

# Rapport de projet Kaggle

## I. Introduction au problème d'IA

Au cours de ce projet, notre but va être de prédire si un champignon est comestible ou bien empoisonné en se basant sur ces caractéristiques physiques.

## II. Introduction au jeu de données

Le jeu de données est composé de 2 fichiers, l'un est le set d'entraînement et l'autre est le set de test.

Les fichiers sont au format csv.

Le fichier d'entraînement contient 3 116 945 observations et 21 caractéristiques et le fichier de test contient 207 796 4 observations et 20 caractéristiques.

Dans les deux fichiers on retrouve les mêmes types de valeurs, on retrouve des données de type float64 soit des valeurs numériques continues et des données de type object soit des données catégoriques.

## III. Pré-traitement des données

### 1. *Importation des données*

Pour le pré-traitement des données, j'ai tout d'abord importé les 2 fichiers csv en enlevant les colonnes « id » qui ne sont pas pertinentes à traiter.

```
test = pd.read_csv("test.csv").drop('id', axis=1)
train = pd.read_csv("train.csv").drop('id', axis=1)
```

Afin de gagner du temps, je vérifie si les fichiers train et test\_processed sont déjà présents, ces fichiers contiennent les données déjà traitées afin de ne pas avoir à le refaire à chaque lancement du programme. À la fin du premier lancement du programme je les sauvegarde dans un csv.

```
if os.path.exists(processed_train_file) and os.path.exists(processed_test_file):
    train = pd.read_csv(processed_train_file)
    test = pd.read_csv(processed_test_file)
    print(f'Train shape: {train.shape}')
    print(f'Test shape: {test.shape}')
    print("Données chargées depuis les fichiers prétraités.")
```

```
train.to_csv(processed_train_file, index=False)
test.to_csv(processed_test_file, index=False)
print(f"Données sauvegardées dans {processed_train_file} et {processed_test_file}")
```

## **2. Suppression des colonnes avec plus de 50% de valeurs manquantes**

Ensuite, j'ai enlevé les colonnes qui contenaient plus de 50% de valeurs manquantes car celles-ci ne sont pas pertinentes à traiter. En raison de leur trop grand nombre de valeurs manquantes j'ai préféré les retirer plutôt que de remplacer les valeurs par la moyenne ou la médian car cela ne serait pas assez représentatif en raison du peu d'observations à ma disposition.

Pour faire cela, j'ai créé un data frame de booléen contenant la valeur True si la cellule est NaN, sinon la valeur est False.

Ensuite j'ai calculé la moyenne pour chaque colonne, sachant que True est considéré comme 1 et False comme 0 la valeur moyenne trouvée correspond donc à la proportion de valeurs manquantes.

Puis je parcours mon data frame en stockant le nom des colonnes qui possèdent plus de 50% de valeurs manquantes et enfin je supprime les colonnes que j'ai récupéré précédemment.

```
missing_threshold = 0.5
missing_train = train.isnull().mean()
columns_to_drop = [col for col in
missing_train[missing_train > missing_threshold].index]
train.drop(columns=columns_to_drop, inplace=True)
test.drop(columns=columns_to_drop, inplace=True)
```

## **3. Imputation des valeurs manquantes**

Tout d'abord, je récupère les colonnes catégoriques et numériques dans des variables séparées.

J'impute les valeurs manquantes des colonnes numériques par la médian.

Pour les colonnes catégoriques, je récupère les colonnes également présente dans le dataset de test pour ne pas prendre en compte la colonne class puis j'impute les valeurs manquantes avec la valeur « not\_present »

```

numerical_cols = train.select_dtypes(include=[np.number]).columns.tolist()
categorical_cols = train.select_dtypes(include=[object]).columns.tolist()

train[numerical_cols] = train[numerical_cols].fillna(train[numerical_cols].median())
test[numerical_cols] = test[numerical_cols].fillna(train[numerical_cols].median())

categorical_cols_in_test = [col for col in categorical_cols if col in test.columns]

train[categorical_cols_in_test] = train[categorical_cols_in_test].fillna('not_present')
test[categorical_cols_in_test] = test[categorical_cols_in_test].fillna('not_present')

