DEVLOG RÉSEAU DU DEVELOPPEMENT LOGICIEL Recherche et Enseignement Supérieur

cnrs dépasser les frontières www.cnrs.fr

INSTITUT de MATHEMATIQUES de TOULOUSE

ANITI ARTIFICIAL & NATURAL INTELLIGENCE TOULOUSE INSTITUTE

cimi Centre International de Mathématiques et d'Informatique TOULOUSE

Introduction au Deep Learning avec PyTorch

Partie 3 : PyTorch

Laurent Risser

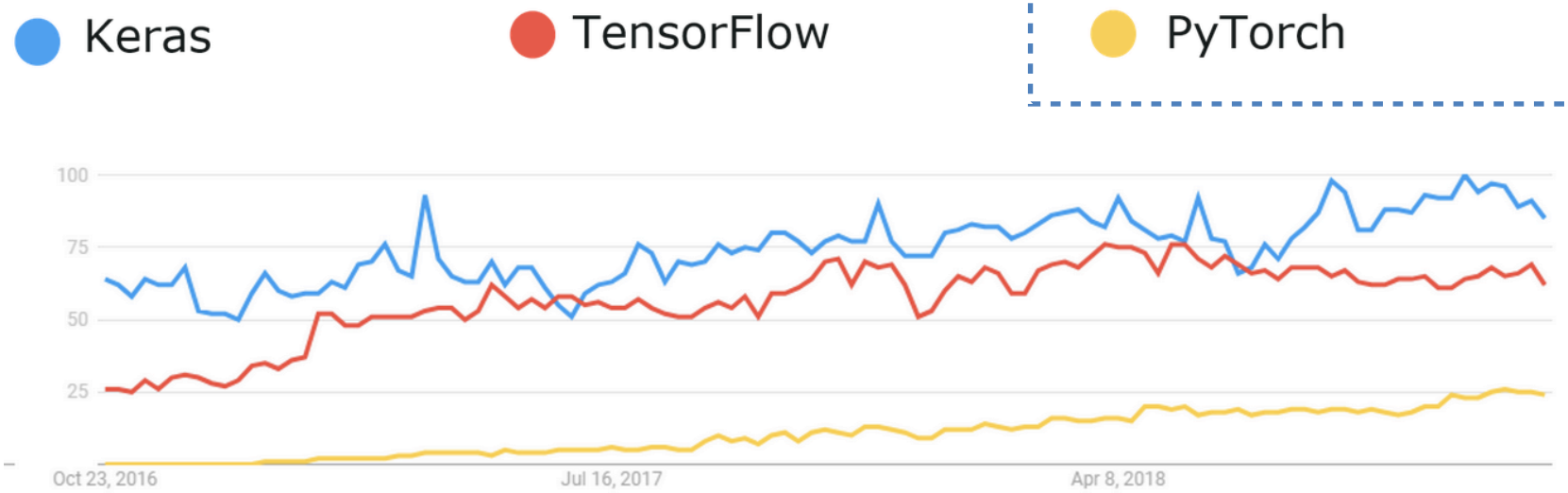
Ingénieur de Recherche à l'Institut de Mathématiques de Toulouse et au 3IA ANITI

