with the current NN parameters
            curr_loss = loss(Y_pred_batch.to('cpu'), var_Y_batch)  #compute the current loss
            curr_loss.backward()                                   #compute the loss gradient
            optimizer.step()                                       #update the NN parameters

            #prepare the next mini-batch of the epoch
            batch_start+=BATCH_SIZE
```

Récupération des observations nécessaires pour le mini-batch de l'itération courante

```python
def fit(model,X_train,Y_train,X_test,Y_test, EPOCHS = 5, BATCH_SIZE = 32):
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(),lr=1e-3) #lr is the learning rate
    model.train()

    n=X_train.shape[0] #number of observations in the training data

    #stochastic gradient descent
    for epoch in range(EPOCHS):
        batch_start=0
        epoch_shuffler=np.arange(n)
        np.random.shuffle(epoch_shuffler) #remark that 'utilsData.DataLoader' could be used instead

        while batch_start+BATCH_SIZE < n:
            #get mini-batch observation
            mini_batch_observations = epoch_shuffler[batch_start:batch_start+BATCH_SIZE]
            var_X_batch = X_train[mini_batch_observations,:,:].float().to(DEVICE)
            var_Y_batch = Y_train[mini_batch_observations]

            #gradient descent step
            optimizer.zero_grad()                #set the parameters gradients to 0
            Y_pred_batch = model(var_X_batch)    #predict y with the current NN parameters
            curr_loss = loss(Y_pred_batch.to('cpu'), var_Y_batch)  #compute the current loss
            curr_loss.backward()                                   #compute the loss gradient
            optimizer.step()                                       #update the NN parameters

            #prepare the next mini-batch of the epoch
            batch_start+=BATCH_SIZE
```

- Mise à zéro des gradients de paramètres du modèle
- Prédiction des sorties avec les paramètres du modèle courant (→calcul de $\{h_\Theta(x_i)\}_{i\in B}$)
- Calcul du *loss* pour toutes les observations du mini-batch
- Calcul des gradients (backpropagation) puis mise à jour des paramètres en fonction des gradients

Simple DNN classification using PyTorch

Notebook : 01-DNN-MNIST_PyTorch-2022.ipynb

Objective :

Donner un exemple d'utilisation élémentaire de PyTorch

Dataset :

MNIST que l'on ne présente plus :-)

3 : Aller plus loin …

FIDLE

→ Poids tirés au hasard

```
import torchvision.models as models
resnet18 = models.resnet18()
alexnet = models.alexnet()
vgg16 = models.vgg16()
squeezenet = models.squeezenet1_0()
densenet = models.densenet161()
inception = models.inception_v3()
googlenet = models.googlenet()
shufflenet = models.shufflenet_v2_x1_0()
mobilenet = models.mobilenet_v2()
resnext50_32x4d = models.resnext50_32x4d()
wide_resnet50_2 = models.wide_resnet50_2()
mnasnet = models.mnasnet1_0()
```

→ Poids pré-entrainés

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
squeezenet = models.squeezenet1_0(pretrained=True)
vgg16 = models.vgg16(pretrained=True)
densenet = models.densenet161(pretrained=True)
inception = models.inception_v3(pretrained=True)
googlenet = models.googlenet(pretrained=True)
shufflenet = models.shufflenet_v2_x1_0(pretrained=True)
mobilenet = models.mobilenet_v2(pretrained=True)
resnext50_32x4d = models.resnext50_32x4d(pretrained=True)
wide_resnet50_2 = models.wide_resnet50_2(pretrained=True)
mnasnet = models.mnasnet1_0(pretrained=True)
```

3.1 Aller plus loin — Charger et adapter une architecture

```python
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
```

```python
print(resnet18)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
                    ⋮                              ⋮                              ⋮

     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

← **Entrée : Image 2D à 3 canaux**

→ **Sortie : Vecteur de taille 1000**

Besoin d'adapter la première et dernière couche si par exemple (MNIST)

- Les entrées n'ont qu'un canal
- Les sorties sont en dimension 10

Remarque : il existe un package « torchsummary » pour résumer l'architecture comme en Keras

3.1 Aller plus loin — Charger et adapter une architecture

```python
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
```

```python
print(resnet18)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
                    ⋮                          ⋮                          ⋮

