Jean-Baptiste Gaillot **Master 2 CHPS**

Projet de HPC pour l'intelligence artificielle
Reconnaissance des panneaux de signalisation

Jeu de données : https ://www.kaggle.com/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign

On importe les modules nécessaires :

[1]:
```
# tensorflow et keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout,␣
↪BatchNormalization, GlobalAveragePooling2D
from tensorflow.keras.regularizers import l2
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
```

[2]:
```
# sklearn
from sklearn.metrics import confusion_matrix
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from sklearn.utils.class_weight import compute_class_weight
```

[3]:
```
# Modules complémentaires
import os
import json
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as mcolors
import math
from PIL import Image
```

1 Récupération et préparation des données

On dispose d'un jeu d'entraînement (de 39209 images) et d'un jeu de test (de 12630 images). Dans le jeu d'entraînement, il y a en réalité 1307 images différentes mais qui existent en 30 versions (de qualités différentes). Il n'y a qu'une image qui possède 29 versions. Pour cette image, une de ses 29 versions à été dupliquée pour avoir 30 versions.

On initialise les données du problème :

[4]:
```
nom_repertoire_dataset = "../Dataset"
taille_image = 32
nb_classe = 43
```

On récupère le jeu de données en écrivant la fonction `lire_donnees` :

- Entrées :
  - `nom_fichier_csv` est un fichier `csv` qui contient les informations sur le jeu de données (tailles des images, classes, chemins relatifs, ...),

— `nom_repertoire` est le nom du répertoire téléchargé,
— `taille_image` est la taille de l'image recadrée.

— Sorties :
— `X` contient les images,
— `y` contient les étiquettes.

[5]:
```
def lire_donnees(nom_fichier_csv, nom_repertoire_dataset, taille_image) :

    df = pd.read_csv(nom_fichier_csv)
    images = []
    etiquettes = []

    for _, ligne in df.iterrows():
        chemin_image = os.path.join(nom_repertoire_dataset, ligne["Path"])
        image = Image.open(chemin_image).convert("RGB")
        image = image.resize((taille_image, taille_image))
        image = np.array(image)

        images.append(image)
        etiquettes.append(ligne["ClassId"])

    X = np.array(images)
    y = np.array(etiquettes)

    res = X, y

    return res
```

On appelle la fonction `lire_donnees` pour le jeu de données :

[6]:
```
X_entrainement, y_entrainement = lire_donnees(os.path.join(nom_repertoire_dataset, "Train.
↪csv"), nom_repertoire_dataset, taille_image)
X_test, y_test = lire_donnees(os.path.join(nom_repertoire_dataset, "Test.csv"),␣
↪nom_repertoire_dataset, taille_image)
```

On obtient les dimensions des jeux de données :

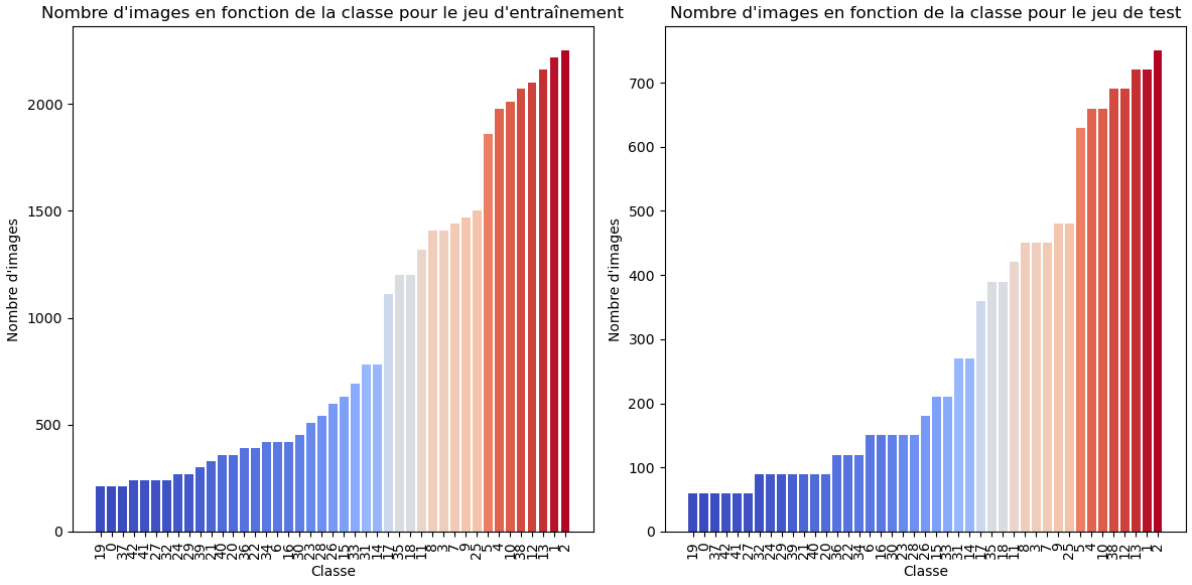
[7]:
```
print("Dimensions de X_entrainement =\n", X_entrainement.shape,
      "\n\nDimensions de y_entrainement =\n", y_entrainement.shape,
      "\n\nDimensions de X_test = \n", X_test.shape,
      "\n\nDimensions de y_test = \n", y_test.shape)
```

```
Dimensions de X_entrainement =
 (39210, 32, 32, 3)

Dimensions de y_entrainement =
 (39210,)

Dimensions de X_test =
 (12630, 32, 32, 3)

Dimensions de y_test =
 (12630,)
```

