Master Calcul Haute Performance et Simulation

Projet de HPC pour l'intelligence artificielle

Reconnaissance des panneaux de signalisation

par Jean-Baptiste Gaillot

Informations

- Dataset : *GTSRB - German Traffic Sign Recognition Benchmark*.
- Choix de l'implémentation : "from scratch".
- Principaux modules `Python` utilisés : `tensorflow`, `keras` et `sklearn`.
- Modules complémentaires `Python` utilisés : `os`, `json`, `time`, `numpy`, `pandas`, `matplotlib`, `PIL`.

Informations

On a un problème de classification avec 43 classes possibles :

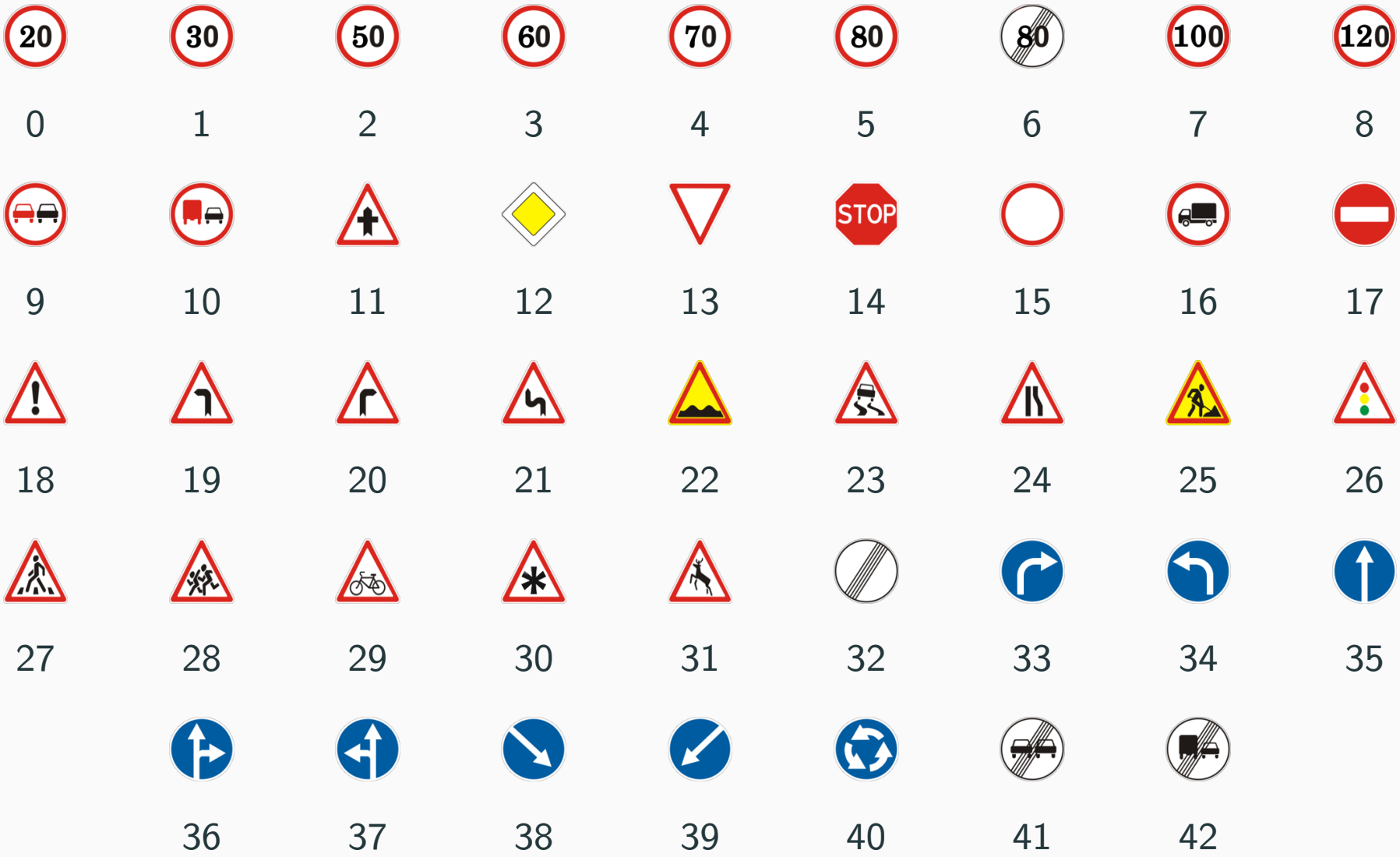


Table des matières

Récupération et préparation des données

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 :

```
nom_repertoire_dataset = "../Dataset"
taille_image = 32
nb_classe = 43
```

Récupération et préparation des données

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

```
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
```

