LISV Laboratoire d'ingénierie des systèmes de Versailles
UVSQ université PARIS-SACLAY

TP - Keras

Halim Djerroud

révision 0.1

Objectifs

Ce TP a pour but de familiariser les participants avec :
— La création, l'entraînement et l'évaluation de modèles de réseaux de neurones en utilisant Keras.
— La distinction entre les réseaux monocouches et multicouches.
— L'utilisation d'hyperparamètres pour améliorer les performances.
— L'interprétation des résultats et leur visualisation.

Prérequis

— Connaissances de base en Python.
— Familiarité avec les notions fondamentales des réseaux de neurones.
— Installation des bibliothèques `TensorFlow/Keras`, `NumPy`, et `Matplotlib`.

1 Partie 1 : Jeu de donnée Iris

Objectif

L'objectif de cet exercice est de construire et d'entraîner un réseau de neurones en utilisant la bibliothèque `Keras`. Ce réseau sera utilisé pour classifier le dataset `Iris`, un jeu de données bien connu en apprentissage automatique.

Description du Dataset Iris

Le dataset `Iris` contient 150 observations réparties en trois classes (*setosa*, *versicolor*, *virginica*). Chaque observation comprend quatre caractéristiques :

— longueur des sépales,
— largeur des sépales,
— longueur des pétales,
— largeur des pétales.

Le but est de prédire la classe d'une fleur en fonction de ses caractéristiques.

Instructions

1. Chargement des données

Utilisez la bibliothèque `sklearn` pour charger le dataset Iris. Voici un exemple de code pour commencer :

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Chargement des données
iris = load_iris()
X = iris.data
y = iris.target

# Encodage One-Hot pour les labels
ohe = OneHotEncoder(sparse=False)
y = ohe.fit_transform(y.reshape(-1, 1))

# Division en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

2. Construction du Modèle

Créez un réseau de neurones en utilisant `Keras`. Le modèle doit :

— avoir une couche d'entrée correspondant aux 4 caractéristiques du dataset,
— comporter une ou deux couches cachées avec des neurones et une activation `relu`,
— avoir une couche de sortie avec 3 neurones (une pour chaque classe) et une activation `softmax`.

Voici un exemple de code de base :

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Définition du modèle
model = Sequential([
    Dense(16, input_dim=4, activation='relu'),
    Dense(8, activation='relu'),
    Dense(3, activation='softmax')
])

# Compilation du modèle
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

3. Entraînement du Modèle

Entraînez le modèle sur les données d'entraînement :

```
history = model.fit(X_train, y_train, epochs=50, batch_size=8, validation_split=0.2)
```

4. Évaluation du Modèle

Évaluez les performances du modèle sur les données de test :

```
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Loss: {loss}, Accuracy: {accuracy}")
```

5. Visualisation des Performances

Tracez les courbes de la précision et de la perte pour l'entraînement et la validation :

```
import matplotlib.pyplot as plt

# Courbe de précision
plt.plot(history.history['accuracy'], label='Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Précision')
plt.show()

# Courbe de perte
plt.plot(history.history['loss'], label='Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title('Perte')
plt.show()
```

2 Partie 2 : Réseaux de neurones monocouches avec MNIST

2.1 Chargement et exploration des données

Nous utiliserons le jeu de données **MNIST** pour classer des chiffres manuscrits.

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

# Charger les données
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalisation des données
x_train = x_train / 255.0
x_test = x_test / 255.0

# Afficher quelques images
for i in range(9):
    plt.subplot(3, 3, i + 1)
    plt.imshow(x_train[i], cmap='gray')
    plt.title(f"Label: {y_train[i]}")
    plt.axis('off')
plt.show()
```

2.2 Préparation des données

```
# Aplatir les images en vecteurs
x_train_flattened = x_train.reshape(x_train.shape[0], -1)
x_test_flattened = x_test.reshape(x_test.shape[0], -1)

# Encodage des labels
from tensorflow.keras.utils import to_categorical
y_train_onehot = to_categorical(y_train, num_classes=10)
y_test_onehot = to_categorical(y_test, num_classes=10)
```

2.3 Création d'un modèle monocouche

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Modèle monocouche
model = Sequential([
    Dense(10, input_shape=(784,), activation='softmax')
])

# Compilation du modèle
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

2.4 Entraînement et évaluation du modèle

```
# Entraînement
history = model.fit(x_train_flattened, y_train_onehot,
                    validation_data=(x_test_flattened, y_test_onehot),
                    epochs=10, batch_size=32)

# Évaluation
test_loss, test_accuracy = model.evaluate(x_test_flattened, y_test_onehot)
print(f"Test Accuracy: {test_accuracy:.2f}")
```

2.5 Visualisation des performances

```
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Test Accuracy')
plt.title('Performance du modèle monocouche')
plt.xlabel('Époch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

