CIFAR-10 Linear Model

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

Les algorithmes à étudier :

Modèles précédents

- Modèle Linéaire
- Perceptron Multicouches

Nouveaux modèles

- ConvNet(s)
- ResNets / HighwayNets
- RNN(s)

Pour chacun des algorithmes cités, il faut :

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.

Méthodologie

1. Créer un modèle classique.
2. Entraîner le modèle pendant 500 epochs.
3. Examiner sa courbe TensorBoard et son accuracy.
4. Augmenter un des hyperparamètre.
5. Réduire ce même hyperparamètre.
6. Dire l'influence de cette hyperparamètre sur la courbe et sur le modèle de manière générale.
7. Recommencer à partir de l'étape 3 pour tout les hyperparamètres possibles.

Hyperparamètres

- Batch size
- Learning rate
- Momentum
- Structure
- Fonction d'activation
- Initialization de kernel
- Regularizers (Dropout, L1 Norm, L2 Norm)

Qui s'occupe de quoi ?

- Perceptron (Mamadian)
- MLP (Réda M.)
- ConvNet (Reda B.)

In [ ]:
```
import os
import numpy as np
from numpy.random import seed
import tensorflow as tf
from tensorflow.keras.layers import Flatten, Dense, Conv2D, BatchNormalization, Input, Ave
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
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")
```

Fixer les seeds

In [ ]:
```
set_seed(42) # TensorFlow
seed(42) # NumPy
```

Modèle linéaire

In [ ]:
```
EPOCHS = 500
SHUFFLE = True
BATCH_SIZE = 256
```

In [ ]:
```
class Perceptron(Model):
    def __init__(self, activation_function, kernel_initializer, l2_val=0, l1_val=0):
        super(Perceptron, self).__init__()
        self._flatten_layer = Flatten()
        if l1_val > 0 and l2_val > 0:
            self._dense_layer = Dense(IMG_SIZE[0] * IMG_SIZE[1] * IMG_SIZE[2],
                                      activation=activation_function,
                                      kernel_initializer=kernel_initializer,
                                      kernel_regularizer=l1_l2(l1_val, l2_val),
                                      bias_regularizer=l1_l2(l1_val, l2_val)
                                      )
        elif l1_val > 0:
            self._dense_layer = Dense(IMG_SIZE[0] * IMG_SIZE[1] * IMG_SIZE[2],
                                      activation=activation_function,
                                      kernel_initializer=kernel_initializer,
                                      kernel_regularizer=l1(l1_val),
                                      bias_regularizer=l1(l1_val)
                                      )
        elif l2_val > 0:
            self._dense_layer = Dense(IMG_SIZE[0] * IMG_SIZE[1] * IMG_SIZE[2],
                                      activation=activation_function,
                                      kernel_initializer=kernel_initializer,
                                      kernel_regularizer=l2(l2_val),
                                      bias_regularizer=l2(l2_val)
                                      )
        else:
            self._dense_layer = Dense(IMG_SIZE[0] * IMG_SIZE[1] * IMG_SIZE[2],
                                      activation=activation_function,
                                      kernel_initializer=kernel_initializer
                                      )
        self._output_layer = Dense(NUM_CLASSES, activation=softmax)


    def call(self, inputs):
        x = self._flatten_layer(inputs)
        x = self._dense_layer(x)
        return self._output_layer(x)
```

Main Linear Model

Hyperparameters

In [ ]:
```
learning_rate = 0.01
momentum = 0.0
l2_val = 0.00
l1_val = 0.00
using_l1 = l1_val > 0
using_l2 = l2_val > 0
ki = glorot_uniform
fa = tanh
version = '_pft'
```

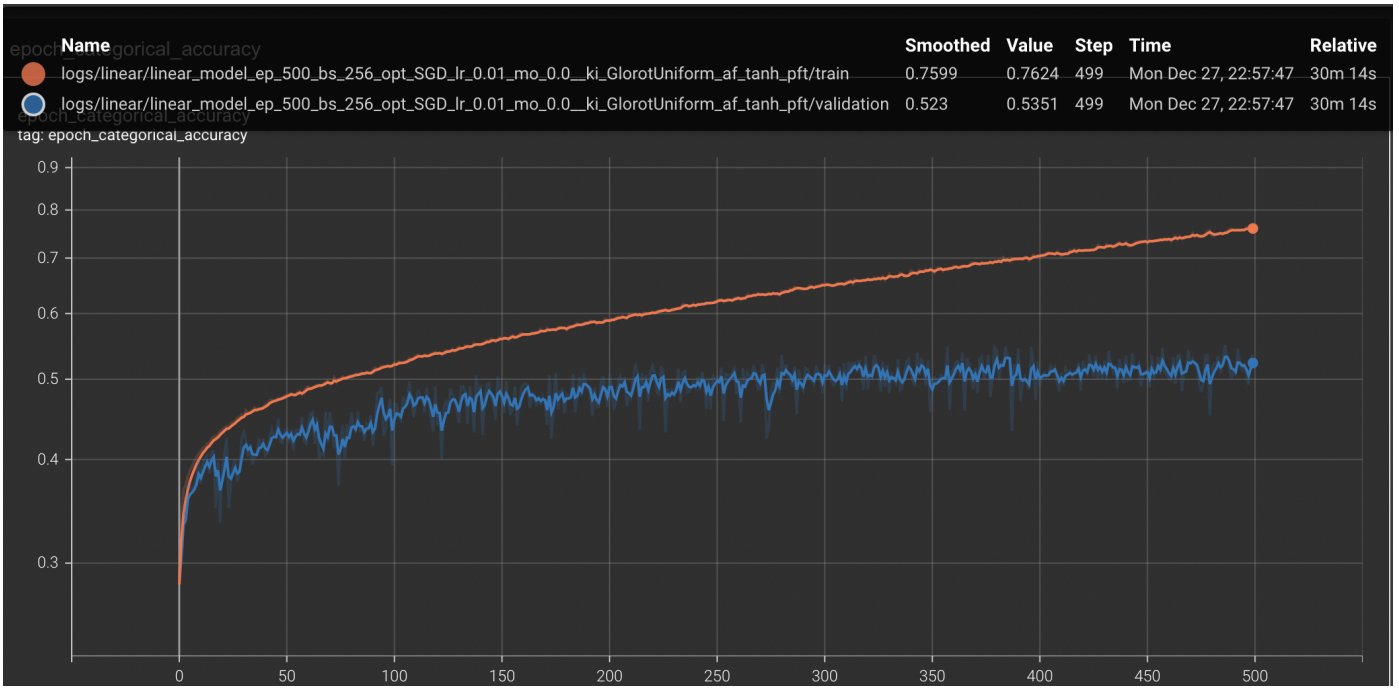
training

In [ ]:
```
perceptron = Perceptron(fa, ki, l1_val=l1_val, l2_val=l2_val)
perceptron.compile(loss=categorical_crossentropy,
                   optimizer=SGD(learning_rate=learning_rate,
```

