

CIFAR-10 (MLP)

La première étape intermédiaire de notre projet est d'utiliser les algorithmes ci-dessous sur le célèbre dataset CIFAR-10.

Il faut étudier :

1. L'influence de tous les hyperparamètres des modèles

- Structure
- Fonctions d'activations
- etc.

2. Les paramètres des algorithmes d'apprentissages

- Learning Rate
- Momentum
- etc.

In []:

```
import os
import numpy as np
from typing import List
from numpy.random import seed
import tensorflow as tf
from tensorflow.keras.layers import Flatten, Dense, Conv2D, BatchNormalization, Input, AveragePooling2D
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.metrics import categorical_accuracy
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.activations import relu, softmax, tanh
from tensorflow.keras.initializers import he_normal, glorot_uniform, Zeros
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.callbacks import TensorBoard
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.regularizers import l2, l1, l1_l2
from tensorflow.random import set_seed
```

In []:

```
print("Version de TensorFlow :", tf.__version__)
print("Nom du GPU :", tf.test.gpu_device_name())

tf.keras.backend.clear_session()
tf.config.optimizer.set_jit(False)
```

Importation du dataset

In []:

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

In []:

```
NUM_CLASSES = 10
IMG_SIZE = x_train[0].shape
```

In []:

```
x_train = x_train.astype('float32') / 256
x_test = x_test.astype('float32') / 256
```

```
y_train = to_categorical(y_train, num_classes=NUM_CLASSES)
y_test = to_categorical(y_test, num_classes=NUM_CLASSES)
```

In []:

```
LOG_DIR = os.path.join("logs")
MODELS_DIR = os.path.join("models")
```

Fixer les seeds

In []:

```
set_seed(42) # TensorFlow
seed(42) # NumPy
```

MLP

Pour voir l'influence des hyperparamètres sur un MLP, nous avons d'abord besoin de fixer le nombre d'époques et le nombre de layer sur notre couche de sortie.

In []:

```
EPOCHS = 500
SHUFFLE = True
NUM_CLASSES = 10
```

Pour nous faciliter la tâche, nous allons créer un modèle principal appelé *main* puis nous allons ajouter, modifier ou enlever un hyperparamètre à la fois pour que l'on puisse apercevoir son influence sur le modèle. Pour chacun des hyperparamètres déjà présent dans notre modèle principal, nous allons prendre une valeur excessivement élevée et une valeur très basse. En revanche, pour les hyperparamètres qui n'y s'y trouvent pas (ex: Dropout, L2), nous allons les ajouter à notre modèle et comparer.

Modèle main

Le modèle *main* est notre modèle principale, ce sera celui avec lequelle nous comparerons tout les autres prochains modèles et essayer de distinguer le plus clairement possible l'influence des hyperparamètres que nous modifierons. Nous avons choisi les hyperparamètres par défaut lorsque c'était possible et avons choisi le reste selon notre intuition.

In []:

```
def MLP_main(num_layer: int, nodes_by_layers: List[int]) -> Model:
    input_layer = Input(shape=(32, 32, 3))
    hidden_layers = Flatten()(input_layer)

    for n in range(num_layer):
        hidden_layers = Dense(nodes_by_layers[n], activation=relu, kernel_initializer=he_r

    output_layer = Dense(NUM_CLASSES, activation=softmax)(hidden_layers)
    return Model(input_layer, output_layer)
```

Voici les hyperparamètres de notre modèle :

In []:

```
batch_size = 256
num_layer = 6
nodes_by_layers = [32, 16, 64, 32, 16, 64]
learning_rate = 0.01
momentum = 0.45
```

Tout au long de notre section sur le MLP, nous n'utiliserons comme optimiseur uniquement Stochastic Gradient Descent (SGD). Nous pouvons désormais instancier et entraîner notre modèle.

In []:

```
mlp = MLP_main(num_layer, nodes_by_layers)

mlp.compile(
    loss=categorical_crossentropy,
    optimizer=SGD(learning_rate=learning_rate, momentum=momentum),
    metrics=categorical_accuracy
)

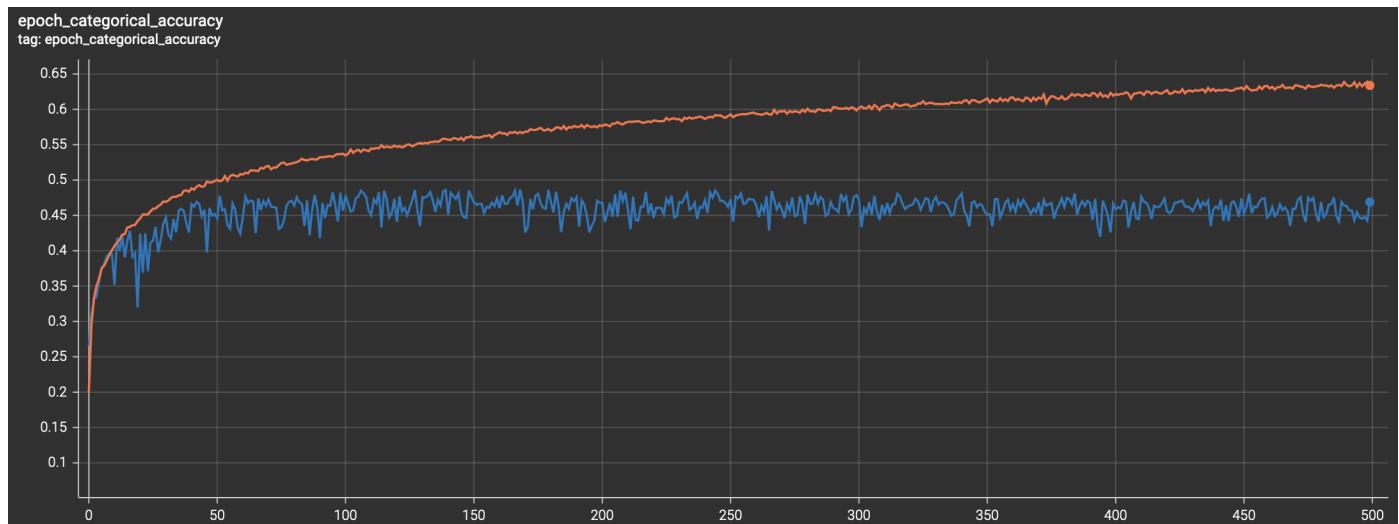
MLP_LOG = os.path.join(LOG_DIR, "mlp",
                      f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{lr}_mom_{mom}_af_relu_layers_{_}")

MLP_MODELS = os.path.join(MODELS_DIR, "mlp",
                          f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{lr}_mom_{mom}_af_relu_layers_{_}")

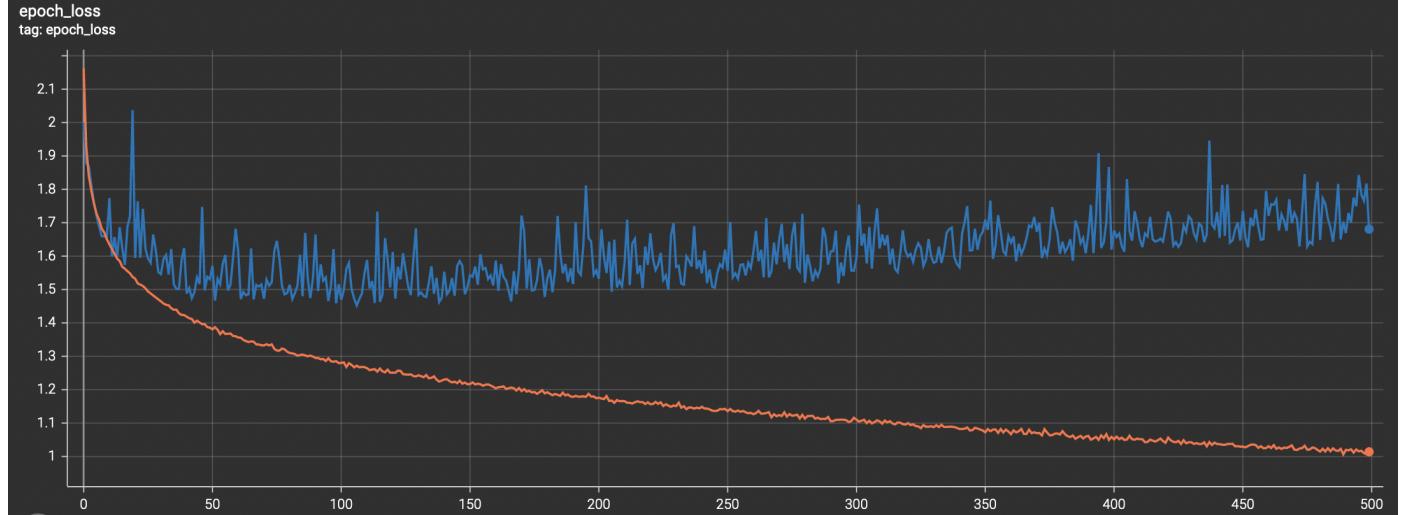
mlp.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=EPOCHS,
        validation_data=(x_test, y_test),
        shuffle=SHUFFLE,
        callbacks=[TensorBoard(MLP_LOG)])
)
mlp.save(MLP_MODELS)
```