     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

Entrée : Image 2D à 3 canaux

Sortie : Vecteur de taille 10

Modification des sorties (basique)

```python
import torch.nn as nn

output_classes=10
resnet18.fc = nn.Linear(512, output_classes)
```

3.1 Aller plus loin — Charger et adapter une architecture

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
```

```
print(resnet18)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
                    ⋮                          ⋮                          ⋮

    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Sequential(
    (fc1): Linear(in_features=512, out_features=128, bias=True)
    (relu): ReLU()
    (fc2): Linear(in_features=128, out_features=10, bias=True)
    (output): Sigmoid()
  )
)
```

Entrée : Image 2D à 3 canaux

Sortie : Vecteur de taille 10

Modification des sorties (avancé) :

```
import torch.nn as nn
from collections import OrderedDict

output_classes=10

resnet18.fc = nn.Sequential(OrderedDict([
    ('fc1', nn.Linear(512, 128)), ('relu', nn.ReLU()),
    ('fc2', nn.Linear(128, output_classes)), ('output', nn.Sigmoid())
]))
```

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
```

```
print(resnet18)
```

```
ResNet(
  (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
                    ⋮                    ⋮                    ⋮

    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Sequential(
)   (fc1): Linear(in_features=512, out_features=128, bias=True)
    (relu): ReLU()
    (fc2): Linear(in_features=128, out_features=10, bias=True)
    (output): Sigmoid()
  )
)
```

Entrée : Image 2D à 1 canal

Sortie : Vecteur de taille 10

Modification des entrées :

```
resnet18.conv1=nn.Conv2d(1, 64, kernel_size=(7, 7),  stride=(2, 2), padding=(3, 3), bias=False)
```

3.1 Aller plus loin — Charger et adapter une architecture

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
```

```
print(resnet18)
```

```
ResNet(
  (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Identity()
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                    ⋮                          ⋮                          ⋮
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Sequential(
)   (fc1): Linear(in_features=512, out_features=128, bias=True)
    (relu): ReLU()
    (fc2): Linear(in_features=128, out_features=10, bias=True)
    (output): Sigmoid()
  )
)
```

Entrée : Image 2D à 1 canal

Sortie : Vecteur de taille 10

Simplification du réseau :

```
class Identity(nn.Module):
    def __init__(self):
        super(Identity, self).__init__()
    def forward(self, x):
        return x
resnet18.layer1=Identity()
```

Poids du réseau sous la forme d'un dictionnaire : « resnet18.state_dict() »

```
for param_tensor in resnet18.state_dict():
    print(param_tensor, "\t", resnet18.state_dict()[param_tensor].size())
```

```
conv1.weight     torch.Size([64, 1, 7, 7])
bn1.weight       torch.Size([64])
bn1.bias         torch.Size([64])
bn1.running_mean         torch.Size([64])
bn1.running_var          torch.Size([64])
bn1.num_batches_tracked          torch.Size([])
layer2.0.conv1.weight    torch.Size([128, 64, 3, 3])
layer2.0.bn1.weight      torch.Size([128])
layer2.0.bn1.bias        torch.Size([128])
    ⋮                     ⋮                    ⋮
layer4.1.conv2.weight    torch.Size([512, 512, 3, 3])
layer4.1.bn2.weight      torch.Size([512])
layer4.1.bn2.bias        torch.Size([512])
layer4.1.bn2.running_mean        torch.Size([512])
layer4.1.bn2.running_var         torch.Size([512])
layer4.1.bn2.num_batches_tracked         torch.Size([])
fc.fc1.weight    torch.Size([128, 512])
fc.fc1.bias      torch.Size([128])
fc.fc2.weight    torch.Size([10, 128])
fc.fc2.bias      torch.Size([10])
```

Poids du réseau sous la forme d'un dictionnaire : « resnet18.state_dict() »

```
resnet_dict=resnet18.state_dict()

print(resnet_dict['layer2.0.conv2.weight'])
```

```
tensor([[[[-0.0074, -0.0098,  0.0028],
          [-0.0108,  0.0258,  0.0455],
          [-0.0272,  0.0053,  0.0132]],

         [[ 0.0354,  0.0251,  0.0078],
          [ 0.0040,  0.0199,  0.0274],
          [ 0.0353,  0.0355,  0.0133]],