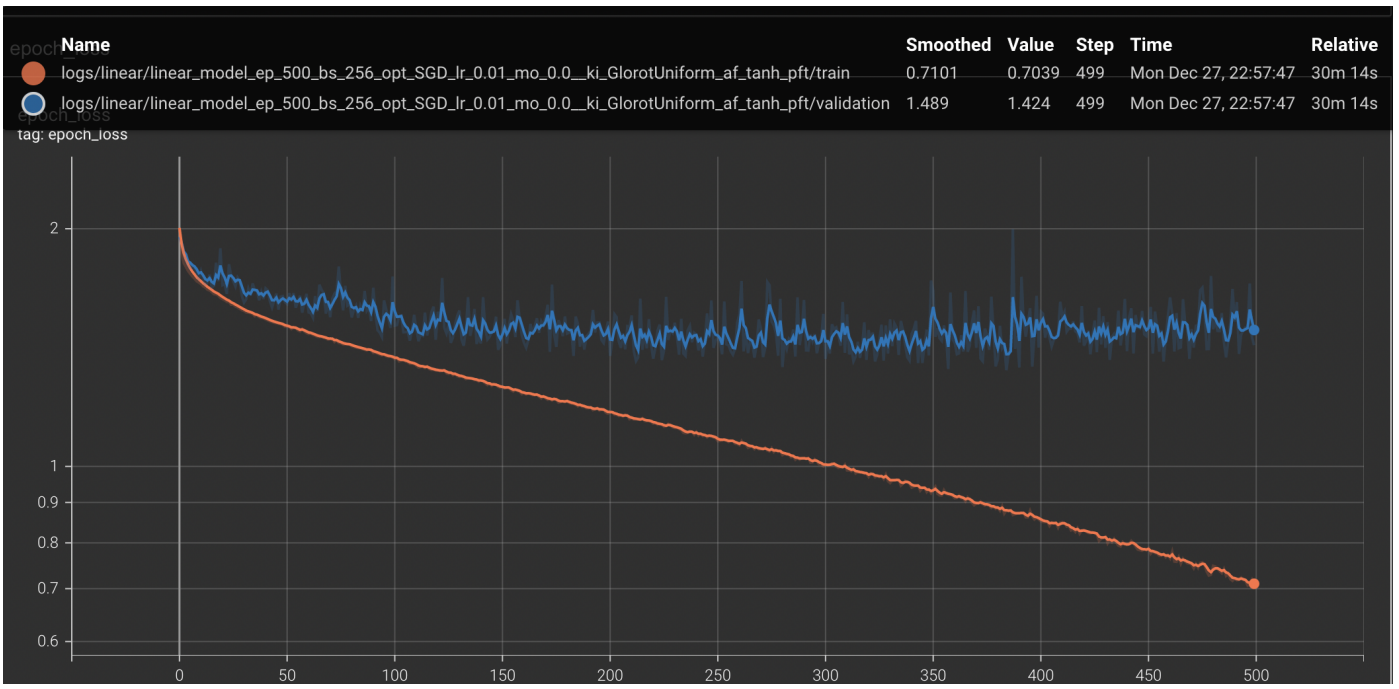
```
                                    momentum=momentum),
                  metrics=categorical_accuracy)

log_name = os.path.join(LOG_DIR,
                        "linear",
                        f"linear_model_ep_{EPOCHS}_bs_{BATCH_SIZE}_opt_SGD_lr_{learnir

print(log_name)
perceptron.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, validation_data=(x_
```

resultat:

Main model accuracy



Main model loss



learning rate analysis Models

hyperparameters

In [ ]:

```
learning_rate = [0.05,0.2,0.001]
```

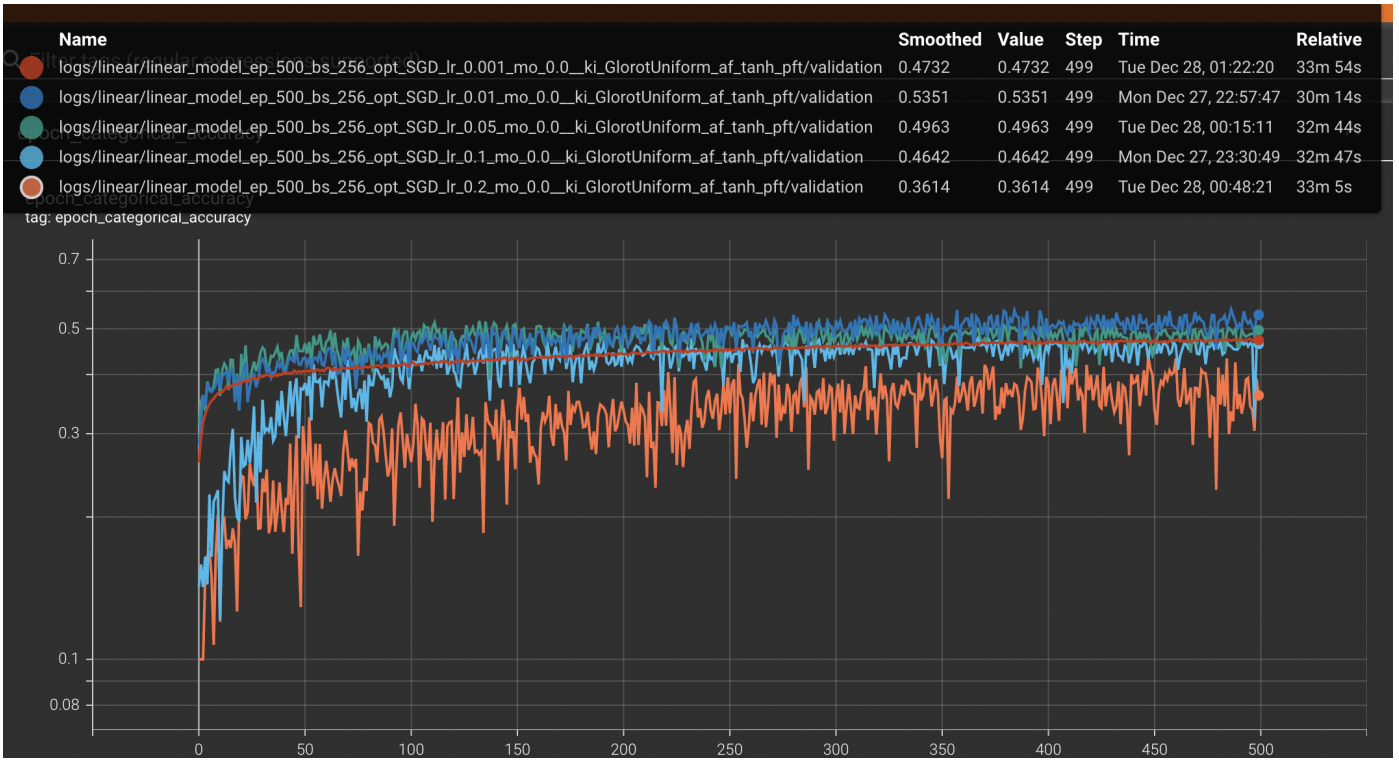
```
momentum = 0.0
l2_val = 0.00
l1_val = 0.00
using_l1 = l1_val > 0
using_l2 = l2_val > 0
ki = glorot_uniform
fa = tanh
version = '_pft'
```