```

#### **4. Normalisation des données**

Tout d'abord, je fais en sorte que les données soient toutes comprises entre 0 et 1 en utilisant la méthode Min-Max.

```

min_vals = train[numerical_cols].min()
max_vals = train[numerical_cols].max()
train[numerical_cols] = (train[numerical_cols] - min_vals) / (max_vals - min_vals)
test[numerical_cols] = (test[numerical_cols] - min_vals) / (max_vals - min_vals)

```

Ensuite, je remplace toutes les valeurs négatives ou égal à 0 par un très petit nombre positif, car avec Min-Max il y a un risque que des très petites valeurs négatives soient calculer. Afin de pouvoir appliquer une transformation logarithmique avec les données elles doivent être strictement supérieures à 0.

```

train[numerical_cols] = train[numerical_cols].apply(lambda x: np.maximum(x, 1e-9))
test[numerical_cols] = test[numerical_cols].apply(lambda x: np.maximum(x, 1e-9))

```

Enfin, j'applique une transformation logarithmique afin de rendre la distribution plus normale. Cela permet de réduire l'impact des valeurs extrêmes et de compenser l'asymétrie des données.

```

train[numerical_cols] = np.log1p(train[numerical_cols])
test[numerical_cols] = np.log1p(test[numerical_cols])

```

#### **5. Traitement des catégories rares**

Je défini une fonction « process\_column » qui va traiter les données qui ont peu d'influences. Le traitement de ces données peu influentes permet de réduire le bruit et de faciliter la généralisation du modèle.

Tout d'abord, je vérifie si la colonne à traiter est également dans le jeu correspondant afin d'être sûr que la colonne est bien présente. Puis je calcul les fréquences relatifs sous forme de proportion. J'applique cela pour les catégories du dataset de train et de test et je donne également un nom.

Si jamais la colonne a traité pour test est class, la fréquence sera un data frame vide.

```
def process_column(col, train, test):  
    if col in train.columns:  
        freq_train = train[col].value_counts(normalize=True)  
        freq_train.name = 'freq_train'  
  
        freq_test = test[col].value_counts(normalize=True) if col in test.columns else pd.Series()  
        freq_test.name = 'freq_test'
```

Ensuite je regroupe toutes les fréquences d'entraînement et de test pour pouvoir calculer la fréquence maximale de chaque catégorie. Je compare les fréquences maximales de chaque catégories avec 1% et remplacent les catégories inférieurs à 1% avec la valeur « infrequent ».

```
freq = pd.merge(freq_train, freq_test, how='outer', left_index=True, right_index=True)  
freq['max'] = freq.max(axis=1)  
rare_categories = freq[freq['max'] < 0.01].index  
if len(rare_categories) > 0:  
    train[col] = train[col].replace(rare_categories, 'infrequent')  
    if col in test.columns:  
        test[col] = test[col].replace(rare_categories, 'infrequent')
```

Afin que le traitement des données se fasse plus rapidement j'ai utilisé la bibliothèque joblib qui permet d'utiliser plusieurs cœurs et donc de faire le traitement en parallèle. J'ai défini une fonction « replace » qui exécute la fonction précédente en utilisant plusieurs cœurs. La valeur de « n\_jobs » permet de définir combien de cœurs alloué, lorsque celle-ci vaut -1 cela signifie que tous les cœurs seront alloué à cette tâche. J'exécute ensuite cette fonction en passant en paramètre le data frame de test, test et la liste des colonnes catégoriques.

```
def replace(train, test, categorical_cols):  
    Parallel(n_jobs=-1)(delayed(process_column)(col, train, test) for col in categorical_cols)  
  
replace(train, test, categorical_cols)
```

## 6. Encodage des catégories

Afin de pouvoir utiliser mes valeurs catégorielles pour mon entraînement je les ai encodés en données numériques. J'effectue une vérification pour éviter la colonne class et pouvoir traiter mes autres colonnes. Ensuite, je récupère toutes les catégories présentes. Je fais un mapping entre chaque catégorie unique et un entier unique. J'applique ensuite l'encode sur train et test

en utilisant la map.

```
for col in categorical_cols:
    if col in train.columns:
        unique_categories = np.unique(train[col])
        category_to_int = {category: idx for idx, category in enumerate(unique_categories)}

        train[col] = train[col].map(category_to_int)

    if col in test.columns:
        test[col] = test[col].map(category_to_int)
```

