TensorFlow et Intelligence Artificielle

m2iformation.fr

Sommaire (1/2)

Sommaire (2/2)

Réseaux de Neurones avec TensorFlow

- API Sequential et Functional
- Couches (Dense, Conv, LSTM, etc.)
- Fonctions d'activation
- Compilation et entraînement
- TP : Classification MNIST

Deep Learning Avancé

- Transfer Learning
- Fine-tuning
- Custom layers et models
- Callbacks et monitoring
- TP : Transfer Learning avec ResNet

Production et Déploiement

- TensorFlow Serving
- TensorFlow Lite
- TensorFlow.js

Introduction à TensorFlow – Historique

TensorFlow est une bibliothèque open-source de **Machine Learning** développée par Google Brain.

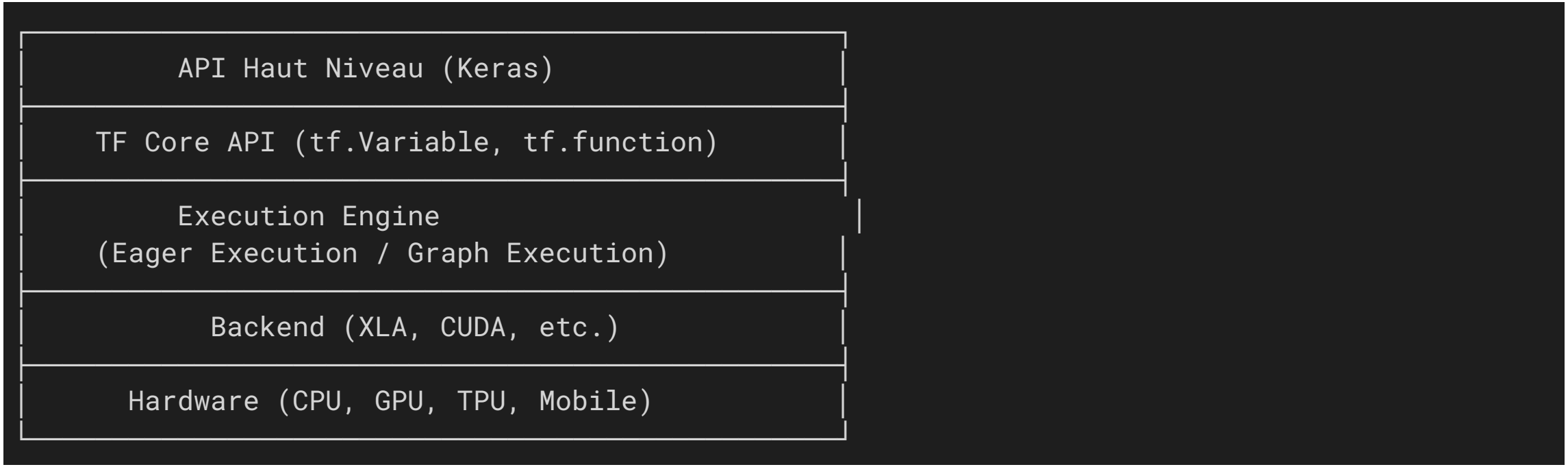
Chronologie :

- **2011** : DistBelief, le prédécesseur interne de Google
- **2015** : Open-sourcing de TensorFlow 1.0
- **2019** : TensorFlow 2.0 avec Keras intégré par défaut
- **2024-2025** : TensorFlow 2.17+ avec support JAX et optimisations

Pourquoi TensorFlow ?

- Écosystème complet (recherche → production)
- Support multi-plateforme (CPU, GPU, TPU, mobile, web)
- Communauté massive et ressources abondantes
- Intégration native avec Keras

Architecture TensorFlow – Vue d'ensemble



Composants clés :

- **tf.keras** : API haut niveau pour construire des modèles
- **tf.data** : Pipeline de données performant
- **tf.function** : Compilation de graphes pour performance
- **TensorBoard** : Visualisation et monitoring

Installation et Environnement

Installation standard :

```
# CPU uniquement
pip install tensorflow

# GPU (CUDA requis)
pip install tensorflow[and-cuda]
```

Vérification de l'installation :

```
import tensorflow as tf

print(f"TensorFlow version: {tf.__version__}")
print(f"GPU disponible: {tf.config.list_physical_devices('GPU')}")
print(f"Eager execution: {tf.executing_eagerly()}")
```

Environnement recommandé :

- Python 3.9-3.12
- CUDA 12.x pour support GPU
- cuDNN 8.9+

Concepts Fondamentaux – Tenseurs

Un **tenseur** est un tableau multidimensionnel, la structure de données fondamentale de TensorFlow.