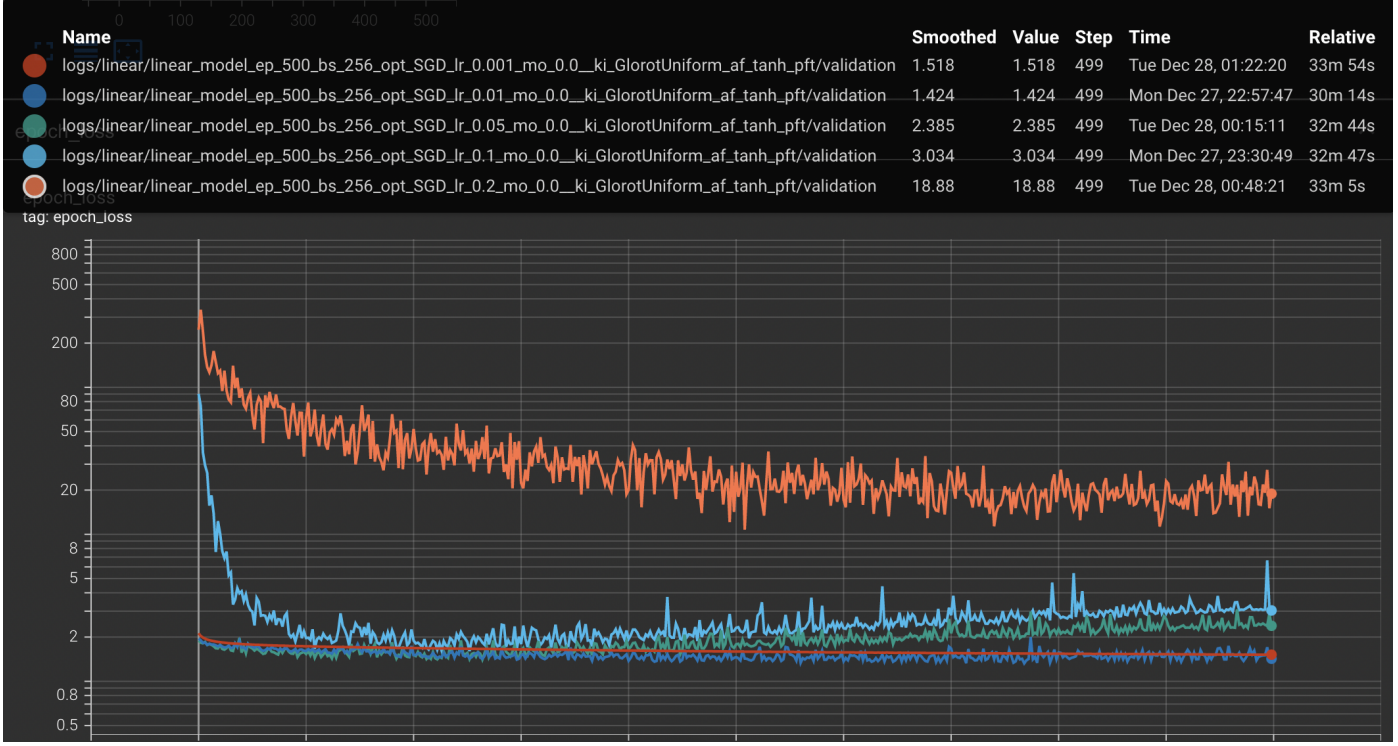
training

In [ ]:

```
for lr in learning_rate:
    perceptron = Perceptron(fa, ki, l1_val=l1_val, l2_val=l2_val)
    perceptron.compile(loss=categorical_crossentropy,
                       optimizer=SGD(learning_rate=lr,
                                     momentum=momentum),
                       metrics=categorical_accuracy)

    log_name = os.path.join(LOG_DIR,
                            "linear",
                            f"linear_model_ep_{EPOCHS}_bs_{BATCH_SIZE}_opt_SGD_lr_{lr}_mo_

    print(log_name)
    perceptron.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, validation_data
```

Resultat:





On peut voir que la courbe rouge (lr à 0,001) est beaucoup plus douce que les autres courbes car elle a un pas d'apprentissage beaucoup plus faible. On peut voir que sur la courbe orange (lr à 0,02) il y a des variations brusque après chaque epochs à cause d'un trop grand pas.