## 7. Préparation des données pour l'entraînement

Tout d'abord, je mélange les données de train avec sample(). Frac = 1 signifie que l'on garde 100% des données, random\_state = 42 permet de mélanger de la même manière à chaque fois et reset\_index(drop = True) réinitialise les index.

```
train_data = train.sample(frac=1, random_state=42).reset_index(drop=True)
```

Ensuite je sépare les features et la class qui est donc la prédiction.

```
X_train = train_data.drop('class', axis=1).values
y_train = train_data['class'].values
X_test = test.values
```

Je défini device, si une carte graphique nvidia est disponible je l'utilise sinon j'utilise le cpu.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Je convertis ensuite les données en tenseur PyTorch, X\_train\_tensor sera de type float32, y\_train\_tensor sera de type float32 et il sera transformé en vecteur colonne pour être utilisé pour la régression binaire. X\_test\_tensor sera de type float32 et redirigé sur le device disponible.

```
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).view(-1, 1)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32).to(device)
```

Création d'un dataset de train PyTorch avec TensorDataset, le premier paramètre est les features et le deuxième les prédictions.

A l'aide de ce dataset, je crée un DataLoader afin de charger les données de manière plus efficace pour l'entraînement. Le batchsize définit la taille des lots, je défini celle-ci en fonction du nombre d'observations et des capacités du gpu. J'active le shuffle afin que les données soient mélangées avant chaque epoch pour éviter que le modèle ne se souvienne de l'ordre des données.

```
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=2048, shuffle=True)
```

## IV. Modèle intégré

J'ai utilisé un réseau de neurones multicouches pour effectuer de la régression binaire. Ce réseau utilise des couches linéaires, des fonctions d'activations non linéaires, de la normalisation par lots, du dropout pour la régularisation et des couches de transformations linéaires.

Tout d'abord l'input\_dim correspond au nombre de features du jeu de données et hidden\_dim à la taille de la première couche caché.

```
def __init__(self, input_dim, hidden_dim):
    super(NeuralNetwork, self).__init__()
```

Je crée 5 couches linéaires, chaque couche fait une transformation linéaire de l'entrée via une multiplication matricielle et un biais. La première couche prend comme paramètre le nombre de features du jeu de données et le nombre de dimensions cachés. La deuxième couche prend en paramètre le nombre de dimension caché et une taille fixe de 128. Les 3 autres couches ont des tailles fixes allant de 128 jusqu'à 1 pour la dernière couche pour produire une sortie de classification binaire.

```
self.fc1 = nn.Linear(input_dim, hidden_dim)
self.fc2 = nn.Linear(hidden_dim, 128)
self.fc3 = nn.Linear(128, 64)
self.fc4 = nn.Linear(64, 32)
self.fc5 = nn.Linear(32, 1)
```

Afin d'éviter le surapprentissage ce modèle utilise deux techniques.

Premièrement, le dropout qui est ajusté sur 40%. Cela signifie que à chaque itération, 40% des neurones ou dropout est appliqué seront ignorés, cela permet d'empêcher que le réseau ne s'ajuste trop fortement aux données d'entraînement.

Deuxièmement, le batch normalization qui est appliqué après chaque transformation linéaire les activations sont normalisés, la normalisation aide à stabiliser l'entraînement et accélère la convergence.

```
self.dropout = nn.Dropout(0.4)
self.batch_norm1 = nn.BatchNorm1d(hidden_dim)
self.batch_norm2 = nn.BatchNorm1d(128)
self.batch_norm3 = nn.BatchNorm1d(64)
self.batch_norm4 = nn.BatchNorm1d(32)
```

La méthode « forward » permet de définir le passage des données dans le réseau de neurones.

L'entrée « x » passe d'abord par la couche linéaire « fc1 » puis une fonction d'activation ReLU. ReLU est défini comme :  $\text{ReLU}(x) = \max(0, x)$ . La fonction d'activation ReLU permet d'introduire de la non-linéarité afin de modéliser des relations plus complexes et évite la simplification excessive du modèle.

La sortie de la première couche est normalisée via batch normalization pour stabiliser l'apprentissage puis soumise à dropout pour éviter le surapprentissage.

Les mêmes opérations sont répétées pour chacune des couches suivantes jusqu'à la quatrième.