lrisser@math.univ-toulouse.fr

Keras

TensorFlow

PyTorch

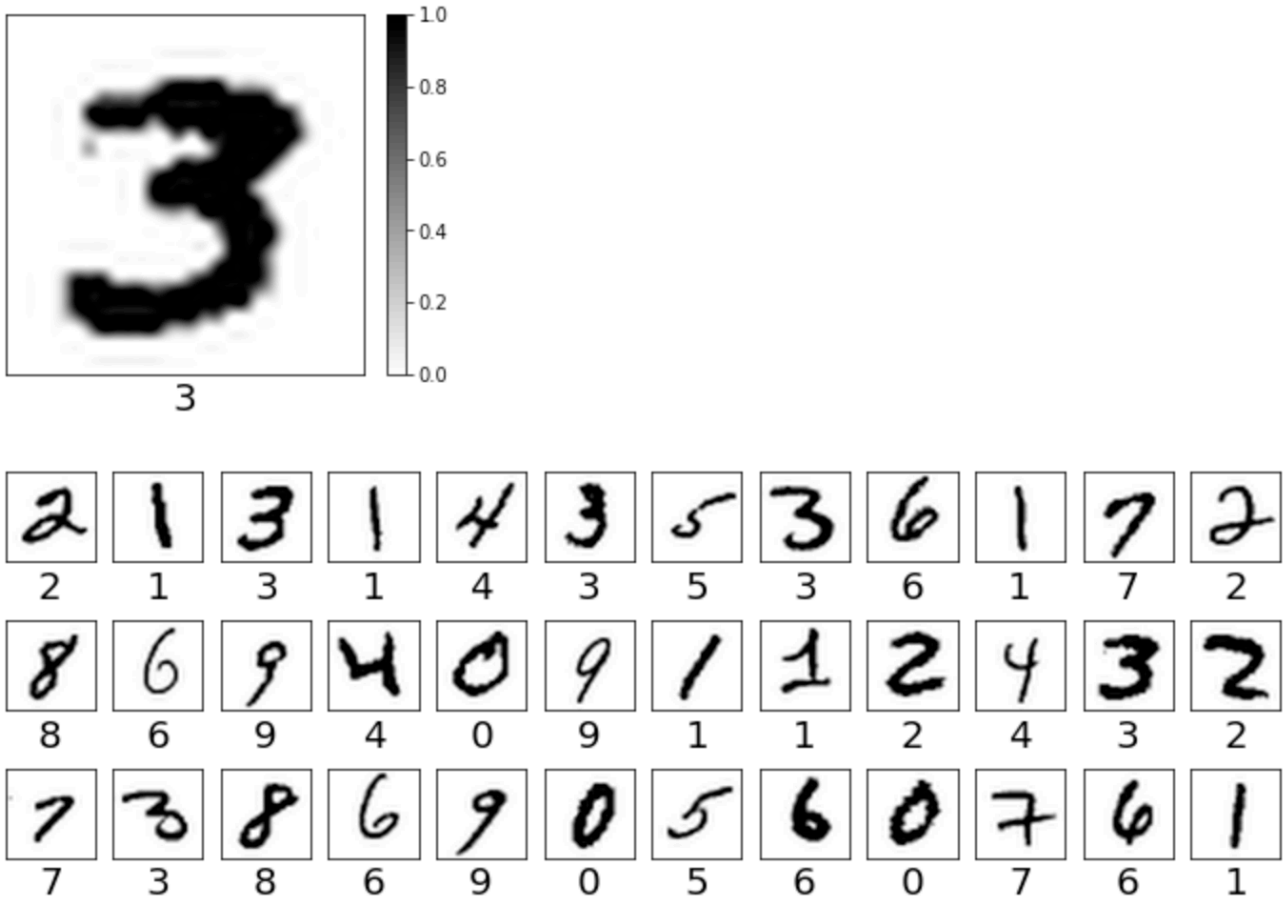


→ Niveau de complexité intermédiaire entre *TensorFlow* et *Keras* — Particulièrement flexible !

Exemple adapté d'un code *fidle sous Keras* de *S. Arias (INRIA)*, *E. Maldonado (INRIAE)*, *J.L. Parouty* (CNRS)

(1) Données d'apprentissage

```
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)
```



Exemple adapté d'un code *fidle sous Keras* de *S. Arias (INRIA)*, *E. Maldonado (INRIAE)*, *J.L. Parouty* (CNRS)

(2) Création d'un modèle de réseau de neurones

```
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()
```

Exemple adapté d'un code *fidle sous Keras* de *S. Arias (INRIA)*, *E. Maldonado (INRIAE)*, *J.L. Parouty* (CNRS)