Hiérarchie des tenseurs :

- **Scalaire** (0D) : un seul nombre → `5`
- **Vecteur** (1D) : liste de nombres → `[1, 2, 3]`
- **Matrice** (2D) : tableau 2D → `[[1,2], [3,4]]`
- **Tenseur 3D+** : données multidimensionnelles

Analogie :

```
Scalaire (0D)    →  Un point
Vecteur (1D)     →  Une ligne
Matrice (2D)     →  Une feuille de papier
Tenseur 3D       →  Un cube
Tenseur 4D+      →  Hypercube
```

Création de Tenseurs (1/2)

```python
import tensorflow as tf

# Scalaire
scalaire = tf.constant(42)
print(scalaire.shape)  # ()

# Vecteur
vecteur = tf.constant([1, 2, 3, 4, 5])
print(vecteur.shape)  # (5,)

# Matrice
matrice = tf.constant([[1, 2], [3, 4], [5, 6]])
print(matrice.shape)  # (3, 2)
```

Création de Tenseurs (2/2)

```
# Tenseur 3D (ex: image RGB)
image = tf.constant([[[255, 0, 0], [0, 255, 0]],
                     [[0, 0, 255], [255, 255, 0]]])
print(image.shape)  # (2, 2, 3) → (hauteur, largeur, canaux)

# Tenseur 4D (batch d'images)
batch = tf.constant(np.random.rand(32, 224, 224, 3))
print(batch.shape)  # (32, 224, 224, 3) → (batch, H, W, C)
```

Types de Données et Shapes

Types de données principaux :

Type TensorFlow	Type NumPy	Usage
`tf.float32`	`np.float32`	Poids des modèles (standard)
`tf.float16`	`np.float16`	Mixed precision training
`tf.int32`	`np.int32`	Labels, indices
`tf.bool`	`np.bool`	Masques, conditions
`tf.string`	-	Texte, chemins de fichiers

Manipulation de shapes :

```
x = tf.constant([[1, 2, 3], [4, 5, 6]])
print(x.shape)              # (2, 3)
print(x.dtype)              # tf.int32
print(tf.rank(x))           # 2
print(tf.size(x))           # 6
# Reshape
x_reshaped = tf.reshape(x, [3, 2])  # (3, 2)
```

Opérations sur les Tenseurs (1/2)

```
a = tf.constant([[1, 2], [3, 4]])
b = tf.constant([[5, 6], [7, 8]])

# Opérations élément par élément
addition = tf.add(a, b)          # ou a + b
soustraction = tf.subtract(a, b) # ou a - b
multiplication = tf.multiply(a, b) # ou a * b
division = tf.divide(a, b)       # ou a / b

# Opérations matricielles
matmul = tf.matmul(a, b)         # Produit matriciel
transpose = tf.transpose(a)       # Transposée
```

Opérations sur les Tenseurs (2/2)

```
# Réductions
somme = tf.reduce_sum(a)          # Somme de tous les éléments
moyenne = tf.reduce_mean(a)       # Moyenne
maximum = tf.reduce_max(a)        # Maximum
minimum = tf.reduce_min(a)        # Minimum

# Réductions avec axes
somme_lignes = tf.reduce_sum(a, axis=0)  # Somme par colonne
somme_colonnes = tf.reduce_sum(a, axis=1) # Somme par ligne
```

Broadcasting

Le **broadcasting** permet d'effectuer des opérations entre tenseurs de shapes différentes.
Règles de broadcasting :

