Introduction au Deep LEarning

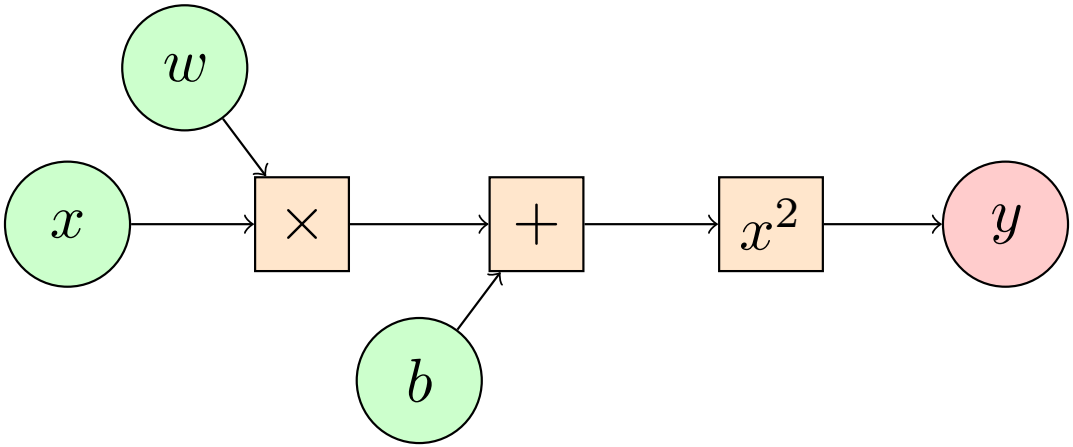
Graphe de calcul, PyTorch et autres

Olivier Schwander <olivier.schwander@sorbonne-universite.fr>

Master MIND
Sorbonne Université

2025-2026

Graphe de calcul



Séquence d'opérations

- Entrées : paramètres du réseau
- Entrées : exemples
- Sortie : loss (on va dériver)
- Sorties : scores (pas de dérivation)

Retour sur la backprop

Règle de la chaîne

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Backprop

- Application récursive la règle de la chaîne de la sortie vers l'entrée
- Somme des contributions : si w contribue à plusieurs sorties, on somme les gradients

Autograd en Pytorch

Feuilles du graphe

- Tenseurs en entrée du graphe (ie pas le résultat d'un calcul)
- `requires_grad=True` $\rightarrow$ calcul des gradients (paramères ou exemples)
- `requires_grad=False` $\rightarrow$ constantes

Calcul

- Accumulation dans `.grad` lors de `backward()`
- Besoin de réinitialiser à chaque epoch
- Interdiction des dépendances circulaires

Descente de gradient en PyTorch

Difficultés

- Ne pas créer de boucles
- Remettre à 0 les gradients

Solutions manuelles

- Modifier en place les valeurs
- Créer un nouveau graphe
- Remettre à 0 explicitement les gradients

Descente de gradient en PyTorch

```
x = torch.tensor(10.0, requires_grad=True)
lr = 0.1

for i in range(20):
    y = x ** 2
    y.backward()

    with torch.no_grad(): # On ne suit pas les gradients
        x -= lr * x.grad # L'opérateur -= modifie en place

    x.grad.zero_() # On remet à 0 le gradient
```

Descente de gradient en PyTorch

```
x = torch.tensor(10.0, requires_grad=True)
lr = 0.1

for i in range(20):
    y = x ** 2
    y.backward()

    with torch.no_grad(): # On ne suit pas les gradients
        x = x - lr * x.grad # Nouvelle variable

    x.requires_grad_() # Une feuille dans le nouveau graph
```

Régression linéaire (version 1)

```
a = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(0.0, requires_grad=True)
lr = 0.01

for epoch in range(1000):
    loss = torch.sum((a * x + b - y)**2) / x.shape[0]
    loss.backward()

    with torch.no_grad(): # On ne stocke pas les gradients
        a = a - lr * a.grad
        b = b - lr * b.grad

    a.requires_grad_() # Mais on voudra quand même les gra
    b.requires_grad_()
```

Régression linéaire (version 2)

```
a = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(0.0, requires_grad=True)
lr = 0.01

for epoch in range(1000):
    loss = torch.sum((a * x + b - y)**2) / x.shape[0]
    loss.backward()

    # Pas de graphe de calcul, on manipule directement les
    a.data = a.data - lr * a.grad.data
    b.data = b.data - lr * b.grad.data

    # On réinitalise les gradients pour préparer la procha
    a.grad.data.zero_()
    b.grad.data.zero_()
```

Régression linéaire (version 3)

```
a = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(0.0, requires_grad=True)
lr = 0.01

for epoch in range(1000):
    loss = torch.sum((a * x + b - y)**2) / x.shape[0]
    loss.backward()

    # Pas de graphe de calcul, on manipule directement les
    a = (a - lr * a.grad).clone().detach()
    b = (b - lr * b.grad).clone().detach()

    # On voudra les gradients dans la suite
    a.requires_grad = True
    b.requires_grad = True
```