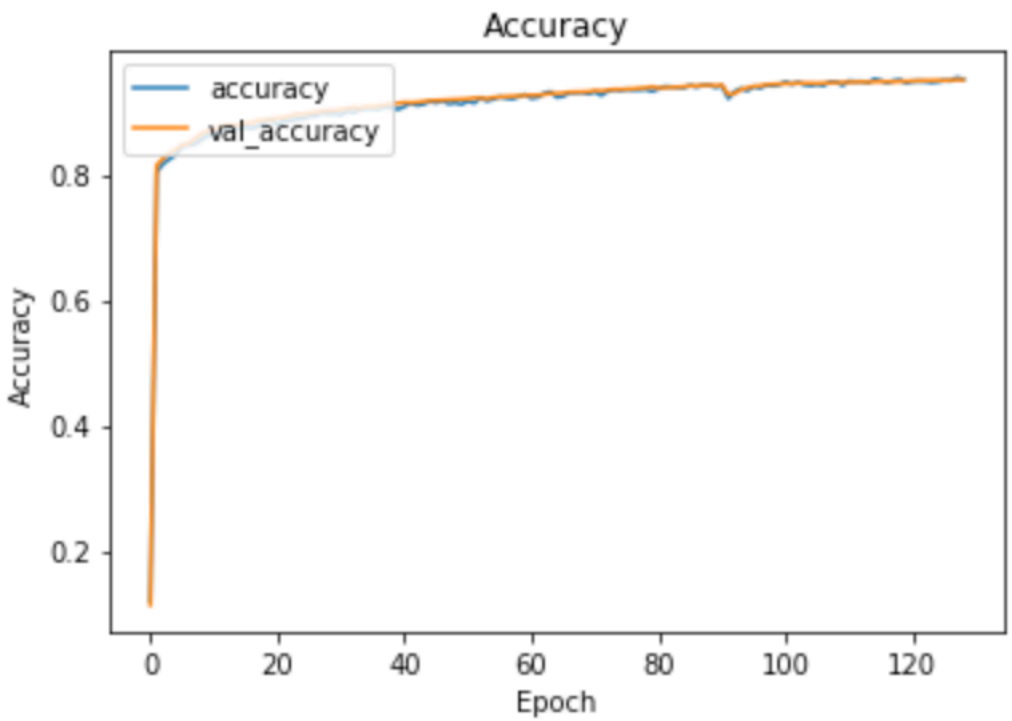
(3) Définition d'une stratégie d'apprentissage

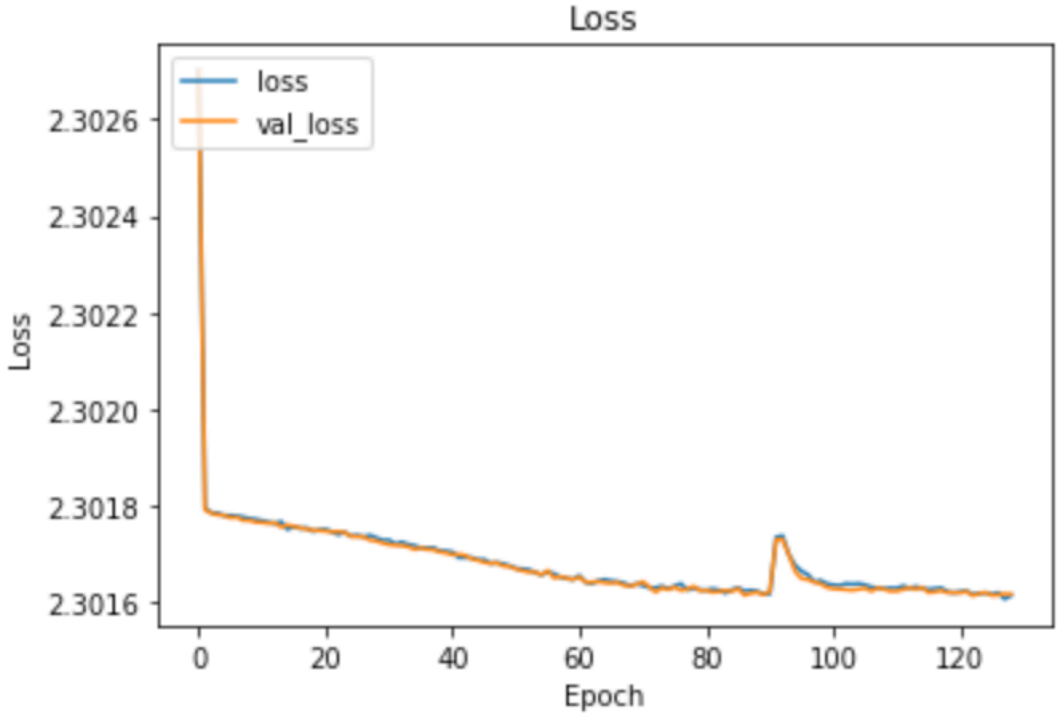
```
def fit(model,X_train,Y_train,X_test,Y_test, EPOCHS = 5, BATCH_SIZE = 32):
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(),lr=1e-3) #lr is the learning rate
    model.train()

    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
```