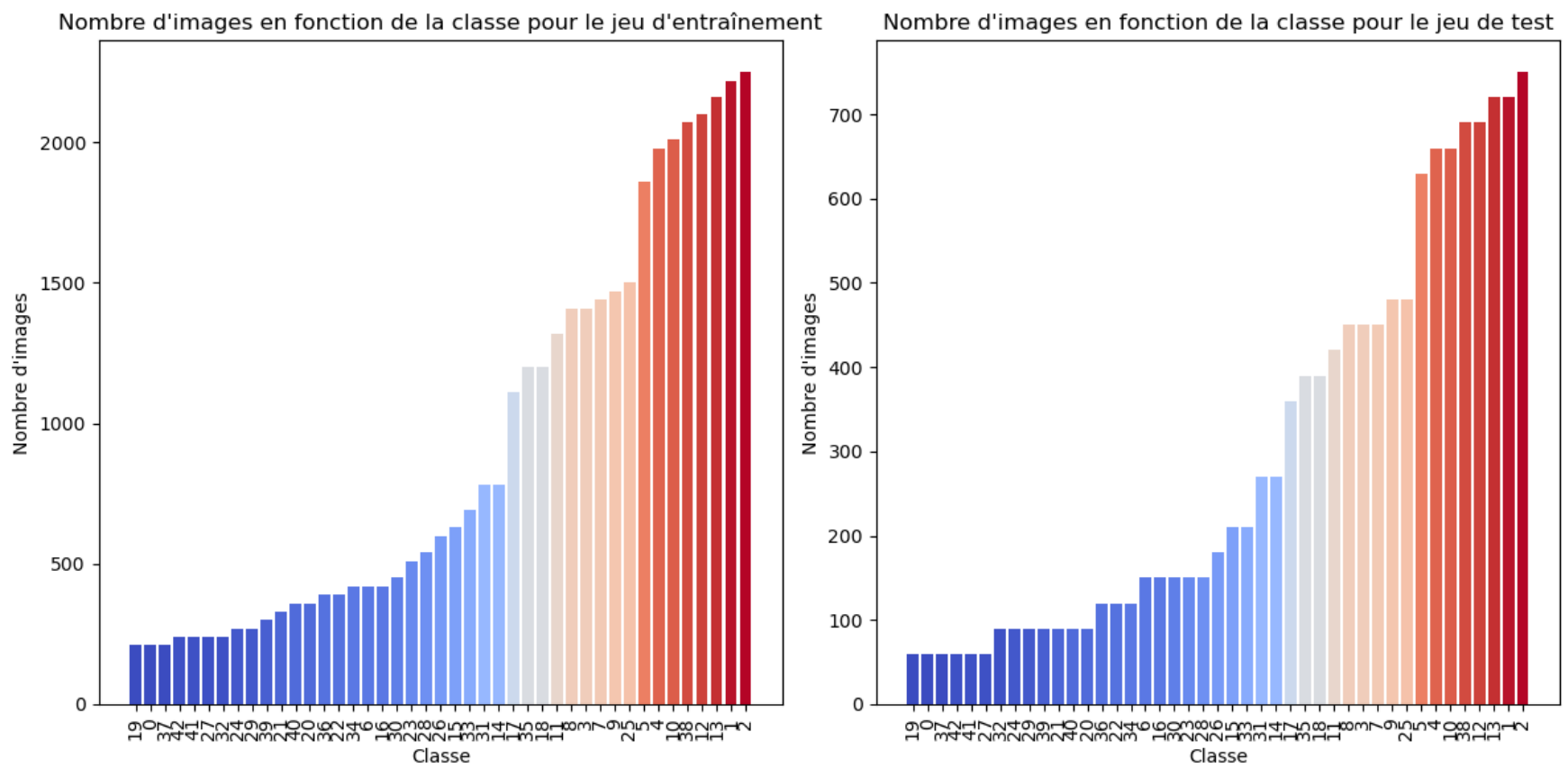
Récupération et préparation des données

On obtient les dimensions des jeux de données :

```
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,)
```

Récupération et préparation des données

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



Récupération et préparation des données

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

Récupération et préparation des données

On normalise les valeurs des pixels entre 0 et 1 :

```
X_entrainement = X_entrainement.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
```

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

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

poids_classes = dict(enumerate(poids_classes))
```

Récupération et préparation des données

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

```
y_entrainement = to_categorical(y_entrainement, nb_classes)
y_test = to_categorical(y_test, nb_classes)
```

Construction et entraînement du modèle

Construction et entraînement du modèle

On initialise les données de l'entraînement :

```
nb_epoque = 180
taille_lot = 32
ratio_validation = 0.2
```

Construction et entraînement du modèle

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

```
def construire_modele(taille_image, nb_classes):

    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_classes, activation = 'softmax'))

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

    return modele
```

Construction et entraînement du modèle

On affiche le résumé du modèle :

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_1 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout (Dropout)	(None, 8, 8, 64)	0
conv2d_2 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_1 (Dropout)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dropout_2 (Dropout)	(None, 2048)	0
dense (Dense)	(None, 256)	524,544
dense_1 (Dense)	(None, 43)	11,051

Total params: 629,739 (2.40 MB)

Trainable params: 629,291 (2.40 MB)

Non-trainable params: 448 (1.75 KB)

Construction et entraînement du modèle

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 :

```
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
```

Construction et entraînement du modèle

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` :

```
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)
```

Construction et entraînement du modèle

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

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

Construction et entraînement du modèle

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

```
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
```

Construction et entraînement du modèle

On entraîne le modèle :

