Rapport de Projet - Kaggle Challenge - LANL Earthaquake Prediction

Matthieu Dagommer, Paul Boulgakoff, Germain L'Hostis, Godefroy Bichon

Introduction

L'objectif du projet est de développer un modèle capable de prédire, à partir d'un signal acoustique unidimensionnel, le temps qui sépare la fin du signal d'une secousse sismique.
Les données ont été récoltées par l'institut LANL (Los Alamos National Laboratory) à l'aide d'un montage permettant de générer de petits séismes artificiels.

Deux architectures de réseau de neurones ont été développées dans le cadre de ce projet, car elles nous sont apparues comme évidentes pour traiter des séquences: un réseau de convolution 1D et un réseau LSTM.

Ce rapport est divisé en 2 parties correspondant aux deux architectures étudiées, convolution 1D et LSTM. Elles se déclinent elles-même en 2 sous-parties:

- Une description du traitement des données préalable
- Une description progressive et méthodique de la constitution et du raffinement de l'architecture concernée

I. Réseau de Convolution 1D

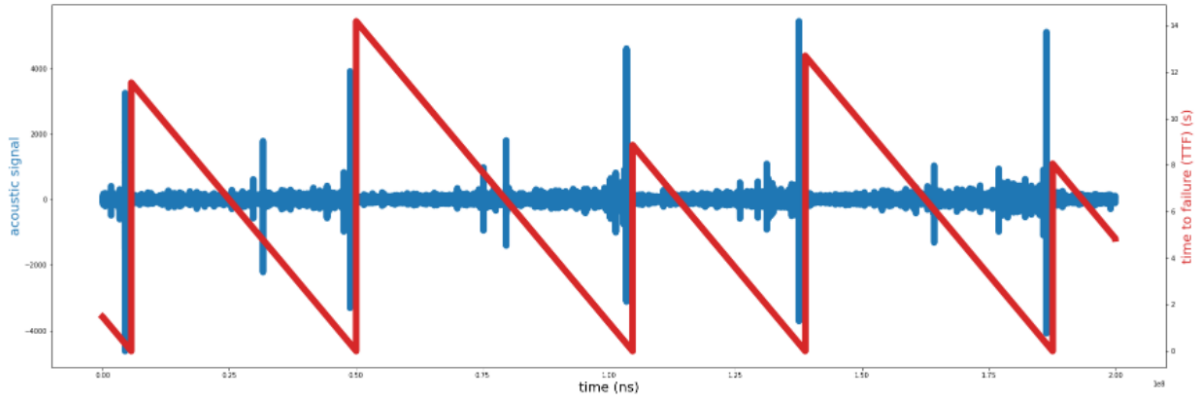
1. Traitement des données

Le jeu de données d'entraînement disponible contient une séquence longue de plus de 600 000 000 points. Chaque point correspond à un couple de scalaires: le signal acoustique au temps t, et la durée avant le prochain séisme au temps t (ou time-to-failure, que nous abrévierons TTF). Par souci de réaliser des expériences de durées raisonnables, nous n'avons pas exploité l'intégralité du jeu d'entraînement disponible. Nous nous sommes contenté, pour toutes les expériences décrites ci-dessous d'une fraction de 200 000 000 points.

Le jeu de données test est composé de séquences de longueur 150000. Chaque séquence est accompagné d'un seul scalaire qui correspond au TTF associé à la fin de la séquence.

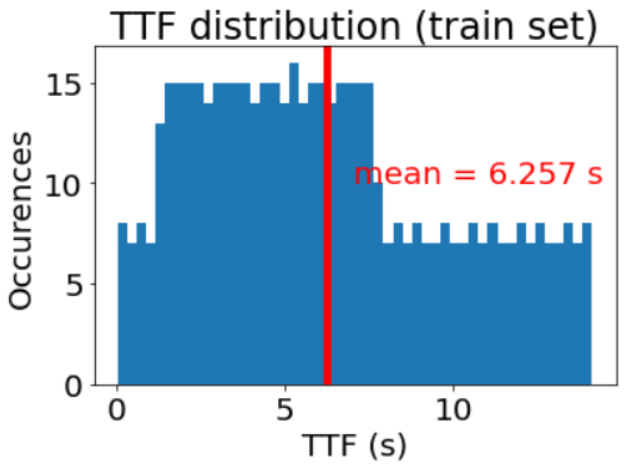
Afin d'entraîner le modèle à effectuer précisément la tâche à laquelle on le destine, il faut que le format des données d'entraînement et des données de validation (ou test) soient concordants. Nous avons pour cela partionné le long signal d'entraînement en plusieurs séquences (ou patchs) d'une longueur de 150000 identique à celle des séquences de test. Nous avons également fait en sorte d'implémenter un recouvrement partiel de ces séquences entre elles.