La dernière couche effectue une transformation linéaire afin d'obtenir la sortie finale.

```
def forward(self, x):
    x = torch.relu(self.fc1(x))
    x = self.batch_norm1(x)
    x = self.dropout(x)
    x = torch.relu(self.fc2(x))
    x = self.batch_norm2(x)
    x = self.dropout(x)
    x = torch.relu(self.fc3(x))
    x = self.batch_norm3(x)
    x = torch.relu(self.fc4(x))
    x = self.batch_norm4(x)
    x = self.fc5(x)
    return x
```

## V. Fonction de perte

La fonction de perte que j'ai utilisée est BCEWithLogitsLoss, elle combine la fonction sigmoïde et la fonction de perte Binary Cross-Entropy.

Tout d'abord, pour un lot de N exemples, où  $y_i$  est la véritable étiquette et  $\hat{z}_i$  est la logit brut produit par le modèle pour l'exemple i, la formule est :

$$\text{BCEWithLogitsLoss} = -\frac{1}{N} \sum_{i=1}^N [y_i \cdot \log(\sigma(\hat{z}_i)) + (1 - y_i) \cdot \log(1 - \sigma(\hat{z}_i))]$$

Où  $\sigma(\hat{z}_i)$  est la fonction sigmoïde appliqué à  $\hat{z}_i$ , défini comme :

$$\sigma(\hat{z}_i) = \frac{1}{1 + e^{-\hat{z}_i}}$$

La sortie brut ou logit est la valeur générée par le modèle qui n'est pas encore transformé en probabilité, ces logits sont transformés en probabilité via la fonction sigmoïde.

La BCE calcule la divergence entre les probabilités prédites et les vraies étiquettes, plus la probabilité prédite est proche de l'étiquette réelle plus la perte sera faible.

L'avantage d'utiliser BCEWithLogitsLoss est d'éviter les erreurs de calculs en évitant de calculer la sigmoïde séparément et de combiner dans une seule opération le calcul de la sigmoïde et de la BCE ce qui est plus rapide.

## VI. Méthode d'optimisation

J'ai choisi la méthode d'optimisation Adam qui est une méthode d'optimisation adaptative adaptant dynamiquement le taux d'apprentissage pour chaque paramètre du modèle. L'algorithme Adam suit deux moments statistiques, le premier qui est la moyenne des gradients et le deuxième qui suit la variance des gradients. Tout d'abord, le calcul des premiers et deuxièmes moments :

$$\begin{aligned} m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\ v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \end{aligned}$$

Où :

$g_t$  est le gradient à l'itération  $t$ .

$\beta_1$  et  $\beta_2$  sont des hyperparamètre de l'optimiseur.

Puis la correction des biais :

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned}$$

Et enfin la mise à jour des paramètres :

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

Où :

$\theta_t$  est le paramètre à l'itération  $t$ .

$\eta$  est le taux d'apprentissage.

$\epsilon$  est un terme ajouté pour éviter de diviser par 0.

Dans la première étape, le premier moment calcul la moyenne exponentielle des gradients et le second moment calcul la moyenne exponentielle des carrés des gradients.

Dans la deuxième étape, on compense les estimations biaisées au départ, pour corriger ce biais on applique un facteur de correction qui permet de compenser les valeurs biaisées.

Dans la troisième étape, on met à jour les différents paramètres à l'aide des valeurs.

## VII. Évaluation du modèle

### 1. Les métriques

Les métriques que j'ai utilisé sont la loss et l'accuracy. J'ai calculer la loss et l'accuracy de mon jeu de train et j'ai utilisé kaggle pour avoir l'accuracy de mon jeu de test.

J'ai défini une fonction calculate\_accuracy, celle-ci prends en paramètres les prédictions effectuées par le modèle et les étiquettes réels associés aux données.



La fonction sigmoïde est appliquée aux prédictions pour les transformer en probabilité entre 0 et 1, puis les prédictions sont converties en étiquettes binaires. Si la prédiction est supérieure à 0.5 elle est classée comme 1 sinon comme 0. Elles sont ensuite converties en float.

Ensuite les étiquettes prédites sont comparées aux étiquettes réelles puis le nombre de valeurs correctes prédites sont stockées dans la variable `correct`.

