ELE778-01 - Intelligence artificielle: réseaux neuroniques et systèmes experts

Laboratoire 3

Présenté à :
Cynthia Moussa

Liam Beguin
BEGL02129304

Louis Laporte
LAPL14128903

École de technologie superieure

9 août 2016

Table des matières

1 Introduction **2**

2 Implémentation du réseau de neurones **2**
2.1 Architecture globale 2
2.2 Extraction des données 3
2.3 Initialisation du réseau 4
2.4 Entrainement 7
2.5 Ajustement des hyper-paramètres 8
2.6 Inspection du reséau 9

3 Discussions **9**
3.1 Comparaison LVQ vs MLP 9
3.2 Conclusion et pistes d'amélioration 9

1 Introduction

Lors du laboratoire 2 nous avons implémenté un réseau de neurone à rétro-propagation du gradient d'erreur. Pour ce laboratoire nous allons réutilliser l'étape de pré-traitement et nous allons implémenter un réseau de neurones de type LVQ (Learning Vector Quantization) dans le but de classifier, le plus précisément possible, le dataset *TiDigits* (english spoken digits from 1 to 9). Enfin, nous comparerons les 2 types d'implémentation quand a leur erreur hors echantillon et leur temps d'apprentissage.

Pour implémenter ce réseau de neurones, nous utiliserons *Python*. Voici une liste d'avantages versus inconvénients qui ont motivé notre choix pour ce language :

— Avantages :
  Rapidité de développement,
  Language interprété donc pas de compilation,
  Modules de calcul matriciel performants,
  Language accessible,

— Inconvenients :
  Language interprété donc moins rapide.

Dans un premier temps, nous entrainerons le réseau sur un set de données puis nous évaluerons sa capacité a généraliser sur de nouvelles données non utilisées lors de l'entrainement.

2 Implémentation du réseau de neurones

2.1 Architecture globale

Comme dit en introduction, nous allons implémenter un réseau de neurones de type LVQ. Le LVQ est un algorithme supervisé dit compétitif dans le sens où les sorties compétitionnent afin de ne donner qu'une seule sortie active.

Le LVQ est une méthode de classification où chaque sortie représente une classe, le nombre de classes doit cependant etre fournit par l'utilisateur. Au cours du processus d'apprentissage les poids (ou prototypes ou encore codebooks) de chaque classe sont attirés ou repoussés dans le but de les positionner optimalement par rapport aux données d'entrainement.

Actualisation des poids : Les poids sont mis à jour à l'aide de expression suivante :

$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + s(t) \cdot \eta \cdot \big(\mathbf{x} - \mathbf{w}_j(t)\big) \tag{1}$$

Où :

$$\begin{cases} j = \textit{numero} \text{ de la classe,} \\ t = \text{instant ou epoch} \\ s = 1 \text{ si } \hat{y} = y \text{ sinon } s = -1 \\ \eta = \text{taux d'apprentissage} \end{cases} \tag{2}$$

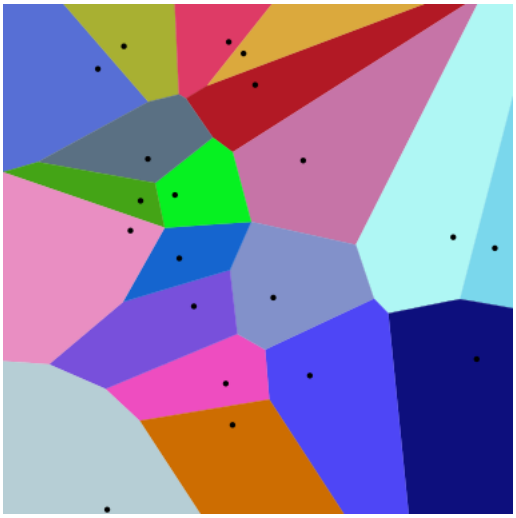

FIGURE 1 – Représentation d'une classification de type dans $\mathbb{R}^2$.

2.2 Extraction des données

Pour extraire les informations de la base de donnée *TiDigits*, nous avons utilisé le code que nous avions développé pour la première partie du laboratoire. Nous nous sommes basés sur les similarités entre la prononciation d'un même chiffre par plusieurs individus. Pour cela, nous avons filtré les énergies statiques et dynamiques afin de pouvoir accentuer les propriétés de chaque chiffre.

Le filtrage consiste dans un premier temps à utiliser une opération de seuillage sur l'énergie statique afin de détécter le début du mot et d'éliminer tout le *blanc* qui se trouve au début de l'enregistrement. La seconde étape permet d'aligner le début de chaque mot en utilisant le premier maximum de l'énergie dynamique. Ceci nous permet de ne pas être trop sensible aux différences d'attaque entre les individus.