3 Partie 3 : Réseaux de neurones multicouches

3.1 Création d'un modèle multicouche

```
# Modèle multicouche
model = Sequential([
    Dense(128, input_shape=(784,), activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

# Compilation du modèle
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

3.2 Entraînement et évaluation

```
# Entraînement
history = model.fit(x_train_flattened, y_train_onehot,
                    validation_data=(x_test_flattened, y_test_onehot),
                    epochs=10, batch_size=32)

# Évaluation
test_loss, test_accuracy = model.evaluate(x_test_flattened, y_test_onehot)
print(f"Test Accuracy: {test_accuracy:.2f}")
```

3.3 Analyse des résultats

Comparez les performances du modèle monocouche avec celles du modèle multicouche en observant les métriques `accuracy` et `loss`.

```
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Test Accuracy')
plt.title('Performance du modèle multicouche')
plt.xlabel('Époch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

4 Partie 4 : Optimisation des hyperparamètres

4.1 Modification du batch size et du learning rate

Ajoutez des arguments pour ajuster les paramètres comme la taille des batchs, le taux d'apprentissage et le nombre d'époques.

```
from tensorflow.keras.optimizers import Adam

# Modèle optimisé
model = Sequential([
    Dense(128, input_shape=(784,), activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

# Changer le learning rate
optimizer = Adam(learning_rate=0.001)

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

4.2 Évaluation après optimisation

Répétez les étapes précédentes pour observer les changements.

5 Partie 5 : Régularisation dans les Réseaux de Neurones

Objectif

Dans cet exercice, nous allons introduire des techniques de régularisation dans un modèle multicouche pour améliorer ses performances et réduire le surapprentissage. Les techniques explorées incluront :

— Dropout
— Régularisation L2

5.1 Ajout de Dropout

Modifiez le modèle multicouche précédent pour inclure des couches de `Dropout`. Ces couches désactivent aléatoirement une fraction des neurones pendant l'entraînement.

```
from tensorflow.keras.layers import Dropout

# Modèle avec Dropout
model = Sequential([
    Dense(128, input_shape=(784,), activation='relu'),
    Dropout(0.3),  # Désactivation de 30% des neurones
    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax')
])

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

5.2 Régularisation L2

Ajoutez une pénalisation de norme L2 dans les couches pour réduire la complexité du modèle.

```
from tensorflow.keras.regularizers import l2

# Modèle avec régularisation L2
model = Sequential([
    Dense(128, input_shape=(784,), activation='relu', kernel_regularizer=l2(0.01)),
    Dense(64, activation='relu', kernel_regularizer=l2(0.01)),
    Dense(10, activation='softmax')
])

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

5.3 Comparaison des Performances

Entraînez les deux modèles (avec `Dropout` et régularisation L2) sur le même jeu de données, puis comparez leurs performances en termes de précision et de perte.

```
# Entraînement avec Dropout
history_dropout = model.fit(x_train_flattened, y_train_onehot,
                            validation_data=(x_test_flattened, y_test_onehot),
                            epochs=10, batch_size=32)

# Entraînement avec régularisation L2
history_l2 = model.fit(x_train_flattened, y_train_onehot,
                       validation_data=(x_test_flattened, y_test_onehot),
                       epochs=10, batch_size=32)
```

5.4 Visualisation des Résultats

Affichez les courbes de précision et de perte pour les deux modèles et interprétez les résultats.

```
# Visualisation des performances
plt.plot(history_dropout.history['val_accuracy'], label='Dropout')
plt.plot(history_l2.history['val_accuracy'], label='L2 Regularization')
plt.title('Comparaison des performances')
plt.xlabel('Époch')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.show()
```

6 Partie 6 : Utilisation des Callbacks dans Keras

Objectif

Cet exercice a pour but d'explorer les fonctionnalités avancées offertes par les `Callbacks` de Keras. Vous apprendrez à :

- Arrêter l'entraînement automatiquement si les performances cessent de s'améliorer (`EarlyStopping`),
- Ajuster dynamiquement le taux d'apprentissage (`ReduceLROnPlateau`),
- Sauvegarder le meilleur modèle (`ModelCheckpoint`).

6.1 Ajout de Callbacks au Modèle

Créez un modèle multicouche similaire aux exercices précédents. Ajoutez des `Callbacks` pour :

- Arrêter l'entraînement si la précision de validation ne s'améliore pas après 5 époques.
- Réduire le taux d'apprentissage de moitié si la perte de validation ne diminue pas après 3 époques.
- Sauvegarder automatiquement les poids du meilleur modèle.