Exemple adapté d'un code *fidle sous Keras* de *S. Arias (INRIA)*, *E. Maldonado (INRIAE)*, *J.L. Parouty* (CNRS)

(4) Apprentissage

```
batch_size  = 512
epochs      = 128


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

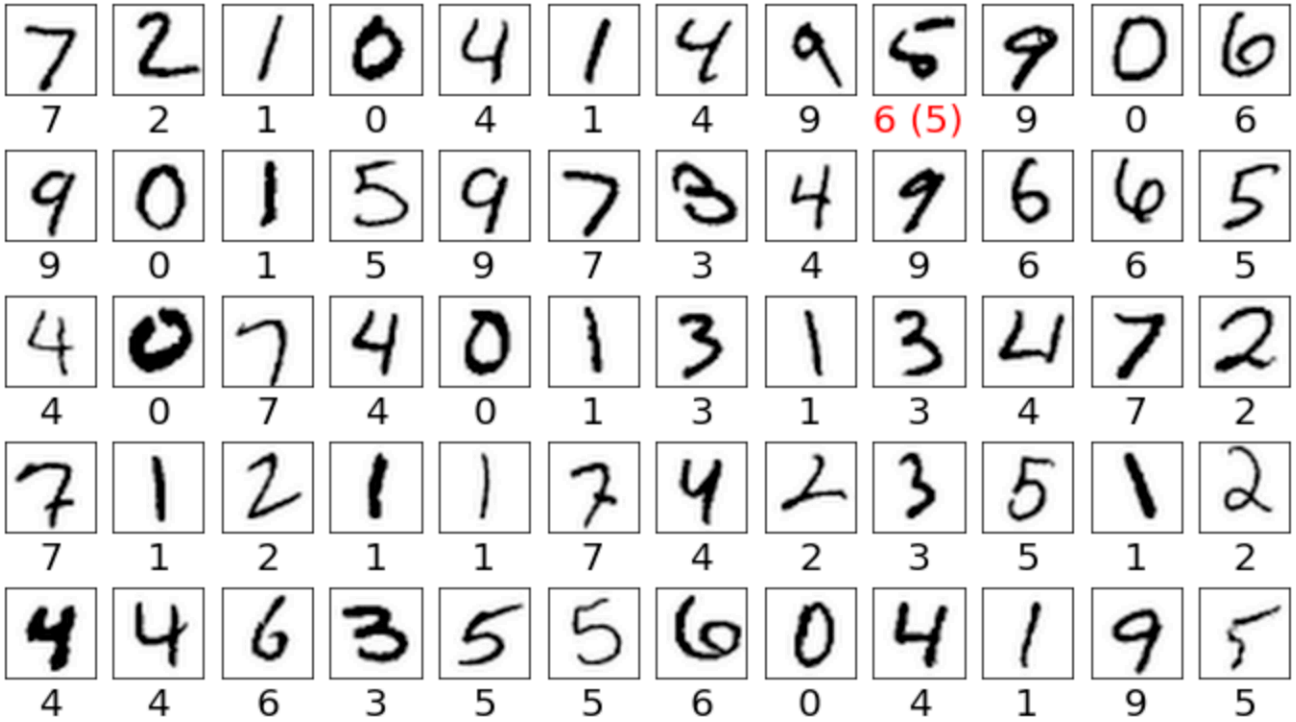




Exemple adapté d'un code *fidle sous Keras* de *S. Arias (INRIA)*, *E. Maldonado (INRIAE)*, *J.L. Parouty* (CNRS)

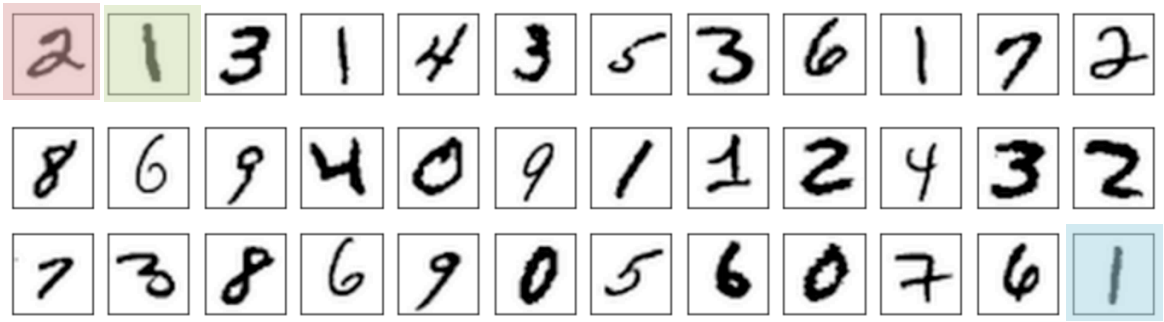
(5) Prédictions

