

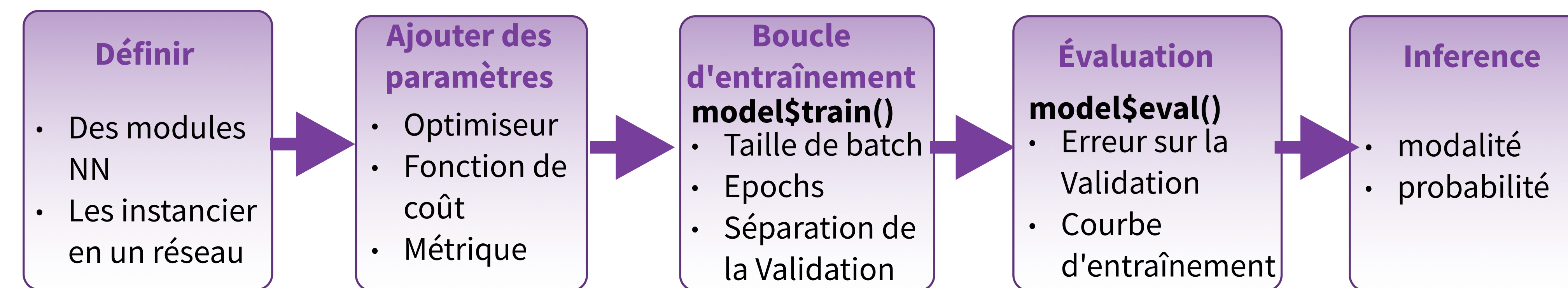
# Le Deep-learning avec torch:: AIDE-MÉMOIRE



**Intro** {torch} est basé sur PyTorch, un environnement de Deep-learning utilisé à grande échelle par les chercheurs.

{torch} profite de l'accélération matérielle de la carte GPU avec une interface conviviale, et couvre un grand nombre de cas d'usage, et pas seulement d'apprentissage profond, grâce à sa flexibilité et l'accès possible à l'interface de bas-niveau.

Il est au centre d'un écosystème de paquets logiciels prévus pour l'interface avec des types de donnée spécifiques comme {torchaudio} pour les données temporelles, {torchvision} pour les images, et {tabnet} pour les données tabulaires.



<https://torch.mlverse.org/>

<https://mlverse.shinyapps.io/torch-tour/>

## INSTALLATION

Le package {torch} s'appuie sur la librairie C++ libtorch. Les prérequis s'installent entièrement depuis R

<https://torch.mlverse.org/docs/articles/installation.html>

```
install.packages("torch")
library(torch)
install_torch()
```

Voir ?install\_torch pour les instructions avec GPU

## La manipulation des modèles

### DEFINIR UN MODULE NN

```
dense ← nn_module(
  "couche_dense_sans_biais",
  initialize = function(in_f, out_f) {
    self$w ← nn_parameter(torch_randn(in_f, out_f))
  },
  forward = function(x) {
    torch_mm(x, self$w)
  }
)
```

Crée un nn module nommé couche\_dense\_sans\_biais

### ASSEMBLER DES MODULES EN RESEAU

```
model ← dense(4, 3)
```

Instancie un réseau avec une seule couche

```
model ← nn_sequential(
  dense(4,3), nn_relu(), nn_dropout(0.4),
  dense(3,1), nn_sigmoid())
```

Instancie un réseau avec une sequence de couches

### ENTRAINER UN MODÈLE

```
model$train()
Active la mise à jour des gradients
```

```
with_enable_grad({
  y_pred ← model(trainset)
  loss ← (y_pred - y)$pow(2)$mean()
  loss$backward()
})
Code détaillé de la boucle d'entraînement (alternatif)
```

### EVALUER UN MODEL

```
model$eval()
Ou
with_no_grad({
  model(validationset)
})
Passe d'inférence du modèle sans mise à jour de gradient
```

### OPTIMISATION

```
optim_sgd()
Optimiseur de descente de gradient stochastique
```

```
optim_adam()
Optimiseur ADAM
```

### FONCTION DE COÛT POUR LA CLASSIFICATION

```
nn_cross_entropy_loss()
nn_bce_loss()
nn_bce_with_logits_loss()
(Binaire) fonction de coût d'entropie croisée
nn_nll_loss()
Fonction de coût de log-vraisemblance négative
nn_margin_ranking_loss()
nn_hinge_embedding_loss()
nn_multi_margin_loss()
nn_multilabel_margin_loss()
(Multinomial) (multi étiquette) fonction de coût de Hinge
```

