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

Laboratoire 2

Présenté à :
Cynthia Moussa

Liam Beguin
BEGL02129304

Louis Laporte
LAPL14128903

École de technologie superieure

---

25 juillet 2016

Table des matières

1 Notations et symboles

$\mathbf{x}$	Entrée du réseau,
$\mathbf{y_x}$	Sortie attendue du réseau pour l'entrée $\mathbf{x}$,
$\hat{\mathbf{y}}_\mathbf{x}$	Prédiction du réseau pour l'entrée $\mathbf{x}$,
$\mathbf{a} \circ \mathbf{b}$	Produit matriciel de Hadamard,
$\epsilon_\mathbf{x}$	taux d'erreur lorsque le réseau evalue l'entree $\mathbf{x}$,
$\sigma(z)$	Fonction d'activation generique,
$\sigma'(z)$	Derivée de la fonction d'activation,
$C(w, b)$	Fonction de cout générique,
$\mathbf{C}'(w, b)$	Dérivée de la fonction de cout générique par rapport a $\hat{\mathbf{y}}$,
$\nabla_b C$	Dérivées de C par rapport aux seuils de la couche (l),
$\nabla_w C$	Dérivées de C par rapport aux poids de la couche (l),
$d^{(l)}$	Nombre d'elements sur la couche (l),
$\mathbf{w}^{(l)}(T)$	Ensemble des poids de la couche (l) à l'époque T,

2 Introduction

Après avoir fait une étape de pré-traitement lors du laboratoire 2.1, nous allons maintenant implémenter un réseau de neurones à rétro-propagation du gradient d'erreur dans le but de classifier, le plus précisément possible, le dataset *TiDigits* (english spoken digits from 1 to 9).

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.

3 Implémentation du réseau de neurones

3.1 Architecture globale

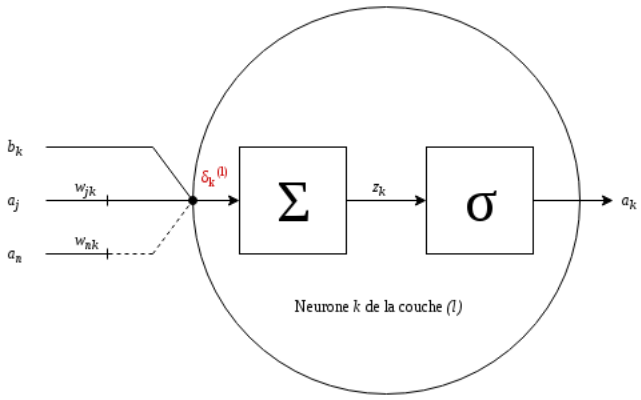

FIGURE 1 – Représentation symbolique d'un neurone.

Comme dit en introduction, Nous allons implémenter un réseau de neurones a rétro-propagation du gradient d'erreur. Ici, nous énoncerons les notations et équations génériques utilisées au sein du réseau.

La majorité des calculs sont effectués sous forme matricielle afin de simplifier et généraliser les équations mais aussi de mieux tirer avantage des ressources de calcul disponibles.

Étant donné que le réseau doit être très flexible quant au nombre de couches et au nombre de neurones, nous utiliserons un tuple (une liste invariante) pour instancier le réseau. Chaque élément de la liste représente une couche du réseau (incluant la couche d'entrée) et sa valeur détermine le nombre de neurones present sur la couche. Pour la couche de sortie, nous utiliserons un vecteur de type *one-hot* ou un seul element est actif à la fois.

D'autre part, on nous demande aussi que le réseau soit capable d'utiliser differentes fonctions d'activation. Base sur cette contrainte, nous avons choisi d'aussi donner a l'utilisateur la possibilite de choisir entre differentes fonctions de cout et differentes fonctions de régularisation.

Notations : Soit, $w_{jk}^{(l)}$ le poid connectant le neurone k de la couche $(l-1)$ au neurone j de la couche (l), où :

$$\begin{cases} 1 \leq l \leq L & \text{layers} \\ 0 \leq k \leq d^{(l-1)} & \text{inputs} \\ 1 \leq j \leq d^{(l)} & \text{outputs} \end{cases}$$

On définit donc $\mathbf{w}^{(l)}$ la matrice de dimension $\Big((l) \times (l-1)\Big)$ connectant la couche $(l-1)$ à la couche (l) créant ainsi, $(L-1)$ matrices de poids sur l'ensemble du réseau.