Regardons la courbe de loss et d'accuracy de notre modèle principal.

En bleu, les courbes sur les données de validation. Et en orange, les courbes sur les données d'entraînement.



On peut désormais en être sûr, notre courbe de loss nous apprend que notre modèle **sur-apprend** mais après environ 150 époques. On peut le voir car notre courbe bleue (données de validation) augmente de plus en plus.



Nous pouvons voir que notre modèle stagne sur nos données de validation à partir de 50 époques, ce qui signifie qu'il sur-apprend après ces 50 époques. La courbe de loss nous permettra d'en être définitivement sûr.

Nombre de couches

Nous allons changer le nombre de couches de notre modèle, notre modèle initial contient 6 couches. Nous allons comparer avec un modèle à 2 puis à 20 couches pour voir l'importance de cette hyperparamètre.

In []:

```
batch_size = 256
learning_rate = 0.01
momentum = 0.45
num_layers = [2, 20]
nodes_by_layers = [32, 16, 64, 32, 16, 64, 32, 16, 64, 32, 16, 64, 32, 16, 64, 32, 16, 64, 32, 16, 64]
```

In []:

```
for i, num_layer in enumerate(num_layers):
    mlp = MLP_main(num_layer, nodes_by_layers)

    mlp.compile(
        loss=categorical_crossentropy,
        optimizer=SGD(learning_rate=learning_rate, momentum=momentum),
        metrics=categorical_accuracy
    )

    MLP_LOG = os.path.join(LOG_DIR, "mlp",
                           f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}")

    MLP_MODELS = os.path.join(MODELS_DIR, "mlp",
                             f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}")

    mlp.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=EPOCHS,
            validation_data=(x_test, y_test),
            shuffle=SHUFFLE,
            callbacks=[TensorBoard(MLP_LOG)])

```

mlp.save(MLP_MODELS)

Comparons les courbe de loss et d'accuracy de notre modèle principal avec nos deux modèles tests.

Rappel : En bleu foncé, les courbes du modèle principal sur les données de validation.

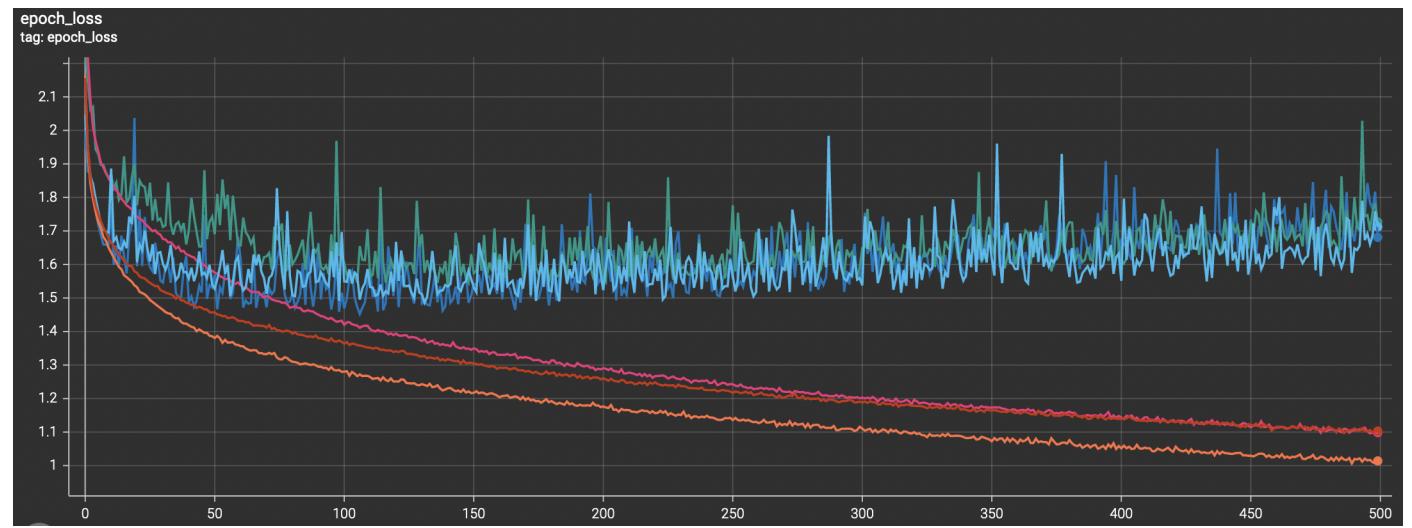
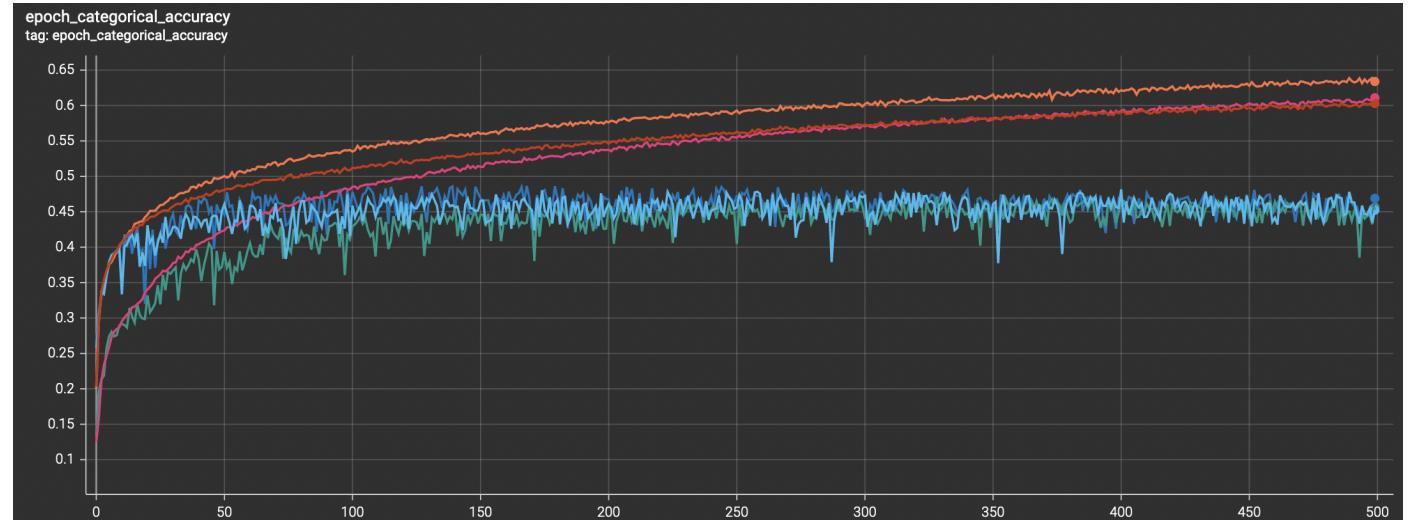
En orange, les courbes du modèle principal sur les données d'entraînement.