Toujours dans un souci de cohérence entre entraînement et utilisation, nous avons veillé à ce qu'aucune des séquences d'entraînement ne contienne de séismes, étant donné que les séquences de test n'en contiennent pas.



Remarque: Il est intéressant de noter qu'on peut déjà voir, à l'oeil nu, un schéma qui se répète avant chaque séisme: l'amplitude du signal semble augmenter de façon croissante avant le pic maximal correspondant au séisme. Lorsque le TTF atteint 0, un séisme se produit. D'ailleurs, comme le montre la superposition des deux courbes, le signal acoustique atteint un pic juste avant que le séisme ne se produise.

Histogramme des TTF :



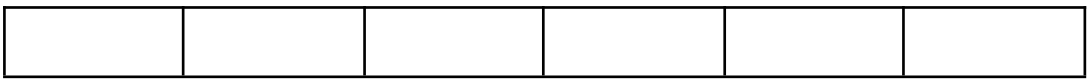
Normalisation des données

Il est possible de normaliser les données d'entraînement, et cela permet parfois d'améliorer la descente de gradient.
Comme les données X sont des signaux acoustiques variant autour de 0, et dont la distribution est gaussienne, nous les normalisons avec StandardScaler().
Les données Y sont des temps avant séismes, toujours supérieurs à 0. Nous les normalisons de façon classique avec MinMaxScaler().

Le patching

Le patching consiste à découper notre signal temporel d'entraînement en signaux de longueur plus courte pour pouvoir créer un ensemble de signaux qui vont nous servir à entraîner le modèle. Nous proposons 2 méthodes pour le patching :

- Le patching « strict » qui consiste à découper strictement le signal en des intervalles régulier qui sont continus lorsqu'ils sont collés les uns aux autres, il s'agit de la façon la plus simple d'imaginer la découpe du signal.



- Le patching avec « recouvrement » qui consiste à superposer les patchs entre eux, c'est-à-dire que le début d'un patch sera composé de la fin de celui qui le précède. Cela permettrait de garder une certaine continuité dans les données d'entraînement, qui ne se retrouve pas dans le cas d'un découpage strict du signal. La taille de ce recouvrement est définie.

Recouvrement

La stratégie de validation

Une partie des séquences obtenues par partition de la séquence d'entraînement a été utilisée comme jeu de validation. Ces séquences ont été tirées au hasard parmi les séquences d'entraînement disponibles après patching. L'objectif du jeu de validation est de régler les "hyperparamètres" (c'est-à-dire tous les paramètres pouvant être réglés manuellement, à l'exception des poids du réseau), donc de trancher parmi différentes déclinaisons d'un modèle celui dont les hyperparamètres permettent d'obtenir les meilleurs résultats de prédiction.

2. Architecture

On utilise comme modèle de départ le réseau de convolution 1D suivant :

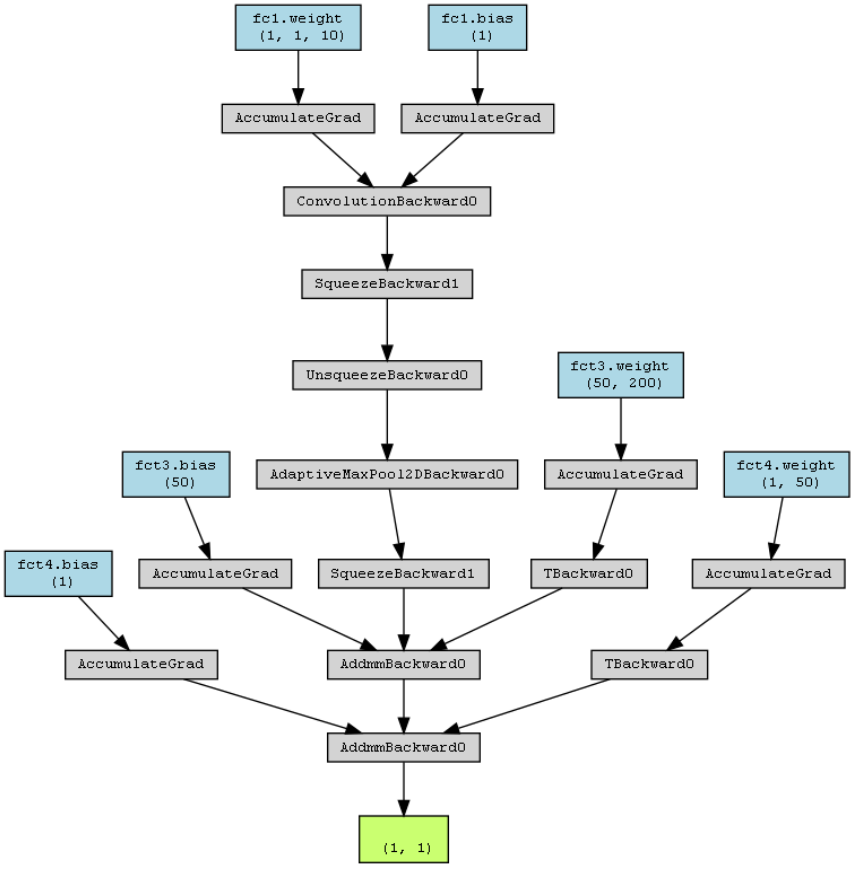
In [12]:
```python
### 1D Convolutional Network #1

class NN(nn.Module):

    def __init__ (self,num_classes=1):
        super(NN, self).__init__()
        self.fc1=nn.Conv1d(in_channels=1, out_channels=1,
                           kernel_size=10, padding=1, stride=5)
        self.fc2=nn.AdaptiveMaxPool1d(200)
        self.fct3=nn.Linear(200,50)
        self.fct4=nn.Linear(50,num_classes)

    def forward (self,x):
        x=self.fc1(x)
        x=self.fc2(x)
        x=self.fct3(x)
        x=self.fct4(x)
        return x
```