Comme tous les fichiers n'ont pas la même taille, nous devons ajouter du padding afin que les données puissent être utilisées comme entrée pour le réseau. En effet, dans le cas où les longueurs ne sont pas identiques, il est impossible d'alimenter le réseau en raison d'erreurs de dimensions sur les produits matriciels. D'autre part, dans le but d'améliorer la vitesse de convergence de l'apprentissage et de limiter la complexité du modèle, nous avons choisi de ne garder que les données statiques et de les normaliser, selon chaque colonne.

La fonction d'extraction des données *extract_datasets()* est conçue pour prendre 2 paramètres : le nombre de features désiré (40, 50, 60, ...) et un booléen permettant de choisir si l'on desir ou non classifier selon le sexe de l'interprète. cette fonction retourne 3 listes : un *training_set*, un *validation_set* et un *test_set* où chaque élément est une combinaison de 2 vecteurs les features $\mathbf{x}$ et les labels $\mathbf{y}$.

Nous avons pour ce nouvel algorithme ajouter une option permettant de fournir des labels sous forme non-vectorisés afin de faciliter le traitement lors des étapes suivantes.

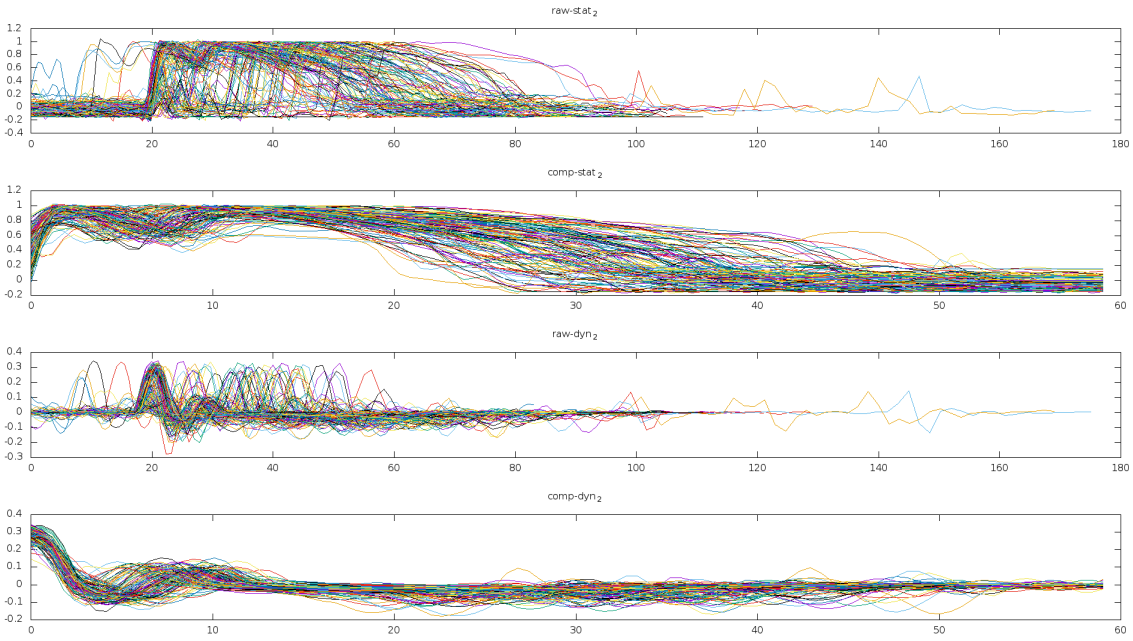

FIGURE 2 – Représentation des énérgies statiques et dynamiques avant et après le traitement

2.3 Initialisation du réseau

Lorsque le réseau est instancié, nous passons le nombre de centroïde que nous voulons par classe. Ensuite nous avons le choix entre 2 types d'initialisation de poids.
Le premier choix permet d'initialiser en choisissant des échantillons de test de manière alétoire parmis le dataset.

La seconde méthode permet de choisir les prototypes initiaux en calculant le centroïde de l'ensemble des points du training set pour chaque classe.

```
1  def random_weight_init(input_size, output_size, ppc, tr_d):
2      """ select codebooks at random within the dataset """
3
4      weights = [ np.zeros(shape=(input_size, 1)) ] * (output_size * ppc)
5
6      np.random.shuffle(tr_d)
7      tr_dX = []
8      tr_dY = []
9      for x, y in tr_d:
10         tr_dX.append(x)
11         tr_dY.append(y)
12
13     tr_dX = np.array(tr_dX)
```

```
    tr_dY = np.array(tr_dY)

    for w_idx, w in enumerate(weights):
        match = np.where(tr_dY == w_idx%output_size)[0][0]
        weights[w_idx] = tr_dX[match]

    return weights
```