### FONCTION DE COÛT POUR LA REGRESSION

```
nn_l1_loss()
Fonction de coût L1
nn_mse_loss()
MSE loss
nn_ctc_loss()
Connectionist Temporal Classification loss
nn_cosine_embedding_loss()
Fonction de coût de plongement cosinus
nn_kl_div_loss()
Fonction de coût de divergence de Kullback-Leibler
nn_poisson_nll_loss()
Fonction de coût de log-vraisemblance négative de Poisson
```

### OTHER MODEL OPERATIONS

```
summary()
Résume le modèle
```

```
torch_save(); torch_load()
enregistre/ restaure le modèle dans un fichier
```

```
load_state_dict()
Restaure un modèle enregistré depuis python
```

## Couches de réseaux de neurones

### COUCHE PRINCIPALES

**nn\_linear()**  
Ajoute une couche de transformation linéaire à une entrée

**nn\_bilinear()** à deux entrées

**nn\_sigmoid(), nn\_relu()**  
Applique une fonction d'activation à une sortie

**nn\_dropout(), nn\_dropout2d(), nn\_dropout3d()**  
Applique une élimination de connexions à l'entrée

**nn\_batch\_norm1d(), nn\_batch\_norm2d(), nn\_batch\_norm3d()**  
Applique une normalisation par batch aux paramètres

### COUCHES DE CONVOLUTION

**nn\_conv1d()** 1D, e.g. convolution temporelle

**nn\_conv\_transpose2d()** convolution 2D transposée (déconvolution)

**nn\_conv2d()** 2D, e.g. convolution spatiale sur les images

**nn\_conv\_transpose3d()** convolution 3D transposée (déconvolution)  
**nn\_conv3d()** 3D, e.g. convolution spatiale sur les volumes

**nnf\_pad()**  
Couche de remplissage avec des zero

### COUCHES D'ACTIVATION

**nn\_leaky\_relu()**  
Activation lineaire rectifiée avec fuite

**nn\_relu6()**  
Activation lineaire rectifiée avec limitation à 6

**nn\_rrelu()**  
Activation lineaire rectifiée avec fuite, aléatoire

**nn\_elu(), nn\_selu()**  
Activation lineaire exponentielle, idem mise à l'échelle

### COUCHES D'EXTRACTION

**nn\_max\_pool1d(), nn\_max\_pool2d(), nn\_max\_pool3d()**  
Extraction des maximum de 1D à 3D

**nn\_avg\_pool1d(), nn\_avg\_pool2d(), nn\_avg\_pool3d()**  
Extraction des moyennes de 1D à 3D

**nn\_adaptive\_max\_pool1d(), nn\_adaptive\_max\_pool2d(), nn\_adaptive\_max\_pool3d()**  
Extraction adaptative des maximum  
**nn\_adaptive\_avg\_pool1d(), nn\_adaptive\_avg\_pool2d(), nn\_adaptive\_avg\_pool3d()**  
Extraction adaptative des moyennes

### COUCHES RÉCURRENTES

**nn\_rnn()**  
Réseau récurrent de noeuds entièrement connectés avec sortie réinjectée à l'entrée

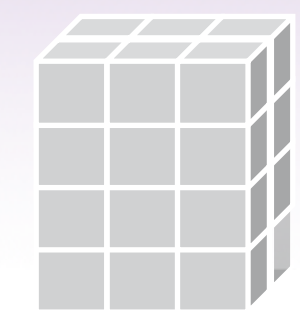
**nn\_gru()**  
Unité récurrente à porte- Cho et al

**nn\_lstm()**  
Unité à mémoire longue-courte - Hochreiter 1997



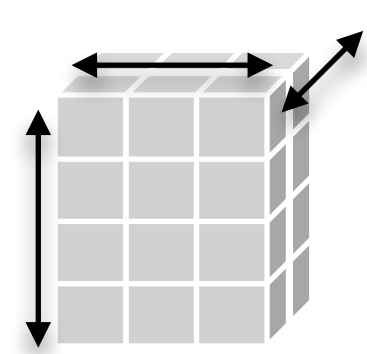
# Manipulation des tenseurs

## CRÉATION DE TENSEUR



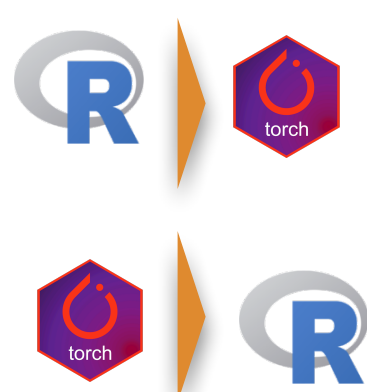
**tt** ← **torch\_rand(4,3,2)** distrib. uniforme  
**tt** ← **torch\_randn(4,3,2)** distrib. normale unitaire  
**tt** ← **torch\_randint(1,7,c(4,3,2))** distrib. uniforme d'entiers dans [1,7)  
Création d'un tenseur de la dimension voulue.

