AVIGNON UNIVERSITÉ

Predicting secondary structures of proteins

Abdou NIANG

14 avril 2023

Master Informatique
Intelligence Artificielle

UE APPRENTISSAGE SUPERVISE
ECUE Application IA

Responsable
Yannick Estèvel

UFR
SCIENCES
TECHNOLOGIES
SANTÉ

AVIGNON UNIVERSITÉ
CERI

CENTRE
D'**ENSEIGNEMENT**
ET DE **RECHERCHE**
EN **INFORMATIQUE**
ceri.univ-avignon.fr

Sommaire

1 Introduction

Ce TP a comme objectif de prédire la structure scondaire d'une proteine. Pour ce faire nous avons utilisé differents algorithmes basés sur l'apprentissage automatique.
Ces algorithmes utilisent des ensembles de données de protéines dont les structures ont été déterminées et dont les séquences d'acides aminés sont connues. L'algorithme est entraîné sur ces données pour apprendre les motifs qui sont associés à chaque type de structure secondaire (alpha-helix, beta-sheet, coil).

2 Données

C'était la partie la plus importante et qui m'avais pris beaucoup plus de temps.
Les données ont été transfomés en One Hot, qui est essentielle pour permettre à ces algorithmes de traiter les données catégorielles de manière efficace [1].

```
ac_am_dict = {
    "1" :[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "A": [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "C": [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "D": [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "E": [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "F": [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "G": [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "H": [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "I": [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "K": [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "L": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "M": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "N": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    "P": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    "Q": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    "R": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    "S": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    "T": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    "V": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    "W": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
    "Y": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],

}
```

Figure 1. One Hot avec dictionnaire

3 Approches

Nous avons testé deux approches :

3.1 Séquence de même taille : Padding

Nous avons constaté que nos séquences de données étaient de longueurs variables. Dans ce contexte, le padding est utilisé pour ajouter des zéros à la fin des séquences plus

courtes pour les rendre de la même longueur que la plus longue séquence dans le jeu de données (498 valeurs) 2.
Nous avons utilisé une fonction prédifinie du tensorflow pour transformer les acides aminés

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```
x_train_padded = pad_sequences(x_train_int, padding='post')
y_train_padded = pad_sequences(y_train_int, padding='post')
x_test_padded = pad_sequences(x_test_int, padding= 'post')
y_test_padded = pad_sequences(y_test_int, padding= 'post')
```

Figure 2. Padding aux Séquences de données

en One Hot 3.

```
x_train = torch.from_numpy(tf.one_hot(x_train_padded, depth =21).numpy())
y_train = torch.from_numpy(tf.one_hot(y_train_padded, depth =4).numpy())
x_test = torch.from_numpy(tf.one_hot(x_test_padded, depth =21).numpy())
y_test = torch.from_numpy(tf.one_hot(y_test_padded, depth =4).numpy())
```

Figure 3. One Hot avec tensorflow

3.2 Fenêtre de cinq acides aminés

L'objectif de cette approche est de construire une séquence de 5 éléments centrée sur l'élément en question.
Les acides animés ont été transformé en One hot avec un dictionnaire.

```
def transformAcideAminToSerieFive(data): #La fonction transforme chaque séquence en une série de 5 acides aminés consécutifs,
                                    #en prenant en compte les acides aminés précédents et suivants
    compteur=0
    array=[]
    Finale = []
    for row in data:
        array=[]
        id=0
        compteur=compteur+(len(row))
        #pour chaque acide aminé la fonction vérifie si l'acide aminé est le premier ou le dernier de la séquence,
        #et ajoute l'acide aminé correspondant dans array
        for id, val in enumerate(row):
            if id==0  :
                #verifier first acide
                array.append(ac_am_dict["1"])
                array.append(ac_am_dict["1"])
                array.append(ac_am_dict[val])

            elif id==len(row)-1 :
                #vérifier last acide
```

```
            array.append(ac_am_dict[val])
            array.append(ac_am_dict["1"])
            array.append(ac_am_dict["1"])
        else :
            array.append(ac_am_dict[val])
    #parcourt chaque élément de array et construit une séquence de 5 acides aminés
        consécutifs

    for idx, val in enumerate(array):
        element=[]
        if (idx+1 < len(array) and idx - 1 >= 0 and idx+2 < len(array) and idx - 2
            >= 0 ):
            #construction des séquence de 5
            avant2=idx - 2
            avant1=idx - 1
            cur=idx
            apres1=idx+1
            apres2=idx+2
            element =
                [*array[avant2],*array[avant1],*array[cur],*array[apres2],*array[apres1]]
            print(element)
            Finale.append(element)#et j'ajoute à la liste finale

print('nombre d\'acide aminé',compteur)


return Finale
```

4 Choix du réseau de neurone

Nous avons choisit d'implémenter le MLP et le GRU pour entrainer et tester nos données. L'Early Stopping a été appliqué sur chaque modèle lors de l'entrainement du reuseau de neurone.

4.1 MLP

Puisqu'on n'a pas des données tres larges, nous avons choisit de mettre un réseau à 3 couches :