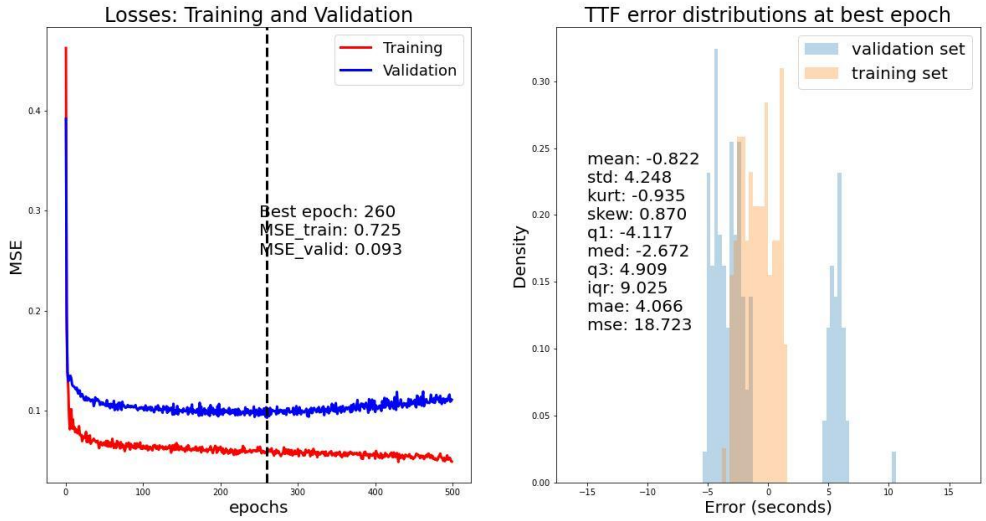
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv1d-1              [-1, 1, 29999]              11
 AdaptiveMaxPool1d-2                [-1, 1, 200]               0
            Linear-3                 [-1, 1, 50]          10,050
            Linear-4                  [-1, 1, 1]              51
================================================================
Total params: 10,112
Trainable params: 10,112
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 0.23
Params size (MB): 0.04
Estimated Total Size (MB): 0.84
----------------------------------------------------------------
```



Pour le plaisir des yeux, petite visualisation avec torchviz

Afin de réaliser des entraînements relativement courts, nous avons choisi de ne prélever que les 200 000 000 premiers points du jeu d'entraînement complet. Une comparaison des résultats obtenus avec davantage de données est exposée ci-dessous.

size_batch = 100 # nombres de patchs par batchs
valid_rate = 0.1 # proportion des données disponibles consacrée à la validation
overlap_rate = 0.02 # taux de recouvrement
model = NN()
learning_rate=0.0001
num_epochs=500



A gauche sont tracées les courbes d'apprentissage et à droite un histogramme des erreurs de validation. L'intérêt de l'histogramme est de caractériser les erreurs faites par le réseau, y déceler des tendances (Par exemple, est-ce que le modèle a tendance à sur-estimer ou à sous-estimer la sortie ?). L'histogramme a été normalisé sous la forme d'une densité afin de pouvoir comparer graphiquement les distributions d'erreur des jeux d'entraînement (training set) et de validation (validation set), dont les tailles en nombres d'échantillons sont asymétriques par ailleurs.

