

# Apprentissage statistique

## Plateformes pour le deep learning

Olivier Schwander <[olivier.schwander@sorbonne-universite.fr](mailto:olivier.schwander@sorbonne-universite.fr)>

Polytech Sorbonne - MAIN5  
Sorbonne Université

2023-2024

# Installation de PyTorch

<https://pytorch.org/get-started/locally/>

```
conda install pytorch torchvision cpuonly -c pytorch
```

# Keras

<https://keras.io>

## Très haut niveau

- ▶ Description des couches
- ▶ Sur-couche pour TensorFlow (entre autre)
- ▶ Interface conforme à `sklearn`
- ▶ Pratique pour tester rapidement un modèle simple
- ▶ Relativement peu de possibilité de modifier des choses

## Définition du modèle

```
model = Sequential()  
model.add(Dense(units=64, activation='relu', input_dim=100))  
model.add(Dense(units=10, activation='softmax'))
```

## Loss, optimisation, etc

```
model.compile(loss='categorical_crossentropy',  
              optimizer='sgd',  
              metrics=['accuracy'])
```

# Apprentissage et évaluation

```
model.fit(x_train, y_train, epochs=5, batch_size=32)  
  
classes = model.predict(x_test, batch_size=128)
```

# Tensorflow

<https://www.tensorflow.org>

## Bas niveau

- ▶ Description des opérations de calcul
- ▶ ou des couches
- ▶ Très générique
- ▶ Complètement personnalisable pour ses propres besoins
- ▶ Support du multi-CPU, du multi-GPU, du multi-serveurs
- ▶ Plateforme déclarative
- ▶ Pas évident pour débbugger

## Modèle

```
X = tf.placeholder("float", [None, num_input])  
Y = tf.placeholder("float", [None, num_classes])
```

```
h1 = tf.Variable(tf.random_normal([num_input, n_hidden_1]))  
h2 = tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2]))  
out = tf.Variable(tf.random_normal([n_hidden_2, num_classes]))  
b1 = tf.Variable(tf.random_normal([n_hidden_1])),  
b2 = tf.Variable(tf.random_normal([n_hidden_2])),  
b3 = tf.Variable(tf.random_normal([num_classes]))
```

```
def neural_net(x):  
    layer_1 = tf.add(tf.matmul(x, h1), b1)  
    layer_2 = tf.add(tf.matmul(layer_1, h2), b2)  
    out_layer = tf.matmul(layer_2, out + b3]  
    return out_layer
```



## Loss, optimisation, évaluation

```
logits = neural_net(X)

loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

# Apprentissage

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)

        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x, Y: batch_y})

    sess.run(accuracy, feed_dict={X: mnist.test.images, Y: mnist.test.labels})
```

# PyTorch

<https://pytorch.org>

## Niveau intermédiaire

- ▶ Description des couches
- ▶ ou de n'importe quel calcul
- ▶ Très générique
- ▶ Complètement personnalisable pour ses propres besoins
- ▶ Plateforme impérative
- ▶ Très proche de NumPy
- ▶ Facile à débbugger

# Modèle

```
class Net(nn.Module):  
    def __init__(self):  
        super(Net, self).__init__()  
        self.conv1 = nn.Conv2d(3, 6, 5)  
        self.pool = nn.MaxPool2d(2, 2)  
        self.conv2 = nn.Conv2d(6, 16, 5)  
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  
        self.fc2 = nn.Linear(120, 84)  
        self.fc3 = nn.Linear(84, 10)  
  
    def forward(self, x):  
        x = self.pool(F.relu(self.conv1(x)))  
        x = self.pool(F.relu(self.conv2(x)))  
        x = x.view(-1, 16 * 5 * 5)  
        x = F.relu(self.fc1(x))  
        x = F.relu(self.fc2(x))
```

# Loss et optimisation

```
criterion = nn.CrossEntropyLoss()  
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=
```

# Apprentissage

```
for epoch in range(epoch):
```

```
    running_loss = 0.0
```

```
    for i, data in enumerate(trainloader, 0):
```

```
        inputs, labels = data
```

```
        optimizer.zero_grad()
```

```
        outputs = net(inputs)
```

```
        loss = criterion(outputs, labels)
```

```
        loss.backward()
```

```
        optimizer.step()
```

```
    running_loss += loss.item()
```

```
    if i % 2000 == 1999:      # print every 2000 mini-batches
```

```
        print('[%d, %5d] loss: % 3f' %
```

# Évaluation

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

# Questions

- ▶ Et les gradients ?
- ▶ Comment fait-on la descente de gradient ?
- ▶ Où et comment sont calculés les gradients ?