MLP en PyTorch

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

model = MLP()
model(x) # Inférence
```

Entraînement

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(10):
    model.train()

    for batch_x, batch_y in train_loader:
        optimizer.zero_grad() # Réinitialisation des gradi
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
```

Inférence et évaluation

```
model.eval()

with torch.no_grad():
    for batch_x, batch_y in test_loader:
        outputs = model(batch_x)
        _, predicted = torch.max(outputs, 1)

        total += batch_y.size(0)
        correct += (predicted == batch_y).sum().item()
```

(Il y a des bibliothèques pour les scores.)

Vue d'ensemble des frameworks deep learning

Catalogue

- **Keras** : abstraction de haut niveau, empilement de couches
- **TensorFlow 1** : graphe statique (déclaration puis exécution)
- **TensorFlow 2** : graphe dynamique (eager : exécution immédiate) + compilation
- **PyTorch** : graphe dynamique (eager : exécution immédiate)
- **JAX** : Fonctionnel pur, compilation JIT, API Flax et Haiku

Tensorflow 1

Séparation déclaration/exécution

- **Phase de déclaration** : description du graphe de calcul
- **Phase d'exécution** : calcul effectif en donnant des valeurs d'entrées au graphe

```
x = tf.placeholder(tf.float32, name='x')
w = tf.get_variable("w", initializer=tf.constant(10.0))
y = tf.pow(x, 2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(y, feed_dict={x: 5.0})
```

Descente de gradient en TF1

```
w = tf.get_variable("w", initializer=tf.constant(10.0))
x_placeholder = tf.placeholder(tf.float32, name="x")
loss = tf.pow(w, 2)   # f(w) = w^2

# Optimiseur gère les gradients automatiquement
optimizer = tf.train.GradientDescentOptimizer(learning_rate
train_op = optimizer.minimize(loss)

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

    for i in range(20):
        _, w_val, loss_val = sess.run([train_op, w, loss])
```

Apprentissage TF1 : MLP

```
X = tf.placeholder(tf.float32, [None, 784])
y_true = tf.placeholder(tf.float32, [None, 10])

h1 = tf.layers.dense(X, 256, activation=tf.nn.relu)
h2 = tf.layers.dense(h1, 128, activation=tf.nn.relu)
logits = tf.layers.dense(h2, 10)

loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits_v2(
        labels=y_true, logits=logits
    )
)

optimizer = tf.train.AdamOptimizer(0.001)
train_op = optimizer.minimize(loss)
```

Apprentissage TF1 : train

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for epoch in range(10):
        for batch_x, batch_y in train_loader:
            batch_y_onehot = tf.one_hot(batch_y, 10).eval()

            sess.run(train_op, feed_dict={
                X: batch_x,
                y_true: batch_y_onehot
            })

        val_loss = sess.run(loss, feed_dict={
            X: val_x,
            y_true: val_y_onehot
        })
```

Tensorflow 2

Eager execution

- Exécution immédiate comme du Python ou Numpy

```
x = tf.constant(5.0)
y = x ** 2 # Valeur concrète
```

GradientTape

- Enregistrement explicite des gradients

```
x = tf.Variable(10.0)

with tf.GradientTape() as tape:
    y = x ** 2

dy_dx = tape.gradient(y, x)
```

Descente de gradient en TF2

```
w = tf.Variable(10.0)
lr = 0.1

for i in range(20):
    with tf.GradientTape() as tape:
        loss = w ** 2

    dw = tape.gradient(loss, w)
    w.assign_sub(lr * dw)
```

Apprentissage TF2 : MLP

Version séquentielle

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(256, activation='relu', input_sha
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

Version fonctionnelle

```
inputs = tf.keras.Input(shape=(784,))
h1 = tf.keras.layers.Dense(256, activation='relu')(inputs)
h2 = tf.keras.layers.Dense(128, activation='relu')(h1)
outputs = tf.keras.layers.Dense(10, activation='softmax')(h
model = tf.keras.Model(inputs, outputs)
```

Apprentissage TF2 : train

Compilation

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(0.001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Entraînement

```
model.fit(
    x_train, y_train,
    epochs=10,
    batch_size=32,
    validation_data=(x_val, y_val)
)
```

Remarque : on peut aussi faire la boucle manuellement pour avoir

Jax

Descente de gradient en JAX

```
import jax
import jax.numpy as jnp

def loss_fn(w):
    return jnp.power(w, 2)  # f(w) = w^2

loss_grad = jax.grad(loss_fn)

w = 10.0
lr = 0.1

for i in range(20):
    dw = loss_grad(w)  # dy/dw = 2w
    w = w - lr * dw
```

Compilation JIT

JIT Just-in-time

- Compilation au moment où on a besoin