```
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)
```

Analyse des résultats

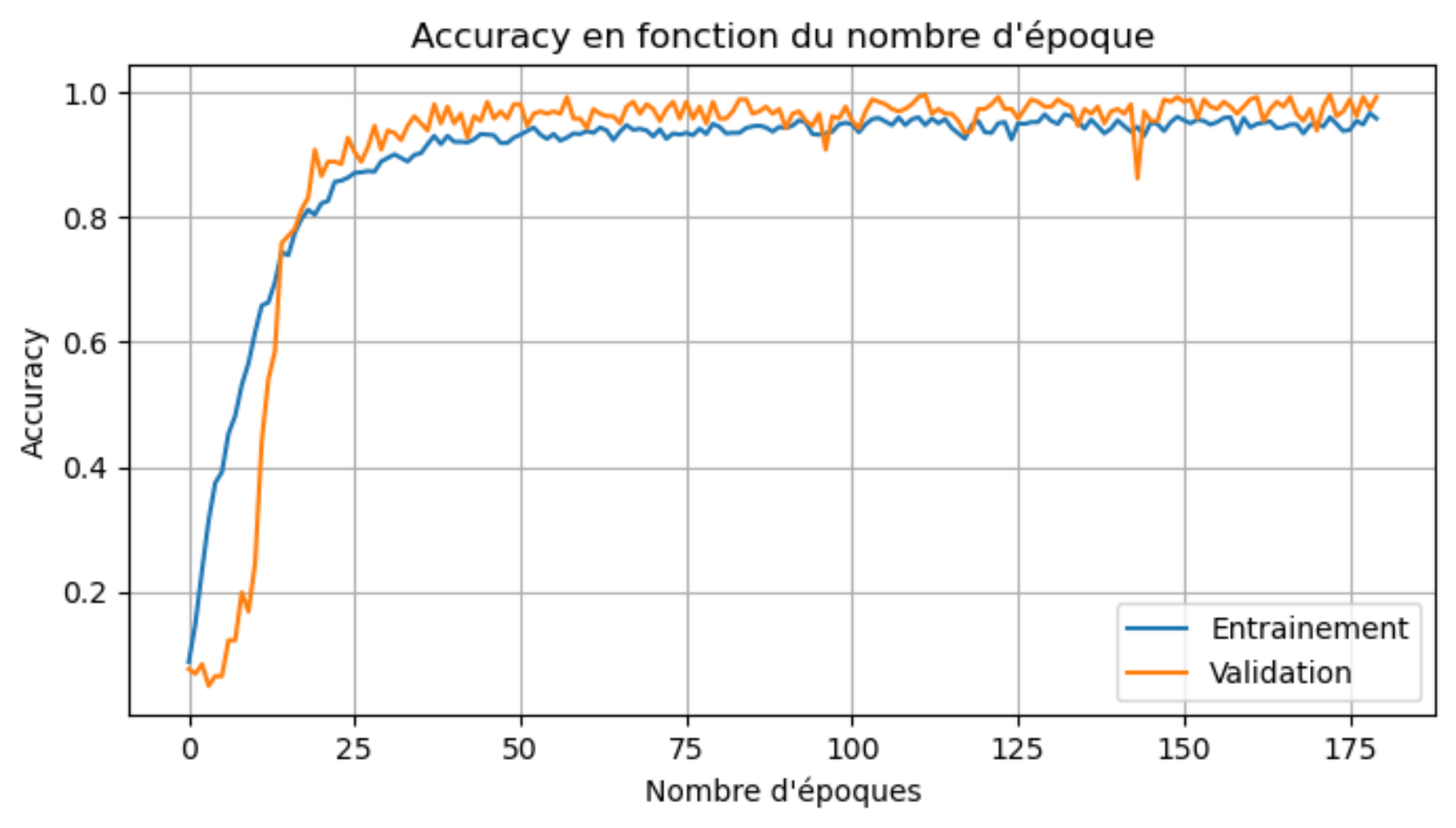
Analyse des résultats

On évalue le modèle :