1. Si les tenseurs n'ont pas le même rang, ajouter des dimensions de taille 1 à gauche
2. Les dimensions sont compatibles si elles sont égales ou si l'une vaut 1
3. Les tenseurs sont étendus virtuellement le long des dimensions de taille 1

```
# Exemple 1 : Vecteur + Scalaire
a = tf.constant([1, 2, 3])  # Shape: (3,)
b = tf.constant(10)          # Shape: ()
resultat = a + b             # Shape: (3,) → [11, 12, 13]

# Exemple 2 : Matrice + Vecteur
matrice = tf.constant([[1, 2, 3], [4, 5, 6]])  # (2, 3)
vecteur = tf.constant([10, 20, 30])             # (3,)
resultat = matrice + vecteur  # (2, 3) → [[11,22,33], [14,25,36]]
```

Indexing et Slicing (1/2)

```
tenseur = tf.constant([
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]],
    [[9, 10], [11, 12]]
])  # Shape: (3, 2, 2)

# Indexing simple
print(tenseur[0])           # Première "tranche" (2, 2)
print(tenseur[0, 1])        # (2,)
print(tenseur[0, 1, 1])     # Scalaire → 4

# Slicing
print(tenseur[:2])          # Deux premières tranches (2, 2, 2)
print(tenseur[:, 0, :])     # Première ligne de chaque tranche (3, 2)
print(tenseur[::2])         # Une tranche sur deux (2, 2, 2)
```

Indexing et Slicing (2/2)

```
# Slicing avancé
print(tenseur[1:, :1, :])   # (2, 1, 2)

# Utilisation de tf.gather
indices = tf.constant([0, 2])
selection = tf.gather(tenseur, indices)  # (2, 2, 2)
```

TP : Manipulation de Tenseurs

Objectif : Maîtriser les opérations de base sur les tenseurs
Exercices :

1. **Créer un tenseur 3D** représentant un batch de 10 images 28×28 en niveaux de gris (valeurs aléatoires entre 0 et 1)
2. **Normalisation :** Normaliser ce batch (moyenne=0, écart-type=1) en utilisant `tf.reduce_mean` et `tf.math.reduce_std`
3. **Extraction :** Extraire les 5 premières images et afficher leur shape
4. **Opération :** Calculer la différence entre chaque image et la moyenne du batch
5. **Reshape :** Aplatir chaque image en vecteur 1D (shape finale : `(10, 784)`)

Bonus : Créer une fonction qui prend un tenseur et retourne ses statistiques (min, max, mean, std)

Calcul Automatique de Gradients

Le **calcul automatique de gradients** (autodiff) est au cœur du Deep Learning.

Pourquoi ?

- L'apprentissage = minimiser une fonction de perte
- Minimisation = suivre le gradient (descente de gradient)
- Calcul manuel de gradients = impossible pour des millions de paramètres

TensorFlow solution : `tf.GradientTape`

Cette classe enregistre les opérations pour calculer automatiquement les dérivées.

Principe :

$$\theta_{nouveau} = \theta_{ancien} - \alpha \cdot \nabla_\theta L(\theta)$$

Où :

- θ = paramètres du modèle
- α = learning rate
- $\nabla_\theta L$ = gradient de la loss par rapport à θ

GradientTape – Exemple Simple (1/2)

```python
import tensorflow as tf

# Variable à optimiser
x = tf.Variable(3.0)

# Contexte d'enregistrement
with tf.GradientTape() as tape:
    # Fonction à dériver : f(x) = x²
    y = x ** 2

# Calcul du gradient : dy/dx = 2x
gradient = tape.gradient(y, x)
print(f"x = {x.numpy()}")
print(f"y = x² = {y.numpy()}")
print(f"dy/dx = {gradient.numpy()}")  # 6.0 (car 2*3)
```

GradientTape – Exemple Simple (2/2)

```
# Exemple plus complexe
with tf.GradientTape() as tape:
    # f(x) = x³ + 2x² - 5x + 3
    y = x**3 + 2*x**2 - 5*x + 3

gradient = tape.gradient(y, x)
# df/dx = 3x² + 4x - 5
print(f"Gradient: {gradient.numpy()}")  # 34.0 (car 3*9 + 4*3 - 5)
```