Plusieurs remarques :

- Sur les courbes d'apprentissage (gauche), on observe un overfitting: il y a un écart quasi-permanent entre les erreurs commises par le modèle sur les données de validation et celles commises sur les données d'entraînement. Cela peut vouloir dire plusieurs choses: soit le modèle est trop complexe, auquel cas il finit par apprendre par coeur le jeu d'entraînement et se généralise de moins en moins à des données externes. Soit la quantité de données d'entrainement est insuffisante pour que le modèle apprenne à être général.

- La distribution des erreurs sur le jeu de validation (droite) est très étonnante: la majorité des erreurs se concentre autour de -5 s et de 5 s. Après quelques

expériences, nous nous sommes aperçus que cette distribution dépendait de la source aléatoire en début de code, puisque la répartition des données entre le jeu d'entraînement et le jeu de validation se fait avec numpy.random.

Nous sommes à la fois dans une situation de surajustement et de sous-ajustement: le modèle se généralise mal aux données de validation, mais ce n'est peut-être qu'un effet de la faible quantité de données. En effet, il ne donne pas non plus une erreur d'entraînement parfaite ! On peut légitimement se demander si notre modèle actuel est suffisamment complexe pour traiter nos données dans l'absolu.

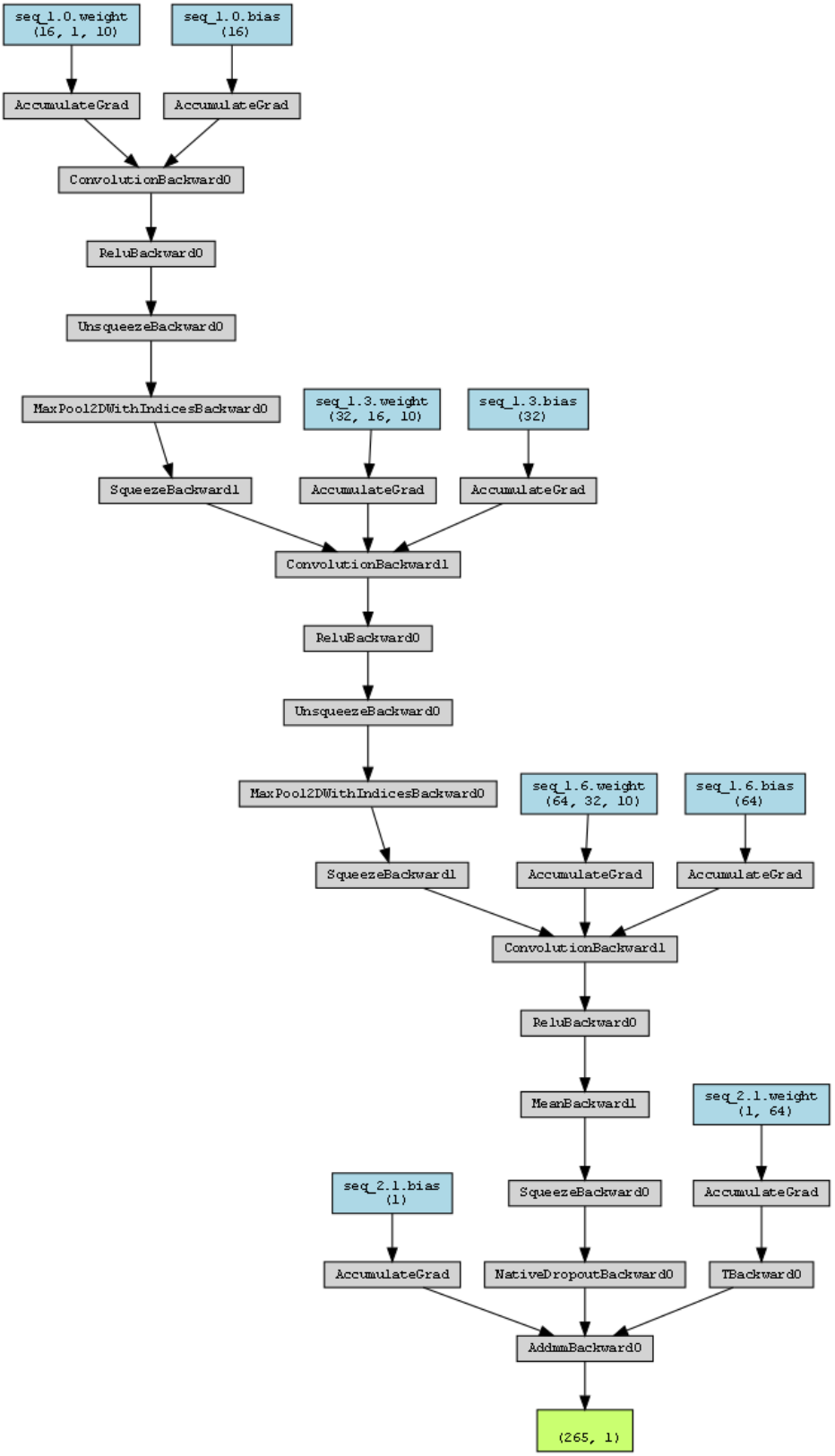
Pour pallier ce problème, nous avons mis au point un deuxième modèle en nous inspirant des source ci-dessous:
https://machinelearningmastery.com/cnn-models-for-human-activity-recognition-time-series-classification/
https://github.com/Gruschtel/1D-CNN/blob/master/1D_CNN_02.ipynb

In [14]:
```
### 1D Convolutional Network #2