```
var_x_test = Variable(x_test[:,:,:]).float()
var_y_test = Variable(y_test[:])
y_pred = model(var_x_test)
np_y_pred_label = torch.argmax(y_pred, dim= 1).numpy().astype(np.uint8)

np_x_test=x_test.numpy().astype(np.float64)
np_y_test=y_test.numpy().astype(np.uint8)

ooo.plot_images(np_x_test, np_y_test, range(0,60), columns=12, x_size=1, y_size=1, y_pred=np_y_pred_label)
```



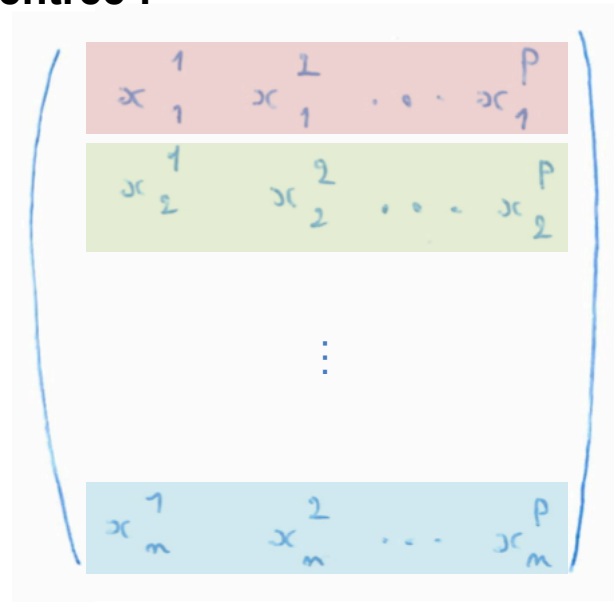
Données d'entrée :

```
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   ...
```



Matrice des données d'entrée :

x_test :



x_test[0,:]

x_test[1,:]

⋮

x_test[n,:]

Modification de la forme d'un tenseur pytorch avec view().

→ Equivalent du reshape() sous numpy

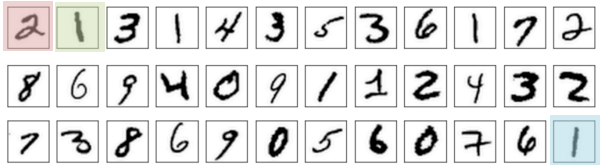
Mise à plat de l'image i de taille $c \times l$:

- $(x_i^1, \ldots, x_i^l)$ est la *1ere ligne* de l'image
- $(x_i^{l+1}, \ldots, x_i^{2l})$ est la *2eme ligne* de l'image
- …
- $(x_i^{(c-1)l+1}, \ldots, x_i^{cl})$ est la *ceme ligne* de l'image

Différence entre « numpy arrays », « pytorch tensors » et « pytorch variables »

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
```

```
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   ...
```