**tt** ← **torch\_ones(4,3,2)**  
**torch\_ones\_like(a)**  
Création d'un tenseur unitaire de la dimension voulue ou de la dimension de 'a'. Voir **torch\_zeros**, **torch\_full**, **torch\_arange**,...



**tt\$shape** [1] 4 3 2  
**tt\$ndim** [1] 3  
**tt\$dtype** torch\_Float  
**tt\$requires\_grad** [1] FALSE  
**tt\$device** torch\_device(type='cpu')  
Dimension et attributs d'un tenseur

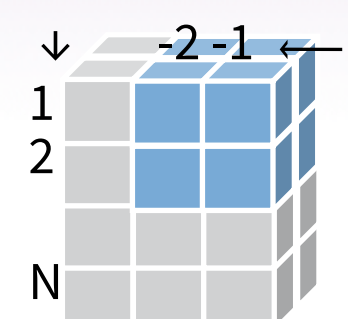
**tt\$stride()** [1] 6 2 1  
Saut de valeurs nécessaire entre deux dimensions du tenseur



**tt** ← **torch\_tensor(a, dtype=torch\_float(), device="cuda")**  
**a** ← **as.matrix(tt\$to(device="cpu"))**  
Copie une matrice 'a' de R dans un tenseur de reals en GPU et vice-versa

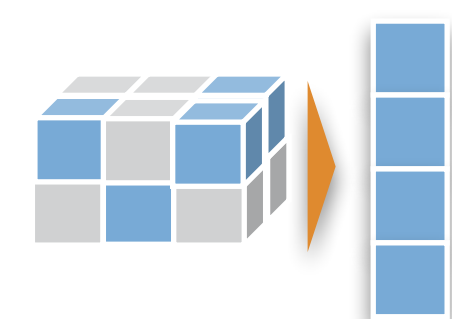
## EXTRACTION DE VALEURS

**tt[1:2, -2:-1, ]**  
Extrait un tenseur 3D  
**tt[5:N, -2:-1, ..]**  
Extrait un tenseur 3D ou plus, N le 'dernier'



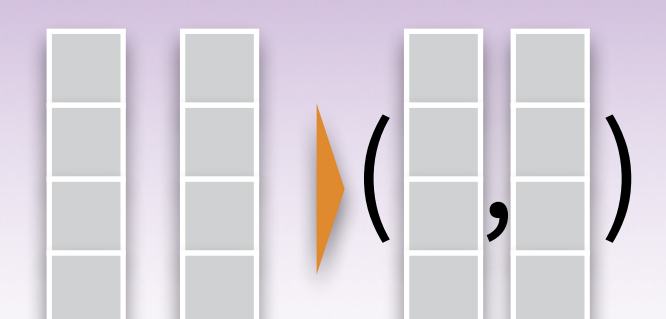
**tt[1:2, -2:-1, 1:1]**  
**tt[1:2, -2:-1, 1, keep=TRUE]**  
Extrait un tenseur 3D en conservant la dimension unitaire.