Listing 1 – Initialisation des poids avec la méthode de selection aléatoire

```
def average_weight_init(input_size, output_size, ppc, tr_d):
    """ init codebooks using the mean point """

    weights = [ np.zeros(shape=(input_size, 1)) ] * (output_size * ppc)

    np.random.shuffle(tr_d)
    tr_dX = []
    tr_dY = []
    for x, y in tr_d:
        tr_dX.append(x)
        tr_dY.append(y)

    tr_dX = np.array(tr_dX)
    tr_dY = np.array(tr_dY)

    for w_idx, w in enumerate(weights):
        match = np.where(tr_dY == w_idx%output_size)
        weights[w_idx] = np.mean(tr_dX[match], axis=0)

    return weights
```

Listing 2 – Initialisation des poids avec la méthode des centres

Validation croisée : Pour évaluter l'erreur commise sur des échantillons jamais vu auparavant, il est important de séparer le set de données en deux une partie pour l'entrainement l'autre pour la validation. Plusieurs méthodes différentes sont disponibles pour déterminer comment séparer le dataset. Plus on à de données dans le set d'entrainement plus la prédiction sera bonne mais moins on sera capable de quantifier la capacité de généralisation du réseau et inversement.

Une méthode intéressante à citer est le *V-folds* ou *K-folds* qui permet de maximiser le nombre d'échantillons d'entrainement en divisant le set complet en V sous ensembles. Le réseau est ensuite entrainé sur tous les sous-ensembles moins un qui est utilisé pour la validation (le sous-ensemble de validation est choisi aléatoirement et change à chaque époque d'apprentissage). Il faut noter qu'il est important de sélectionner les échantillons de manière aléatoire afin de ne pas biaiser l'apprentissage du réseau.

Cependant, notre set de données étant déjà séparé en plusieurs sous-ensembles, nous garderons ces méthodes comme pistes d'amélioration futures.

Le early stopping est utilisé ici aussi mais dans un autre but. N'ayant plus de *sur-apprentissage*, il consiste simplement à monitorer les erreurs au cours de l'entrainement du réseau et de l'interrompre sous certaines conditions de seuil.

2.4 Entrainement

Lors de l'entrainement on passe comme paramètre le nombre maximal d'epochs, le taux d'apprentissage η et η_decay, un booleen activant ou non la diminution du taux d'apprentissage. L'entrainement du réseau est un processus gourmand en calculs et demandant beaucoup de ressources. Il est donc important de bien l'optimiser.

```
1  def train(self, tr_d, eta, epochs, eta_decay=False, va_d=None, estop=True):
2
3      va_err, tr_err = [], []
4      for i in xrange(0, epochs):
5          for x, y in tr_d:
6              d = []
7              for w in self.weights:
8                  # Compute distances
9                  d.append(self.distance(x, w))
10
11             # find closest centroid
12             bmu = np.argmin(d)
13             # Update closest weight: Best Matching Unit
14             if bmu % self.output_size == y:
15                 s = 1
16             else:
17                 s = -1
18             self.weights[bmu] += s * eta * (x - self.weights[bmu])
19
20         if eta_decay:
21             # Compute optimized learning rate
22             eta = eta / (1 + s * eta) if eta < 1 else 1
23
24         tr_err.append(self.eval_error_rate(tr_d))
25         # Validation
26         if va_d:
27             va_err.append(self.eval_error_rate(va_d))
28             # Stop early if validation error is very low
29             if estop and va_err[-1] < 0.01: break
30
31         self.learn_time = datetime.datetime.now() - self.learn_time
32
33         return tr_err, va_err
```

Listing 3 – Definition de notre methode d'apprentissage

2.5 Ajustement des hyper-paramètres

dataset	out	η	η_decay	epoch	proto	tr(%)	va(%)	test(%)	t(s)
40	18	0.1	True	6	10	3.4	0.9	9.9	20
40	18	0.5	True	10	10	2.8	5.5	8.4	29
40	18	0.6	True	10	10	3.6	3.7	9.7	29
40	18	0.5	True	20	20	1.7	6.5	8.8	112
40	18	0.1	True	20	20	2.9	2.7	10.2	112
40	18	0.01	True	20	20	6.6	3.7	12.7	121
40	18	0.1	False	20	20	4.8	0.9	10.7	121
40	9	0.1	True	1	20	4.0	0	5.7	3
40	9	0.5	True	2	20	2.2	0	4.3	6
40	9	0.6	True	2	20	2.3	0	4.3	6
40	9	0.5	True	2	21	2.4	0.9	3.2	7
50	9	0.5	True	2	21	2.4	0.9	3.2	7
60	9	0.5	True	20	21	2.4	0.9	3.2	38