On calcule et trace le graphique du nombre d'images en fonction de la classe pour les jeux d'entraînement et de test :

[8]:
```
nb_images_par_classe_entrainement = pd.Series(y_entrainement).value_counts().
↪sort_values(ascending = True)
nb_images_par_classe_test = pd.Series(y_test).value_counts().sort_values(ascending = True)
nb_images_par_classe = pd.DataFrame({"classe" : nb_images_par_classe_entrainement.index.
↪astype(str),
                                     "nb_images_entrainement" :␣
↪nb_images_par_classe_entrainement.values,
                                     "nb_images_test" : nb_images_par_classe_test.values})
```

[9]:
```
norme = mcolors.Normalize(vmin = nb_images_par_classe["nb_images_entrainement"].min(),
                          vmax = nb_images_par_classe["nb_images_entrainement"].max())
cmap = plt.colormaps["coolwarm"]
couleurs = cmap(norme(nb_images_par_classe["nb_images_entrainement"].values))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (12, 6))
ax1.bar(nb_images_par_classe["classe"], nb_images_par_classe["nb_images_entrainement"],␣
↪color = couleurs)
ax1.set_xlabel("Classe")
ax1.set_ylabel("Nombre d'images")
ax1.set_title("Nombre d'images en fonction de la classe pour le jeu d'entraînement")
plt.sca(ax1)
plt.xticks(rotation = 90)
ax2.bar(nb_images_par_classe["classe"], nb_images_par_classe["nb_images_test"], color =␣
↪couleurs)
ax2.set_xlabel("Classe")
ax2.set_ylabel("Nombre d'images")
ax2.set_title("Nombre d'images en fonction de la classe pour le jeu de test")
plt.sca(ax2)
plt.xticks(rotation = 90)

plt.tight_layout()
plt.show()
```



On affiche un exemple d'une image du jeu d'entraînement (avec ses 30 versions) :

Commentaires :

— Les jeux d'entraînement de de test ont a peu près la même répartition.
— Les classes sont très déséquilibrées dans les jeux.
— Pour le jeu d'entraînement, il faudrait diviser le nombre d'images de chaque classe par 30 pour avoir le nombre d'image par classe sans compter les 30 versions.

[10]:
```
fig, axes = plt.subplots(3, 10, figsize = (16, 8))
axes = axes.flatten()

for i, axe in enumerate(axes):
    idx = 60 + i
    axe.imshow(X_entrainement[idx])
    label = y_entrainement[idx]
    axe.axis("off")

plt.tight_layout()
plt.show()
```

On normalise les valeurs des pixels entre 0 et 1 :

[11]:
```
X_entrainement = X_entrainement.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
```

On crée les poids associés au nombre d'images par classe du jeu d'entraînement :

[12]:
```
poids_classes = compute_class_weight(class_weight = "balanced",
                                     classes = np.unique(y_entrainement),
                                     y = y_entrainement)
```

On transforme les valeurs numériques associés aux labels solutions en vecteurs one-hot :

[13]:
```
y_entrainement = to_categorical(y_entrainement, nb_classe)
y_test = to_categorical(y_test, nb_classe)
```

2 Construction et entraînement du modèle

On initialise les données de l'entraînement :

[14]:
```
nb_epoque = 180
taille_lot = 32
ratio_validation = 0.2
```

On construit le modèle en écrivant la fonction `construire_modele` :