GradientTape – Gradients Multiples (1/2)

```
# Plusieurs variables
x = tf.Variable(2.0)
y = tf.Variable(3.0)

with tf.GradientTape() as tape:
    # z = x²y + y³
    z = x**2 * y + y**3

# Gradients par rapport à x et y
grad_x, grad_y = tape.gradient(z, [x, y])

print(f"∂z/∂x = {grad_x.numpy()}")  # 2xy = 2*2*3 = 12
print(f"∂z/∂y = {grad_y.numpy()}")  # x² + 3y² = 4 + 27 = 31
```

GradientTape – Gradients Multiples (2/2)

```
# Gradient de gradients (dérivée seconde)
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
        y = x ** 3

    # Première dérivée
    dy_dx = inner_tape.gradient(y, x)

# Deuxième dérivée
d2y_dx2 = outer_tape.gradient(dy_dx, x)
print(f"d²y/dx² = {d2y_dx2.numpy()}")  # 6x = 12
```

Descente de Gradient from Scratch (1/2)

```
# Données synthétiques : y = 2x + 1 + bruit
X = tf.constant(np.random.rand(100, 1), dtype=tf.float32)
y_true = 2 * X + 1 + tf.random.normal([100, 1], stddev=0.1)

# Paramètres à apprendre
w = tf.Variable(tf.random.normal([1, 1]))
b = tf.Variable(tf.zeros([1]))

# Hyperparamètres
learning_rate = 0.1
epochs = 100
```

Descente de Gradient from Scratch (2/2)

```
# Entraînement
for epoch in range(epochs):
    with tf.GradientTape() as tape:
        # Prédiction : y_pred = wx + b
        y_pred = tf.matmul(X, w) + b

        # Loss : Mean Squared Error
        loss = tf.reduce_mean(tf.square(y_true - y_pred))

    # Calcul des gradients
    gradients = tape.gradient(loss, [w, b])

    # Mise à jour des paramètres
    w.assign_sub(learning_rate * gradients[0])
    b.assign_sub(learning_rate * gradients[1])

    if epoch % 20 == 0:
        print(f"Epoch {epoch}: Loss = {loss.numpy():.4f}, w = {w.numpy()[0,0]:.4f}, b = {b.numpy()[0]:.4f}")

print(f"\nParamètres finaux: w ≈ 2.0, b ≈ 1.0")
```

TP : Implémentation de Gradient Descent

Objectif : Implémenter une régression logistique from scratch avec TensorFlow
Dataset : Classification binaire (2 classes, 2 features)
Étapes :

1. Générer un dataset synthétique avec `sklearn.datasets.make_classification`
2. Initialiser les poids `W` et le biais `b`
3. Définir la fonction sigmoid : $\sigma(z) = \frac{1}{1+e^{-z}}$
4. Calculer la loss (binary cross-entropy) : $L = -\frac{1}{n}\sum[y\log(\hat{y}) + (1-y)\log(1-\hat{y})]$
5. Utiliser `GradientTape` pour calculer les gradients
6. Mettre à jour les paramètres pendant 1000 epochs
7. Calculer l'accuracy finale

Formules :

- $\hat{y} = \sigma(WX + b)$
- $\nabla_W L$, $\nabla_b L$ calculés automatiquement

Introduction à Keras

Keras est l'API haut niveau de TensorFlow pour construire des réseaux de neurones.

Avantages :

- Syntaxe intuitive et pythonique
- Prototypage rapide
- Support de multiples backends (TensorFlow historiquement)
- Production-ready

Trois façons de construire un modèle :

1. **Sequential API** : Modèles linéaires simples
2. **Functional API** : Modèles complexes avec branches
3. **Subclassing** : Flexibilité maximale, contrôle total

Nous allons explorer chaque approche avec des exemples concrets.