         [[ 0.0193, -0.0213, -0.0362],
          [-0.0196, -0.0189, -0.0595],
          [-0.0218, -0.0077,  0.0039]],

         ...,

         [[-0.0068,  0.0108, -0.0037],
          [ 0.0135,  0.0114, -0.0013],
          [ 0.0081,  0.0002,  0.0006]],

         [[ 0.0077,  0.0077,  0.0044],
```

```
print(resnet_dict['fc.fc2.bias'])
```

```
tensor([ 0.0310, -0.0075, -0.0521,  0.0285,  0.0192, -0.08
11,  0.0776,  0.0501,
        -0.0611,  0.0259])
```

Fixer (freezer) l'apprentissage de paramètres spécifiques

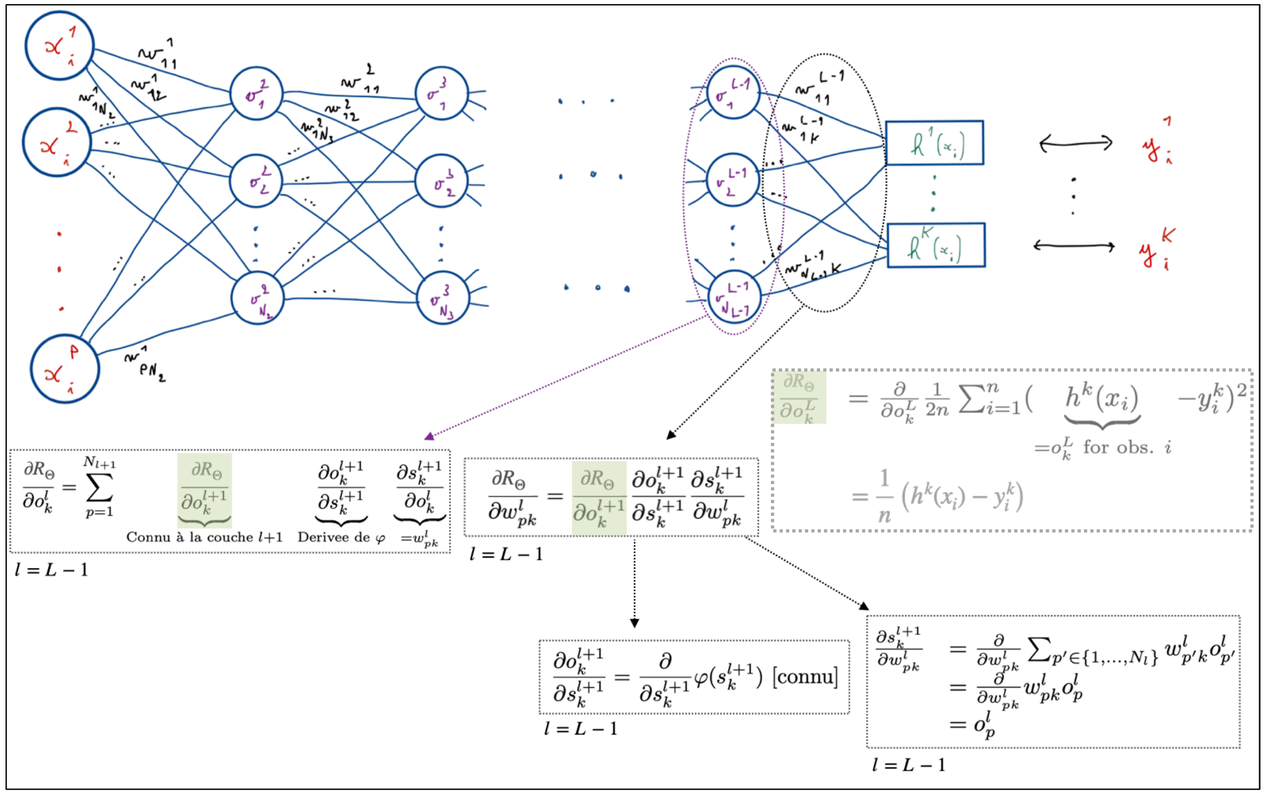
```
for param in resnet18.conv1.parameters():
    param.requires_grad = False

for name, param in resnet18.named_parameters():
        print(name, param.requires_grad)
```