En bleu clair, les courbes du modèle à 2 couches cachées sur les données de validation.

En rouge, les courbes du modèle à 2 couches cachées sur les données d'entraînement.

En vert, les courbes du modèle à 20 couches cachées sur les données de validation.

En rose, les courbes du modèle à 20 couches cachées sur les données d'entraînement.



Commençons par comparer avec le modèle à **2 couches cachées**, nous pouvons voir que sur les données de validation, ce modèle à presque les mêmes résultats que notre modèle principal. La majeure différence se fait sur les données d'entraînement où l'on peut voir que le modèle moins profond apprend moins bien que le modèle initial.

Pour le modèle à **20 couches cachées**, nous pouvons voir que l'apprentissage est plus lent de manière générale, cela est dû à la profondeur du modèle.

Learning rate

Nous allons ajuster notre taux d'apprentissage (*learning rate* en anglais) de notre modèle, notre modèle initial contient un taux d'apprentissage de 0.01 (valeur par défaut). Nous allons comparer avec un modèle ayant 0.005 puis 0.05 en taux d'apprentissage pour voir l'importance de cette hyperparamètre sur notre modèle.

In []:

```
batch_size = 256
```

```

learning_rates = [0.0005, 0.05]
momentum = 0.45
num_layer = 6
nodes_by_layers = [32, 16, 64, 32, 16, 64]

```

```

In [ ]:
for i, learning_rate in enumerate(learning_rates):
    mlp = MLP_main(num_layer, nodes_by_layers)

    mlp.compile(
        loss=categorical_crossentropy,
        optimizer=SGD(learning_rate=learning_rate, momentum=momentum),
        metrics=categorical_accuracy
    )

    MLP_LOG = os.path.join(LOG_DIR, "mlp",
                           f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}")

    MLP_MODELS = os.path.join(MODELS_DIR, "mlp",
                             f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}")

    mlp.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=EPOCHS,
            validation_data=(x_test, y_test),
            shuffle=SHUFFLE,
            callbacks=[TensorBoard(MLP_LOG)])

```

)

```

    mlp.save(MLP_MODELS)

```

Comparons les courbes de loss et d'accuracy de notre modèle principal avec nos deux modèles tests.

Rappel : En bleu foncé, les courbes du modèle principal sur les données de validation.

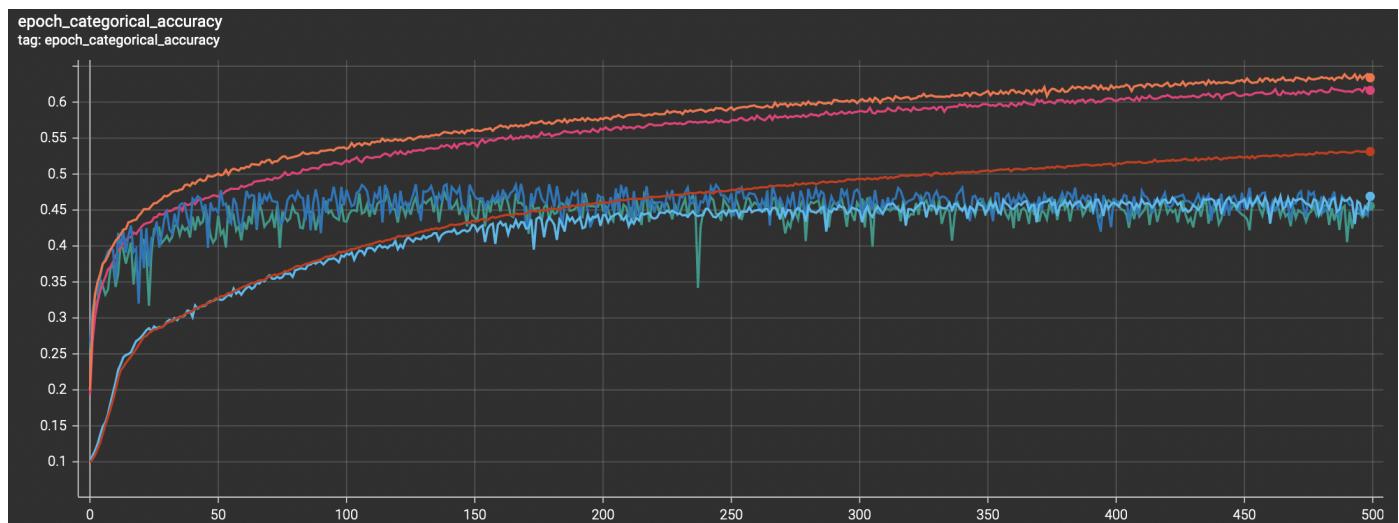
En orange, les courbes du modèle principal sur les données d'entraînement.

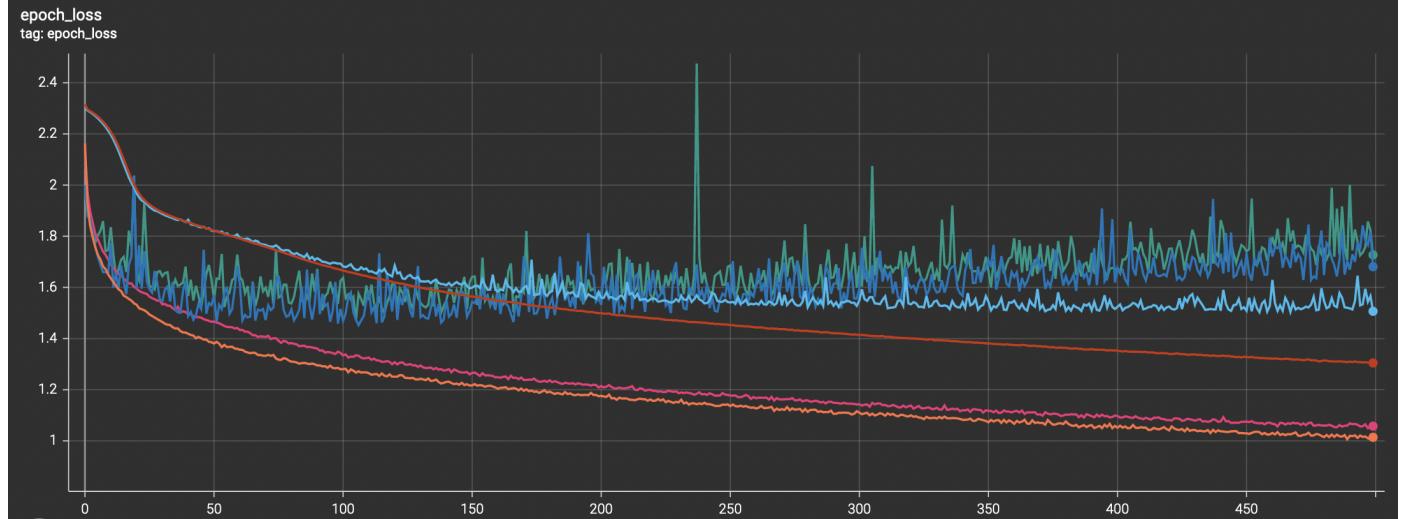
En bleu clair, les courbes du modèle ayant un taux d'apprentissage de 0.005 sur les données de validation.