class NN_2(nn.Module):
    def __init__(self,num_classes=1):
        super(NN_2, self).__init__()
        self.seq_1=nn.Sequential(nn.Conv1d(in_channels = 1, out_channels = 16, kernel_size = 10, stride = 5),
                                 nn.ReLU(),
                                 nn.MaxPool1d(kernel_size = 2),
                                 nn.Conv1d(in_channels = 16, out_channels = 32, kernel_size = 10, stride = 5, \
                                           padding = 'valid'),
                                 nn.ReLU(),
                                 nn.MaxPool1d(kernel_size = 2),
                                 nn.Conv1d(in_channels = 32, out_channels = 64, kernel_size = 10, stride = 5, \
                                           padding = 'valid'),
                                 nn.ReLU()
                                )

        self.seq_2=nn.Sequential(nn.Dropout(p = 0.4),
                                 nn.Linear(in_features = 64, out_features = 1)
                                )
    def forward(self,x):
        x=self.seq_1(x)
        x = th.mean(x, dim = 2, keepdim = True)
        x = th.squeeze(x)
        x=self.seq_2(x)
        return (x)
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv1d-1           [-1, 16, 29999]             176
              ReLU-2           [-1, 16, 29999]               0
         MaxPool1d-3           [-1, 16, 14999]               0
            Conv1d-4            [-1, 32, 2998]           5,152
              ReLU-5            [-1, 32, 2998]               0
         MaxPool1d-6            [-1, 32, 1499]               0
            Conv1d-7             [-1, 64, 298]          20,544
              ReLU-8             [-1, 64, 298]               0
           Dropout-9                  [-1, 64]               0
           Linear-10                   [-1, 1]              65
================================================================
Total params: 25,937
Trainable params: 25,937
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 11.28
Params size (MB): 0.10
Estimated Total Size (MB): 11.95
----------------------------------------------------------------
```

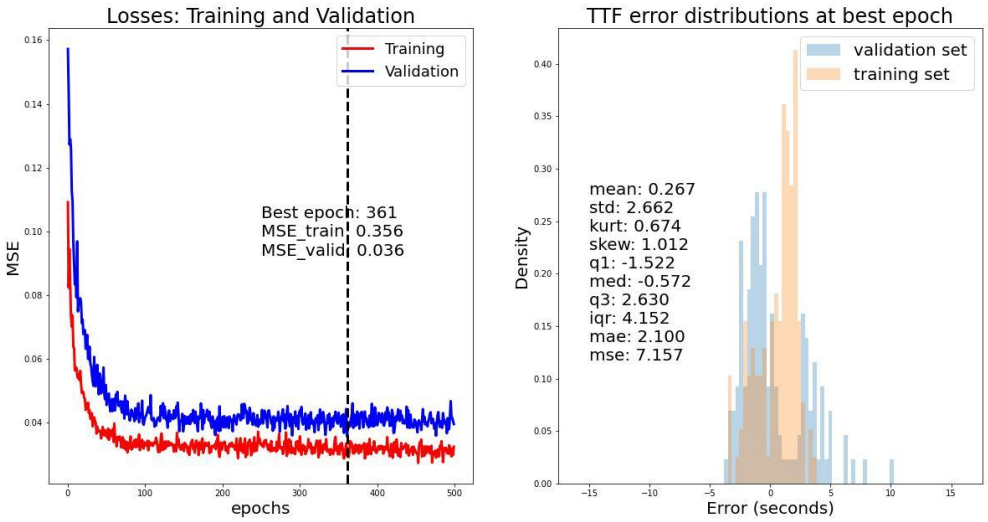


A nouveau, petit graphe sympathique.