```
class NeuralNetwork(nn.Module):
 def __init__(self):

    super(NeuralNetwork, self).__init__()
    self.flatten = nn.Flatten()
    self.linear_relu_stack = nn.Sequential(
        nn.Linear(22*5, 128),
        nn.ReLU(),
        nn.Linear(128, 64),
        nn.ReLU(),
        nn.Linear(64, 3),

    )
    self.drops = nn.Dropout(0.0)
 def forward(self, x):
    x = self.flatten(x)
    x=self.drops(x)
    logits = self.linear_relu_stack(x)
```

```
        out = F.softmax(logits, dim=1)
        return out
```

La méthode **–init–** définit les couches de la réseau de neurones, en utilisant les modules PyTorch **nn.Linear** pour les couches connectée et **nn.Sequential** pour les combiner en une séquence. Les couches sont activées par la fonction d'activation **ReLU** à l'exception de la dernière couche qui est activée par la fonction **softmax**. En outre, un objet de la classe **nn.Flatten** est créé pour aplatir l'entrée du réseau de neurones.

La méthode **forward** définit la propagation avant du réseau de neurones. Les données d'entrée sont aplaties par **self.flatten**, puis traitées par **self.linear-relu-stack** et enfin activées par la fonction **softmax** pour produire une sortie.

Il est également à noter qu'un objet nn.Dropout est créé dans la méthode –init–, mais celui-ci est défini avec une probabilité de dropout de 0.0, donc cela n'aura pas d'effet sur les sorties.

4.2 GRU

Nous avaons utilisé un GRU à deux Couches :

```
class NeuralNetwork(nn.Module):

 def __init__(self, input_size, hidden_size, num_layers):

     super(NeuralNetwork, self).__init__()
     self.flatten = nn.Flatten()
     self.num_layers = 2
     self.hidden_size = 16
     self.gru = nn.GRU(input_size, 17, num_layers*2,
         batch_first=True,bidirectional=True)
     self.fc = nn.Linear(34, 4)
     self.relu = nn.ReLU()
 def forward(self, x,h0):
     out, o = self.gru(x, h0)
     a = nn.Sequential(
        self.relu,
        self.fc)
     out = a(out)
     return out , o

 def init_hidden(self, batch_size):