Feedforward : cette étape propage l'entrée a travers l'ensemble du réseau et enregistre toutes les variables intermediaires.

$$\begin{aligned} z_j^{(l)} &= \sum_{k}^{d^{(l-1)} w_{jk}^{(l)}} \cdot a_k^{(l-1)} + b_j^{(l)} \\ \mathbf{z}_x^{(l)} &= \mathbf{w}^{(l)} \cdot \mathbf{a}_x^{(l-1)} + \mathbf{b}^{(l)} \\ \mathbf{a}_x^{(l)} &= \sigma\left(\mathbf{z}_x^{(l)}\right) \end{aligned} \tag{1}$$

Pour avoir la prédiction finale du réseau $\hat{\mathbf{y}}$, il faut répéter l'étape ci-dessus jusqu'à atteindre la sortie où on obtient :

$$\begin{aligned} \mathbf{z}_x^{(L)} &= \mathbf{w}^{(L)} \cdot \mathbf{a}_x^{(L-1)} + \mathbf{b}^{(L)} \\ \hat{\mathbf{y}}_x &= \sigma\left(\mathbf{z}_x^{(L)}\right) \end{aligned} \tag{2}$$

La fonction de cout est la fonction utilisée lors du processus d'entrainement pour évaluer l'erreur entre la prédiction du réseau $\hat{\mathbf{y}}$ et la sortie attendue $\mathbf{y}$. Les différents fonctions disponibles seront detaillées dans la section 3.5. Pour la suite, nous l'appelerons de façon generique : $C(\mathbf{w}, \mathbf{b})$.

La fonction de regularization est la fonction utilisee lors du processus d'entrainement pour pénaliser la minimisation de la fonction de cout. Ceci sera couvert plus en detail dans la section 3.6. Pour la suite, nous l'appelerons de façon générique : $\Omega(\mathbf{w})$.

retro-propagation : Cette étape calcule l'impacte de chaque poids (et seuil) sur l'erreur finale. Pour obtenir l'erreur δ_j^l due au neurone j de la couche l, il faut propager l'erreur de la sortie vers l'entrée avec les équations suivantes :

$$\begin{aligned}\boldsymbol{\delta}^{(L)} &= \frac{\partial C}{\partial \mathbf{z}^{(L)}} = \frac{\partial C}{\partial \mathbf{a}^{(L)}} \circ \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} \\ &= \frac{\partial C}{\partial \mathbf{a}^{(L)}} \circ \sigma'\left(\mathbf{z}^{(L)}\right) = \frac{\partial C}{\partial \hat{\mathbf{y}}} \circ \sigma'\left(\mathbf{z}^{(L)}\right) \\ &= C'(\mathbf{w}, \mathbf{b}) \circ \sigma'\left(\mathbf{z}^{(L)}\right)\end{aligned} \tag{3}$$

De la même manière, on définit une notation récursive :

$$\begin{aligned}\delta_j^{(l)} &= \frac{\partial C}{\partial z_j^{(l)}} = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \\ &= \sum_k \delta_k^{(l+1)} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}\end{aligned} \tag{4}$$

Or,

$$z_k^{(l+1)} = \sum_j^{d^{(l)}} w_{jk}^{(l+1)} \cdot a_j^{(l)} + b_k^{(l+1)}$$

Donc,

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^l} = w_{kj}^{(l+1)} \cdot \sigma'\left(z_j^{(l)}\right)$$

Et,

$$\delta_j^{(l)} = \sigma'\left(z_j^{(l)}\right) \cdot \sum_k \delta_k^{(l+1)} \cdot w_{kj}^{(l+1)}$$

Nous pouvons maintenant utiliser $\delta_j^{(l)}$ pour determiner l'impacte de chaque seuil sur la fonction de cout :

$$\begin{aligned}\frac{\partial C}{\partial b_j^{(l)}} &= \frac{\partial C}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} \\ &= \delta_j^{(l)}\end{aligned} \tag{5}$$

Idem pour les poids :

$$\begin{aligned}\frac{\partial C}{\partial w_{jk}^{(l)}} &= \frac{\partial C}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} \cdot \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} \cdot a_k^{(l-1)} \\ &= \delta_j^{(l)} \cdot \sigma\left(z_k^{(l-1)}\right)\end{aligned} \tag{6}$$