[15]:
```
def construire_modele(taille_image, nb_classe):

    modele = Sequential()

    # Couche 1
    modele.add(Conv2D(32, (3, 3),
                      activation = 'relu',
                      input_shape = (taille_image, taille_image, 3),
                      padding = 'same'))
    modele.add(BatchNormalization())
    modele.add(MaxPooling2D((2, 2)))

    # Couche 2
    modele.add(Conv2D(64, (3, 3), activation = 'relu', padding = 'same', kernel_regularizer␣
↪= l2(0.005)))
    modele.add(BatchNormalization())
    modele.add(MaxPooling2D((2, 2)))
    modele.add(Dropout(0.125))

    # Couche 3
    modele.add(Conv2D(128, (3, 3), activation = 'relu', padding = 'same', kernel_regularizer␣
↪= l2(0.005)))
    modele.add(BatchNormalization())
    modele.add(MaxPooling2D((2, 2)))
    modele.add(Dropout(0.25))

    # Connection
    modele.add(Flatten())
    modele.add(Dropout(0.5))
    modele.add(Dense(256, activation = 'relu', kernel_regularizer = l2(0.005)))
    modele.add(Dense(nb_classe, activation = 'softmax'))

    # Compilation
    modele.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics =␣
↪['accuracy'])

    return modele
```

Commentaires :

- On a choisi un modèle classique avec 3 couches convolutives de 32, 64 et 128 filtres convolutifs.
- On utilise `kernel_regularizer` pour ajouter une pénalité à la fonction loss proportionnellement aux poids.
- On utilise `BatchNormalization` pour normaliser les poids.
- On utilise `MaxPooling2D` pour diviser par 2 le nombre de poids à la sortie du filtre.
- On utilise `Dropout` pour supprimer aléatoirement une partie des poids.
- On utilise 256 paramètres pour représenter l'image à la fin du réseau pour calculer les probabilités.

On appelle la fonction `construire_modele` :

[16]:
```
modele = construire_modele(taille_image, nb_classe)
```

```
/home/gaillot/.conda/envs/1/lib/python3.10/site-
packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

On affiche le résumé du modèle :

[17]:
```
modele.summary()
```

```
Model: "sequential"


 Total params: 629,739 (2.40 MB)


 Trainable params: 629,291 (2.40 MB)


 Non-trainable params: 448 (1.75 KB)
```

On écrit la fonction `generer_echantillon_epoque` qui permettra de choisir une unique version de chaque image du jeu d'entraînement et de validation lors d'une époque :

[18]:
```
def generer_echantillon_epoque(X, y, nb_versions):

    nb_images = len(X) // nb_versions
    indices_selectionnes = []

    for i in range(nb_images):
        nombre_aleatoire = np.random.randint(0, nb_versions)
        indices_selectionnes.append(i * nb_versions + nombre_aleatoire)

    X_selectionne = X[indices_selectionnes]
    y_selectionne = y[indices_selectionnes]

    return X_selectionne, y_selectionne
```

Commentaire : Cette approche permet d'exploiter le fait d'avoir 30 versions de chaque image au lieu de faire un générateur de perturbations.

On écrit la fonction `separer_echantillon_epoque` qui permettra de séparer l'échantillon généré par la fonction `generer_echantillon_epoque` en un jeu d'entraînement et un jeu de validation selon le ratio `ratio_validation` :

[19]:
```
def separer_echantillon_epoque(X_epoque, y_epoque, ratio_validation):

    indices = np.arange(len(X_epoque))
    np.random.shuffle(indices)

    X_epoque = X_epoque[indices]
    y_epoque = y_epoque[indices]

    separation = int(len(X_epoque) * ratio_validation)
    X_entrainement_epoque = X_epoque[:-separation]
    y_entrainement_epoque = y_epoque[:-separation]
    X_validation_epoque = X_epoque[-separation:]
    y_validation_epoque = y_epoque[-separation:]
```