Sequential API (1/2)

Utilisation : Empiler des couches séquentiellement (pas de branches)

```
from tensorflow import keras
from tensorflow.keras import layers

# Méthode 1 : Liste de couches
model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(784,)),
    layers.Dropout(0.2),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(10, activation='softmax')
])
```

Sequential API (2/2)

```
# Méthode 2 : Ajout progressif
model = keras.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(784,)))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

# Afficher l'architecture
model.summary()
```

Functional API

Utilisation : Modèles avec architectures complexes (multi-inputs, skip connections, etc.)

```
from tensorflow.keras import Input, Model
from tensorflow.keras.layers import Dense, Concatenate

# Définition des inputs
input_a = Input(shape=(32,), name='input_a')
input_b = Input(shape=(64,), name='input_b')

# Branches séparées
branch_a = Dense(64, activation='relu')(input_a)
branch_b = Dense(64, activation='relu')(input_b)

# Fusion des branches
merged = Concatenate()([branch_a, branch_b])

# Couches communes
output = Dense(128, activation='relu')(merged)
output = Dense(10, activation='softmax')(output)

# Création du modèle
```

Model Subclassing

Utilisation : Maximum de flexibilité, logique custom dans `call()`

```
class CustomModel(keras.Model):
    def __init__(self):
        super(CustomModel, self).__init__()
        self.dense1 = layers.Dense(64, activation='relu')
        self.dense2 = layers.Dense(64, activation='relu')
        self.dropout = layers.Dropout(0.2)
        self.output_layer = layers.Dense(10, activation='softmax')

    def call(self, inputs, training=False):
        x = self.dense1(inputs)
        if training:
            x = self.dropout(x)
        x = self.dense2(x)
        if training:
            x = self.dropout(x)
        return self.output_layer(x)

# Instanciation
model = CustomModel()
```

Couches Principales

Couche	Usage	Paramètres clés
Dense	Couche fully-connected	`units`, `activation`
Conv2D	Convolution 2D (images)	`filters`, `kernel_size`, `strides`
MaxPooling2D	Réduction spatiale	`pool_size`, `strides`
LSTM	Séquences (texte, time series)	`units`, `return_sequences`
GRU	Variante LSTM plus légère	`units`, `return_sequences`
Dropout	Régularisation	`rate` (ex: 0.2 = 20%)
BatchNormalization	Normalisation des activations	`momentum`, `epsilon`
Embedding	Représentation de tokens	`input_dim`, `output_dim`
Flatten	Aplatir tenseur multidim	-
GlobalAveragePooling2D	Pooling global	-

Couches Principales

Exemple de combinaison :

```
model = keras.Sequential([
    layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D(2),
    layers.Flatten(),
    layers.Dense(10, activation='softmax')
])
```

Fonctions d'Activation

Activation	Formule	Usage
ReLU	$f(x) = \max(0, x)$	Standard pour couches cachées
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	Classification binaire (output)
Softmax	$f(x_i) = \frac{e^{x_i}}{\sum e^{x_j}}$	Classification multi-classe
Tanh	$f(x) = \tanh(x)$	Centré sur 0, alternative à sigmoid
LeakyReLU	$f(x) = \max(\alpha x, x)$	Évite dying ReLU
ELU	$f(x) = x$ si $x > 0$ sinon $\alpha(e^x - 1)$	Moyenne proche de 0
Swish	$f(x) = x \cdot \sigma(x)$	Performances parfois meilleures

Fonctions d'Activation

```
# Utilisation
layers.Dense(64, activation='relu')
# ou
layers.Dense(64, activation=tf.nn.relu)
# ou
layers.Dense(64)
layers.Activation('relu')
```

Compilation du Modèle

Compilation = Configuration de l'entraînement

```
model.compile(
    optimizer='adam',           # Algorithme d'optimisation
    loss='sparse_categorical_crossentropy',  # Fonction de perte
    metrics=['accuracy']        # Métriques à suivre
)
```

Principaux optimiseurs :

- `adam` : Adaptive Moment Estimation (standard)
- `sgd` : Stochastic Gradient Descent
- `rmsprop` : RMSProp
- `adamw` : Adam avec weight decay

Compilation du Modèle

Principales loss functions :

- `binary_crossentropy` : Classification binaire
- `categorical_crossentropy` : Multi-classe (one-hot)
- `sparse_categorical_crossentropy` : Multi-classe (labels entiers)
- `mse` (Mean Squared Error) : Régression

Entraînement du Modèle (1/2)

```
# Entraînement simple
history = model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=32,
    validation_split=0.2,  # 20% pour validation
    verbose=1
)

# Avec dataset de validation séparé
history = model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=32,
    validation_data=(X_val, y_val)
)
```