Et sous formes matricielles :

$$\begin{aligned} \boldsymbol{\delta}^{(l)} &= \left(\left(\mathbf{w}^{(l+1)}\right)^T \cdot \boldsymbol{\delta}^{(l+1)}\right) \circ \sigma'\left(\mathbf{z}^{(l)}\right) \\ \nabla_b C &= \frac{\partial C}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)} \\ \nabla_w C &= \frac{\partial C}{\partial \mathbf{w}^{(l)}} = \boldsymbol{\delta}^{(l)} \cdot \left(\sigma\left(\mathbf{z}^{(l-1)}\right)\right)^T \end{aligned} \tag{7}$$

Actualisation des poids : maintenant que le réseau est en mesure de quantifier l'erreur due a chaque neurone, nous pouvons mettre a jour les poids à l'aide des expressions suivantes :

$$\begin{aligned} \mathbf{b}^{(l)}(T+1) &= \mathbf{b}^{(l)}(T) - \eta \nabla_b C \\ \mathbf{w}^{(l)}(T+1) &= \mathbf{w}^{(l)}(T) - \eta \cdot \left(\nabla_b C + \Omega'(\mathbf{w})\right) \end{aligned} \tag{8}$$

3.2 Extraction des données

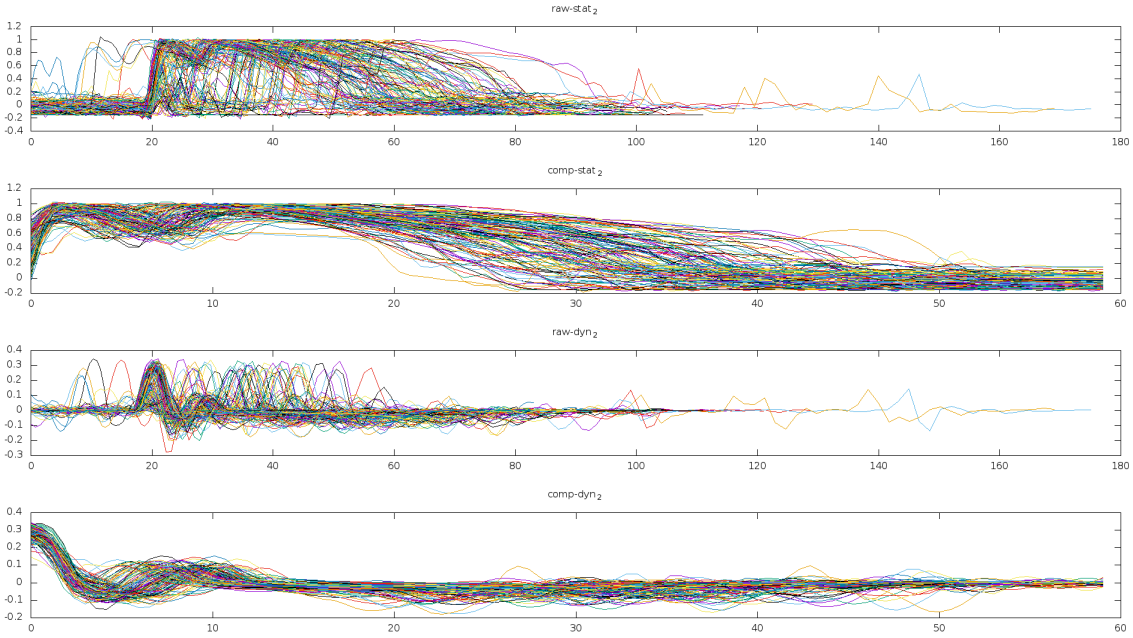

FIGURE 2 – Représentation des énérgies statiques et dynamiques avant et après le traitement

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 d'étécter 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 figure 12 donne un aperçu de l'impacte du pré-traitement sur les performances du réseau.

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 **x** et les labels **y**.

```
1  def extract_sample(file_, size=60, sex=False, out_size=9):
2      # Preprocess file ...
3      x = prep.Preprocessing(file_, size)
4      x.start_point_detection(threshold=0.5, n=10)
5      x.cut_first_max(n=20)
6      x.normalize()
7      x.fit()
8      x.get_subset('static')
9
10     num = int(re.search(r'(?=.*)[0-9](?=.*)', file_).group(0))
11
12     # make a column of the whole array
13     features = x.data.reshape((len(x.data)*len(x.data[0]), 1) )
14
15     if sex:
16         if re.search(r'.*woman.*', file_):
17             labels = vectorize_output( num - 1 + out_size, shape=(out_size*2, 1))
18         else:
19             labels = vectorize_output(num-1, shape=(out_size*2, 1))
20     else:
21         labels = vectorize_output(num-1, shape=(out_size, 1))
22
23     return (features, labels)
```

Listing 1 – Fonction appeleé pour extraire les données d'un unique fichier

3.3 Initialisation du réseau

Lorsque le réseau est instancié, les paramètres tels que η et λ sont copiés dans l'object *Network* et les matrices de poids et de seuils (*weights* et *biases*) sont générées. Les dimensions de ces matrices sont determinées en fonction du nombre de neurones sur chaque couche et du nombre de couches L.