**tt[1:2, -2:-1, 1]**  
La dimension unitaire est supprimée par défaut.

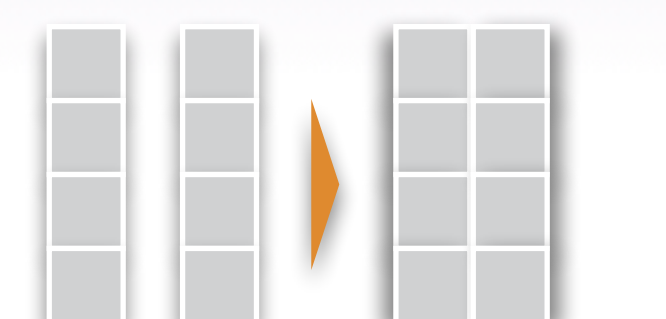


**tt[ tt > 3.1]**  
Filtrage booléen (le résultat est plat)

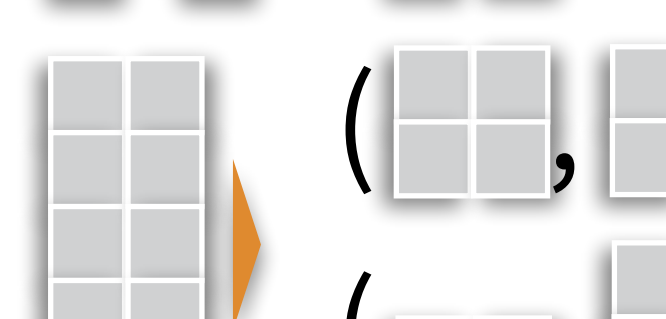
## CONCATENATION DE TENSEURS



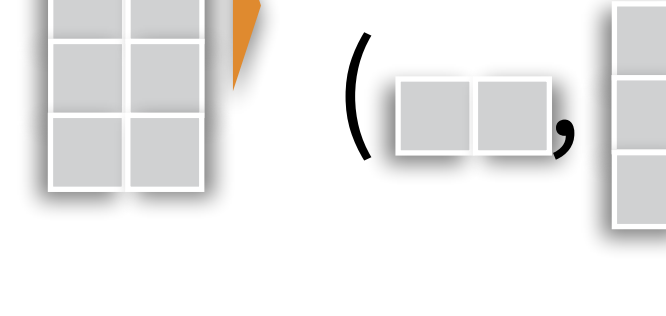
**torch\_stack()**  
Pile de tenseurs



**torch\_cat()**  
Assemble les tenseurs

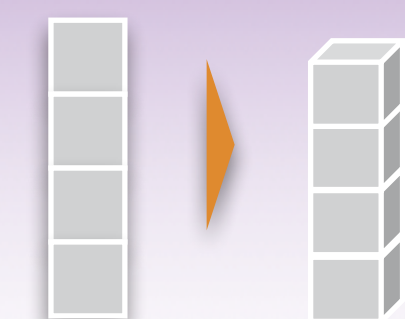


**torch\_split(2)**  
Découpe en sections de taille 2

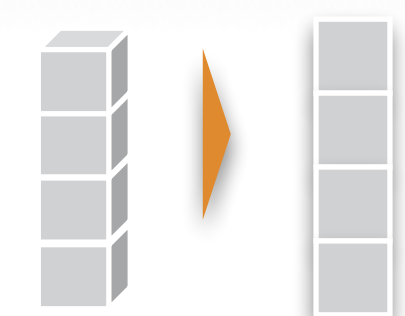


**torch\_split(c(1,3,1))**  
Découpe en sections de taille explicites

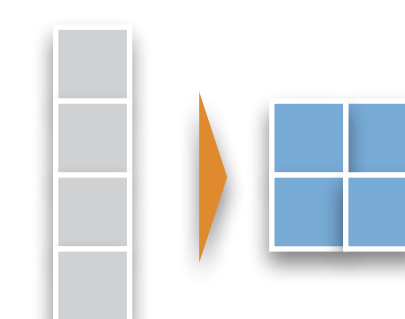
## OPÉRATION SUR LES DIMENSIONS



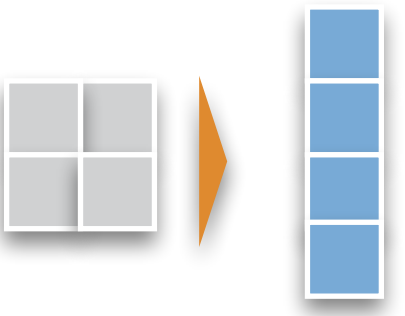
**tt\$unsqueeze(1)** **torch\_unsqueeze(tt,1)**  
Ajoute une première dimension unitaire à "tt"



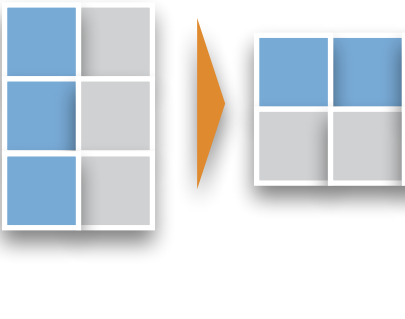
**tt\$squeeze(1)** **torch\_squeeze(t,1)**  
Supprime la première dimension unitaire de "tt"