Affichage des images de x_train

```
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()
    var_Y_batch = Variable(Y_train[mini_batch_observations])

    #gradient descent step
    optimizer.zero_grad()
    Y_pred_batch = model(var_X_batch)
    curr_loss = loss(Y_pred_batch, var_Y_batch)
    curr_loss.backward()
    optimizer.step()

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

Un epoch dans la descente de gradient

Numpy arrays : format standard pour les matrices et tenseurs en Python

PyTorch tensors :

- Equivalent d'un numpy array dans la librairie PyTorch
- Se transfert facilement sur un GPU

```
if torch.cuda.is_available():
  tensor = tensor.to('cuda')
```

PyTorch variables :

- Objet qui « hérite » des pyTorch tensor
- On peut calculer son gradient si en lien direct ou non avec un *loss* (autograd ← différentiation automatique)

Réseau dense de l'exemple

```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)   #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()
```

Petit réseau convolutionnel (CNN)

```python
class ConvolutionalNetwork(nn.Module):
    def __init__(self, output_neurons):
        super().__init__()
        # First convolution/pooling layer of 20 filters (for MNIST, transforms size : 1*28*28 -> 20*26*26 -> 20*13*13)
        self.conv1 = nn.Conv2d(1, 20, kernel_size=3, stride=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second convolution/pooling layer of 40 filters (for MNIST, transforms size : 20*13*13 -> 40*11*11 -> 40*5*5)
        self.conv2 = nn.Conv2d(20, 40, kernel_size=3, stride=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Third convolution/pooling layer of 80 filters (for MNIST, transforms size : 40*5*5 -> 80*3*3 -> 80*1*1)
        self.conv3 = nn.Conv2d(40, 80, kernel_size=3, stride=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Fourth dense layer of K neurons and a final sigmoid function (for MNIST, transforms size : 80*1*1 -> output_classes)
        self.dense = nn.Linear(80*1*1, output_neurons)
        self.sigmoid = nn.Sigmoid()

    def forward(self, X):
        # Forward pass of data through the first convolution/pooling layer
        output_layer1 = self.pool1(self.conv1(X))
        # Forward pass of data through the second convolution/pooling layer
        output_layer2 = self.pool2(self.conv2(output_layer1))
        # Forward pass of data through the third convolution/pooling layer
        output_layer3 = self.pool3(self.conv3(output_layer2))
        # Flatten the output of convolution/pooling layers to pass through dense layers
        output_layer3 = output_layer3.view(output_layer3.size(0), -1)
        # Forward pass of data through the fourth dense layer
        return self.sigmoid(self.dense(output_layer3))
```

Charger une architecture *deep* proposée dans PyTorch (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()
```

Charger une architecture *deep* proposée dans PyTorch (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)
```

```python
def fit(model,X_train,Y_train, 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 = 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
```

Voyons maintenant l'algorithme d'apprentissage …

```python
def fit(model,X_train,Y_train, 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 = 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
```

- 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, 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 = 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
```

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, 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 = 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
```

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

```python
def fit(model,X_train,Y_train, 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 = 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
```

- Récupération des observations nécessaires pour le mini-batch de l'itération courante
- Conversion des *PyTorch tensors* en *PyTorch variables* (données utilisées pour le calcul des gradients)

```python
def fit(model,X_train,Y_train, 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 = 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
```

- Mise à zéro des gradients de paramètres du modèle (se sont des objets Variable)
- 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

À vous maintenant …

MERCI !

Tout sur PyTorch : https://pytorch.org/
Tutoriel pour aller un peu plus loin : https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html