```
    return (X_entrainement_epoque,
            y_entrainement_epoque,
            X_validation_epoque,
            y_validation_epoque)
```

On initialise l'historique de l'entraînement qui contiendra les résultats de performances :

[20]:
```
historique = {'loss_entrainement': [],
              'loss_validation': [],
              'accuracy_entrainement': [],
              'accuracy_validation': []}
```

On écrit la fonction `ajouter_historique` qui permettra d'ajouter à l'historique global l'historique de chaque époque :

[21]:
```
def ajouter_historique(historique_epoque, historique):

    historique['loss_entrainement'].append(historique_epoque.history['loss'][0])
    historique['loss_validation'].append(historique_epoque.history['val_loss'][0])
    historique['accuracy_entrainement'].append(historique_epoque.history['accuracy'][0])
    historique['accuracy_validation'].append(historique_epoque.history['val_accuracy'][0])

    return None
```

On entraîne le modèle :

[22]:
```
temps_debut = time.time()
```

[23]:
```
for epoque in range(nb_epoque):

    X_epoque, y_epoque = generer_echantillon_epoque(X_entrainement, y_entrainement, 30)

    (X_entrainement_epoque,
     y_entrainement_epoque,
     X_validation_epoque,
     y_validation_epoque) = separer_echantillon_epoque(X_epoque,
                                                       y_epoque,
                                                       ratio_validation)

    historique_epoque = modele.fit(X_entrainement_epoque, y_entrainement_epoque,
                                   batch_size = taille_lot,
                                   validation_data = (X_validation_epoque,␣
↪y_validation_epoque),
                                   epochs = 1,
                                   class_weight = None,
                                   verbose = 0)

    ajouter_historique(historique_epoque, historique)
```

[24]:
```
temps_fin = time.time()
```

[25]:
```
temps_entrainement = temps_fin - temps_debut
```

3 Analyse des résultats

On évalue le modèle :

[26]:
```
loss_test, accuracy_test = modele.evaluate(X_test, y_test, verbose = 0)
```

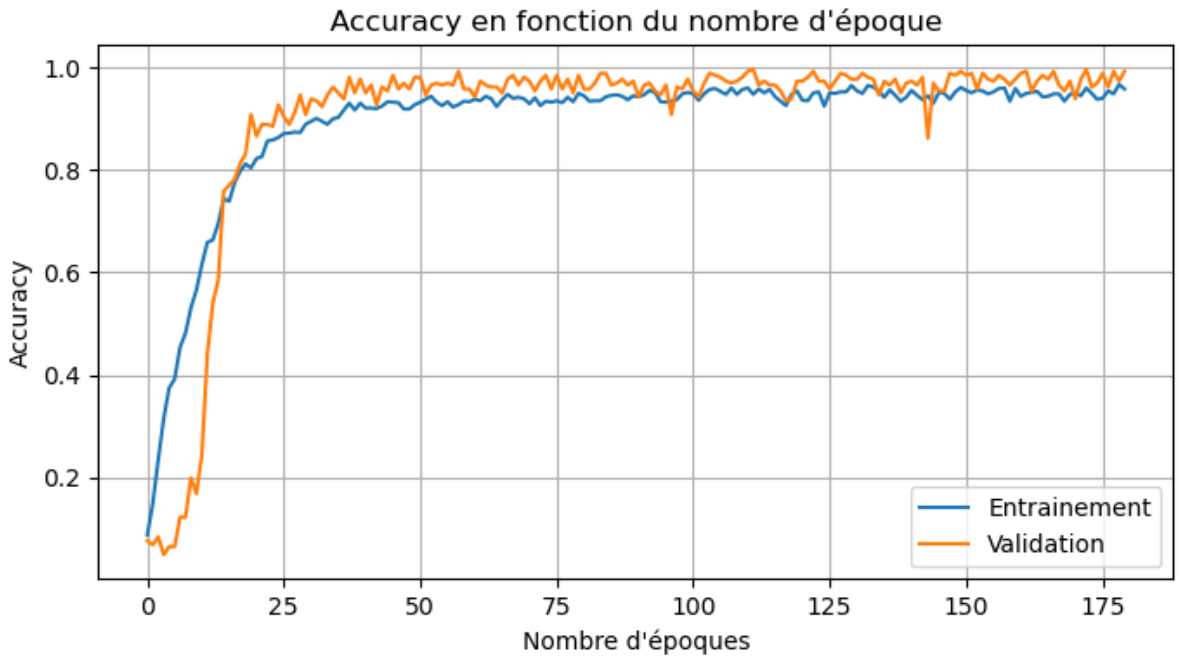
On affiche les résultats de performances :

[27]:
```
print("Accuracy en entraînement (%) =\n", historique['accuracy_entrainement'][-1] * 100.0,
      "\n\nAccuracy en test (%) =\n", accuracy_test * 100.0,
      "\n\nDifférence d'accuracy (%) =\n", (accuracy_test -␣
 ↪historique['accuracy_entrainement'][-1]) * 100.0,
      "\n\nLoss en entraînement =\n", historique['loss_entrainement'][-1],
      "\n\nLoss en test =\n", loss_test,
      "\n\nDifférence de loss =\n", (loss_test - historique['loss_entrainement'][-1]),
      "\n\nTemps d'exécution (s) =\n", temps_entrainement)
```