```
conv1.weight False
bn1.weight True
bn1.bias True
layer2.0.conv1.weight True
layer2.0.bn1.weight True
layer2.0.bn1.bias True
layer2.0.conv2.weight True
layer2.0.bn2.weight True
layer2.0.bn2.bias True
layer2.0.downsample.0.weight True
   ⋮        ⋮        ⋮
layer4.1.bn2.weight True
layer4.1.bn2.bias True
fc.fc1.weight True
fc.fc1.bias True
fc.fc2.weight True
fc.fc2.bias True
```

Fixer (freezer) l'apprentissage de paramètres spécifiques

```
for name, param in resnet18.named_parameters():
    if name[:3]!='fc.':
        param.requires_grad = False

for name, param in resnet18.named_parameters():
        print(name, param.requires_grad)
```

```
conv1.weight False
bn1.weight False
bn1.bias False
layer2.0.conv1.weight False
layer2.0.bn1.weight False
layer2.0.bn1.bias False
layer2.0.conv2.weight False
layer2.0.bn2.weight False
layer2.0.bn2.bias False
   ⋮        ⋮        ⋮
layer4.1.bn2.weight False
layer4.1.bn2.bias False
fc.fc1.weight True
fc.fc1.bias True
fc.fc2.weight True
fc.fc2.bias True
```

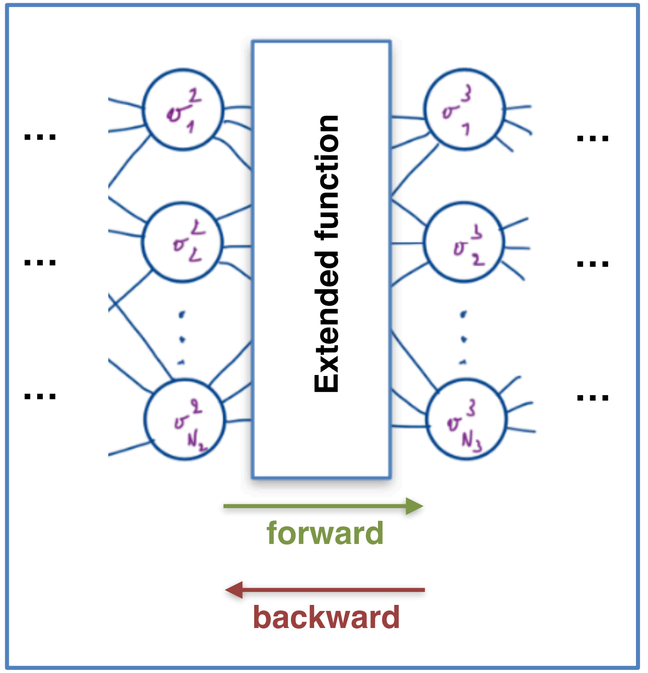


On défini : $\text{out} = f\left(\text{in}, \text{param}_1, \cdots, \text{param}_p\right)$

On connait : $\frac{\partial R}{\partial \text{out}}$

- $\frac{\partial R}{\partial \text{in}} = \frac{\partial R}{\partial \text{out}} \frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial R}{\partial \text{out}} \frac{\partial f\left(\text{in}, \text{param}_1, \cdots, \text{param}_p\right)}{\partial \text{in}}$
- …
- $\frac{\partial R}{\partial \text{param}_p} = \frac{\partial R}{\partial \text{out}} \frac{\partial \text{out}}{\partial \text{param}_p} = \frac{\partial R}{\partial \text{out}} \frac{\partial f\left(\text{in}, \text{param}_1, \cdots, \text{param}_p\right)}{\partial \text{param}_p}$

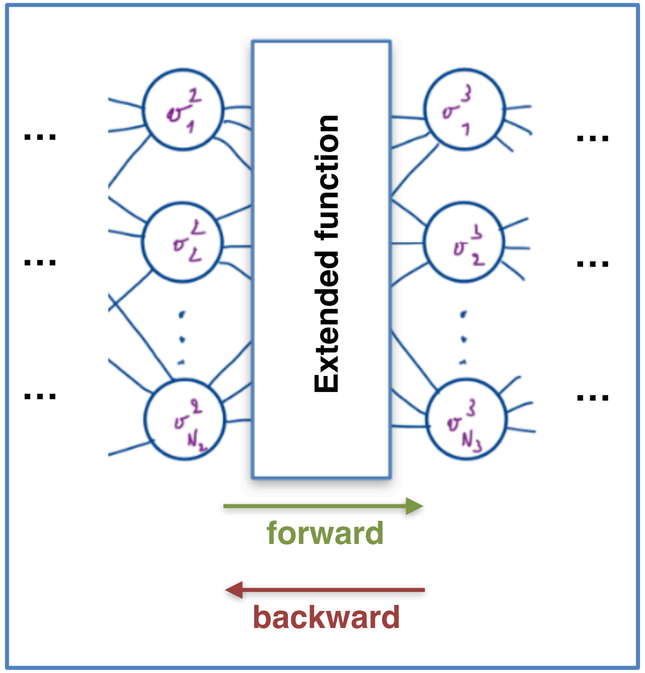
```python
from torch.autograd import Function

