# Machine Learning

Martin Guyard et Guillaume Brizolier

TP2 : Renewable Energy Prediction

## 1 Preprocessing des données

### 1.1 Problèmes éventuels dans les données

1. Redondance : cela peut éventuellement ralentir les calculs mais ce n'est pas un problème en soi.
2. Normalisation : Il faut toujours normaliser pour des problèmes informatiques ! On divise chaque colonne par la valeur max et on soustrait la moyenne pour centrer les données.
3. Indépendance : fortement lié à la redondance. Dans notre cas, ce n'est pas un problème.
4. Format : Utilisation de one-hot vector pour convertir les chaînes ASCII en un nombre de colonnes égales au nombre de villes. Il s'agit d'un vecteur où toutes les valeurs sont nulles sauf 1 à 0 qui correspond à l'indicateur de catégorie.

   $$0 \mid 0 \mid 0 \mid 1 \mid 0 \mid 0$$

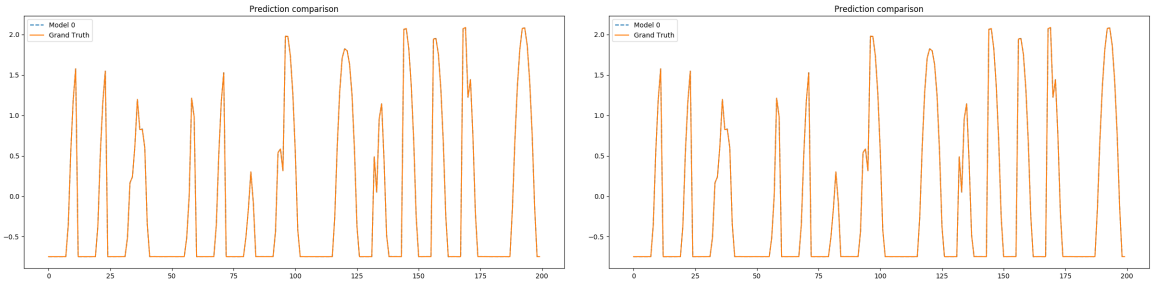
   On utilise la même technique pour la date avec cette fois-ci 3 one-hot vectors : un de 12 cases pour les mois, un de 31 pour les jours, et un de 24 pour les heures.
5. Trous : Si on a des trous, on peut soit faire des moyennes pondérées en fonction de points suivants et précédents, ou tout simplement remplacée par la moyenne (nulle ici car nous avons normalisé les données).

### 1.2 Input

[9, Nbcol, 1] : on a un seul neurone qui est un réel. Output : [batch, 1] Dans notre couche finale, on fait une combinaison linéaire qui sort un nombre sans mettre de fonction d'activation, puisqu'on veut des valeurs dans R.
Au milieu : on met une couche dense fully connected, et ça marche. Les fonctions d'activation au sein du réseau sont celles qu'on veut, mais par contre à celle de la fin on n'en met pas.

## 2 Prévision

target : une ligne!
on peut utiliser une fenêtre de 4 lignes. Input : [9 : nb de lignes, 4 : nb de dimensions tempo que je veux, NbCol : nbre de features] : b(batch), t(time), c(colonnes)
Je ne peux prédire qu'une valeur ! (pas toute la ligne). Du coup la sortie ne change pas.




(a) Comparaison 1 (b) Comparaison 2

Figure 1: Données de test (orange solide) vs données prédites par le modèle (bleu pointillé)

# 3 Modèles utilisés

## 3.1 Modèle régressif

Ce problème de prédiction pouvant s'assimiler à un problème de régression, nous avons choisi d'utiliser un SVR (*Support Vector Regression*), importé depuis le module *sklearn*.

```
    from sklearn.svm import SVR

    def model_no_ann(name, data, idx, target):
""" Train a model on train + valid data set
# Argument
    :param name: str, name
    :param data: numpy array, data
    :param idx: dict, data sets indexes
    :param target: int, position of target
"""
    fn = join(model_path, name)
    dt = np.concatenate((data[idx['train'], :target], data[idx['train'], target + 1:]), axis=-1)
    dv = np.concatenate((data[idx['valid'], :target], data[idx['valid'], target + 1:]), axis=-1)
    dtv = np.concatenate((dt, dv))
    ltv = np.concatenate((data[idx['train'], target], data[idx['valid'], target]))
    model = SVR(epsilon=0.001)
    model.fit(dtv, ltv)
    '''
    === Put some code here ===
    '''
    with open(fn, 'wb') as f:
        pidump(model, f)
    dtest = np.concatenate((data[idx['test'], :target], data[idx['test'], target + 1:]), axis=-1)
    graph_comparison([model.predict(dtest)], data, idx, target, 1, 0, t_idx='test', step=200)
```

Ici, le kernel utilisé est 'rbf' (paramètre par défaut) car les données ne sont pas linéairement séparables. On garde tous les autres paramètres à leur valeur par défaut, mis à part epsilon qu'on choisit à 0.001 pour un maximum de précision. Comme le montre les courbes de prédictions suivantes, le modèle prédit parfaitement les données de test : Nous obtenons donc des performances très satisfaisantes pour ce modèle.