En rouge, les courbes du modèle ayant un taux d'apprentissage de 0.005 sur les données d'entraînement.

En vert, les courbes du modèle ayant un taux d'apprentissage de 0.5 sur les données de validation.

En rose, les courbes du modèle ayant un taux d'apprentissage de 0.5 sur les données d'entraînement.





Commençons par comparer avec le modèle ayant un **taux d'apprentissage de 0.005**, nous pouvons voir que ce modèle est de manière générale plus stable mais apprend plus lentement, ce qui est tout à fait normal car notre modèle réalise de tout petit pas vers le minimum local (si, on a de la chance vers le minimum global).

Pour le modèle ayant un **taux d'apprentissage de 0.05**, nous pouvons voir que ce modèle à un apprentissage plus instable que le modèle initial, cela est dû au fort taux d'apprentissage qui réalise des pas très grand et diverge du minimum local.

Batch size

Nous allons changer le nombre de couches de notre modèle, notre modèle initial contient un batch size de 256. Nous allons comparer avec un modèle contenant un batch size de 32 puis de 1024 pour voir l'importance de cette hyperparamètre.

In []:

```
batch_sizes = [32, 1024]
learning_rate = 0.01
momentum = 0.45
num_layers = 6
nodes_by_layers = [32, 16, 64, 32, 16, 64]
```

In []:

```
for i, batch_size in enumerate(batch_sizes):
    mlp = MLP_main(num_layers, nodes_by_layers)

    mlp.compile(
        loss=categorical_crossentropy,
        optimizer=SGD(learning_rate=learning_rate, momentum=momentum),
        metrics=categorical_accuracy
    )

    MLP_LOG = os.path.join(LOG_DIR, "mlp",
                           f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}")

    MLP_MODELS = os.path.join(MODELS_DIR, "mlp",
                             f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}")

    mlp.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=EPOCHS,
            validation_data=(x_test, y_test),
            shuffle=SHUFFLE,
            callbacks=[TensorBoard(MLP_LOG)]
```

```
)  
mlp.save(MLP_MODELS)
```

Comparons les courbe de loss et d'accuracy de notre modèle principal avec nos deux modèles tests.

Rappel : En bleu foncé, les courbes du modèle principal sur les données de validation.

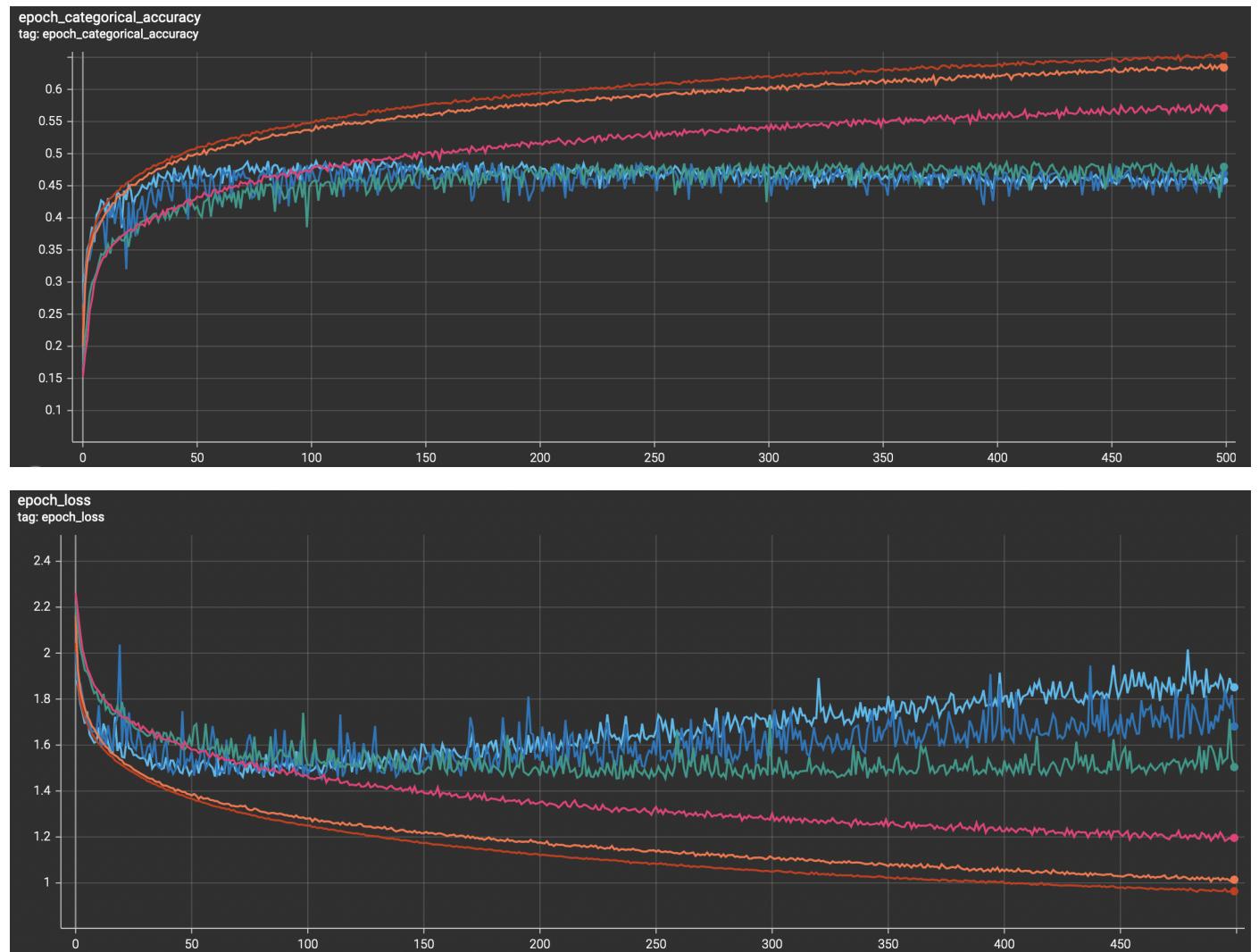
En orange, les courbes du modèle principal sur les données d'entraînement.

En bleu clair, les courbes du modèle ayant un batch size de 32 sur les données de validation.

En rouge, les courbes du modèle ayant un batch size de 32 sur les données d'entraînement.

En vert, les courbes du modèle ayant un batch size de 1024 sur les données de validation.