class MyReLU(Function):
    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input

relu = MyReLU.apply
```



https://pytorch.org/docs/stable/notes/extending.html

https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

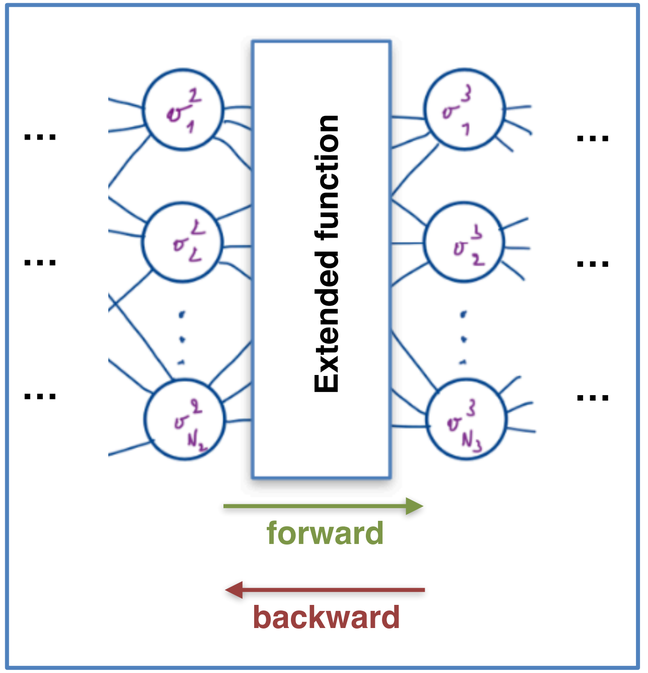
```python
from torch.autograd import Function

class MulConstant(Function):
    @staticmethod
    def forward(ctx, tensor, constant):
        ctx.constant = constant
        return tensor * constant

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output * ctx.constant, None

mulCst = MulConstant.apply
```



https://pytorch.org/docs/stable/notes/extending.html

https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

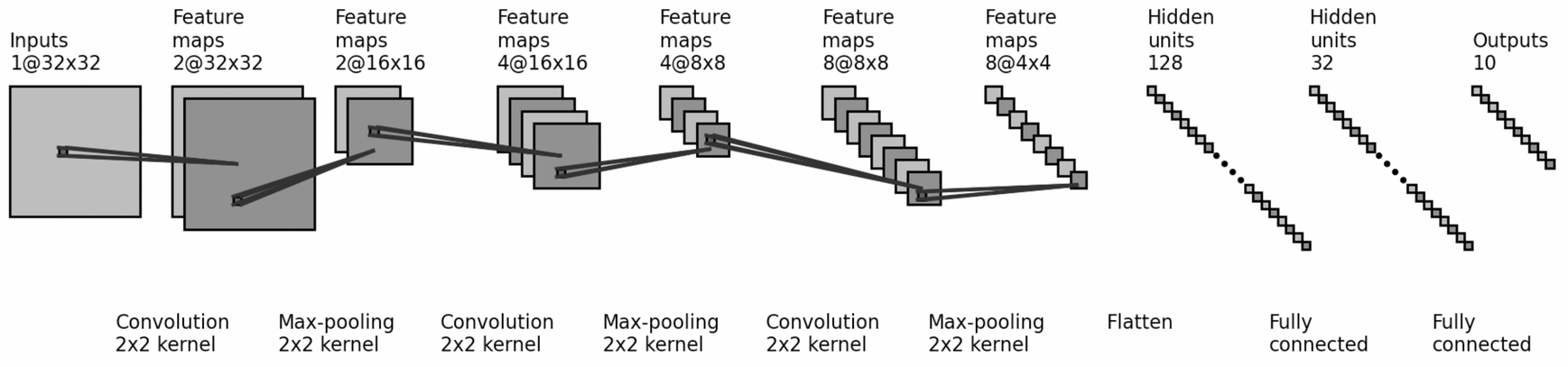
```python
class LinearFunction(Function):