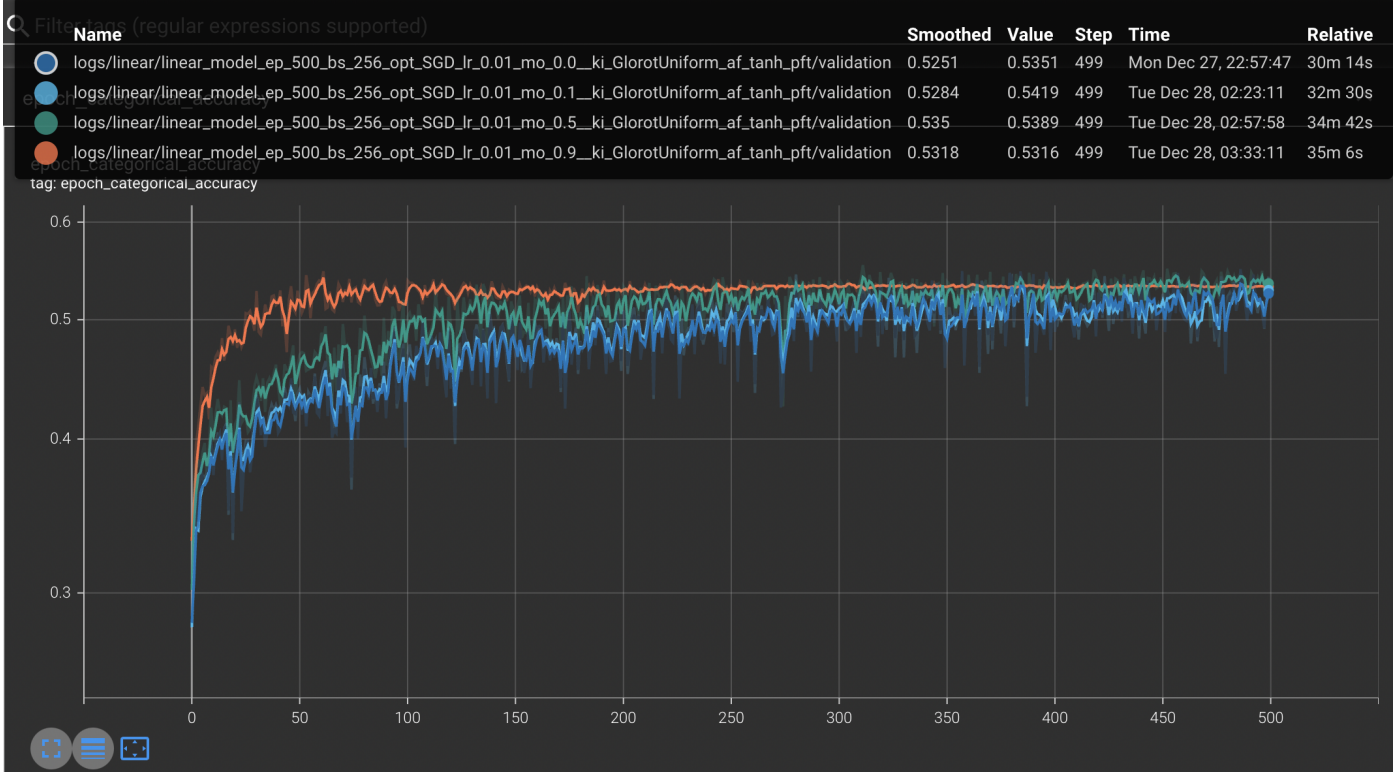
Momentum analysis Models

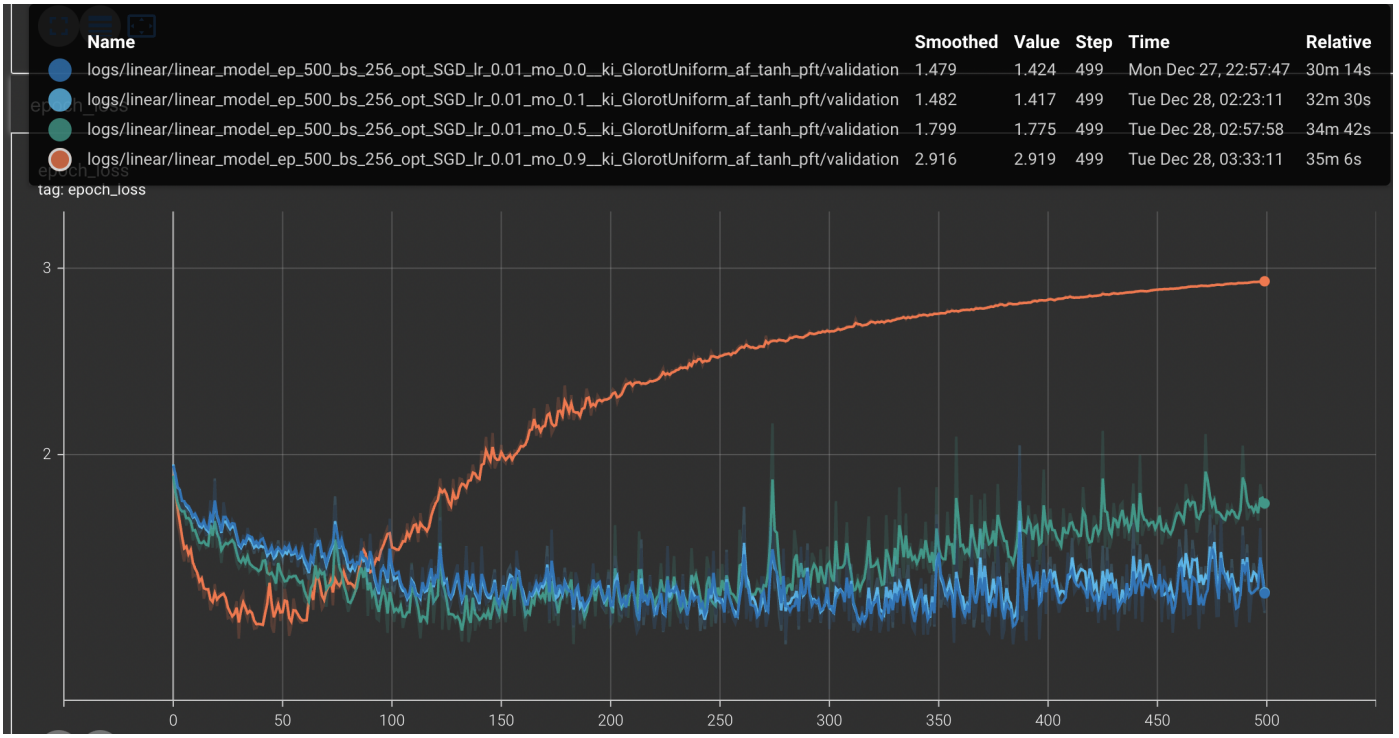
In [ ]:
```
learning_rate = 0.01
momentum = [0.1, 0.5, 0.9]
l2_val = 0.00
l1_val = 0.00
using_l1 = l1_val > 0
using_l2 = l2_val > 0
ki = glorot_uniform
fa = tanh
version = '_pft'
```