     hidden = torch.zeros(self.num_layers, batch_size, self.hidden_size)
```

La méthode **–init–** définit les couches du réseau de neurones. Elle prend en entrée la taille de l'entrée, la taille cachée, et le nombre de couches. L'objet **nn.Flatten** est créé pour aplatir l'entrée du réseau de neurones. L'objet **nn.GRU** est créé avec les paramètres spécifiés pour construire une couche GRU bidirectionnelle avec input-size dimensions en entrée, 17 dimensions cachées, num-layers*2 couches, et une disposition des données par lots (batch) avec batch-first=True. L'objet **nn.Linear** est créé avec 34 dimensions en entrée et 4 dimensions en sortie.

La méthode **forward** définit la propagation avant du réseau de neurones. Les données d'entrée x et l'état caché initial h0 sont passés à la couche GRU **self.gru**, qui renvoie une sortie out et un nouvel état caché o. La sortie out est ensuite traitée par une séquence de la fonction d'activation **ReLU** et la couche linéaire **self.fc**, en utilisant la méthode **nn.Sequential**. La sortie finale est la sortie de la couche linéaire out et l'état caché o.

La méthode init-hidden initialise l'état caché du réseau de neurones. Elle prend en entrée la taille des lots batch-size et renvoie un tenseur de zéros avec la forme appropriée pour

l'état caché initial.

5 Résultats

Ci-dessous le tableau comparatif entre le modèle MLP et le modèle GRU 1. Les meilleurs

Modèle	Accuracy Max	Loss	optimizer	learning rate
MPL	60.94	0.1676 (MSELoss)	Adam	lr=0.0001
GRU	64.4	0.062 (MSELoss)	Adam	lr=0.001

Table 1. Tableau de comparaison entre le modèle MLP et le modèle GRU

résultats obtenus avec le GRU sont dus au fait que ce modèle est récurrent grâce à sa mémoire interne, il se souvient d'éléments importants sur les données qu'il a reçues et permet d'avoir des informations sur ses voisins, et du fait que ce dernier utilise moins de paramètres que son homologue lstm ce qui le rend plus rapide. Les résultats obtenus avec l'ensemble de validation sont médiocres, ce qui est tout à fait normal puisque les données d'entraînement ont déjà été réduites de 20% et de base elles sont déjà peu nombreuses.

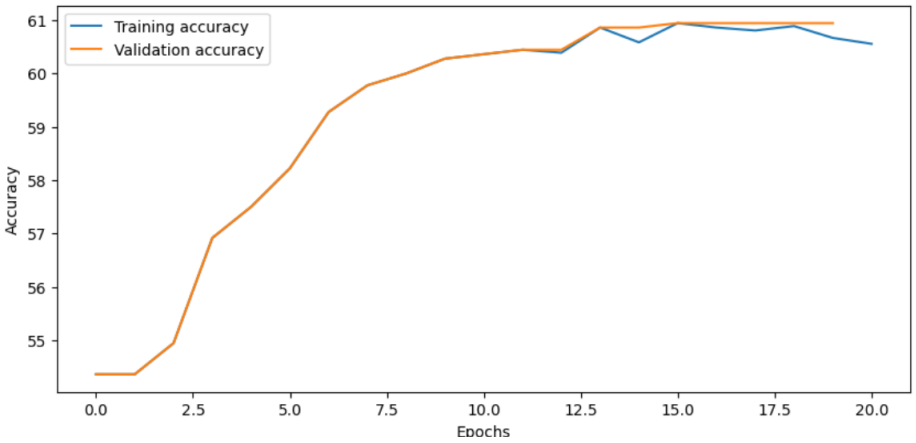

Figure 4. Accurency avec le modèle MLP

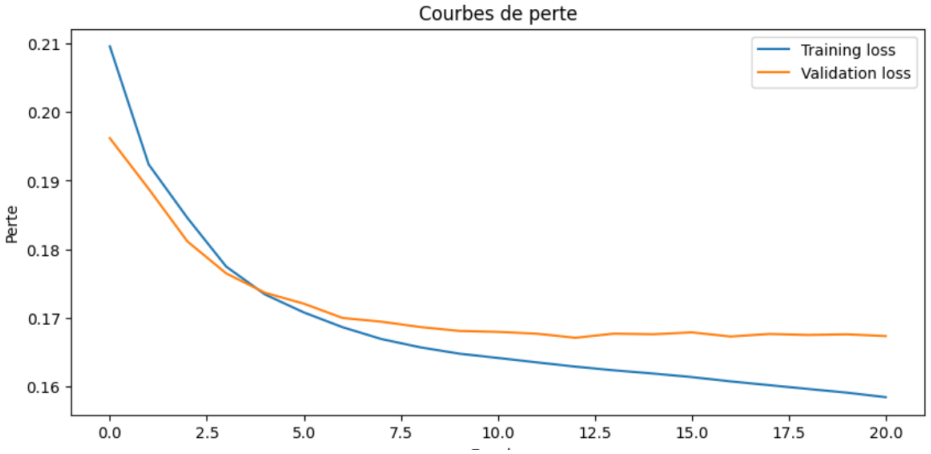

Figure 5. Loss avec le modèle MLP

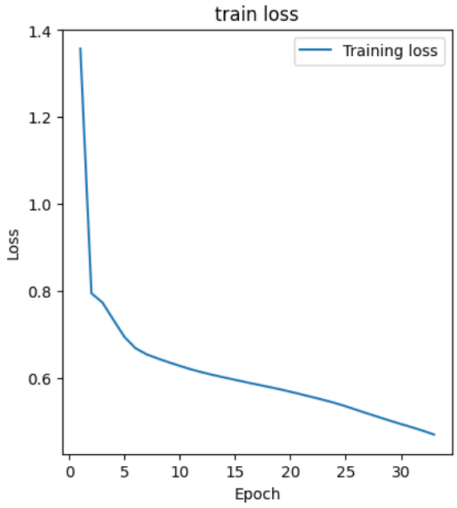

Figure 6. Loss Train avec le modèle GRU

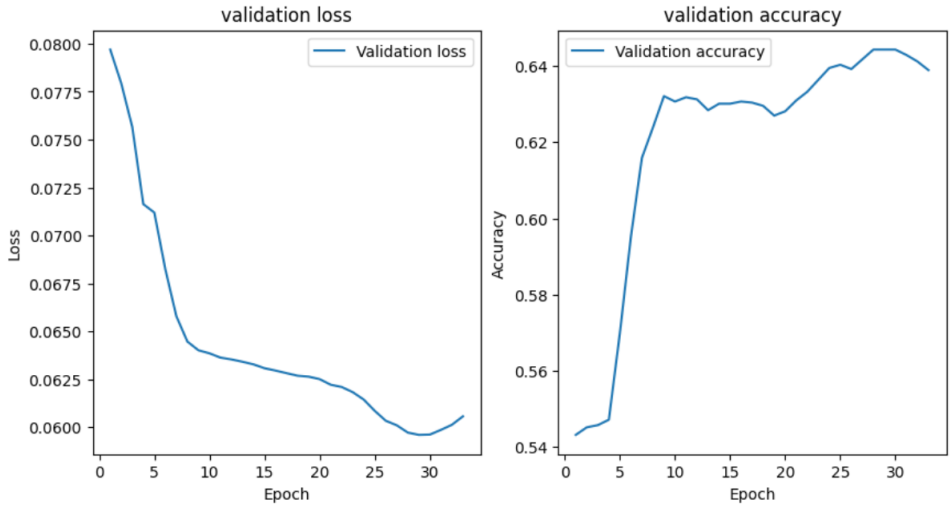

Figure 7. Loss et Accuracy test avec le modèle MLP

6 Conclusion

Ce projet nous a permis de mettre en pratique, de tester différents modèles d'apprentissage automatique et de mieux comprendre le Réseau de Neurone Récursif, RNN à savoir le GRU, qui permet de mieux contrôler la mémoire à court et long terme du réseau.