# Calcul

En NumPy:

```
A = np.zeros(10, 10)
```

```
B = np.ones(10, 10)
```

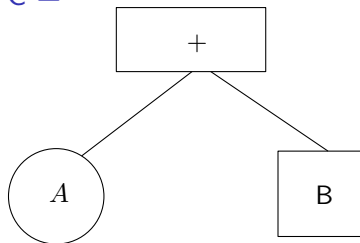
```
C = A + B
```

Que contient C ?

Chaque ligne exécute un calcul et renvoie le résultat.

# Graphe de calcul

$C =$



- ▶ Représentation abstraite de la façon d'arriver au résultat
- ▶ Et donc: représentation manipulable informatiquement

# Effectuer le calcul

## Graphe de calcul

- ▶ Description abstraite du calcul

## Vraies valeurs

- ▶ Valeurs concrètes au lieu de variables abstraites

## Évaluation

- ▶ Transformer la description du calcul en un résultat concret

## Style déclaratif

En TensorFlow:

```
a = tf.Variable(123, name="a")  
b = tf.Variable(54, name="b")  
c = a + b
```

Que contient c ?

Chaque ligne décrit une partie du calcul.

Évaluation:

```
c.eval()
```

ou

```
session.run(...)
```

## Style impératif

En PyTorch:

```
a = torch.tensor(123)
b = torch.tensor(54)
c = a + b
```

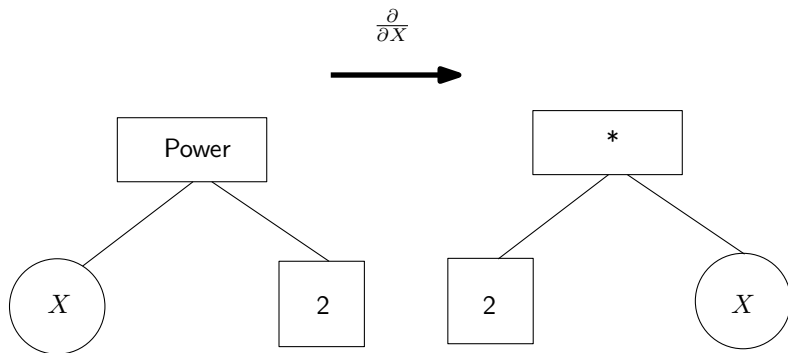
Que contient c ?

Chaque ligne décrit une partie du calcul ET calcule le résultat

Évaluation:

- ▶ C'est fait au fur et à mesure
- ▶ Mais on se souvient de comment on est arrivé au résultat

# Dérivation automatique



## Calcul du gradient

```
x0 = 5.0
```

```
x = torch.tensor(x0, requires_grad=True)
```

```
y = x**2
```

```
y.backward()
```

```
grad_y_wrt_x = x.grad
```

- Pas besoin d'écrire le gradient manuellement

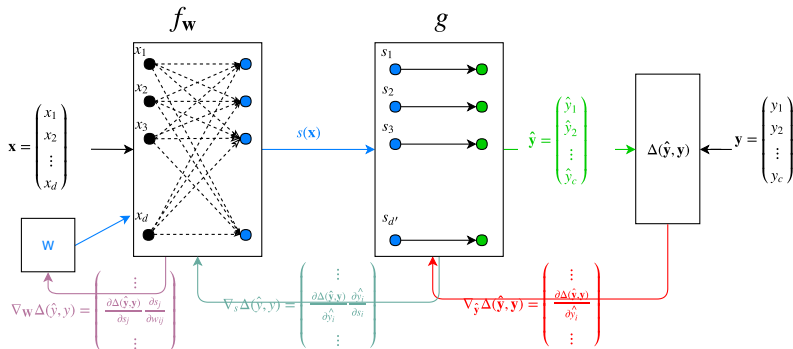
# Descente de gradient

À vous

$$\min_x x^2$$



# Rétropropagation



# Régression linéaire

À vous.

- ▶ En dimension quelconque
- ▶  $Y = AX + B$
- ▶ Loss Mean Square Error
- ▶ Essais sur ce que vous voulez

# MNIST

À vous.

## Deux modèles simples

- ▶ Réseau dense (2 couches)
- ▶ Réseau convolutionnel (2 conv + 1 dense)

Évaluer les différents modèles, en fonction des hyper-paramètres.

**À finir chez vous**

## Chargement MNIST

```
train_loader = torch.utils.data.DataLoader(  
    datasets.MNIST('data', train=True, download=True,  
        transform=transforms.Compose([  
            transforms.ToTensor(),  
            transforms.Normalize((0.1307,), (0.3081,))  
        ])),  
    batch_size=args.batch_size, shuffle=True)  
test_loader = torch.utils.data.DataLoader(  
    datasets.MNIST('data', train=False,  
        transform=transforms.Compose([  
            transforms.ToTensor(),  
            transforms.Normalize((0.1307,), (0.3081,))  
        ])),  
    batch_size=args.test_batch_size, shuffle=True)
```