En rose, les courbes du modèle ayant un batch size de 1024 sur les données d'entraînement.



Commençons par comparer avec le modèle ayant un **batch size de 32**, l'apprentissage est meilleur mais beaucoup plus lent, on passe de 2 secondes pour un batch size de 256 à 11 secondes par époque. Cela est dû au découpage en batch du dataset, une époque correspond à un passage dans tout notre dataset. Plus le batch size est petit, plus nous réduisons notre nombre d'exemples dans chacun de nos batch et donc nos poids sont mis à jour plus souvent. C'est pour cela qu'un modèle avec un batch size faible est généralement meilleur.

Pour le modèle ayant un **batch size de 1024**, nous pouvons voir que le modèle apprend moins bien que les modèles comportant moins en batch size, car chacun des batch de notre dataset comprend plus d'images mais itére moins souvent les poids, ce qui nous donne un résultat moins bon pour notre nombre d'époques.

En revanche, il existe un lien direct entre le nombre d'époques et le batch size, nous devons entraîner le modèle plus longtemps que si nous avions un batch size moins élevé.

Fonction d'activations

Nous allons changer la fonction d'activation de toutes nos couches cachées, notre modèle initial contient comme fonction d'activation ReLU avec pour initialiseur de noyau He Normal. Nous allons comparer avec un modèle ne contenant pas de fonction d'activation avec pour initialiseur de noyau Glorot Uniform puis avec un modèle contenant comme fonction d'activation Tanh avec pour initialiseur de noyau Glorot Uniform pour voir l'importance de cette hyperparamètre.

In []:

```
def MLP_activation_function(num_layer: int, nodes_by_layers: List[int], activation_function):
    input_layer = Input(shape=(32, 32, 3))
    hidden_layers = Flatten()(input_layer)

    if activation_function.lower() == "linear":
        for n in range(num_layer):
            hidden_layers = Dense(nodes_by_layers[n], activation=None, kernel_initializer="he_normal")(hidden_layers)

    elif activation_function.lower() == "tanh":
        for n in range(num_layer):
            hidden_layers = Dense(nodes_by_layers[n], activation=tanh, kernel_initializer="glorot_uniform")(hidden_layers)

    else:
        for n in range(num_layer):
            hidden_layers = Dense(nodes_by_layers[n], activation=relu, kernel_initializer="glorot_uniform")(hidden_layers)

    output_layer = Dense(NUM_CLASSES, activation=softmax)(hidden_layers)
    return Model(input_layer, output_layer)
```

In []:

```
batch_size = 256
learning_rate = 0.01
momentum = 0.45
num_layers = 6
nodes_by_layers = [32, 16, 64, 32, 16, 64]
activation_functions = ["linear", "tanh"]
```

In []:

```
for i, activation_function in enumerate(activation_functions):
    mlp = MLP_activation_function(num_layers, nodes_by_layers, activation_function)

    mlp.compile(
        loss=categorical_crossentropy,
        optimizer=SGD(learning_rate=learning_rate, momentum=momentum),
        metrics=categorical_accuracy
    )

    MLP_LOG = os.path.join(LOG_DIR, "mlp",
                           f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}_{activation_function}")

    MLP_MODELS = os.path.join(MODELS_DIR, "mlp",
                             f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}_{activation_function}")

    mlp.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=EPOCHS,
            validation_data=(x_test, y_test),
            shuffle=SHUFFLE,
            callbacks=[TensorBoard(MLP_LOG)]
```

```
)  
mlp.save(MLP_MODELS)
```

Comparons les courbe de loss et d'accuracy de notre modèle principal avec nos deux modèles tests.

Rappel : En bleu foncé, les courbes du modèle principal sur les données de validation.

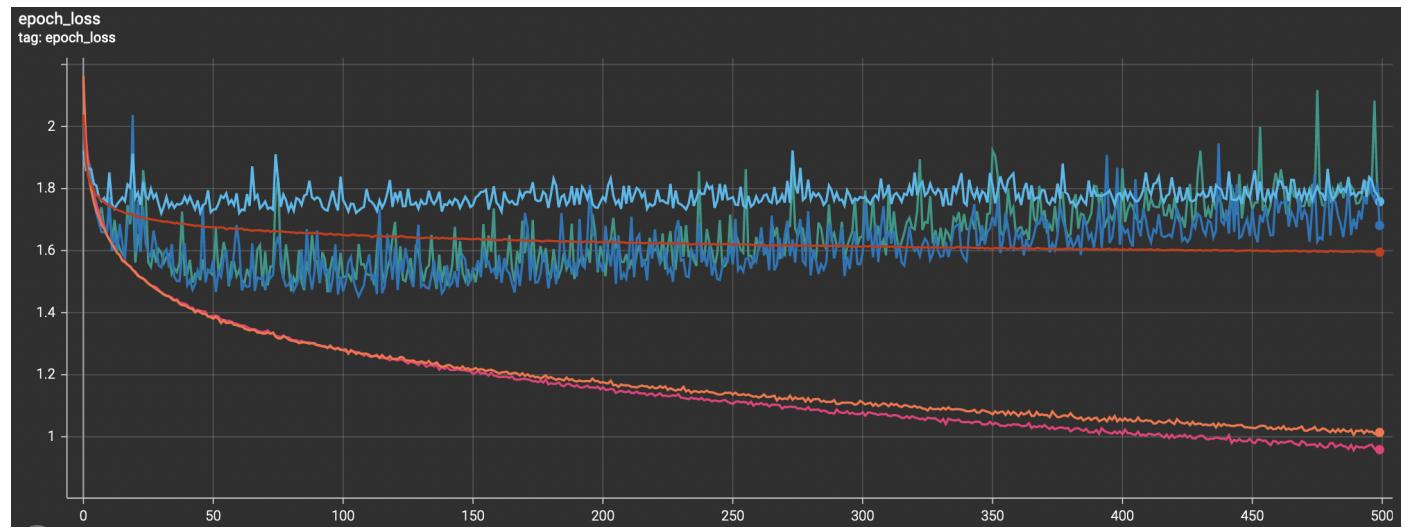
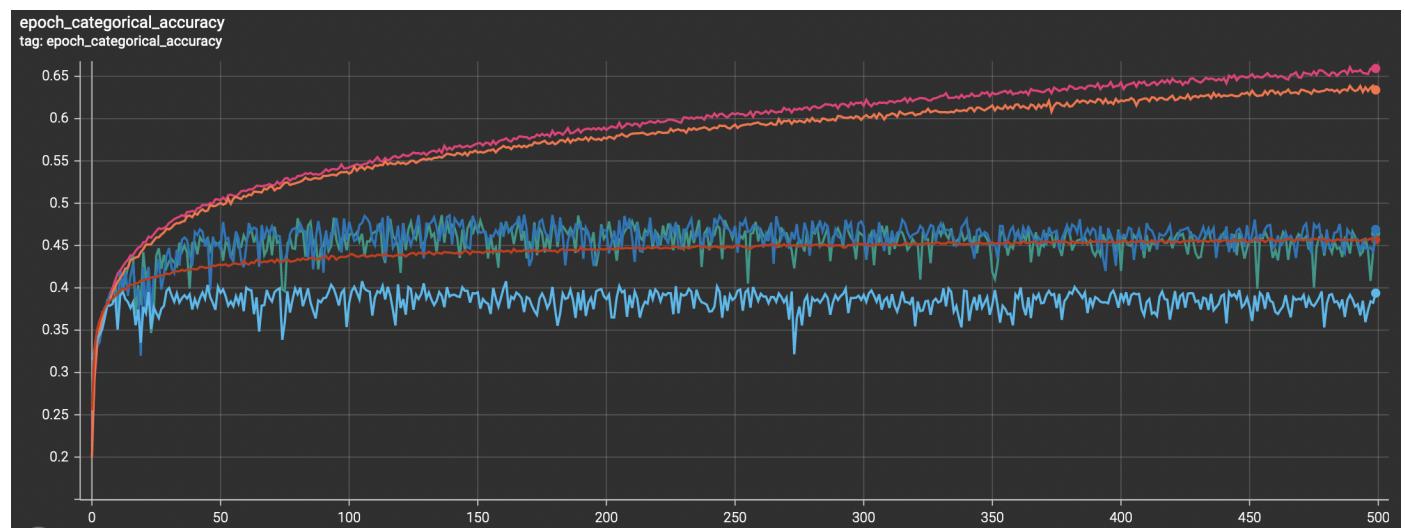
En orange, les courbes du modèle principal sur les données d'entraînement.