Entraînement du Modèle (2/2)

```
# Avec callbacks
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

callbacks = [
    EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True
    ),
    ModelCheckpoint(
        'best_model.keras',
        save_best_only=True
    )
]

history = model.fit(
    X_train, y_train,
    epochs=50,
```

Évaluation et Prédiction (1/2)

```
# Évaluation sur le test set
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test accuracy: {test_accuracy:.4f}")

# Prédictions
predictions = model.predict(X_test)
print(predictions.shape)  # (n_samples, n_classes)

# Classe prédite
predicted_classes = np.argmax(predictions, axis=1)
```

Évaluation et Prédiction (2/2)

```
# Prédiction sur un seul échantillon
single_prediction = model.predict(X_test[0:1])
print(f"Prédiction: classe {np.argmax(single_prediction)}")

# Obtenir les probabilités
probabilities = predictions[0]
for i, prob in enumerate(probabilities):
    print(f"Classe {i}: {prob:.4f}")
```

TP : Classification MNIST

Objectif : Créer un réseau de neurones pour classifier les chiffres manuscrits (MNIST)
Dataset MNIST :

- 60 000 images d'entraînement
- 10 000 images de test
- Images 28×28 pixels en niveaux de gris
- 10 classes (chiffres 0-9)

Tâches :

1. Charger le dataset MNIST avec `keras.datasets.mnist`
2. Prétraiter les données (normalisation 0-1, reshape si nécessaire)
3. Créer un modèle Sequential avec :
    - Couche Flatten (28×28 → 784)
    - 2-3 couches Dense avec ReLU
    - Dropout pour régularisation
    - Couche de sortie avec Softmax

TP : Classification MNIST

4. Compiler avec optimizer Adam, loss sparse_categorical_crossentropy
5. Entraîner pendant 10 epochs
6. Évaluer sur le test set et afficher l'accuracy
7. Visualiser quelques prédictions

Réseaux de Neurones Convolutifs (CNN)

Les **CNN** sont spécialisés pour les données structurées spatialement (images, vidéos).
Principes clés :

1. **Convolution** : Détection de features locales via des filtres
2. **Pooling** : Réduction de dimensionnalité
3. **Hiérarchie** : Features simples → features complexes
4. **Invariance** : Robustesse aux translations

Architecture typique :

```
Input → [Conv → ReLU → Pool] × N → Flatten → Dense → Output
```

Avantages :

- Moins de paramètres que Dense (poids partagés)
- Capture les patterns spatiaux
- Invariance par translation

Couche de Convolution

Opération de convolution :
Une convolution applique un **filtre** (ou kernel) sur l'image pour extraire des features.

```
# Exemple de filtre 3×3 pour détecter les contours verticaux
filtre_vertical = np.array([
    [-1, 0, 1],
    [-1, 0, 1],
    [-1, 0, 1]
])
```

Paramètres :

- **Filters** : Nombre de filtres à apprendre
- **Kernel size** : Taille du filtre (ex: 3×3, 5×5)
- **Strides** : Pas de déplacement du filtre
- **Padding** : `'valid'` (pas de padding) ou `'same'` (garde taille)

Couche de Convolution

```
layers.Conv2D(
    filters=32,
    kernel_size=(3, 3),
    strides=(1, 1),
    padding='same',
    activation='relu'
)
```

Pooling (1/2)

Le **pooling** réduit la taille spatiale des feature maps.
Types de pooling :

1. **Max Pooling** : Garde la valeur maximale dans chaque région
2. **Average Pooling** : Moyenne des valeurs

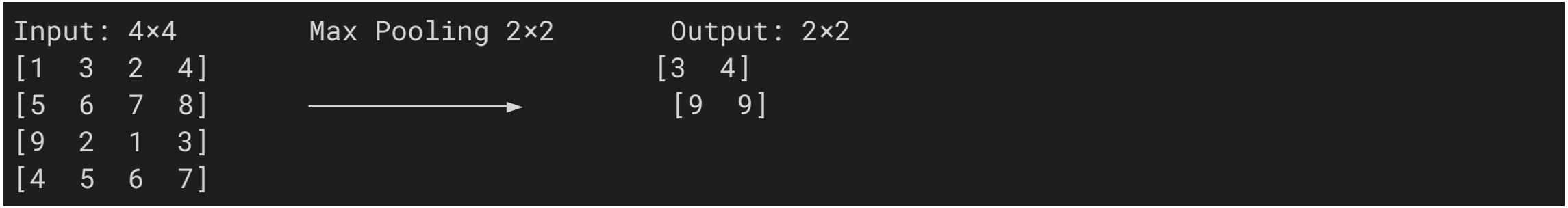
```
# Max Pooling 2×2
layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2))

# Résultat : (H, W) → (H/2, W/2)
```