training

In [ ]:
```
for mt in momentum:
    perceptron = Perceptron(fa, ki, l1_val=l1_val, l2_val=l2_val)
    perceptron.compile(loss=categorical_crossentropy,
                       optimizer=SGD(learning_rate=learning_rate,
                                     momentum=mt),
                       metrics=categorical_accuracy)

    log_name = os.path.join(LOG_DIR,
                            "linear",
                            f"linear_model_ep_{EPOCHS}_bs_{BATCH_SIZE}_opt_SGD_lr_{learnir

    print(log_name)
    perceptron.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, validation_data
```

Resultat:





On peut voir que lorsque le momentum est grand (courbe orange) on converge beaucoup plus rapidement vers le minimum global. On remarque aussi que le modele avec un momentum élevé (courbe orange) sur-apprend beaucoup plus tot car il est dans un minimum local bien plus tôt.

Kernel Initializer analysis Models

hyperparameters

In [ ]:

```
learning_rate = 0.01
momentum = 0.0
l2_val = 0.00
l1_val = 0.00
using_l1 = l1_val > 0
using_l2 = l2_val > 0
ki = he_normal
```

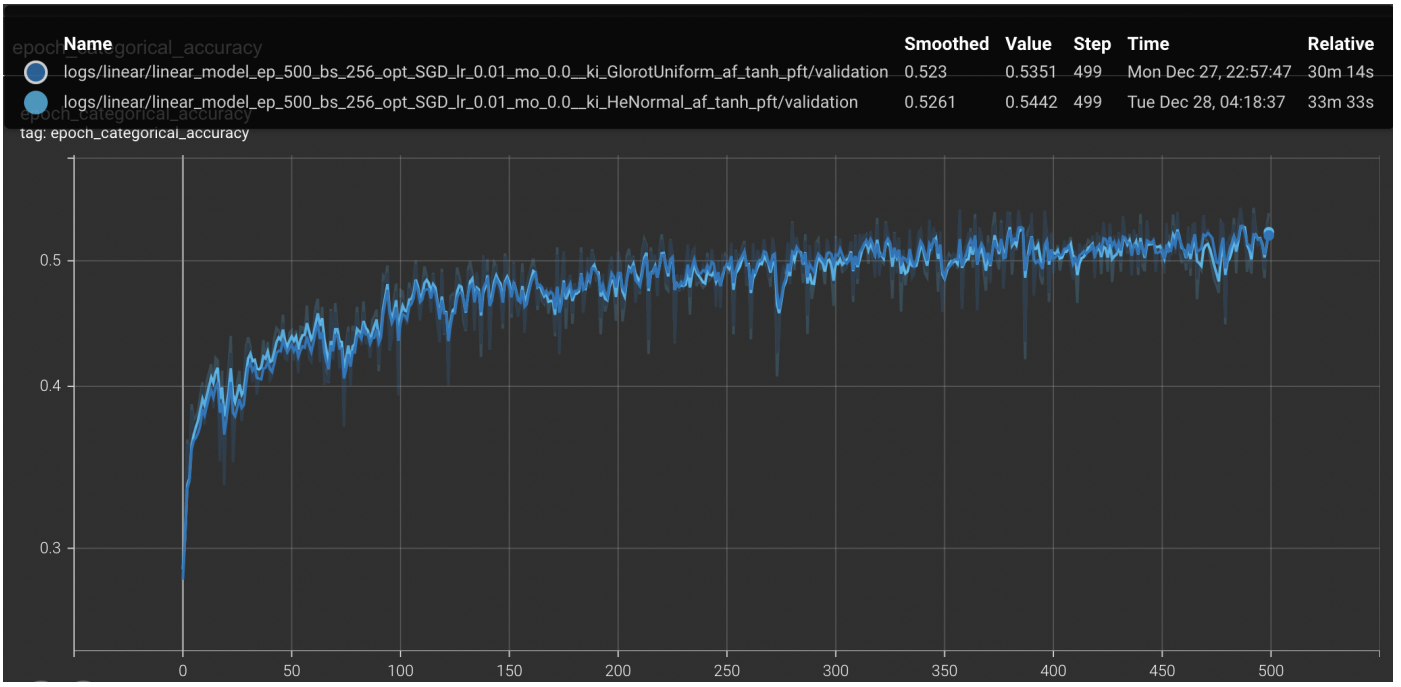
```
fa = tanh
version = '_pft'
```

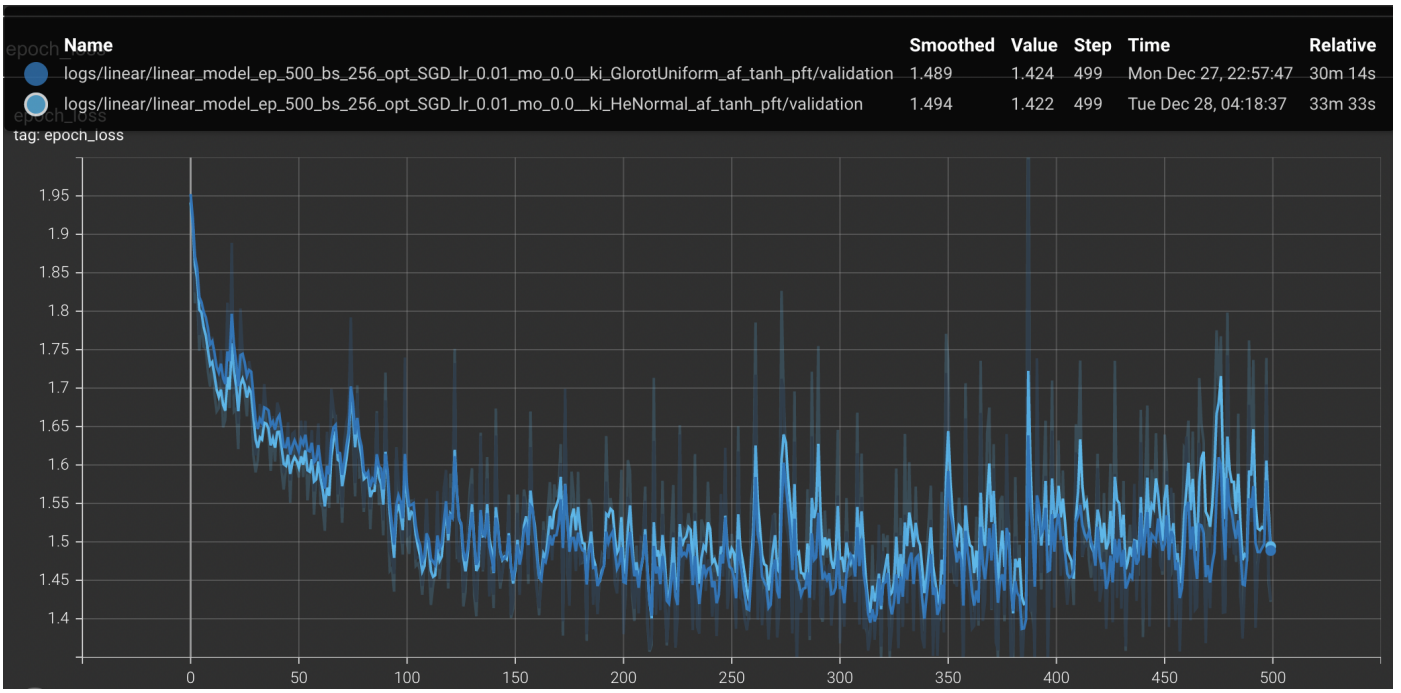
training

In [ ]:

```
perceptron = Perceptron(fa, ki, l1_val=l1_val, l2_val=l2_val)
perceptron.compile(loss=categorical_crossentropy,
                   optimizer=SGD(learning_rate=learning_rate,
                                 momentum=momentum),
                   metrics=categorical_accuracy)

log_name = os.path.join(LOG_DIR,
                        "linear",
                        f"linear_model_ep_{EPOCHS}_bs_{BATCH_SIZE}_opt_SGD_lr_{learnir

print(log_name)
perceptron.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, validation_data=(x_
```

Resultat:





Sur l'analyse du kernel initializer, nous pouvons emettre l'hypothese que dans notre modele lineaire le changement du kernel initializer seul n'a pas de réel impact. Les courbes sont assez identique.