En bleu clair, les courbes du modèle n'ayant pas de fonction d'activation sur les données de validation.

En rouge, les courbes du modèle n'ayant pas de fonction d'activation sur les données d'entraînement.

En vert, les courbes du modèle ayant comme fonction d'activation Tanh sur les données de validation.

En rose, les courbes du modèle ayant comme fonction d'activation Tanh sur les données d'entraînement.



Commençons par comparer avec le modèle ayant comme **fonction d'activation Tanh**, nous pouvons voir que l'apprentissage est quasiment similaire à notre modèle initial.

Pour le modèle n'ayant **pas de fonction d'activation**, on peut voir que l'apprentissage est stable mais beaucoup moins efficace. On peut en déduire que la fonction d'activation joue un rôle important dans l'accuracy d'un modèle

Momentum

Nous allons changer la valeur du momentum de notre optimiseur, notre modèle initial contient 0.45 en

momentum. Nous allons comparer avec un modèle n'en contenant pas puis avec un modèle contenant un très fort momentum (0.95) pour voir l'importance de cette hyperparamètre.

In []:

```
batch_size = 256
learning_rate = 0.01
momentums = [0, 0.95]
num_layers = 6
nodes_by_layers = [32, 16, 64, 32, 16, 64]
```

In []:

```
for i, momentum in enumerate(momentums):
    mlp = MLP_main(num_layers, nodes_by_layers)

    mlp.compile(
        loss=categorical_crossentropy,
        optimizer=SGD(learning_rate=learning_rate, momentum=momentum),
        metrics=categorical_accuracy
    )

    MLP_LOG = os.path.join(LOG_DIR, "mlp",
                           f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}")

    MLP_MODELS = os.path.join(MODELS_DIR, "mlp",
                             f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}")

    mlp.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=EPOCHS,
            validation_data=(x_test, y_test),
            shuffle=SHUFFLE,
            callbacks=[TensorBoard(MLP_LOG)])
)

mlp.save(MLP_MODELS)
```

Comparons les courbe de loss et d'accuracy de notre modèle principal avec nos deux modèles tests.

Rappel : En bleu foncé, les courbes du modèle principal sur les données de validation.

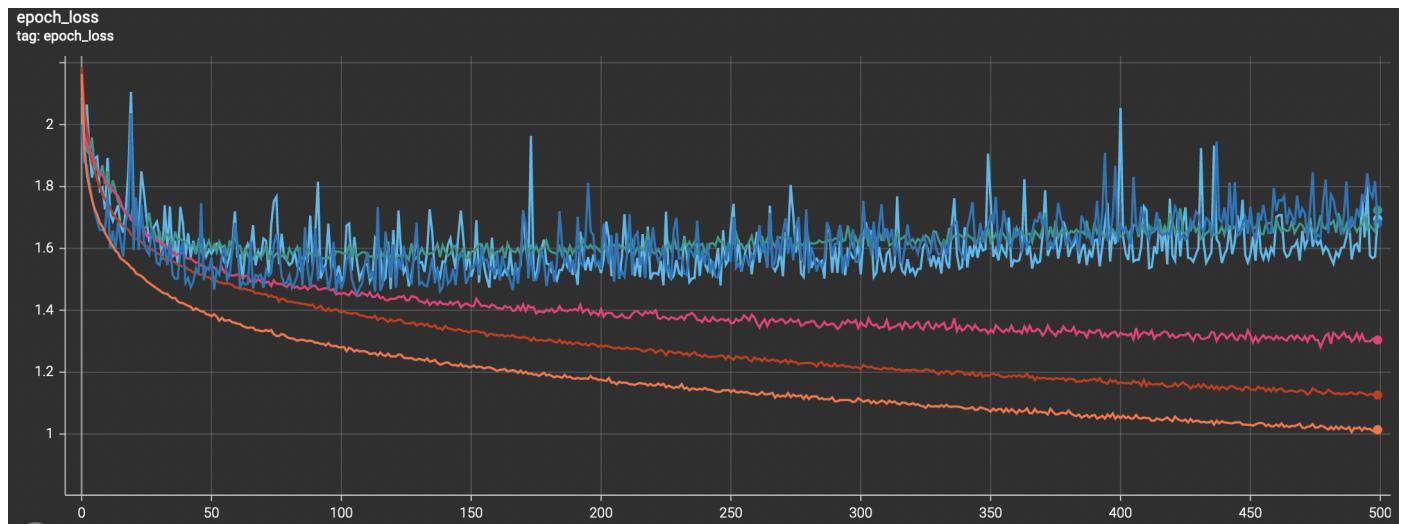
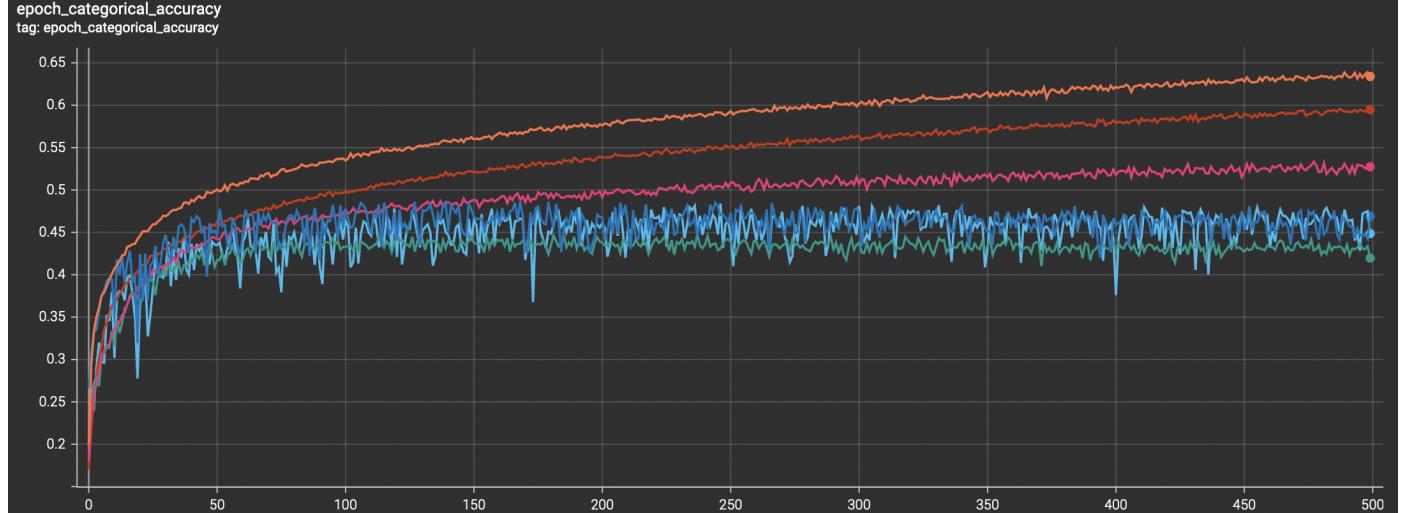
En orange, les courbes du modèle principal sur les données d'entraînement.