Le nombre de paramètre est passé d'environ 10 000 à 25 000. La taille des batchs a dû être réduite, la mémoire du GPU étant devenue insuffisante pour stocker à la fois les paramètres du modèle et les calculs intermédiaires.

size_batch = 100 # nombres de patchs par batchs

valid_rate = 0.1 # proportion des données disponibles consacrée à la validation
overlap_rate = 0.02 # taux de recouvrement
model = NN_2()
learning_rate=0.0001
num_epochs=500



On a vu que les erreurs commises sur le jeu de validation se divisent très nettement en deux groupes dans le cas du modèle 1, et on sait que ce résultat est avant tout lié à l'échantillon de validation. En revanche le modèle 2, qui a été entraîné avec les mêmes jeux de données, semble considérablement effacer cette tendance, même si deux blocs restent facilement discernables (voir l'équart inter-quartile "iqr" ci-dessous).

Tableau comparatif:

	mean (s)	std (s)	kurt (s)	skew (s)	q1 (s)	med (s)	q3 (s)	iqr (s)	mae (s)	mse (s²)
cnn_1	-0.822	4.248	-0.935	0.870	-4.117	-2.672	4.909	9.025	4.066	18.723
cnn_2	0.267	2.662	0.674	1.012	-1.522	-0.572	2.63	4.152	2.100	7.157

mean: moyenne; std: écart-type; kurt: kurtosis; skew: skewness; q1: quantile(0.25); med: médiane; q3: quantile(0.75); iqr: équart interquartile; mae: mean absolute error; mse: mean square error

Avec des paramètres identiques, le modèle 2 réalise un meilleur score de prédiction sur le jeu de validation que le modèle 1. L'erreur moyenne absolue est de 4,066 s pour le modèle 1 contre 2,100 s pour le modèle 2. Les autres données statistiques disponibles sur la distribution des erreurs laissent peu de doute sur la supériorité du modèle 2: son écart interquartile est deux fois plus faible et la moyenne est plus proche de 0.

En toute rigueur, ces résultats doivent être confirmés avec le jeu de données test. Nous ne réalisons pas cette étape ici, pour nous concentrer directement sur le modèle 2 uniquement.

Influence du recouvrement

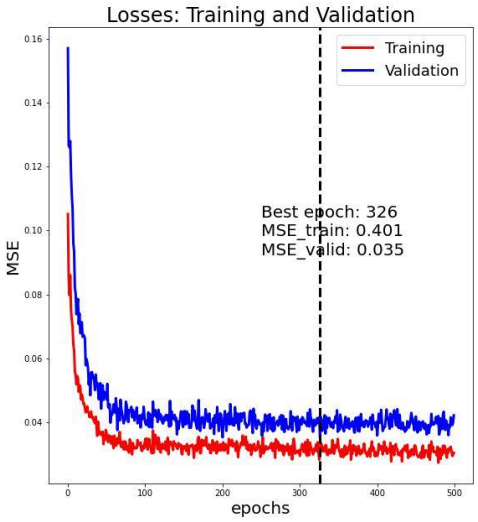
Le recouvrement (ou “overlap”) qui intervient lors du patching des séquences est un hyperparamètre dont nous avons souhaité connaître l’influence sur le modèle. Nous avons pour cela réalisé plusieurs entraînements du modèle 2 en faisant varier le recouvrement.

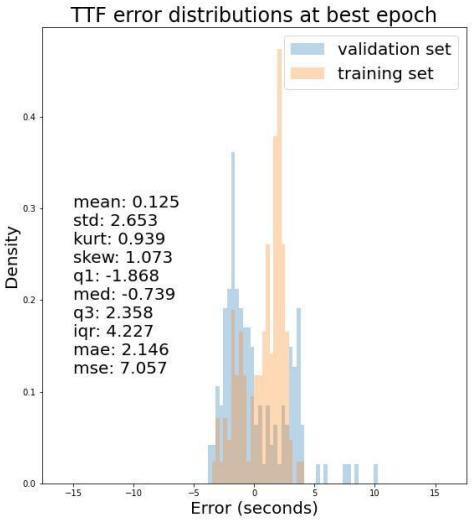