TABLE 1 – Tableau récapitalif des tests effectués sur les hyper-paramètres sur un *i5 dual-core, 2.2GHz* pour le LVQ

Pour obtenir la meilleur solution nous avons cherché en premier lieu le taux d'apprentissage η qui donnait la plus petite erreur sur le training set, ensuite nous avons ajusté le nombre de prototype par classe avec et sans la réduction du taux d'apprentissage. Nous pouvons conclure qu'aux vues des résultats la meilleur configuration est pour un dataset contenant de 40 éléments avec un taux d'apprentissage de 0.5. Lorsque nous avons fait la classification homme/femme, nous avons observé qu'á partir de 10 prototypes (par classe) les performances sur le taux d'erreur du dataset de test n'était pas améliorées.

En ce qui concerne la classification des chiffres uniquement le choix du nombre de prototypes est le meilleur entre 21 et 22, avant ou après les valeurs les performances se dégradent.

2.6 Inspection du reséau

Parmis les contraintes obligatoires, il nous est aussi demandé de pouvoir inspecter les couches cachées du réseau. Pour ce faire, nous tirons parti du fait que *Python* est un language interprété et qu'il fournit à l'utilisateur une interface intéractive(c.f. laboratoire 2.2).

En combinaison avec la possibilité de sauvegarder l'état des objects *LVQ* sous la forme de fichiers (compressés ou non), nous pouvons aisément analyser chaque couche.

3 Discussions

3.1 Comparaison LVQ vs MLP

dataset	out	σ	cost	Ω	η	λ	epoch	batch	hidden	tr(%)	va(%)	test(%)	t(s)
40	18	s	C	L2	0.5	0.001	30	10	0	0	2	14	20
40	9	s	C	L2	0.5	0.001	30	10	0	1	1	7	20

TABLE 2 – Tableau récapitalif des tests effectués sur les hyper-paramètres sur un *i5 dual-core, 2.2GHz* pour le MLP

dataset	out	η	η_{decay}	epoch	proto	tr(%)	va(%)	test(%)	t(s)
40	18	0.5	True	10	10	2.8	5.5	8.4	29
40	9	0.5	True	2	21	2.4	0.9	3.2	7

TABLE 3 – Tableau récapitalif des tests effectués sur les hyper-paramètres sur un *i5 dual-core, 2.2GHz* pour le LVQ

Nous pouvons constaté qu'avec le *LVQ* nous avons de meilleure performance sur le dataset de test et en moins d'epochs. Il faut cependant noter que pour le *MLP*, une fois que les paramètres donnant les meilleurs performances sont trouvées il n'y a plus besoin de les modifier si on change le nombre d'entrée. De plus les temps de calcul pour trouver les meilleurs réponses sont identiques alors que pour le *LVQ* il est nécessaire de changer les paramètres. Le temps de calcul augmente par 4 fois entre 9 et 18 entrées pour le *LVQ*. On peut en clonclure que le *MLP* est plus flexible quant au nombre d'entrée mais a de moins bonne performance sur la dataset de test. Le *LVQ* est plus performant mais lorsqu'on augmente le nombre de d'entrée il devient plus lent. Le *LVQ* est donc plus rapide pour nombre d'entrées limitées et fixes tandis que le *MLP* est plus générique.

3.2 Conclusion et pistes d'amélioration

Encore une fois, du a un manque de temps, nous n'avons pas poussé plus que necessaire sur les optimisations de type (convertion des listes de np.array en matrices). Ceci permettrai certainement de rendre le système un peu plus rapide.

En revanche, malgré cela, on observe tout de même de bonnes performances. On remarque aussi une bonne capacité de généralisation avec une erreur d'environs 5-10% sur le set de test après l'apprentissage.

Ici aussi, nous avons implémenté la génération des matrices de confusion. En les observant attentivement, on remarque que le réseau commet principalement des erreurs lors de la classification homme/femme. Ceci pourrait probablement être du à un pré-traitement trop *agressif* ou encore à un manque d'échantillons d'entrainement. Cette classification erronée pourrait peut-être être ajustée en implémentant une méthode de validation-croisée plus avancée telle que le *V-folds* cité précédemment.

D'autres pistes d'améliorations seraient d'implémenter un ajustement dynamique du nombre de prototypes par classes ou encore d'implementer des variantes du *LVQ* plus performantes telles que le *LVQ2*, *LVQ3* ou encore le *DVQ*. On pourrait aussi penser à combiner plusieurs types de réseaux (c.f. *DTW*) en utilisant des methodes d'agrégation.