    @staticmethod
    def forward(ctx, input, weight, bias):
        ctx.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        output += bias.unsqueeze(0).expand_as(output)
        return output

    @staticmethod
    def backward(ctx, grad_output):
        input, weight, bias = ctx.saved_tensors

        grad_input = grad_output.mm(weight)
        grad_weight = grad_output.t().mm(input)
        grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias

linear = LinearFunction.apply
```



https://pytorch.org/docs/stable/notes/extending.html

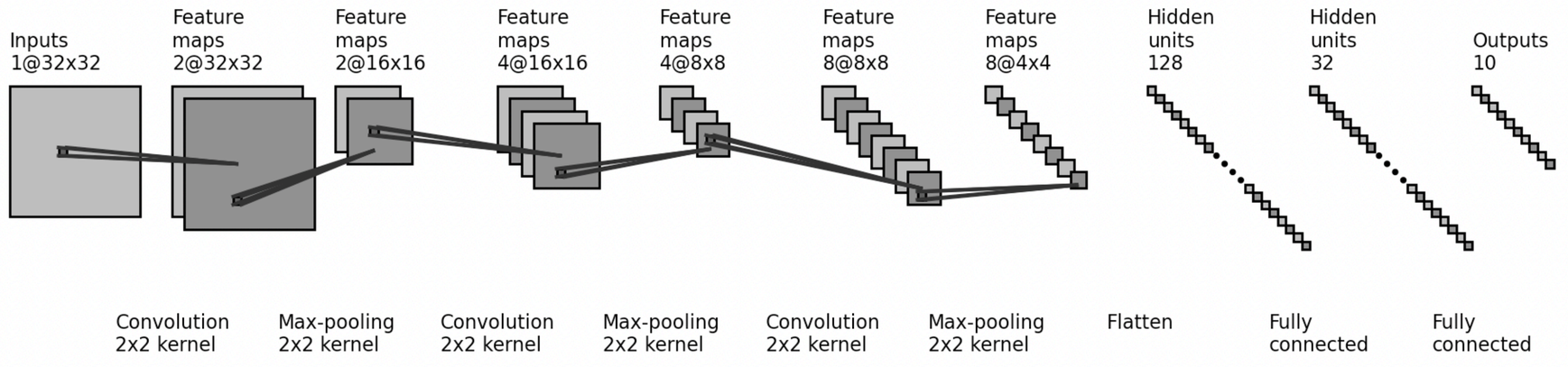
https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

3.5 Aller plus loin — Récupérer les valeurs des neurones




```
class basicCNN(nn.Module):
    def __init__(self):
        super(basicCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 2, kernel_size=2, stride=1, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(2, 4, kernel_size=2, stride=1, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(4, 8, kernel_size=2, stride=1, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(8 * 4 * 4, 32)
        self.fc2 = nn.Linear(32, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = F.relu(self.conv3(x))
        x = self.pool3(x)
        x = x.view(-1, 8*4*4)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return(x)


basicCNN_model = basicCNN()
```

3.5 Aller plus loin — Récupérer les valeurs des neurones



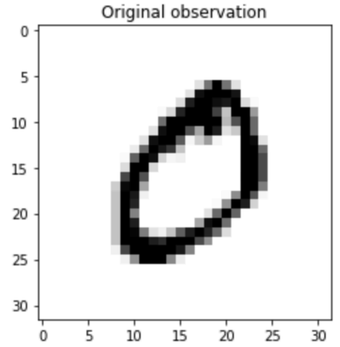

```
#get 100 observations
var_X = Variable(torch_X_train[0:100,:,:].view(-1,1,32,32)).float()

#Create a hook function that will get the transformed data in an hidden
#layer. These data will be stored in the 'activation' dictionary
activation = {}
def get_activation(name):
    def hook(model, input, output):
        activation[name] = output.detach()
    return hook