**torch\_reshape()** **\$view()**  
Change les dimensions du tenseur  
Avec copie ou (potentiellement) sans



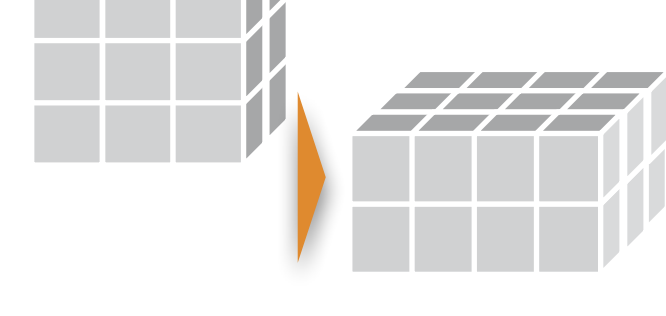
**torch\_flatten()**  
aplatit le tenseur



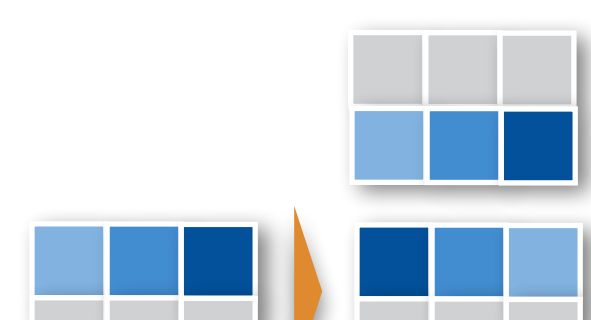
**torch\_transpose()**



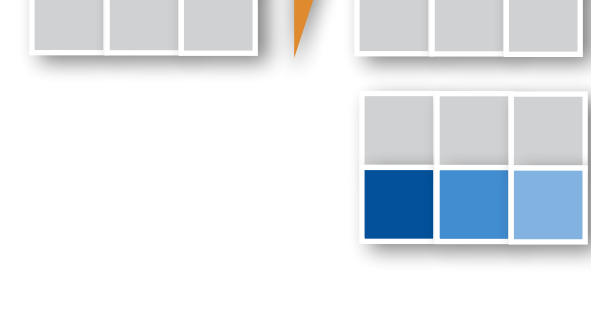
**torch\_movedim(c(1,2))**  
Inverse la dimension 1 avec 2



**torch\_movedim(c(1,2,3), c(3,1,2))**  
déplace dim 1 en 3, dim 2 en 1, dim 3 en 2  
**torch\_permute(c(3,1,2))**  
Idem uniquement sur la base de la destination



**torch\_flip(1)** retourne les valeurs de la dim 1



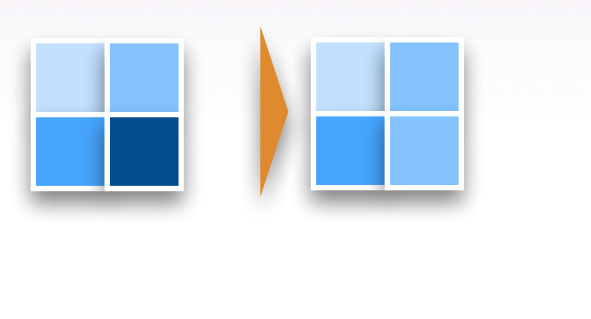
**torch\_flip(2)** 2

**torch\_flip(c(1,2))** des deux dim

## OPÉRATION SUR LES VALEURS DU TENSEUR



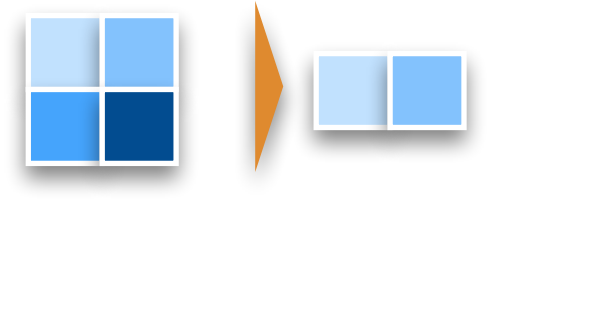
**+, -, \***  
Opérations entre deux tenseurs