```
Accuracy en entraînement (%) =
 95.79349756240845

Accuracy en test (%) =
 94.8060154914856

Différence d'accuracy (%) =
 -0.9874820709228516

Loss en entraînement =
 0.9317676424980164

Loss en test =
 0.9974979162216187

Différence de loss =
 0.0657302737236023

Temps d'exécution (s) =
 115.49490451812744
```

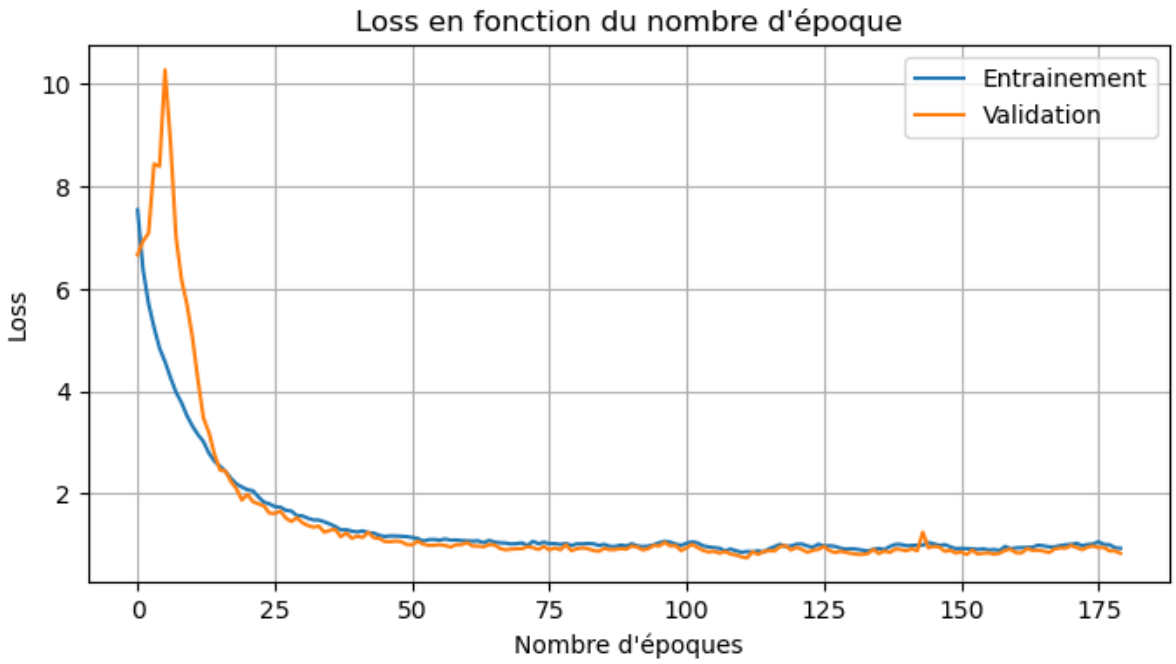
On trace le graphique de l'accuracy en fonction du nombre d'époque :

[28]:
```
plt.figure(figsize = (8, 4))
plt.plot(historique['accuracy_entrainement'], label = "Entrainement")
plt.plot(historique['accuracy_validation'], label = "Validation")
plt.title("Accuracy en fonction du nombre d'époque")
plt.xlabel("Nombre d'époques")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.show()
```



On trace le graphique de la loss en fonction du nombre d'époque :