Activation function analysis Models

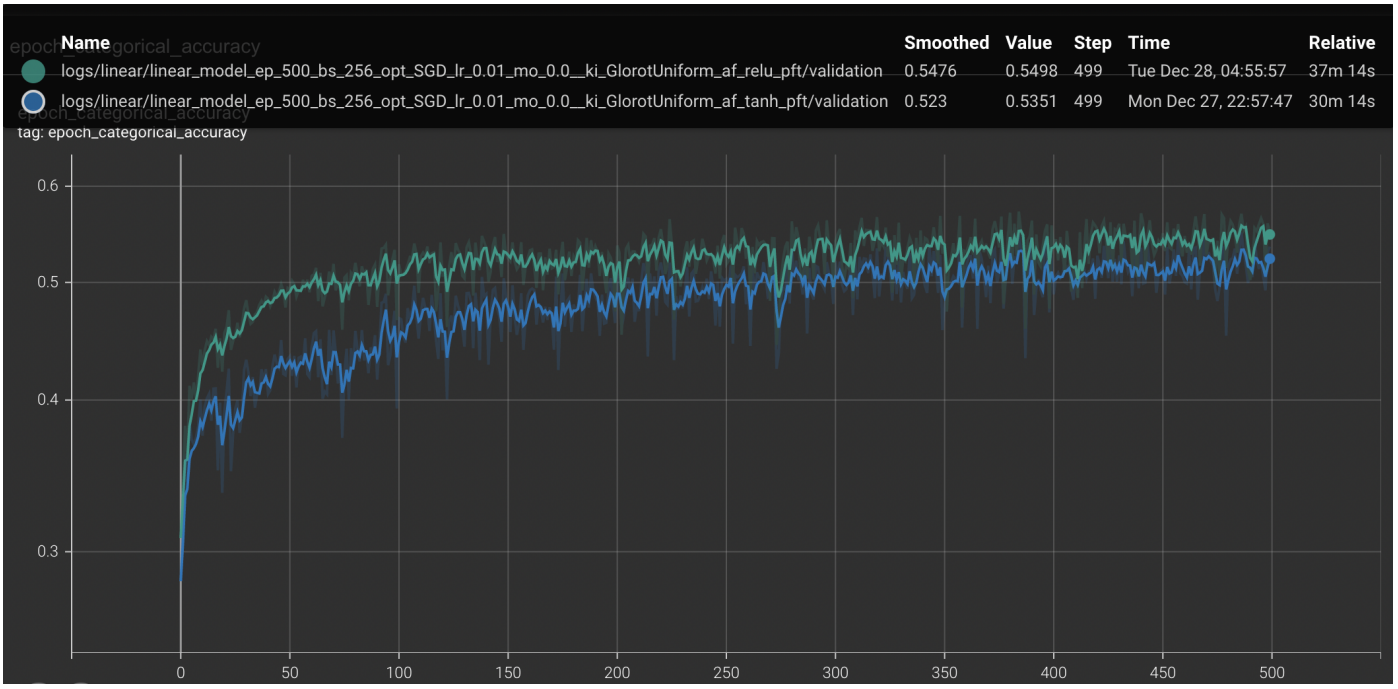
Hyperparameters

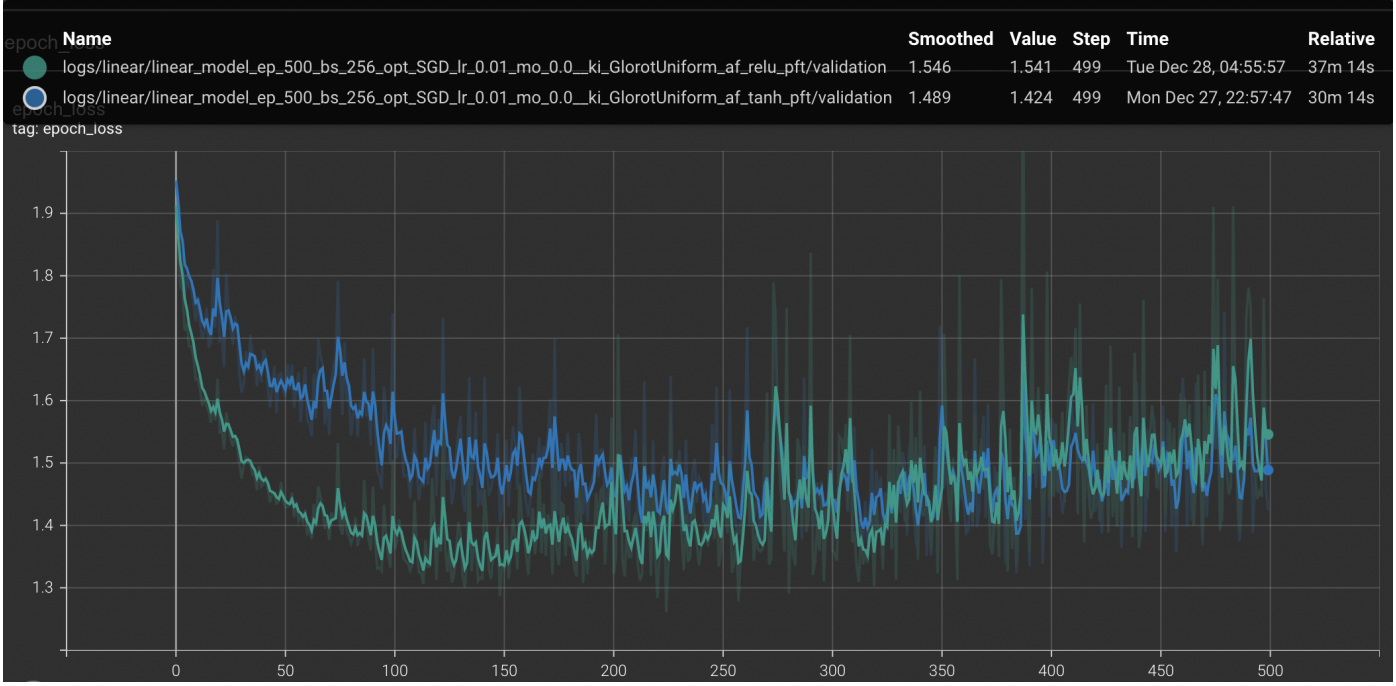
In [ ]:
```
learning_rate = 0.01
momentum = 0.0
l2_val = 0.00
l1_val = 0.00
using_l1 = l1_val > 0
using_l2 = l2_val > 0
ki = glorot_uniform
fa = relu
version = '_pft'
```

training

In [ ]:
```
perceptron = Perceptron(fa, ki, l1_val=l1_val, l2_val=l2_val)
perceptron.compile(loss=categorical_crossentropy,
                   optimizer=SGD(learning_rate=learning_rate,
                                 momentum=momentum),
                   metrics=categorical_accuracy)

log_name = os.path.join(LOG_DIR,
                        "linear",
                        f"linear_model_ep_{EPOCHS}_bs_{BATCH_SIZE}_opt_SGD_lr_{learnir

print(log_name)
perceptron.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, validation_data=(x_
```

Resultat:





On peut voir que la courbe verte (af relu) a des meilleurs resultats que la courbe principal(en bleu), Mais on constate qu'elle sur-apprend beaucoup plus tôt et que le temps d'entrainement est un peu plus long.