```
@jax.jit
def update(w, lr):
    grad_w = jax.grad(loss_fn)(w)
    return w - lr * grad_w

for i in range(20):
    w = update(w, 0.1)
```

PyTorch : boucle de train simple

```
model = MLP()
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0.0

    for batch_idx, (data, target) in enumerate(train_loader
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)

        loss.backward()
        optimizer.step()
```

Boucle de train

Ce qui manque ici

- Sauvegarde des loss, des scores, epoch par epoch
- Affichage des courbes à la fin
- Instrumentation : sortir les valeurs précédents pour les afficher *pendant* le train
- Sauvegarde du modèle à la fin
- Sauvegarde du modèle régulièrement (pour reprendre là où on en était)

À peu près tout le temps la même chose

- Pas évident à ne pas répéter, à cause du "à peu près"
- Souvent : copier-coller

PyTorch Lightning

```
class LitMLP(pl.LightningModule):
    ... # tout ce qu'on avait avant +
    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = F.cross_entropy(y_hat, y)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.001

model = LitMLP()
trainer = pl.Trainer(max_epochs=10)
trainer.fit(model)
```

Tensorboard

Logging et visualisation :

```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter('runs/experiment_1')

# Logging de scalaires
for epoch in range(num_epochs):
    loss = train_one_epoch()
    writer.add_scalar('Loss/train', loss, epoch)
    writer.add_scalar('Learning rate', lr, epoch)

# Logging de gradients
for name, param in model.named_parameters():
    writer.add_histogram(f'Gradients/{name}', param.grad, e
```

MLFlow

Tracking complet d'expériences :

```
import mlflow

mlflow.set_experiment('mnist_classification')

with mlflow.start_run():
    # Logging de paramètres
    mlflow.log_params({
        'learning_rate': 0.001,
        'batch_size': 32,
        'architecture': 'MLP-256-128-10'
    })

    # Entraînement
    for epoch in range(num_epochs):
```

Intérêt des dataloaders

Pourquoi ne pas charger tout en mémoire ?

- Datasets massifs
- Minibatchs
- Data augmentation dynamique
- Multiprocessing

```
loader = DataLoader(
    dataset,
    batch_size=32,
    shuffle=True,
    num_workers=4,
    pin_memory=True,
    drop_last=True
)
```

Dataloader existant

```
from torchvision import datasets, transforms

train_dataset = datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)
train_loader = DataLoader(train_dataset, batch_size=32, shu

cifar10 = datasets.CIFAR10(root='./data', train=True, downl
 imagenet = datasets.ImageFolder('path/to/imagenet')
```

Dataset personnalisé

On crée un `Dataset`, pas un `DataLoader` :

```
class CustomDataset(Dataset):
    def __init__(self, data_path, labels_path):
        self.data = np.load(data_path) # ce qu'on veut
        self.labels = np.load(labels_path) # ici aussi

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        sample = self.data[idx]
        label = self.labels[idx]

        return sample, label
```

torch.save et torch.load

Sauvegarde des paramètres

```
torch.save(model.state_dict(), 'model.pth')

model = ModeleToto()
model.load_state_dict(torch.load('model.pth'))
```

Sauvegarde du modèle complet

- À éviter : pas portable (dépend des numéros de version précis)
- Ne pas oublier le `.state_dict()`

torch.export

Exportation du modèle

- Format intermédiaire portable : plus besoin de Python
- Exécution sur un exemple et analyse du bytecode Python

```
exported_dynamic = torch.export.export(
    model,
    args=(example_input,),
    dynamic_shapes=dynamic_shapes
)
```

- Les structures de contrôle sont conservées

torch.compile

Accélération du code

```
def fonction_complexe(x):
    if x.sum() > 0:
        return x * 2
    else:
        return x * 0.5

fonction_compile = torch.compile(fonction_complexe)
```

- Plus souple que export
- Reste en Python si trop compliqué

ONNX (Open Neural Network Exchange)

Interopérabilité

- Passage d'une bibliothèque à l'autre
- Diffusion des modèles
- Très utilisé pour l'embarqué

Quantification

Objectifs

- Accélérer l'entraînement et l'inférence
- Faciliter la distribution
- Réduire l'empreinte mémoire

Entraînement classique

- Flottant double précision (64 bits)
- Traditionnellement la façon normale de faire du calcul numérique qui se passe bien
- Trop coûteux : stockage et transferts

Nouveaux types

- BF16 float 16 bits, FP8 float 8 bits
- TF32 float 19 bits (mais vu comme un double FP32)
- Int 16, 8 ou 4 bits

Conclusion

Foisenement d'outils

- Bibliothèques
- Implémentation des modèles (cf HuggingFace)

PyTorch

- Défintion du modèle
- Boucle d'entraînement simple

Instrumentation

- Sauvegarder tout
- Les valeurs, les poids, etc
- Tracer des courbes
- **Les analyser**