Plan d’expérience:

Entraînement. n°	1	2	3
Recouvrement (%)	2	10	20

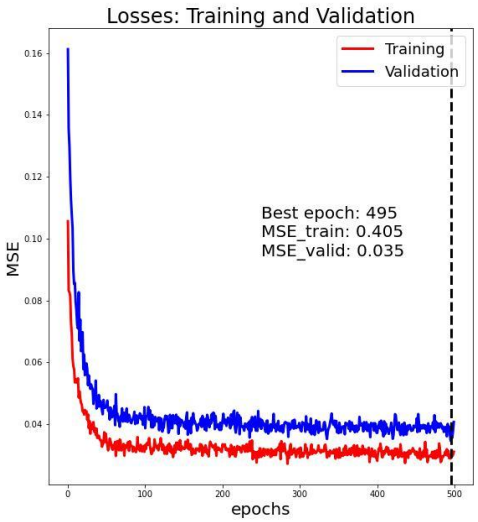
Le premier entraînement se trouve plus haut

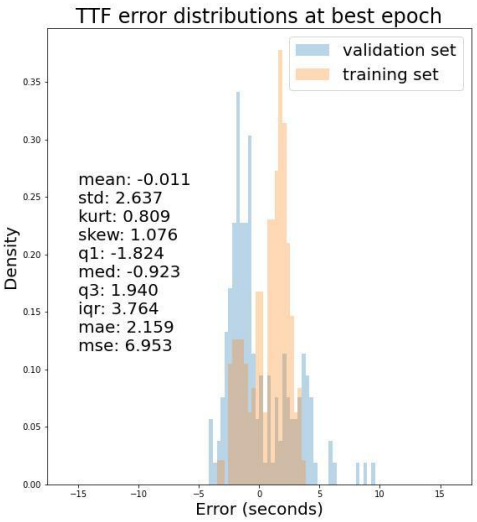
- Overlap = 10





- Overlap = 20





A vue d'oeil, le recouvrement ne semble pas exercer d'influence sur le modèle: l'erreur moyenne absolue stagne autour de 2,1 dans les trois cas. Il n'a pas été possible d'aller à des recouvrements supérieurs à 20 %, faute de mémoire dans le GPU.

Un problème vient du fait que dans l'implémentation actuelle, augmenter le recouvrement augmente la quantité de données utilisée pour l'entraînement. On pourrait imaginer faire l'expérience de fixer la taille des jeux d'entraînement pour isoler l'influence du recouvrement seul.

II. LSTM

Le LSTM est un type de réseau récurrent conçu pour traiter des séries temporelles.

1. Traitement des données

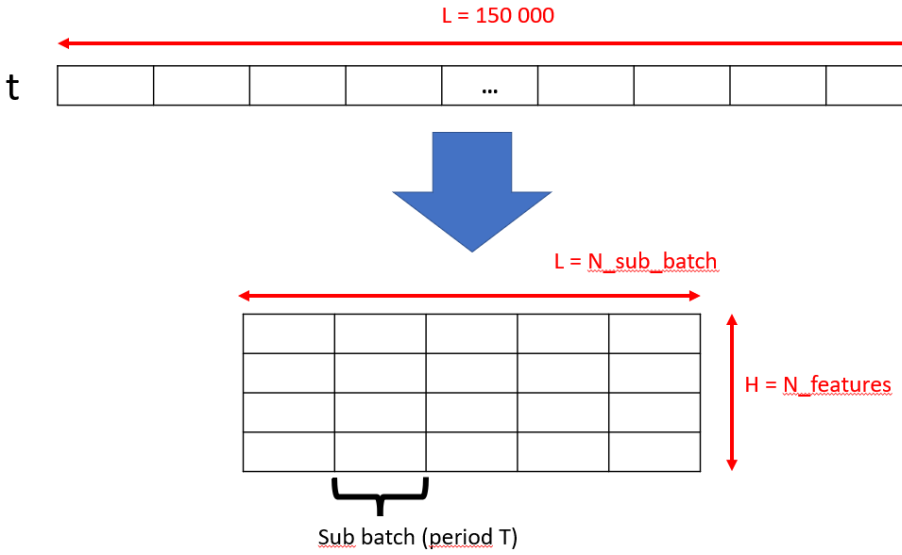
Contrairement au cas de la convolution 1D, nous avons envisagé deux traitements de données différents. Le premier consistait à reprendre les séries temporelles de taille 150,000 telles qu'elles étaient exploitées dans le réseau de convolution 1D.
Cependant, l'inconvénient majeur d'une telle approche est que le LSTM se retrouve à devoir calculer des gradients sur 150000 couches, ce qui est énorme et demande un temps de calcul considérable. Nous avons donc cherché à rétrécir le format des données d'entrée, et d'appliquer un premier prétraitement aux données d'entrée.

Nouveau traitement de données

Prenons la séquence de longueur 150,000. On la subdivise en sous-patchs (sub_patches dans le code) dont le nombre est un hyper-paramètre. On calcule pour chaque sous-patch des données statistiques élémentaires, avec entre autres :

- moyenne
- écart-type
- coefficient d'asymétrie (skewness)
- coefficient d'aplatissement (kurtosis)
- quantiles

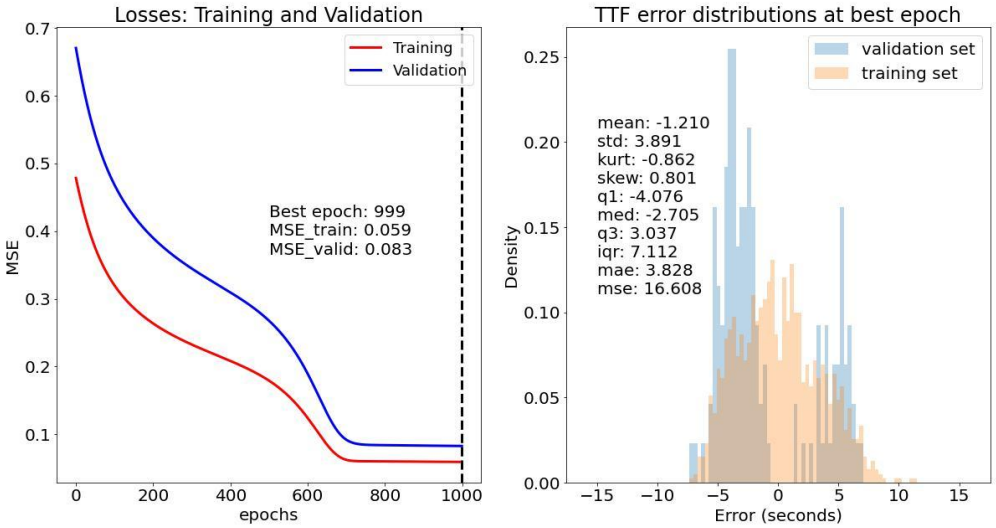
Au final, un échantillon n'est plus une séquence unidimensionnelle mais une séquence multidimensionnelle. Cette nouvelle séquence est une matrice [# sous-patchs * # params statistiques]



Entraînements avec le nouveau traitement de données:

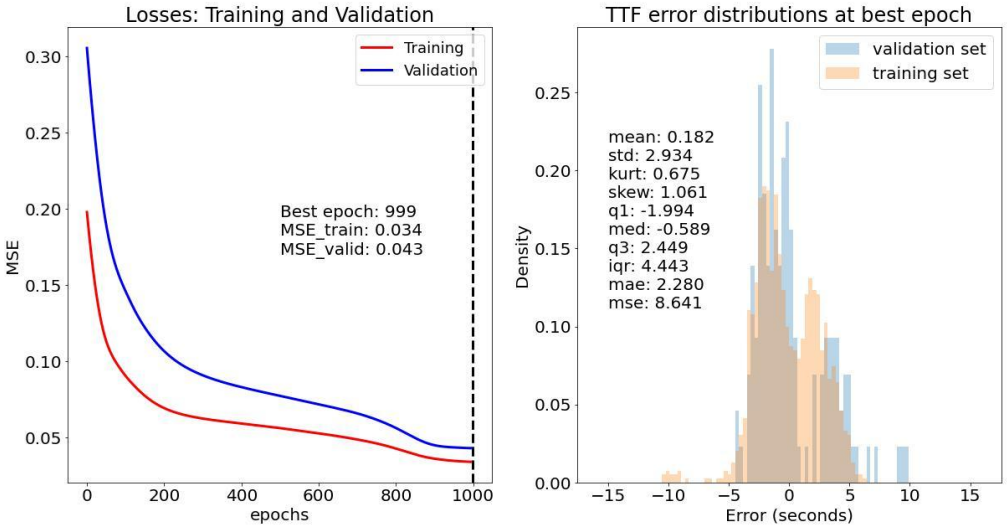
Nous avons entraîné le modèle avec deux sources aléatoires différentes, 0 et 1:

```
seed = 0
th.manual_seed(seed)
np.random.seed(seed)
size_batch = 100 # nombres de patchs par batchs
valid_rate = 0.1 # proportion des données disponibles consacrée à la validation
overlap_rate = 0.02 # taux de recouvrement
num_epochs = 1000 #1000 epochs
learning_rate = 0.001 #0.001 lr
hidden_size = 1 #number of features in hidden state
num_layers = 1 #number of stacked lstm layers
N_sub_patches = 250
```



De manière étonnante, on retrouve la même forme de distribution d'erreur en validation même en changeant la source (seed = 1), avec deux pôles :

```
seed = 1
th.manual_seed(seed)
np.random.seed(seed)
size_batch = 100 # nombres de patchs par batchs
valid_rate = 0.1 # proportion des données disponibles consacrée à la validation
overlap_rate = 0.02 # taux de recouvrement
num_epochs = 1000 #1000 epochs
learning_rate = 0.001 #0.001 lr
hidden_size = 1 #number of features in hidden state
num_layers = 1 #number of stacked lstm layers
N_sub_patches = 250
```



On peut s'apercevoir que l'erreur moyenne absolue évaluée sur l'échantillon de validation peut varier de façon relativement importante lorsqu'on passe d'une source à une autre. Dans ce cas précis, la source change à la fois l'initialisation des paramètres du modèle et le jeu de validation, on ne peut donc pas séparer ces deux variables ici pour décider ce qui a impacté le plus l'erreur de validation. En revanche, on s'aperçoit également que la MSE à la meilleure époque a diminué remarquablement au second entraînement, passant de 0.059 à 0.034. Ces résultats suggèrent une grande sensibilité de la convergence du modèle à l'initialisation.

Conclusion

Ce projet a été l'occasion de prendre en main des notions fondamentales de l'apprentissage profond, tout en nous permettant de nous familiariser avec un large éventail de librairies Python fréquemment utilisés en Data Science (pandas, scikit-learn, pytorch). Nous avons également appris à créer un environnement de travail avec conda et à exploiter la puissance de calcul de nos cartes graphiques via cuda.

Versions utilisées:

python 3.10.4
torch 1.11