Finalement, on retourne le nombre de valeurs prédites correctement par rapport au nombre total de valeurs dans le batch.

```
def calculate_accuracy(predictions, targets):
    predicted_labels = (torch.sigmoid(predictions) >= 0.5).float()
    correct = (predicted_labels == targets).sum().item()
    return correct / len(targets)
```

La `loss` est calculée dans la boucle d'entraînement. Tout d'abord le `running_loss` est initialisé à 0. On passe ensuite les inputs dans le modèle puis on récupère les outputs. On utilise notre fonction de perte définie dans la variable `criterion` et on donne en paramètre les outputs du modèle et les étiquettes réelles correspondantes, on récupère l'écart entre les prédictions du modèle et les étiquettes réelles dans la variable `loss`. `Loss.backward()` calcule les gradients de la perte par rapport aux paramètres du modèle et `optimizer.step()` met à jour les paramètres du modèle à l'aide des gradients calculés. On récupère ensuite la valeur de la `loss` calculée pour le batch dans la variable `running_loss`. Enfin, on calcule la valeur de toutes les `loss` des batch par rapport au nombre de batch pour récupérer la `loss` moyenne.

```

for epoch in range(config["epochs"]):
    model.train()
    running_loss, running_accuracy = 0, 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        accuracy = calculate_accuracy(outputs, labels)
        running_loss += loss.item()
        running_accuracy += accuracy

    avg_loss = running_loss / len(train_loader)
    avg_accuracy = running_accuracy / len(train_loader)

```

## 2. Hyperparamètres Tuning

L'ajustement des hyperparamètres est une étape cruciale, les principaux paramètres que j'ai modifiés sont le batch\_size et le learning rate.

Afin d'affiner le learning rate j'ai mis en place un système de scheduler. Celui-ci réduit le learning rate d'un ratio de 0.5 lorsque la loss stagne pendant plus de 3 epochs.

```

scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=3,
verbose=True)

```

Si la loss moyenne ne bouge pas pendant 3 epochs, le scheduler sera activé via cette fonction qui récupère la loss moyenne comme paramètre.




```

scheduler.step(avg_loss)

```

Afin de trouver le meilleur batch\_size j'ai commencé à partir de la valeur 128 et ai multiplié par 2 la valeur jusqu'à ce que mon accuracy de test n'augmente plus. La meilleure valeur de batch\_size que j'ai trouvée est de 2048. J'ai effectué ces tests sur 200 epochs et on peut voir que l'accuracy augmente jusqu'à 2048 puis diminue avec une

valeur supérieure.

|   |  |                |
|---|--|----------------|
|  | <b>submission.csv</b><br>Complete (after deadline) · 1m ago · 200 epoch 5092 batch | <b>0.97619</b> |
|  | <b>submission.csv</b><br>Complete (after deadline) · 2h ago · 200 epoch 2048 batch | <b>0.97723</b> |
|  | <b>submission.csv</b><br>Complete (after deadline) · 3h ago · 200 epoch 1024 batch | <b>0.97675</b> |