En bleu clair, les courbes du modèle n'ayant pas de momentum sur les données de validation.

En rouge, les courbes du modèle n'ayant pas de momentum sur les données d'entraînement.

En vert, les courbes du modèle ayant un momentum de 0.95 sur les données de validation.

En rose, les courbes du modèle ayant un momentum de 0.95 sur les données d'entraînement.



Commençons par comparer avec le modèle n'ayant **pas de momentum**, nous pouvons voir que l'apprentissage est moins performant et plus sinueux, car le SGD est naturellement instable.

Pour le modèle ayant un **momentum de 0.95**, nous pouvons remarquer l'apprentissage n'est pas très bon mais beaucoup plus stable. Notre momentum est très élevé, le modèle se base beaucoup trop sur ses directions précédentes.

Norme L2

Nous allons ajouter de la régularisation (norme L2) sur notre modèle, notre modèle initial n'en contient pas. Nous allons comparer avec un modèle contenant un L2 à 0.01 puis avec un modèle contenant un L2 à 0.5 pour voir l'importance de cette hyperparamètre.

In []:

```
def MLP_12(num_layer: int, nodes_by_layers: List[int], l2_val: float) -> Model:
    input_layer = Input(shape=(32, 32, 3))
    hidden_layers = Flatten()(input_layer)

    for n in range(num_layer):
        hidden_layers = Dense(nodes_by_layers[n], activation=relu, kernel_initializer=he_normal())(hidden_layers)
        hidden_layers = Dropout(0.2)(hidden_layers)

    output_layer = Dense(NUM_CLASSES, activation=softmax,
                         kernel_regularizer=l2(l2_val), bias_regularizer=l2(l2_val))(hidden_layers)

    return Model(input_layer, output_layer)
```

In []:

```
batch_size = 256
learning_rate = 0.01
```

```

momentum = 0.45
num_layers = 6
nodes_by_layers = [32, 16, 64, 32, 16, 64]
l2_vals = [0.01, 0.5]

```

In []:

```

for i, l2_val in enumerate(l2_vals):
    mlp = MLP_12(num_layers, nodes_by_layers, l2_val)

    mlp.compile(
        loss=categorical_crossentropy,
        optimizer=SGD(learning_rate=learning_rate, momentum=momentum),
        metrics=categorical_accuracy
    )

    MLP_LOG = os.path.join(LOG_DIR, "mlp",
                           f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}_{l2_val}")

    MLP_MODELS = os.path.join(MODELS_DIR, "mlp",
                             f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentum}_{l2_val}")

    mlp.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=EPOCHS,
            validation_data=(x_test, y_test),
            shuffle=SHUFFLE,
            callbacks=[TensorBoard(MLP_LOG)])
)
mlp.save(MLP_MODELS)

```

Comparons les courbes de loss et d'accuracy de notre modèle principal avec nos deux modèles tests.

Rappel : En bleu foncé, les courbes du modèle principal sur les données de validation.

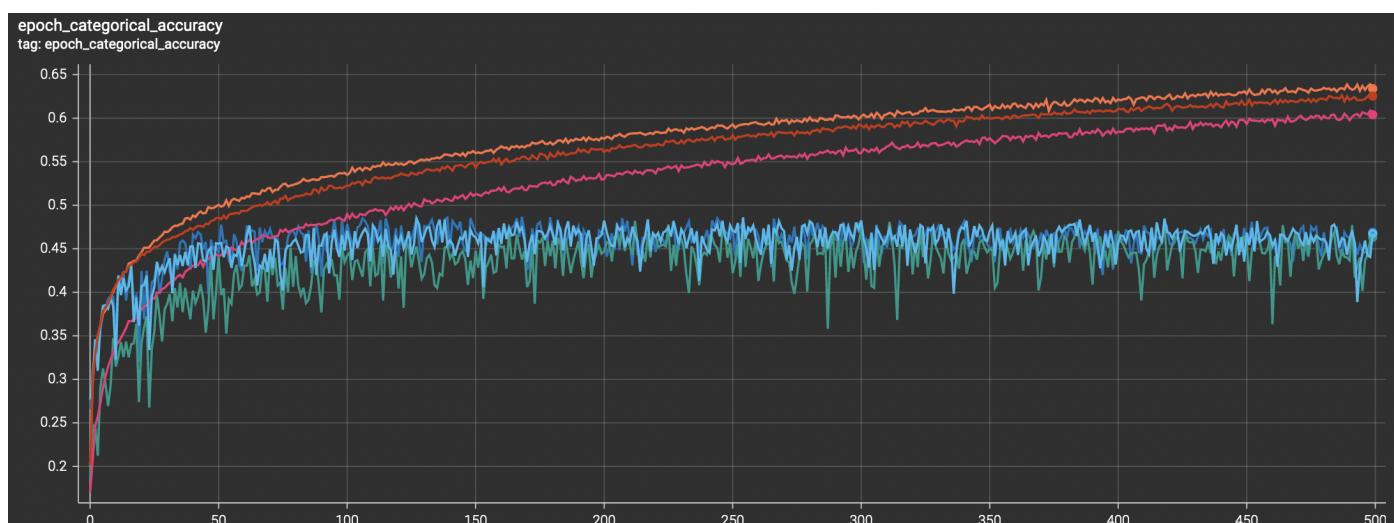
En orange, les courbes du modèle principal sur les données d'entraînement.

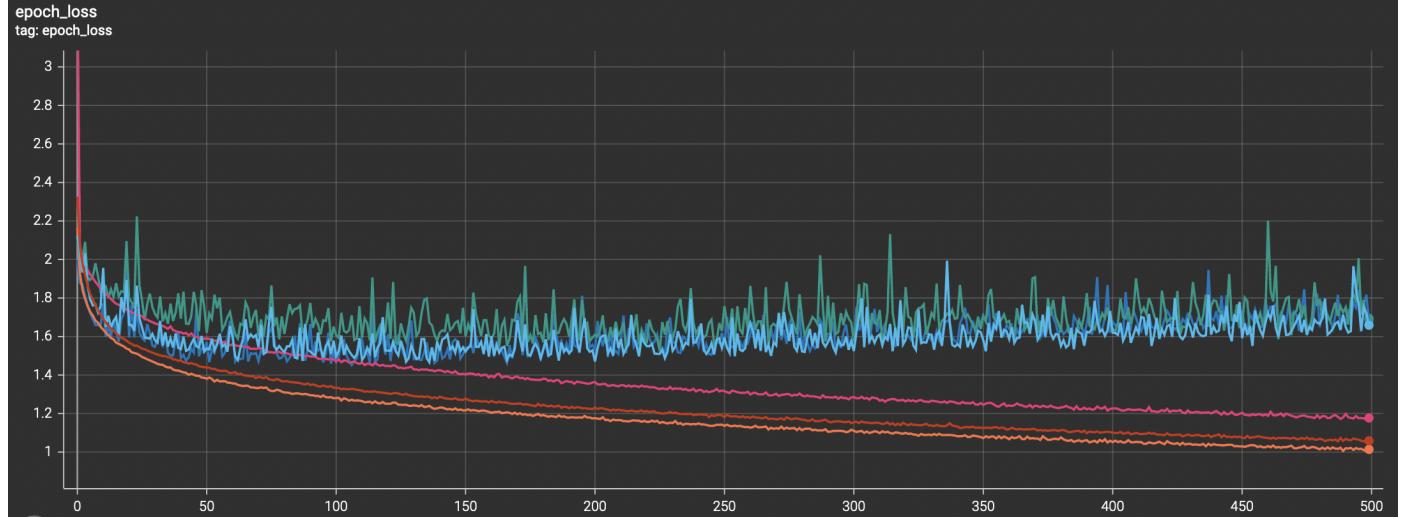
En bleu clair, les courbes du modèle ayant un L2 à 0.01 sur les données de validation.