```
loss_test, accuracy_test = modele.evaluate(X_test, y_test, verbose = 0)
```

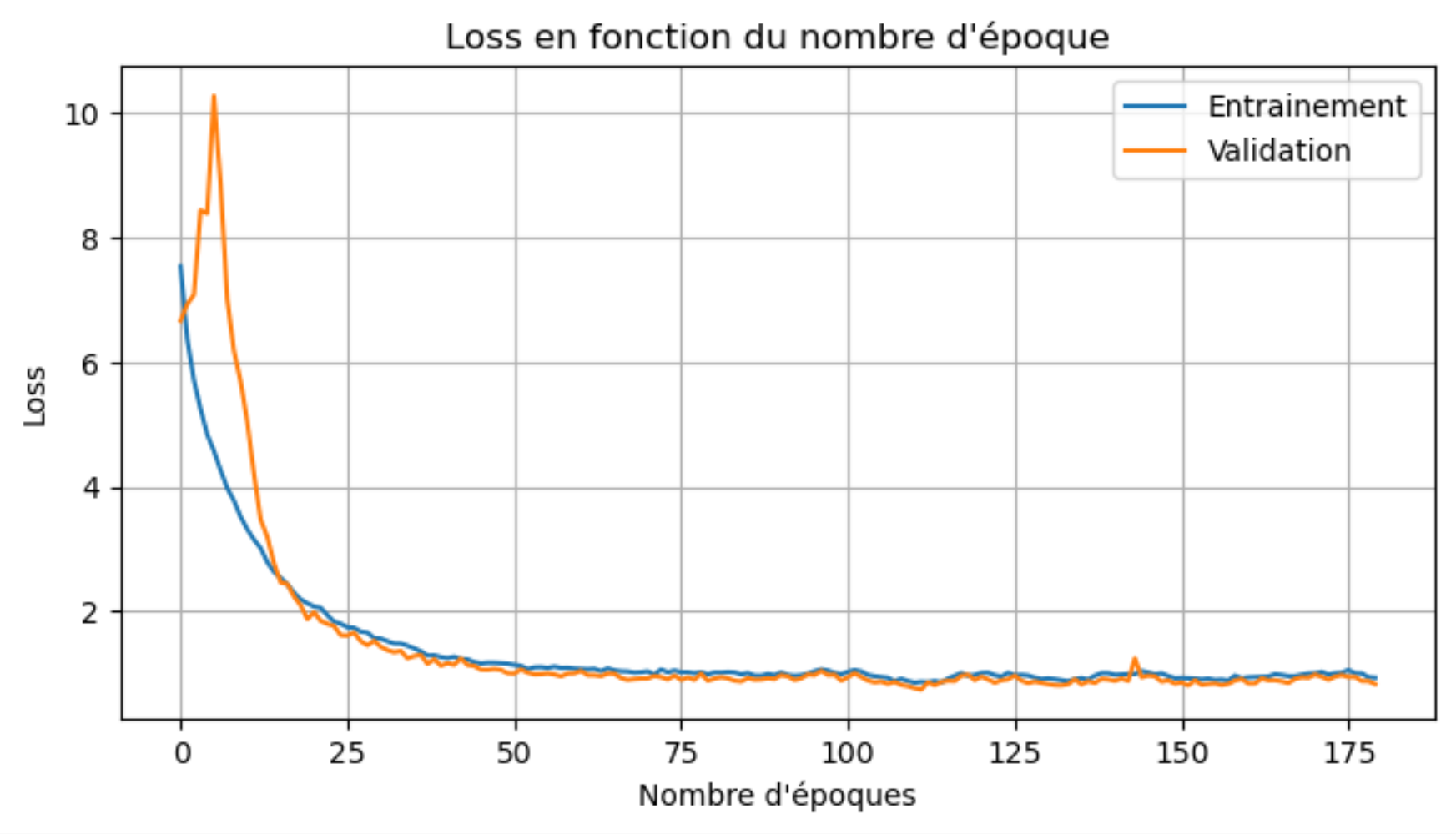
Analyse des résultats

On trace le graphique de l'accuracy en fonction du nombre d'époque (pour 1 exécution) :



Analyse des résultats

On trace le graphique de la loss en fonction du nombre d'époque (pour 1 exécution) :



Analyse des résultats

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

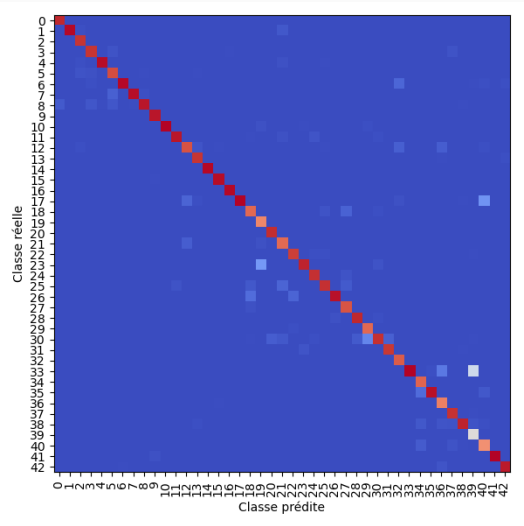
```
y_test_predit = modele.predict(X_test)
y_test_predit = np.argmax(y_test_predit, axis = 1)
y_test_exact = np.argmax(y_test, axis = 1)
```

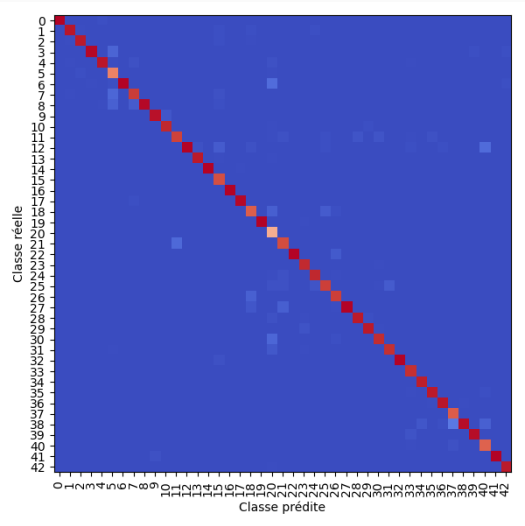
On crée la matrice de confusions :

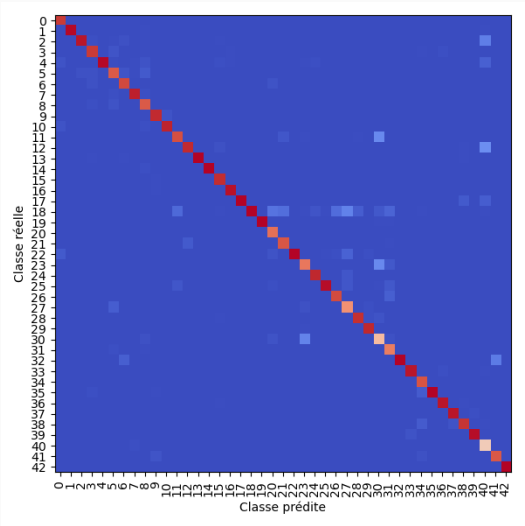
```
matrice_confusion = confusion_matrix(y_exact, y_predit)
```

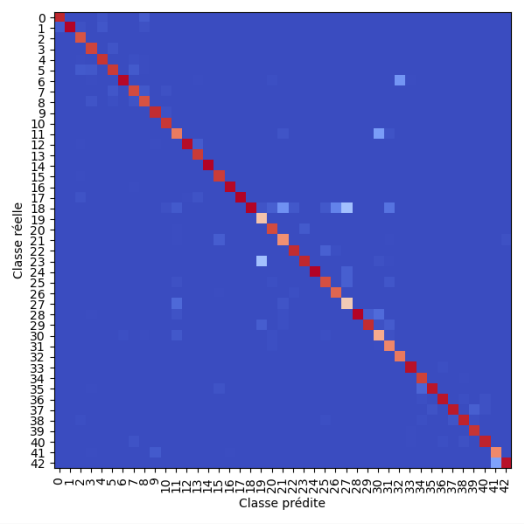
Analyse des résultats

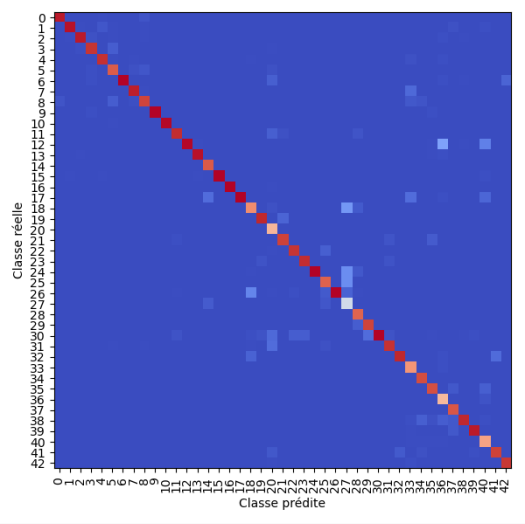
On trace le graphique de la matrice de confusion (pour 5 exécutions) :





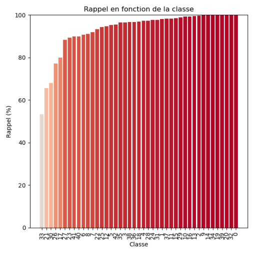