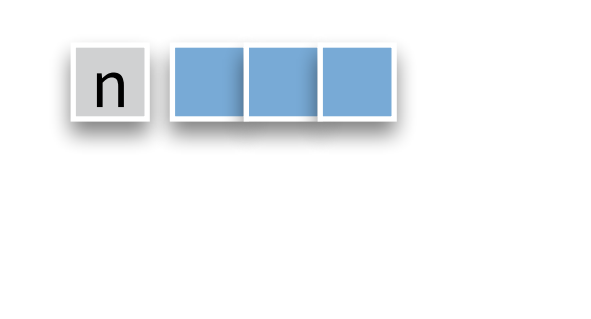
**\$pow(2), \$log(), \$exp(), \$abs(), \$floor(), \$round(), \$cos(), \$fmod(3), \$fmax(1), \$fmin(3)**  
**torch\_clamp(tt, min=0.1, max=0.7)**  
Opérations sur chaque élément

**\$eq(), \$ge(), \$le()**  
Comparaison sur chaque élément

**\$to(dtype = torch\_long())**  
Transformation des types



**\$sum(dim=1), \$mean(), \$max()**  
Fonctions d'agrégation sur un tenseur  
**\$amax()**



**torch\_repeat\_interleave()**  
Répétition d'un tenseur n fois



## UN MODELE DE RECONNAISSANCE D'IMAGE SUR MNIST

5041

Mon premier modèle avec {torch}

## Modèles pré-entraînés

Les modèles pré-entraînés sont des réseaux d'architecture classiques mis à disposition avec leurs paramètres pré-entraînés. On peut les utiliser pour une prédiction directe, une extraction de vecteur de plongement, ou l'entraînement fin.

### MODÈLES R NATIFS

**library(torchvision)**  
**resnet34** ← **model\_resnet34(pretrained=TRUE)**  
Modèles de classification d'image Resnet  
  
**resnet34\_headless** ← **nn\_prune\_head(resnet34, 1)**  
Élagage de la dernière couche d'un modèle

### IMPORT DE MODÈLES PYTORCH

{torchvisionlib} permet d'importer un modèle PyTorch sans avoir à en recoller les modules nn en R.  
On le fait en deux étapes:

1- Dans python, instancier, scripter, et sauvegarder:

```
import torch
import torchvision
```

```
model = torchvision.models.segmentation.\
    fcn_resnet50(pretrained = True)
model.eval()
```

```
scripted_model = torch.jit.script(model)
torch.jit.save(scripted_model, "fcn_resnet50.pt")
```

2- charger le fichier de modèle dans R:

```
library(torchvisionlib)
model ← torch::jit_load("fcn_resnet50.pt")
```

## Résolution de problème

### HELPERS

**with\_detect\_anomaly()**  
Fournit une vue interne du comportement du nn\_module()

## Callbacks

An callback est une ensemble de fonction à appliquer à un moment précis de la boucle d'entraînement du modèle.  
Ils permettent par exemple une vue des états interne ou une collecte de statistiques sur le modèle au cours de l'entraînement

```
# Datasets d'images MNIST
library(torchvision)
train_ds ← mnist_dataset( root = "~/cache",
  download = TRUE,
  transform = torchvision::transform_to_tensor
)
test_ds ← mnist_dataset( root = "~/cache",
  train = FALSE,
  transform = torchvision::transform_to_tensor
)
train_dl ← dataloader(train_ds, batch_size = 32,
  shuffle = TRUE)
test_dl ← dataloader(test_ds, batch_size = 32)

# definition du modèle et de ses couches
net ← nn_module(
  "Net",
  initialize = function() {
    self$fc1 ← nn_linear(784, 128)
    self$fc2 ← nn_linear(128, 10)
  },
  forward = function(x) {
    x %>% torch_flatten(start_dim = 2) %>%
      self$fc1() %>% nnf_relu() %>%
      self$fc2() %>% nnf_log_softmax(dim = 1)
  }
)
model ← net()

# definition de l'optimizer
optimizer ← optim_sgd(model$parameters, lr = 0.01)

# boucle d'entraînement (fit)
for (epoch in 1:10) {
  train_loss ← c()
  test_loss ← c()
  for (b in enumerate(train_dl)) {
    optimizer$zero_grad()
    output ← model(b[[1]]$to(device = device))
    loss ← nnf_nll_loss(output, b[[2]]$to(device = device))
    loss$backward()
    optimizer$step()
    train_loss ← c(train_loss, loss$item())
  }
  for (b in enumerate(test_dl)) {
    model$eval()
    output ← model(b[[1]]$to(device = device))
    loss ← nnf_nll_loss(output, b[[2]]$to(device = device))
    test_loss ← c(test_loss, loss$item())
    model$train()
  }
}
```