[29]:
```
plt.figure(figsize = (8, 4))
plt.plot(historique['loss_entrainement'], label = "Entrainement")
plt.plot(historique['loss_validation'], label = "Validation")
plt.title("Loss en fonction du nombre d'époque")
plt.xlabel("Nombre d'époques")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.show()
```



On convertit les prédictions (sous formes de probabilités) en une unique classe :

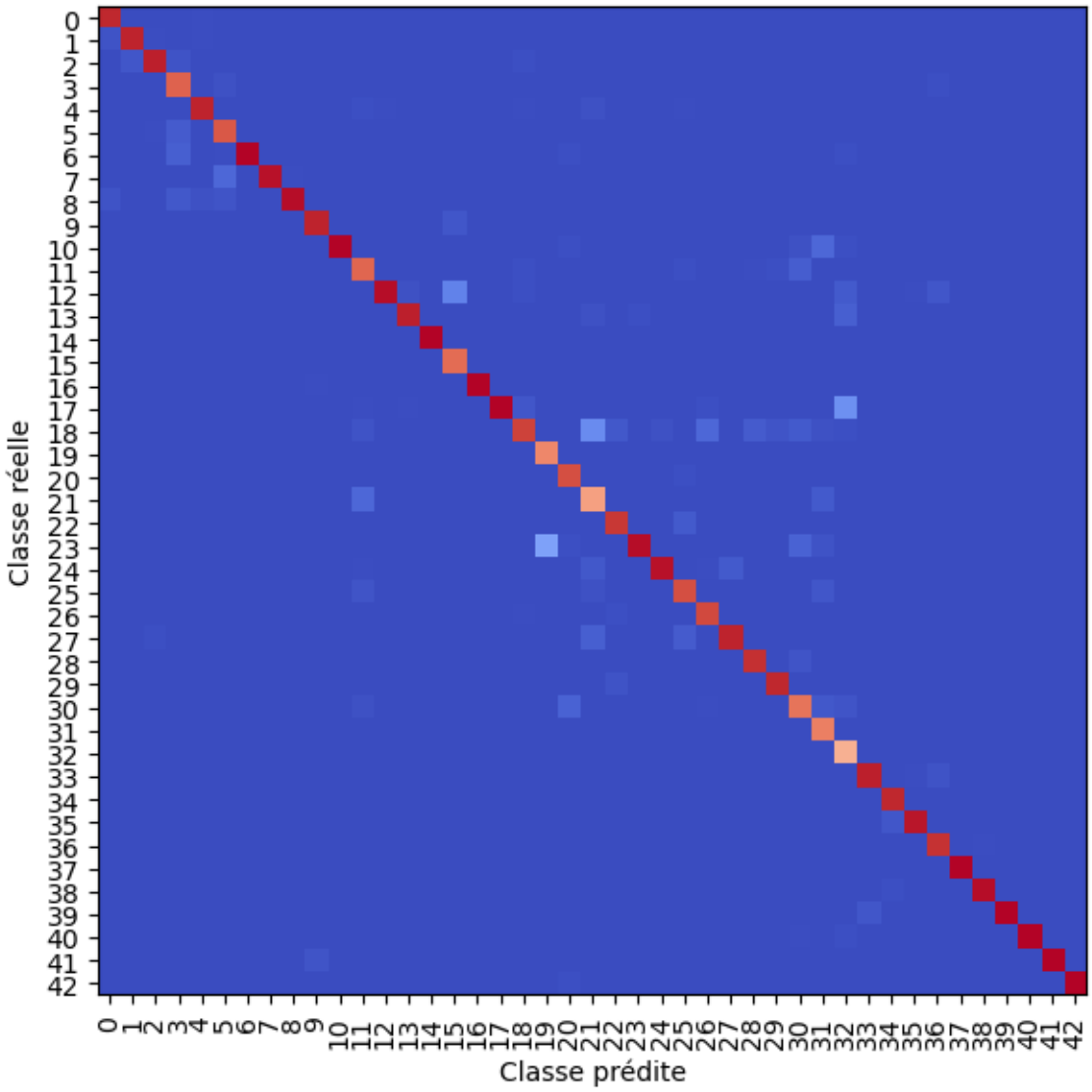
[30]:
```
y_test_predit = modele.predict(X_test, verbose = 0)
y_test_predit = np.argmax(y_test_predit, axis = 1)
y_test_exact = np.argmax(y_test, axis = 1)
```

On crée la matrice de confusion :

[31]:
```
matrice_confusion = confusion_matrix(y_test_exact, y_test_predit)
matrice_confusion_normalisee = matrice_confusion / matrice_confusion.sum(axis = 0, keepdims␣