#Add hooks at the output of different parts of the model.
#-> The hook location is in "[HERE].register_forward_hook(...)"
#-> "get_activation('[HERE]')" is the key where the data are saved
basicCNN_model.pool1.register_forward_hook(get_activation('pool1'))
basicCNN_model.pool2.register_forward_hook(get_activation('pool2'))
basicCNN_model.pool3.register_forward_hook(get_activation('pool3'))
basicCNN_model.fc1.register_forward_hook(get_activation('fc1'))

#propagate the model and get the hooks
output = basicCNN_model(var_X)
```

```
print(activation['pool1'].shape)
print(activation['pool2'].shape)
print(activation['pool3'].shape)
print(activation['fc1'].shape)
```

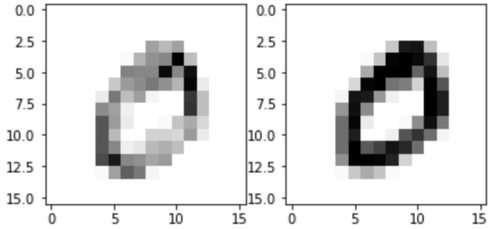
```
torch.Size([100, 2, 16, 16])
torch.Size([100, 4, 8, 8])
torch.Size([100, 8, 4, 4])
torch.Size([100, 32])
```

3.5 Aller plus loin — Récupérer les valeurs des neurones

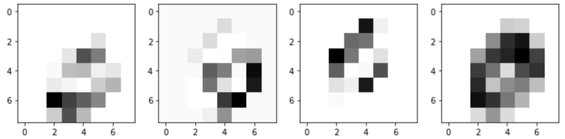
```
image_ID=1
plt.imshow(torch_X_train[image_ID,:,:], cmap='Greys')
plt.colorbar()
plt.title('Original observation')
plt.show()
```




```
fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, sharex=True)
ax0.imshow(activation['pool1'][image_ID,0,:,:], cmap='Greys')
ax1.imshow(activation['pool1'][image_ID,1,:,:], cmap='Greys')
plt.show()
```

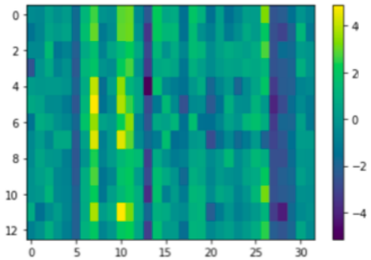


```
fig, (ax0, ax1,ax2, ax3) = plt.subplots(nrows=1, ncols=4,figsize=(12, 6))
ax0.imshow(activation['pool2'][image_ID,0,:,:], cmap='Greys')
ax1.imshow(activation['pool2'][image_ID,1,:,:], cmap='Greys')
ax2.imshow(activation['pool2'][image_ID,2,:,:], cmap='Greys')
ax3.imshow(activation['pool2'][image_ID,3,:,:], cmap='Greys')
plt.show()
```



⋮ ⋮ ⋮ ⋮

```
Obs_with_IDeq0=np.where(var_y[:].numpy()==0)[0]

print('Lattent space:')
plt.imshow(activation['fc1'][Obs_with_IDeq0,:] , aspect='auto')
plt.colorbar()
plt.show()
```



Little things and concepts to keep in mind

- PyTorch c'est :
  - puissant
  - flexible
  - basé sur l'objet torch.tensor et une série de fonctions
- L'utilisation de la rétropropagation peut y être totallement sous jacente…
- … mais ce framework permet aussi d'accéder, de controller et de modifier des aspects fins des DNNs !

Next, on Fidle :

Jeudi 17 janvier, 14h

Séquence n°8

Attention Is All You Need", quand les Transformers changent la donne !

Utilisation et l'architecture classique des transformers ,
Principe du mécanisme d'attention et du multi-head attention
Architectures des transformers (auto-regressive, auto-encoding et encoder decoder)
Pré-entraînement (BERT et GPT)
Fine tuning
Utilisation des transformers dans les autres domaines
Durée : 2h

FIDLE

Next, on Fidle :

Jeudi 17 janvier, 14h

Séquence n° 8

Attention Is All You Need", quand les Transformers changent la donne !

To be continued...

contact@fidle.cnrs.fr
FIDLE https://fidle.cnrs.fr
YouTube https://fidle.cnrs.fr/youtube

FIDLE