### 3. Figures

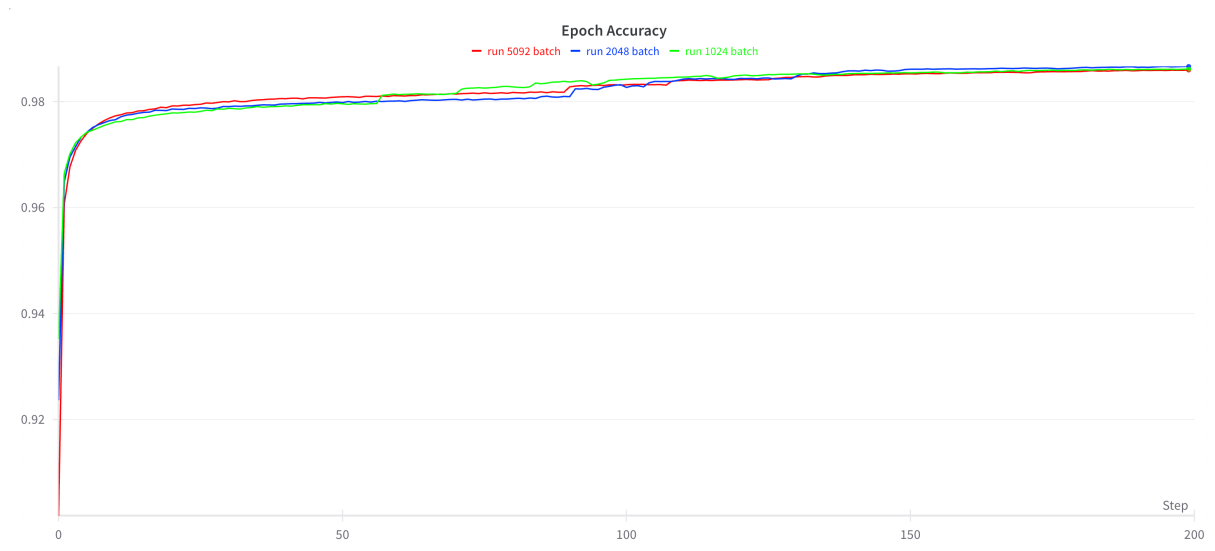
Afin de faire des figures j'ai utilisé wandb pour suivre en temps réels les entraînements et répertorié toutes les figures au même endroit. Tout d'abord, j'importe wandb et j'entre ma clé API. J'initialise, avec le nom de mon projet, le nom de mon entraînement et ma config. Dans ma config, j'initialise mes hyperparamètres et via log je récupère les données que je vais pouvoir visualisé, dans ce cas-ci, la loss, l'accuracy et le learning rate.

```
import wandb

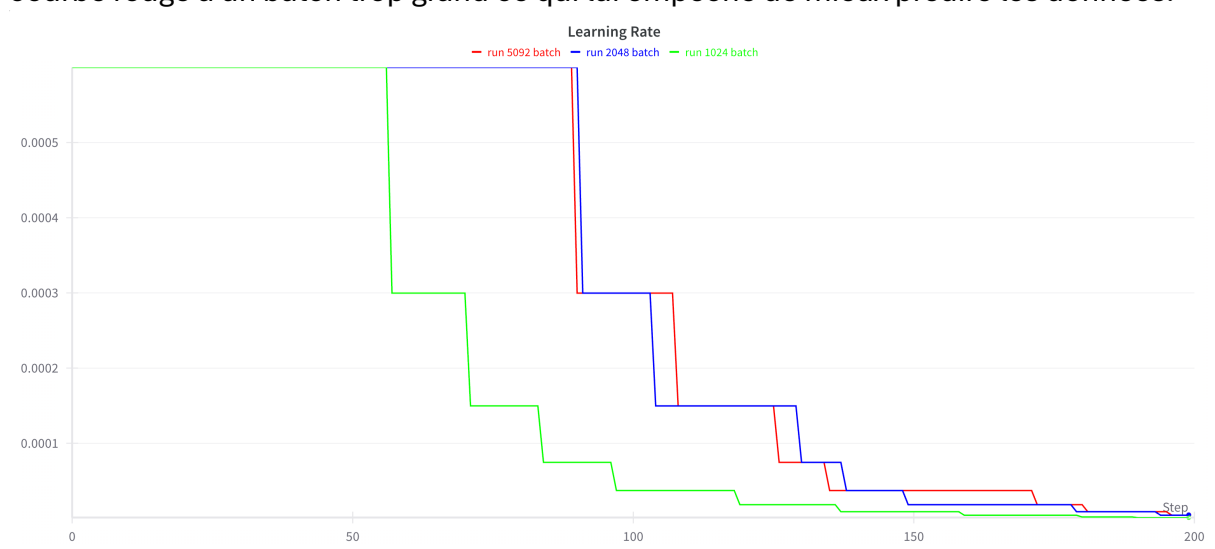
wandb.init(
    project="projet-kaggle",
    name = "run",
    config=config
)

config = {
    "learning_rate": 0.0006,
    "epochs": 200,
    "hidden_dim": 448,
    "weight_decay": 0.0002,
}

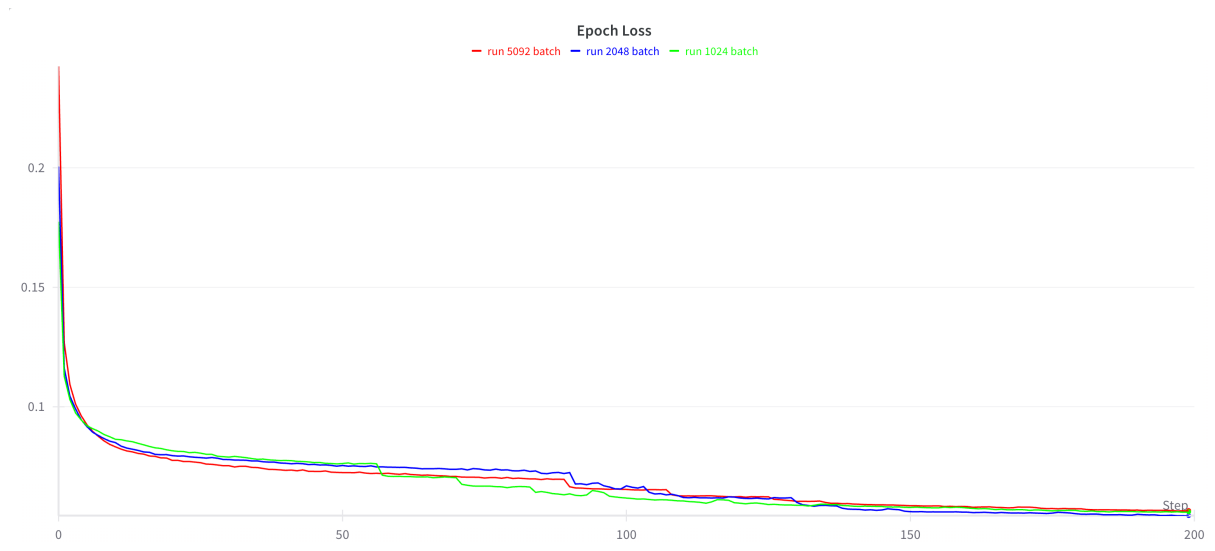
wandb.log({
    "Epoch Loss": avg_loss,
    "Epoch Accuracy": avg_accuracy,
    "Learning Rate": optimizer.param_groups[0]['lr'],
})
```