En rouge, les courbes du modèle ayant un L2 à 0.01 sur les données d'entraînement.

En vert, les courbes du modèle ayant un L2 à 0.5 sur les données de validation.

En rose, les courbes du modèle ayant un L2 à 0.5 sur les données d'entraînement.





Commençons par comparer avec le modèle ayant un **L2 de 0.01**, nous pouvons voir que le modèle performe légèrement moins bien que le modèle initial. L'effet de la norme L2 est beaucoup trop faible pour réellement impacté le modèle test.

Pour le modèle ayant un **L2 de 0.5**, nous pouvons voir que le modèle commence à sur-apprendre plus tard que nos deux modèles (vers 225 époques). La norme L2 permet de retarder le plus possible le sur-apprentissage.

Dropout

Nous allons ajouter de la régularisation (Dropout) sur notre modèle, notre modèle initial n'en contient pas. Nous allons comparer avec un modèle contenant un dropout sur 3 couches cachées. Puis avec un modèle contenant sur toutes ses couches cachées un dropout pour voir l'importance de cette hyperparamètre.

In []:

```
def MLP_dropout(num_layer: int, nodes_by_layers: List[int], full_dropout: bool) -> Model:
    input_layer = Input(shape=(32, 32, 3))
    hidden_layers = Flatten()(input_layer)

    for n in range(num_layer):
        hidden_layers = Dense(nodes_by_layers[n], activation=relu, kernel_initializer=he_r
        if full_dropout or n % 2 == 0:
            hidden_layers = Dropout(0.2)(hidden_layers)

    output_layer = Dense(NUM_CLASSES, activation=softmax)(hidden_layers)
    return Model(input_layer, output_layer)
```

In []:

```
batch_size = 256
learning_rate = 0.01
momentum = 0.45
num_layers = 6
nodes_by_layers = [32, 16, 64, 32, 16, 64]
full_dropouts = [False, True]
```

In []:

```
for i, full_dropout in enumerate(full_dropouts):
    mlp = MLP_dropout(num_layers, nodes_by_layers, full_dropout)

    mlp.compile(
        loss=categorical_crossentropy,
        optimizer=SGD(learning_rate=learning_rate, momentum=momentum),
        metrics=categorical_accuracy
    )
```

```

MLP_LOG = os.path.join(LOG_DIR, "mlp",
                      f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentur")

MLP_MODELS = os.path.join(MODELS_DIR, "mlp",
                         f"ep_{EPOCHS}_bs_{batch_size}_opt_SGD_lr_{learning_rate}_mom_{momentur"

mlp.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=EPOCHS,
        validation_data=(x_test, y_test),
        shuffle=SHUFFLE,
        callbacks=[TensorBoard(MLP_LOG)
                  ]
)
mlp.save(MLP_MODELS)

```

Comparons les courbe de loss et d'accuracy de notre modèle principal avec nos deux modèles tests.

Rappel : En bleu foncé, les courbes du modèle principal sur les données de validation.

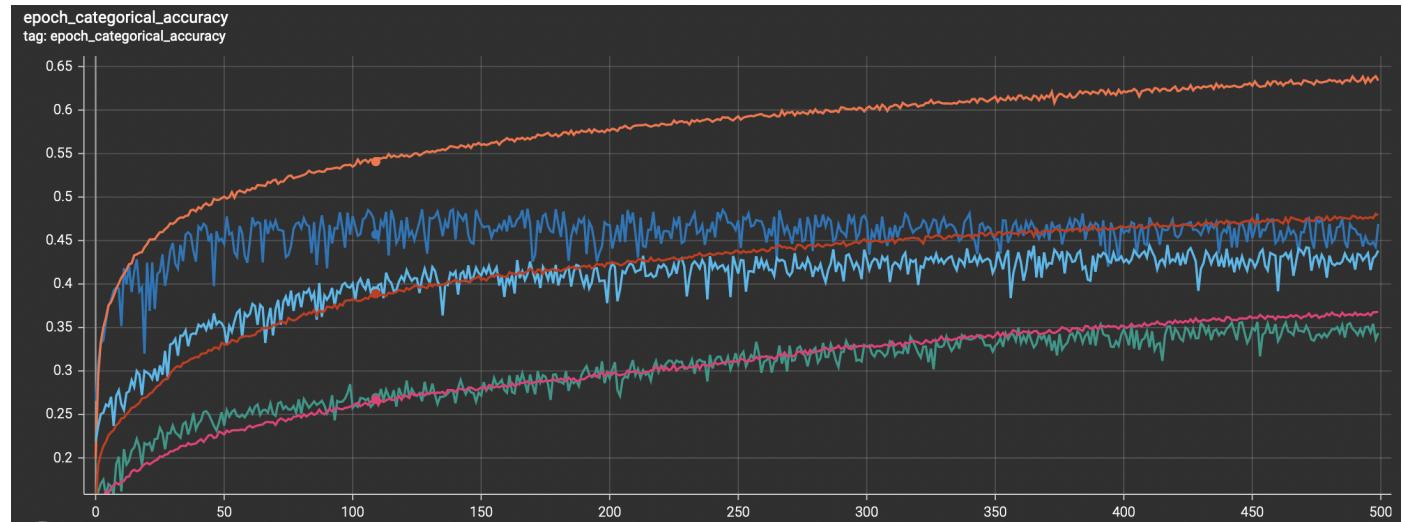
En orange, les courbes du modèle principal sur les données d'entraînement.

En bleu clair, les courbes du modèle ayant un dropout sur 3 couches cachées sur les données de validation.

En rouge, les courbes du modèle ayant un dropout sur 3 couches cachées sur les données d'entraînement.

En vert, les courbes du modèle ayant un dropout sur toutes les couches cachées sur les données de validation.

En rose, les courbes du modèle ayant un dropout sur toutes les couches cachées sur les données d'entraînement.





Commençons par comparer avec le modèle ayant un **dropout sur 3 couches cachées**, nous pouvons voir que le modèle est beaucoup plus longtemps en phase de sous-apprentissage que le modèle initiale.

Pour le modèle ayant un **dropout sur toutes ses couches cachées**, nous pouvons voir que le modèle n'est même pas encore sorti de sa phase de sous-apprentissage après les 500 époques. Le Dropout comme la norme L2 sont des techniques utilisés pour empêcher des modèles de sur-apprendre, dans notre cas, c'est un peu extrême.