↪= True)
```

On trace le graphique de la matrice de confusion :

[32]:
```
plt.figure(figsize = (6, 6))
plt.imshow(matrice_confusion_normalisee,
           interpolation = "nearest",
           cmap = "coolwarm",
           vmin = matrice_confusion_normalisee.min(),
           vmax = matrice_confusion_normalisee.max())
plt.xlabel("Classe prédite")
plt.ylabel("Classe réelle")
plt.xticks(ticks = np.arange(nb_classe), labels = np.arange(nb_classe), rotation = 90)
plt.yticks(ticks = np.arange(nb_classe), labels = np.arange(nb_classe))
plt.tight_layout()
plt.show()
```

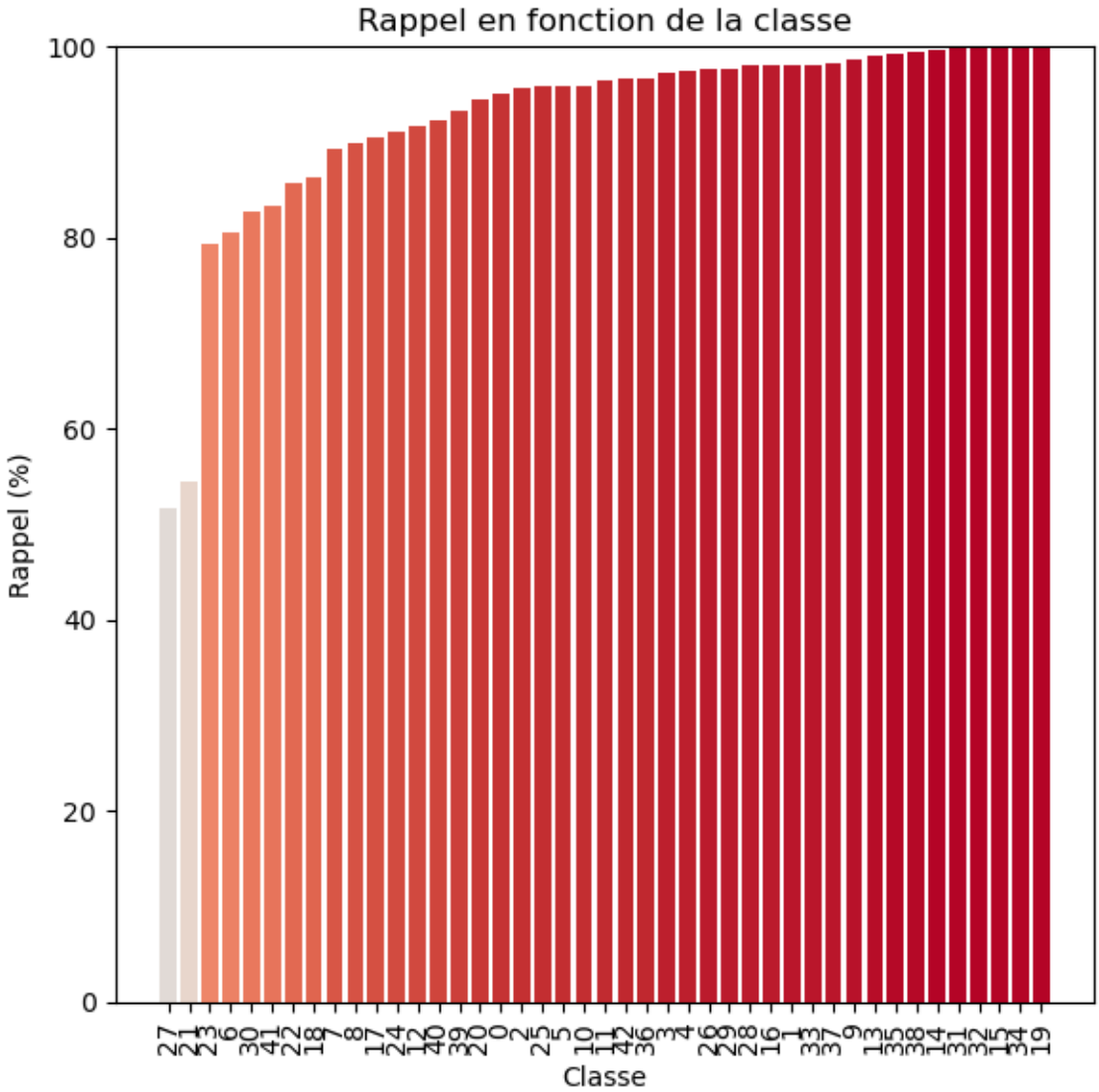


On calcule et trace le graphique du rappel en fonction de la classe :

[33]:
```
matrice_confusion_diagonale = np.diag(matrice_confusion)
nb_image_par_classe = matrice_confusion.sum(axis = 1)