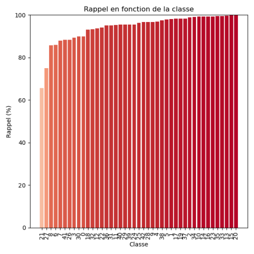


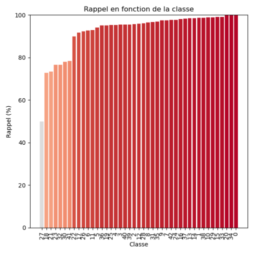


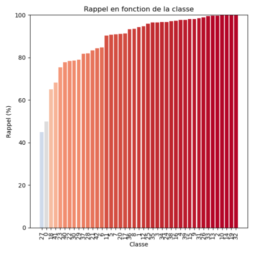
Analyse des résultats

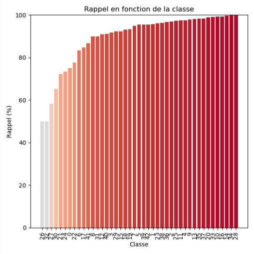
On trace le graphique du rappel en fonction de la classe (pour 5 exécutions) :











Analyse des résultats

On regarde, pour chaque classe parmi les 5 classes ayant le plus faible rappel, les 5 classes les plus prédites (pour 1 exécution) :

Classe exacte :		Classe prédite :					
Nombre :	60	Nombre :	31	18	7	3	1
Classe exacte :		Classe prédite :					
Nombre :	90	Nombre :	49	30	11		
Classe exacte :		Classe prédite :					
Nombre :	150	Nombre :	119	16	8	6	1
Classe exacte :		Classe prédite :					
Nombre :	150	Nombre :	121	23	4	1	1
Classe exacte :		Classe exacte :					
Nombre :	147	Nombre :	124	10	6	5	2

Analyse des résultats

On montre les résultats moyens (pour 5 exécutions) :

Exécutions	1	2	3	4	5	Moyenne
Accuracy en entraînement (%)	97.8	95.8	96.4	95.2	95.9	96.2
Accuracy en test (%)	93.9	95.5	93.6	93.9	94.7	94.3
Différence d'accuracy (%)	-3.9	-0.3	-2.8	-1.3	-1.2	-1.9
Loss en entraînement	0.85	0.88	0.86	0.90	0.92	0.88
Loss en test	0.95	0.92	0.97	0.96	0.98	0.96
Différence de loss	0.10	0.04	0.10	0.06	0.05	0.07
Temps d'exécution (s)	114	113	114	116	114	114

Analyse des résultats

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 des exécutions dans lesquelles on a enregistré le rappel en fonction de la classe dans un fichier `json` et les graphiques du rappel en fonction de la classe et la matrice de confusion.
- On a fait une moyenne de ces exécutions pour créer les poids associés au rappel par classe moyen dans un Notebook qu'on a écrit dans un fichier `json`.
- On a fait des exécutions avec ces nouveaux poids 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.

Analyse des résultats

Commentaires

- 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.

Conclusion

Lien vers le dépôt GitHub du projet

`https://github.com/gaillot18/M2-CHPS-HPC-pour-IA.git`

Merci pour votre attention. Des questions ?