Pooling (2/2)

Avantages :

- Réduit le nombre de paramètres
- Apporte de l'invariance aux petites translations
- Réduit l'overfitting

Exemple :



Architecture CNN Complète (1/2)

```
model = keras.Sequential([
    # Bloc 1
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),

    # Bloc 2
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    # Bloc 3
    layers.Conv2D(64, (3, 3), activation='relu'),

    # Classification
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])
```

Architecture CNN Complète (2/2)

```
Sortie typique:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
max_pooling2d_1 (MaxPooling2D(None, 5, 5, 64)          0
conv2d_2 (Conv2D)            (None, 3, 3, 64)          36928
flatten (Flatten)            (None, 576)               0
dense (Dense)                (None, 64)                36928
dropout (Dropout)            (None, 64)                0
dense_1 (Dense)              (None, 10)                650
=================================================================
```

CNN sur MNIST – Amélioration (1/2)

```
# Dataset MNIST avec canal
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
X_train = X_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0
X_test = X_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0

# Modèle CNN
model = keras.Sequential([
    layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
    layers.BatchNormalization(),
    layers.MaxPooling2D(2),

    layers.Conv2D(64, 3, activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(2),

    layers.Conv2D(64, 3, activation='relu'),
    layers.BatchNormalization(),
```

CNN sur MNIST – Amélioration (2/2)

```
    # Suite du modèle
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

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

history = model.fit(
    X_train, y_train,
    epochs=10,
    validation_split=0.1,
    batch_size=128
)

# Résultat attendu : ~99% accuracy
```

Data Augmentation (1/2)

Data Augmentation = Augmenter artificiellement la taille du dataset en appliquant des transformations.
Objectif : Réduire l'overfitting et améliorer la généralisation

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Définir les transformations
datagen = ImageDataGenerator(
    rotation_range=10,              # Rotation ±10°
    width_shift_range=0.1,          # Translation horizontale 10%
    height_shift_range=0.1,         # Translation verticale 10%
    zoom_range=0.1,                 # Zoom ±10%
    horizontal_flip=True,           # Flip horizontal
    fill_mode='nearest'             # Remplissage des pixels manquants
)
```

Data Augmentation (2/2)

```
# Entraînement avec augmentation
history = model.fit(
    datagen.flow(X_train, y_train, batch_size=32),
    epochs=50,
    validation_data=(X_val, y_val)
)
```

Architectures CNN Célèbres

Modèle	Année	Particularités
LeNet-5	1998	Premier CNN, reconnaissance de chiffres
AlexNet	2012	Révolution ImageNet, ReLU, Dropout
VGG	2014	Couches 3×3 empilées, simple et profond
ResNet	2015	Skip connections, 152 couches possibles
Inception	2014	Multi-scale features, efficient
MobileNet	2017	Optimisé mobile, depthwise separable conv
EfficientNet	2019	Scaling optimal depth/width/resolution

ResNet Skip Connection :

```
x = layers.Conv2D(64, 3, padding='same')(input)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)

# Skip connection
x = layers.Add()([x, input])
```

Transfer Learning

Le **Transfer Learning** consiste à utiliser un modèle pré-entraîné sur une large base de données (ex: ImageNet) et l'adapter à une nouvelle tâche.

Avantages :

- Besoin de moins de données
- Entraînement plus rapide
- Meilleures performances sur petits datasets

Deux approches :

1. **Feature Extraction** : Geler les couches pré-entraînées, entraîner seulement les nouvelles couches
2. **Fine-tuning** : Dégeler quelques couches et les ré-entraîner

Modèles disponibles dans Keras :

- ResNet50, VGG16, InceptionV3, MobileNetV2, EfficientNet, etc.

Transfer Learning – Feature Extraction (1/2)

