Université de Strasbourg

Intelligence artificielle

Licence 3 - UFR Mathématique - Informatique

TP – Classification d'espèces d'iris avec un réseau de neurones artificiel
Partie 2

Le but de ce TP est de travailler à la classification d'espèces d'Iris en utilisant un réseau de neurones.
Objectifs :
- Observer les données, comprendre leur nature et comment les adapter (si besoin) pour les utiliser avec le modèle d'apprentissage réseau de neurones
- Comprendre le fonctionnement d'un réseau de neurones pour le mettre en œuvre dans un programme

Pour faire ce TP vous devez avoir terminé le TP précédent

1 Rétro-propagation de l'erreur

Attention, les formules données ici pour le calcul de l'erreur sur la couche de sorties supposent que l'on utilise un softmax et une mesure d'entropie croisée (si ça n'est pas le cas, il faut que vous dériviez les équations selon les fonctions que vous avez choisi)

Pour la passe arrière, on va cette fois démarrer de la fin. Gardez en tête que le principe de la rétro-propagation de l'erreur est de distribuer l'erreur en sortie (qui est mesurable car nous pouvons comparer les prédictions aux sorties attentudes) dans les couches cachées (pour lesquelles nous n'avons pas de valeurs cibles).

Nous allons mettre en œuvre la fonction `def __backward_pass(self, X, y)`. N'oubliez pas que nous sommes en mode "online", c'est-à-dire que l'on regarde les instances une par une (pour une instance X_i : calcul de la prédiction associée, calcul de l'erreur pour cette prédiction et mise à jour des paramètres ; à répéter pour chaque instance du jeu de données).

Légère modification de votre fonction de propagation avant. Comme on l'a vu au TP précédent, les fonctions d'activation du module `Utility` renvoie aussi la matrice des dérivées de la fonction. Ajustez votre code initial pour enregistrer dans votre classe `NeuralNet` ces dérivées (il vous faut une variable d'instance définie comme suit dans votre constructeur `self.df = [None]*(self.n_layers+1)`) et, dans votre passe avant, récupérer aussi le second membre du tuple renvoyé par vos fonctions d'activations (`self.A[l],self.df[l] = self.activation(self.Z[l])`)

On va avoir besoin de se souvenir des quantités d'ajustement à faire sur nos paramètres : car la mise à jour des paramètre doit se faire après avoir rétro-propagé l'erreur sur toutes les couches. On propose donc de commencer par définir, dans notre fonction, trois listes qui contiendront les erreurs et ajustements à faire :

```
delta = [None] * (self.n_layers + 1)
dW    = [None] * (self.n_layers + 1)
db    = [None] * (self.n_layers + 1)
```

En suite, la marche à suivre est la suivante :

```
delta[L] <- A[L] - y # valable que pour softmax & entropie croisée!
dW[L]    <- delta[L] * A[L-1].transpose
db[L]    <- delta[L]

Pour chaque couche cachée l de L-1 à 0 faire
  # Hadamard: produit de matrices composante par composante.
  #  En python: multiply
  #  https://docs.scipy.org/doc/numpy/reference/generated/numpy.multiply.html
  delta[l] <- hadarmard(W[l+1].transpose * delta[l+1] , df[l])
  Si l == 1 alors # A[l-1] correspond aux _entrées_ du réseau
    dW[l] <- delta[l] * Xi.transpose
  Sinon
    dW[l] <- delta[l] * A[l-1].transpose
  db[l] <- delta[l]
fpour

# Mise à jour des paramètres
Pour chaque couche l du réseau faire
  W[l] <- W[l] - eta*dW[l]
  b[l] <- b[l] - eta*db[l]
fpour
```

Testez une passe avant et arrière pour une instance du jeu de données.

2 Une *epoch* d'entrainement

Nous allons maintenant mettre en œuvre une époque d'entrainement (c'est-à-dire une passe avant/arrière sur chaque instance du jeu de données).

✎ À chaque époque d'entrainement, il va falloir mélanger le jeu d'entrainement : `X_train, y_train = shuffle(X_train, y_train)` [1]. À votre avis, pourquoi cette opération est-elle nécessaire ?

Les étapes pour une époque d'entrainement sont les suivantes :

```
Pour chaque instance Xi du jeu d'entrainement faire
  Xi              <- Xi.transpose
  erreur_train[i] <- passe_avant(Xi)
  passe_arriere(Xi)
fpour
erreur_train = moyenne_erreur(erreur_train[])

Pour chaque instance Xi du jeu de __test__ faire
  Xi             <- Xi.transpose
  erreur_test[i] <- passe_avant(Xi)
fpour
erreur_test = moyenne_erreur(erreur_test[])
```

1. https://scikit-learn.org/stable/modules/generated/sklearn.utils.shuffle.html

3 Entrainement du modèle

Maintenant que vous avez toutes les briques élémentaires, vous pouvez entrainer votre réseau. Pour commencer, vous pouvez lancer n époques d'entrainement (utiliser matplotlip pour afficher l'évolutions des erreurs d'entrainement et de test au fur et à mesure du processus).

Pour aller plus loin, vous pouvez exploiter les erreurs test/train pour mettre en œuvre l'*early stopping !*

La figure 1 montre un exemple de résultat d'entrainement que l'on peut obtenir.

✎ L'entrainement aurait-il dû s'arrêter avant la fin des 100 époques ? Pourquoi ? Quelle est la conséquence d'un entrainement tel que celui illustré sur cette figure ?

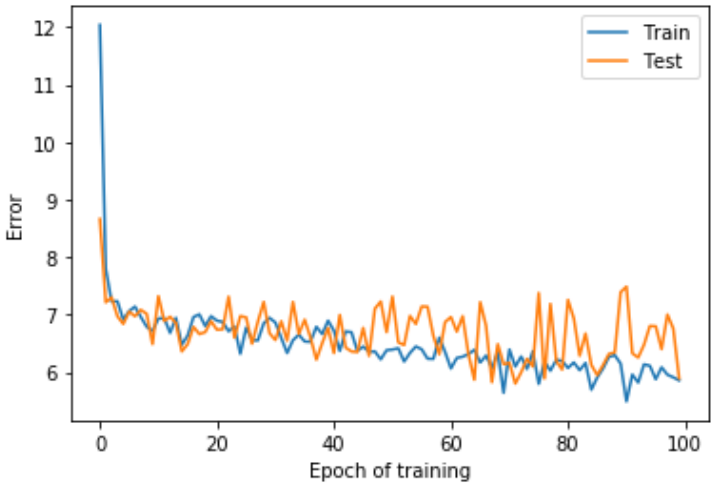

FIGURE *1 – Erreur sur les jeux d'entrainement et de test pendant le processus d'apprentissage.*