### 3.2 Modèle Réseau de Neurones

Nous avons choisi une architecture relativement simple pour notre réseau de neurones. Nous avons choisi de faire un ANN fully-connected, avec 3 couches dense : la première avec 500 neurones, la deuxième avec 100, la troisième avec 10, complétée par une dernière couche d'une seule neurone sans fonction d'activation pour avoir le résultat final de la prédiction.

Le code correspondant, écrit avec l'API fonctionnelle de Keras, est le suivant :

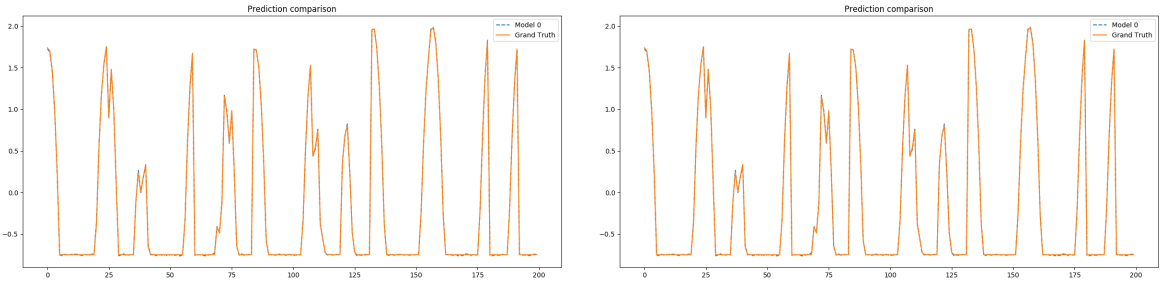
```
def create_model(w, c):
""" Create a keras model
# Arguments
    :param w: int, time dimension
    :param c: int, channel dimension
# Returns
    :return: keras model
"""
l_in = Input(shape=(w, c,))

l_hidden_0 = Dense(500, activation='relu')(l_in)
l_hidden_1 = Dense(100, activation='relu')(l_hidden_0)
l_hidden_2 = Dense(10, activation='relu')(l_hidden_1)
l_hidden_3 = Dense(1)(l_hidden_2)
l_out = Flatten()(l_hidden_3)

return Model(l_in, l_out)
```

On remarque qu'on utilise un layer Flatten au début. Cela est en fait dû aux différences de dimensions de l'input du modèle, qui doit correspondre aux dimensions données par les fonctions de générations de données, à savoir [batch, fenêtre, canaux], tandis que l'output lui correspond simplement à la dimension du batch.

Les résultats pour ce modèle sont un peu moins convaincants, pour une raison que nous ne sommes pas arrivés à déterminer le modèle ne semble pas converger vers une valeur de loss satisfaisante. Voici les résultats que nous obtenons :




(a) caption of first image

(b) caption of second image
second line

Figure 2: Données de validation (orange solide) vs données prédites par le modèle (bleu pointillé)

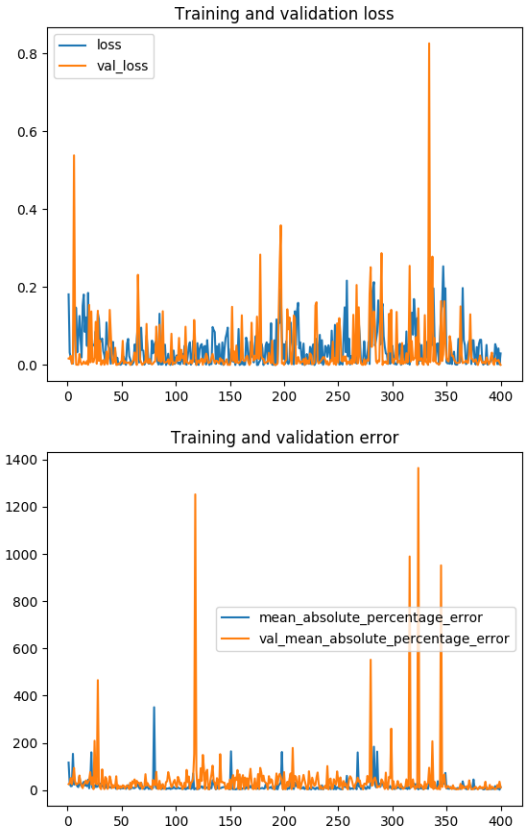



Figure 3: Loss et Mean Squared Error en fonction du nombre de batch