```python
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input

# 1. Charger le modèle pré-entraîné (sans la couche de classification)
base_model = ResNet50(
    weights='imagenet',
    include_top=False,  # Exclure la couche Dense finale
    input_shape=(224, 224, 3)
)

# 2. Geler les poids du modèle de base
base_model.trainable = False

# 3. Ajouter nos propres couches de classification
inputs = keras.Input(shape=(224, 224, 3))
x = preprocess_input(inputs)  # Prétraitement spécifique à ResNet
x = base_model(x, training=False)  # training=False pour BatchNorm
```

Transfer Learning – Feature Extraction (2/2)

```
# Suite : ajout des couches de classification
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(256, activation='relu')(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(10, activation='softmax')(x)  # 10 classes custom

model = keras.Model(inputs, outputs)

# 4. Compilation et entraînement
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

history = model.fit(train_dataset, epochs=10, validation_data=val_dataset)
```

Transfer Learning – Fine-tuning (1/2)

```
# 1. Feature extraction d'abord (voir slide précédente)
# ... entraînement initial ...

# 2. Dégeler les couches supérieures du modèle de base
base_model.trainable = True

# Geler toutes les couches sauf les 20 dernières
for layer in base_model.layers[:-20]:
    layer.trainable = False

print(f"Nombre de couches entraînables: {len(base_model.trainable_variables)}")
```

Transfer Learning – Fine-tuning (2/2)

```
# 3. Recompiler avec un learning rate plus faible
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-5),  # LR faible !
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# 4. Fine-tuning
history_fine = model.fit(
    train_dataset,
    epochs=10,
    validation_data=val_dataset
)

# Le fine-tuning améliore généralement de 1-5% l'accuracy
```

Important : Toujours utiliser un learning rate très faible pour le fine-tuning !

TP : Transfer Learning avec ResNet50

Objectif : Classifier des images de chiens vs chats avec Transfer Learning
Dataset : Dogs vs Cats (disponible sur Kaggle ou TensorFlow Datasets)
Étapes :

1. **Préparer les données**
   - Utiliser `tf.keras.utils.image_dataset_from_directory`
   - Images redimensionnées à 224×224
   - Split train/val (80/20)
2. **Feature Extraction**
   - Charger ResNet50 pré-entraîné sur ImageNet
   - Geler tous les poids
   - Ajouter GlobalAveragePooling + Dense(1, sigmoid)
   - Entraîner 5 epochs

TP : Transfer Learning avec ResNet50

3. **Fine-tuning**
    - Dégeler les 30 dernières couches de ResNet50
    - Learning rate = 1e-5
    - Entraîner 5 epochs supplémentaires
4. **Évaluation**
    - Comparer accuracy avant/après fine-tuning
    - Visualiser quelques prédictions

Réseaux de Neurones Récurrents (RNN)

Les **RNN** sont conçus pour traiter des **séquences** (texte, time series, audio).

Différence avec les réseaux classiques :

- Les réseaux classiques ne considèrent pas l'ordre
- Les RNN ont une "mémoire" des inputs précédents

Architecture de base :

Formule :

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$
$$y_t = W_{hy}h_t + b_y$$

Problème des RNN Simples

Problèmes des RNN vanilles :

1. **Vanishing Gradient** : Les gradients disparaissent lors de la backpropagation dans le temps
2. **Exploding Gradient** : Les gradients explosent
3. **Difficultés à capturer les dépendances long-terme**

Solutions :

- **LSTM** (Long Short-Term Memory) : Portes pour contrôler le flux d'information
- **GRU** (Gated Recurrent Unit) : Version simplifiée de LSTM

Ces architectures permettent de résoudre le problème du vanishing gradient et de capturer des dépendances à long terme.

LSTM (Long Short-Term Memory)

Architecture LSTM :

Une cellule LSTM possède 3 portes :

1. **Forget Gate** (porte d'oubli) : Décide quoi oublier
2. **Input Gate** (porte d'entrée) : Décide quoi ajouter
3. **Output Gate** (porte de sortie) : Décide quoi sortir

Formules :