Dans ce premier graphique, on remarque que les 3 courbes commencent à converger à partir de 140 epochs environ, et celle qui atteint l'accuracy la plus haute sur le jeu d'entraînement est la courbe bleu qui correspond à celle avec un batch de 2048. La courbe rouge à un batch trop grand ce qui lui empêche de mieux prédire les données.



On remarque que la courbe jaune à atteint un learning rate trop bas ce qui empêche d'avoir le meilleur gradient, ceci explique pourquoi elle n'as pas aussi bien convergé que la bleu.



Encore une fois, on voit que la courbe rouge prédit moins bien à cause d'une trop grande valeur de batch elle a la plus grande valeur de loss.

On peut déduire que le modèle avec un batch de 1024 réduit trop son learning rate ce qui lui empêche d'avoir le meilleur gradient tandis que le modèle avec un batch de 5092 a un batch trop élevé pour réussir à atteindre une valeur de loss minimum d'aussi bonne qualité qu'un batch plus petit.

## VIII. Soumission au Kaggle

Pour effectuer la soumission au Kaggle j'ai créé une fonction `make_predictions`. Les paramètres sont le model entraîné, les données d'entrée de test, les ids correspondants et le chemin où sera sauvegardé le fichier de sortie.

Tout d'abord, je passe le model en mode évaluation, je désactive le calcul des gradients puis je génère les prédictions à partir des données du jeu de test. Je transforme les prédictions en probabilité comprises entre 0 et 1 et je défini que celles supérieures à 0.5 valent 1 et celles inférieures valent 0 pour obtenir une classification binaire. Je convertis les valeurs en float, je les déplace sur cpu et convertit le tenseur pytorch en tableau numpy. Je crée un fichier csv contenant les ids et si la class vaut 1 elle est représenté par « p », si elle vaut 0 elle est représenté par « e ». C'est le format demandé par Kaggle pour ce projet.

```
def make_predictions(model, X_test_tensor, test_ids, output_file):
    model.eval()
    with torch.no_grad():
        y_pred = model(X_test_tensor)
        y_pred_class = (torch.sigmoid(y_pred) > 0.5).float().cpu().numpy()

    submission = pd.DataFrame({
        'id': test_ids,
        'class': ["p" if pred == 1 else "e" for pred in y_pred_class]
    })
    submission.to_csv(output_file, index=False)
    print(f"Fichier de soumission sauvegardé : {output_file}")
```