```
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint

# Définition des callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, verbose=1)
model_checkpoint = ModelCheckpoint('best_model.h5', monitor='val_accuracy', save_best_only=True)

callbacks = [early_stopping, reduce_lr, model_checkpoint]

# Création du modèle
model = Sequential([
    Dense(128, input_shape=(784,), activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

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

6.2 Entraînement avec Callbacks

Entraînez le modèle avec les callbacks définis. Observez les ajustements automatiques pendant l'entraînement.

```
history = model.fit(x_train_flattened, y_train_onehot,
                    validation_data=(x_test_flattened, y_test_onehot),
                    epochs=50,
                    batch_size=32,
                    callbacks=callbacks)
```

6.3 Évaluation du Meilleur Modèle

Chargez le meilleur modèle sauvegardé et évaluez ses performances sur le jeu de test.

```
from tensorflow.keras.models import load_model

# Chargement du meilleur modèle
best_model = load_model('best_model.h5')

# Évaluation
test_loss, test_accuracy = best_model.evaluate(x_test_flattened, y_test_onehot)
print(f"Test Accuracy of Best Model: {test_accuracy:.2f}")
```

6.4 Analyse des Résultats

Visualisez les courbes d'entraînement pour comprendre l'impact des callbacks.

```
import matplotlib.pyplot as plt

# Visualisation des courbes
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Impact des Callbacks sur la Précision')
plt.xlabel('Épochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

7 Partie 7 : Visualisation des Activations des Couches

Objectif

Dans cet exercice, nous allons explorer les activations des différentes couches du modèle après avoir passé une image d'entrée. L'objectif est de mieux comprendre comment chaque couche transforme les données pour arriver à la prédiction finale.

7.1 Chargement du Modèle Entraîné

Reprenez le modèle multicouche entraîné dans l'une des parties précédentes.

```
# Charger le modèle entraîné
from tensorflow.keras.models import Model

model = Sequential([
    Dense(128, input_shape=(784,), activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train_flattened, y_train_onehot, epochs=10, batch_size=32)
```

7.2 Sélection d'une Image de Test

Choisissez une image à partir du jeu de données de test.

```
# Choisir une image aléatoire
import numpy as np

image_index = np.random.randint(0, x_test_flattened.shape[0])
test_image = x_test_flattened[image_index].reshape(1, 784)
print(f"Label réel : {np.argmax(y_test_onehot[image_index])}")
```

7.3 Création d'un Modèle pour Extraire les Activations

Créez un modèle Keras intermédiaire pour extraire les activations des couches internes.

```
# Créer un modèle intermédiaire pour accéder aux activations
layer_outputs = [layer.output for layer in model.layers]
activation_model = Model(inputs=model.input, outputs=layer_outputs)

# Calcul des activations
activations = activation_model.predict(test_image)
```

7.4 Visualisation des Activations

Visualisez les activations des différentes couches sous forme de graphiques.

```
import matplotlib.pyplot as plt

# Fonction pour visualiser les activations
```

```python
def plot_activations(activations, model):
    for i, activation in enumerate(activations):
        plt.figure(figsize=(15, 5))
        plt.title(f"Activations de la couche {i + 1} ({model.layers[i].name})")
        if len(activation.shape) == 2:  # Fully connected layers
            plt.plot(activation[0])
        else:  # Feature maps
            for j in range(min(10, activation.shape[-1])):  # Max 10 feature maps
                plt.subplot(1, 10, j + 1)
                plt.imshow(activation[0, :, :, j], cmap='viridis')
                plt.axis('off')
        plt.show()

# Visualisation
plot_activations(activations, model)
```

8 Partie 8 : Interprétation des Prédictions avec SHAP

Objectif

L'objectif de cet exercice est d'introduire une méthode d'interprétation des modèles de réseaux de neurones en utilisant la bibliothèque `SHAP`. Apprendront à expliquer les prédictions d'un modèle en identifiant les caractéristiques les plus importantes qui influencent les décisions du modèle.

8.1 Installation de SHAP

Assurez-vous que la bibliothèque `shap` est installée.

```
pip install shap
```

8.2 Chargement du Modèle Entraîné

Reprenez le modèle multicouche entraîné dans les exercices précédents.

```
# Charger le modèle entraîné
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Création et entraînement du modèle
model = Sequential([
    Dense(128, input_shape=(784,), activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train_flattened, y_train_onehot, epochs=10, batch_size=32)
```

8.3 Utilisation de SHAP pour Interpréter les Prédictions

Utilisez SHAP pour expliquer les prédictions du modèle.

```
import shap
import numpy as np

# Sélectionner un échantillon de test
explainer = shap.Explainer(model, x_train_flattened[:100])  # Explique le modèle
shap_values = explainer(x_test_flattened[:10])  # Obtenez les valeurs SHAP pour les 10 premières
```

8.4 Visualisation des Résultats

Visualisez les contributions des caractéristiques aux prédictions du modèle.

```
# Visualisation globale
shap.summary_plot(shap_values, x_test_flattened[:10])

# Visualisation pour une prédiction individuelle
shap.force_plot(explainer.expected_value[0], shap_values.values[0], x_test_flattened[0])
```