rappel_par_classe = matrice_confusion_diagonale / nb_image_par_classe
rappel_par_classe_df = pd.DataFrame({"classe": range(len(rappel_par_classe)),
                                     "rappel": rappel_par_classe}).sort_values(by =␣
↪"rappel", ascending = False)

rappel_par_classe_classement_df = rappel_par_classe_df.sort_values(by = "rappel", ascending␣
↪= True)
rappel_par_classe_classement_df["rappel"] = 100.0 * rappel_par_classe_classement_df["rappel"]
```

[34]:
```
norme = mcolors.Normalize(vmin = 0, vmax = 100)
cmap = plt.colormaps["coolwarm"]
couleurs = cmap(norme(rappel_par_classe_classement_df["rappel"].values))
```

```
fig, ax = plt.subplots(figsize = (6, 6))
ax.bar(rappel_par_classe_classement_df["classe"].astype(str),
       rappel_par_classe_classement_df["rappel"],
       color = couleurs)
ax.set_xlabel("Classe")
ax.set_ylabel("Rappel (%)")
ax.set_title("Rappel en fonction de la classe")
ax.set_ylim(0, 100)
plt.xticks(rotation = 90)

sm = plt.cm.ScalarMappable(cmap = cmap, norm = norme)
sm.set_array(rappel_par_classe_classement_df["rappel"].values)

plt.tight_layout()
plt.show()
```



On regarde, pour chaque classe parmi les 5 classes ayant le plus faible rappel, les 5 classes les plus prédites :

[35]:
```
classes_rappel_flop5 = rappel_par_classe_classement_df.head(5)["classe"].values

for classe in classes_rappel_flop5:
    ligne = matrice_confusion[classe]
    accuracy_top5 = np.argsort(ligne)[-5:][::-1]
    classes_rappel_flop5_top5 = pd.DataFrame({"classe_predite" : accuracy_top5,
                              "nb_predictions" : ligne[accuracy_top5]})
    print("5 classes les plus prédites pour la classe", classe)
    display(classes_rappel_flop5_top5)
```

```
5 classes les plus prédites pour la classe 27
```

	classe_predite	nb_predictions
0	27	31
1	25	18
2	2	7
3	21	3
4	18	1

```
5 classes les plus prédites pour la classe 21
```

	classe_predite	nb_predictions
0	21	49
1	11	30
2	31	11
3	2	0
4	1	0

```
5 classes les plus prédites pour la classe 23
```

	classe_predite	nb_predictions
0	23	119
1	19	16
2	30	8
3	31	6
4	20	1

```
5 classes les plus prédites pour la classe 6
```

	classe_predite	nb_predictions
0	6	121
1	3	23
2	5	4
3	32	1
4	20	1

```
5 classes les plus prédites pour la classe 30
```

	classe_predite	nb_predictions
0	30	124
1	31	10
2	11	6
3	20	5
4	13	2

Commentaires :

— Avec les poids associés au nombre d'images par classe du jeu d'entraînement, l'accuracy en entraînement n'a pas augmentée. - On a fait 5 exécutions dans laquelles on a enregistré le rappel en fonction de la classe dans un fichier `json` (voir `Rappel-par-classe/*.json`) et le graphique de la matrice de confusion (voir `Matrice-confusion/*.png`).

— On a fait 5 exécutions dans lesquelles on a enregistré le rappel en fonction de la classe dans un fichier `json` (voir `Rappel-par-classe/*.json`) et les graphiques du rappel en fonction de la classe et la matrice de confusion (voir `Rappel-par-classe/*.png` et `Matrice-confusion/*.png`).
— Pour ces 5 exécutions, les 5 classes ayant le plus faible rappels sont en moyennes les classes 27, 21, 30, 26 et 41.
— On a fait des exécutions avec ces nouveaux poids (au lieu de ceux associés au nombre d'images par classe du jeu d'entraînement) mais cela n'a pas permit d'augmenter le rappel par classe pour les classes ayant le plus faible rappel.
— Avec ces résultats, on garde finalement un entraînement sans poids.
— Appliquer des poids au classes lors de l'entraînement ne permet pas de résoudre le problème du rappel par classe.
— Utiliser des tailles d'images différentes (comme 48×48 ou 64×64) ne permet pas d'augmenter l'accuracy et le rappel par classe, et augmente le temps d'exécution.
— Utiliser une couche convolutive supplémentaire (de 256 filtres) et 512 paramètres dans la connection ne permet pas d'augmenter l'accuracy et le rappel par classe, et augmente le temps d'exécution.