## IX. NEF

Voici un exemple du programme qui tourne sur le nef, j'ai utilisé que 10 epochs pour ce test.

```
(proj-kaggle) [lthurst@nef-gpu4 proj-kaggle]$ python main.py
Train shape: (3115945, 21)
Test shape: (2077964, 20)
Columns dropped: ['stem-root', 'stem-surface', 'veil-type', 'veil-color', 'spore-print-color']
Colonne 'class': 0 catégories rares remplacées.
Colonne 'cap-shape': 102 catégories rares remplacées.
Colonne 'does-bruise-or-bleed': 28 catégories rares remplacées.
Colonne 'cap-color': 98 catégories rares remplacées.
Colonne 'cap-surface': 103 catégories rares remplacées.
Colonne 'has-ring': 25 catégories rares remplacées.
Colonne 'season': 0 catégories rares remplacées.
Colonne 'stem-color': 80 catégories rares remplacées.
Colonne 'ring-type': 40 catégories rares remplacées.
Colonne 'gill-attachment': 110 catégories rares remplacées.
Colonne 'gill-spacing': 63 catégories rares remplacées.
Colonne 'gill-color': 75 catégories rares remplacées.
Colonne 'habitat': 61 catégories rares remplacées.
Colonne 'class' encodée.
Colonne 'cap-shape' encodée.
Colonne 'cap-surface' encodée.
Colonne 'cap-color' encodée.
Colonne 'does-bruise-or-bleed' encodée.
Colonne 'gill-attachment' encodée.
Colonne 'gill-spacing' encodée.
Colonne 'gill-color' encodée.
Colonne 'stem-color' encodée.
Colonne 'has-ring' encodée.
Colonne 'ring-type' encodée.
Colonne 'habitat' encodée.
Colonne 'season' encodée.
Données sauvegardées dans processed_train.csv et processed_test.csv
wandb: Currently logged in as: lillian-hurst2004 (lillian-hurst2004-none). Use 'wandb login --relogin' to force relogin
wandb: wandb version 0.19.0 is available! To upgrade, please run:
wandb: $ pip install wandb --upgrade
wandb: Tracking run with wandb version 0.16.6
wandb: Run data is saved locally in /home/lthurst/Apprentissage/projet-kaggle/wandb/run-20241212_172155-v0cf7kf2
wandb: Run 'wandb offline' to turn off syncing.
wandb: Syncing run run
wandb: View project at https://wandb.ai/lillian-hurst2004-none/projet-kaggle
wandb: View run at https://wandb.ai/lillian-hurst2004-none/projet-kaggle/runs/v0cf7kf2
/home/lthurst/.conda/envs/projet-kaggle/lib/python3.12/site-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the learning rate.
warnings.warn(
Epoch 1/10, Loss: 0.23897105155808854, Accuracy: 0.904414259534072
Epoch 2/10, Loss: 0.15138308659842073, Accuracy: 0.9494674684987695
Epoch 3/10, Loss: 0.13836600534549368, Accuracy: 0.9552935998920806
Epoch 4/10, Loss: 0.12777418380850405, Accuracy: 0.9696300443451942
Epoch 5/10, Loss: 0.12091299483696985, Accuracy: 0.962169519946958
Epoch 6/10, Loss: 0.11663719831565676, Accuracy: 0.9638918498839958
Epoch 7/10, Loss: 0.11251634264291762, Accuracy: 0.9654460998443627
Epoch 8/10, Loss: 0.10978906253477692, Accuracy: 0.9667244773507104
Epoch 9/10, Loss: 0.10807128193813931, Accuracy: 0.9673753854132464
Epoch 10/10, Loss: 0.1060298996609401, Accuracy: 0.9680552018211367
Fichier de soumission sauvegardé : submission.csv
wandb: 10.012 MB of 0.012 MB uploaded
wandb: Run history:
wandb: Epoch Accuracy
wandb: Epoch Loss
wandb: Learning Rate
wandb:
wandb: Run summary:
wandb: Epoch Accuracy 0.96806
wandb: Epoch Loss 0.10603
wandb: Learning Rate 0.0006
wandb:
wandb: View run run at: https://wandb.ai/lillian-hurst2004-none/projet-kaggle/runs/v0cf7kf2
wandb: View project at: https://wandb.ai/lillian-hurst2004-none/projet-kaggle
wandb: Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)
wandb: Find logs at: ./wandb/run-20241212_172155-v0cf7kf2/logs
```