Batch size analysis Models

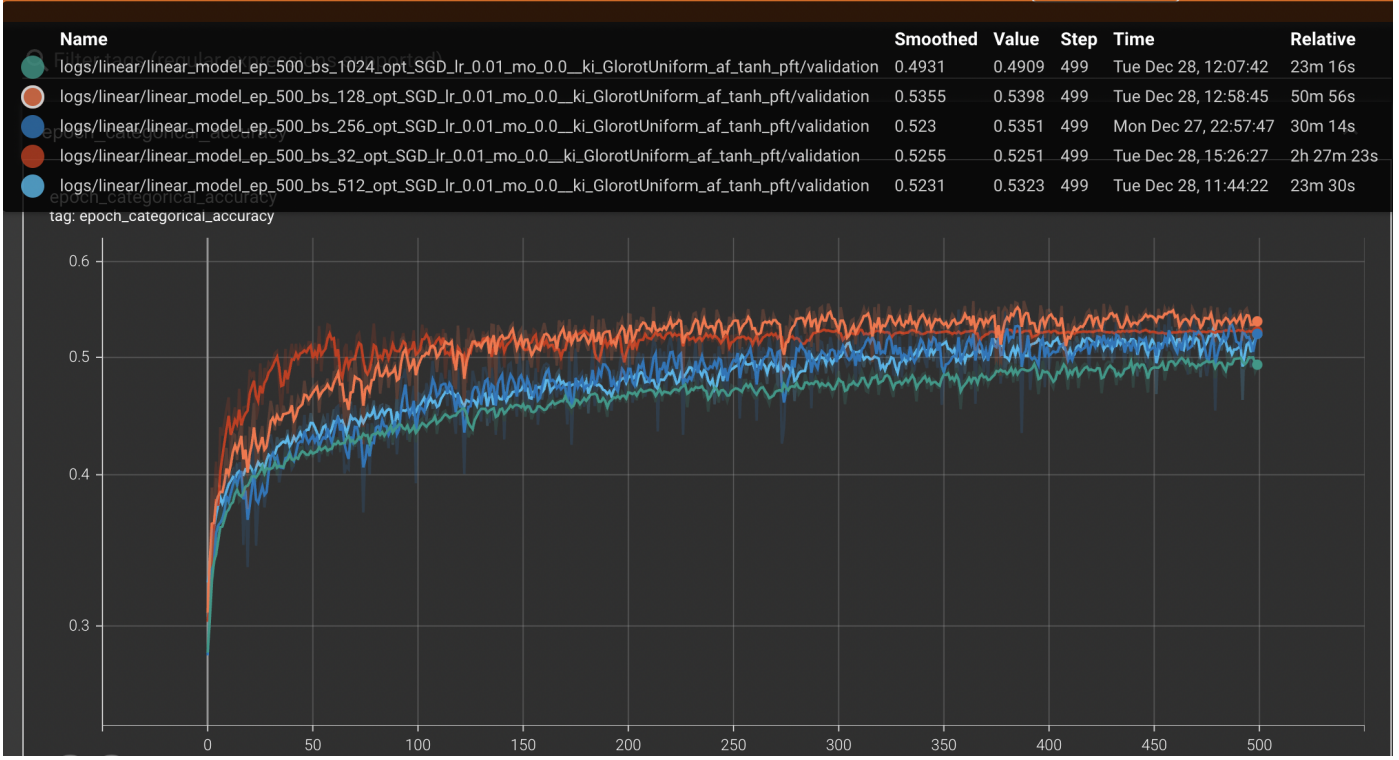
hyperparameters

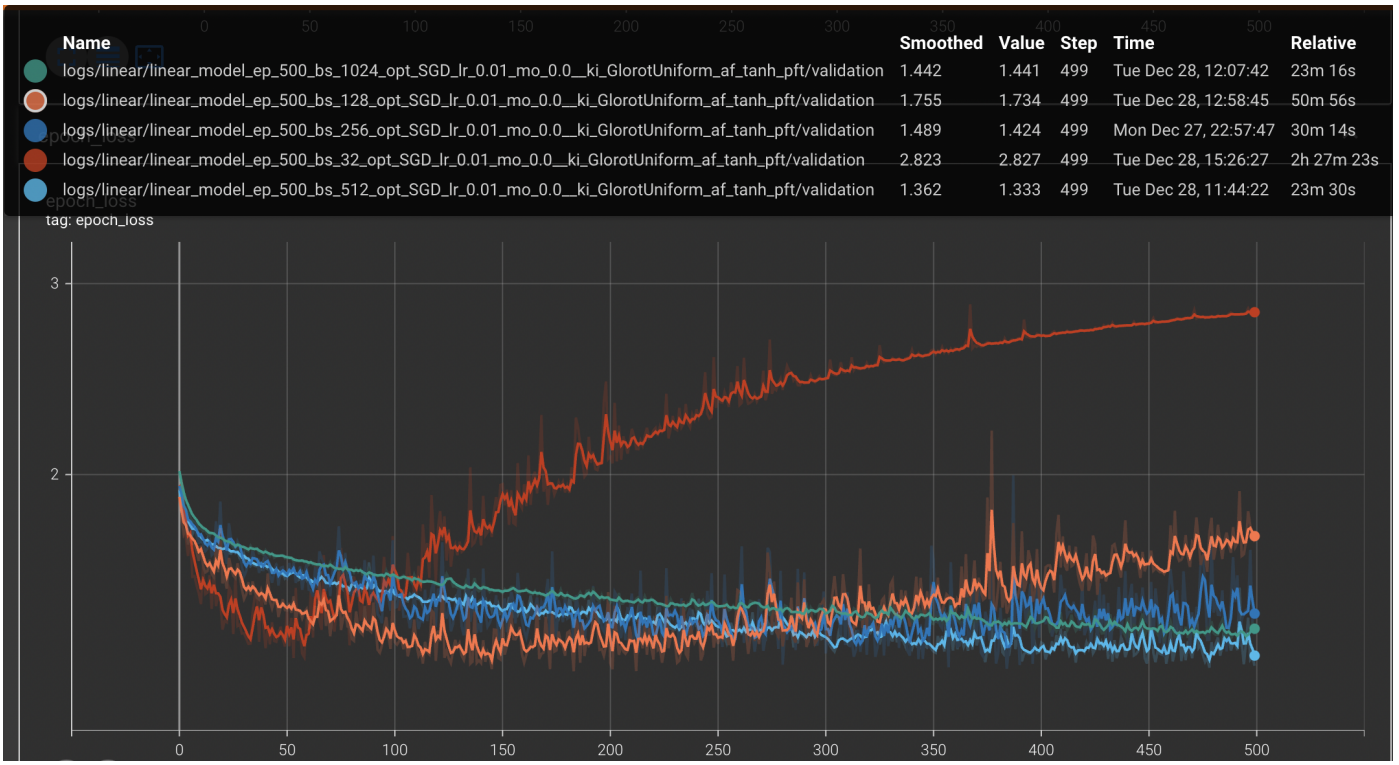
In [ ]:
```
BATCH_SIZE = [512,1024,128,32]
learning_rate = 0.01
momentum = 0.0
l2_val = 0.00
l1_val = 0.00
using_l1 = l1_val > 0
using_l2 = l2_val > 0
ki = glorot_uniform
fa = tanh
version = '_pft'
```

training

In [ ]:
```
for bs in BATCH_SIZE:
    perceptron = Perceptron(fa, ki, l1_val=l1_val, l2_val=l2_val)
    perceptron.compile(loss=categorical_crossentropy,
                  optimizer=SGD(learning_rate=learning_rate,
                                momentum=momentum),
                  metrics=categorical_accuracy)

    log_name = os.path.join(LOG_DIR,
                            "linear",
                            f"linear_model_ep_{EPOCHS}_bs_{bs}_opt_SGD_lr_{learning_rate}_

    print(log_name)
    perceptron.fit(x_train, y_train, batch_size=bs, epochs=EPOCHS, validation_data=(x_test
```

Resultat:





La modification de l'hyperparametre batch-size influe sur le temps d'apprentissage et la generalisation globale. On peut voir que la courbe rouge (bs à 32) a mis 2h27 min pour faire 500 epochs. La courbe orange(bs à 128) a mieux generalisé dans notre c'est peut-etre le batch size le plus adapté à notre probleme, Mais on peut voir que ces deux modeles sur-apprend contrairement aux autres modeles qui ont des plus grands batch-size.