D'autre part, du fait que la fonction aléatoire utilisée pour l'initialisation des poids donne une distribution avec les paramètres suivants $\mu = 0$ et $\sigma = 1$, l'initialisation des poids a tendance à saturer les neurones et ralentire l'apprentissage. car :

$$\begin{aligned} Var(z) &= \sum_{i=1}^{N} (Var(x_i)) \\ Var(z) &= N \cdot Var(x) \\ \sigma(z) &= \sqrt{N} \cdot 1 \\ \sigma(z) &= \sqrt{N} \end{aligned} \tag{9}$$

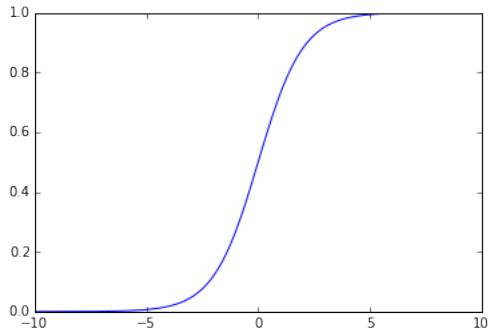

FIGURE 3 – Visualisation de la saturation de la fonction *sigmoid* si $|z| \geq 5$.

Pour contrer cette saturation et donc accelerer les premières époques d'apprentissage, nous utiliserons une distribution plus étroite avec les paramètres $\mu = 0$ et $\sigma = 1/\sqrt{N}$

```
1 self.biases  = [ np.random.randn(y, 1) for y in struct[1:] ]
2 self.weights = [ np.random.randn(y, x) / np.sqrt(x) \
3                  for x, y in zip(struct[:-1], struct[1:]) ]
```

Listing 2 – Initialisation des poids et seuils

3.4 Fonctions d'activation

La fonction sigmoid est la fonctions de référence dans les réseaux de neurones et est definie de la manière suivante :

$$\begin{aligned} \sigma(x) &= \frac{1}{1+e^{-x}} \\ \sigma'(x) &= \sigma(x)(1-\sigma(x)) \end{aligned} \tag{10}$$

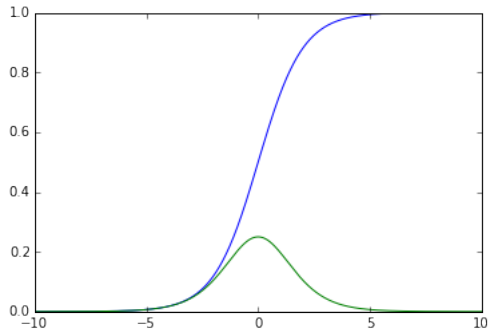

FIGURE 4 – Representation de la fonction sigmoid (blue) et de sa derivee (vert)

elle permet de d'obtenir une sortie comprise dans l'intervalle [0;1] qui peut dans certain cas être assimilé à une probabilité. C'est principalement cette fonction que nous utiliserons.

```
1 def sigmoid(z, prime=False):
2     if prime:
3         return sigmoid(z)*(1-sigmoid(z))
4     else:
5         return 1.0 / (1.0 + np.exp(np.negative(np.clip(z, -50, 50))))
```

Listing 3 – Definition de la fonction *sigmoid* et de sa dérivée

La fonction tangente hyperbolique est aussi tres rependue dans les réseau de neurones et permet de definir un *soft threshold*. En d'autres termes, c'est une version derivable de la fonction de seuil utilisee par le perceptron. Elle se comporte comme une fonction linéaire proche de 0 et comme une fonction de seuil sur le reste de son ensemble de définition. Elle est définie de la manière suivante :

$$\begin{aligned} \tanh(x) &= \frac{\sinh(x)}{\cosh(x)} \\ \tanh'(x) &= 1 - \tanh^2(x) \end{aligned} \tag{11}$$

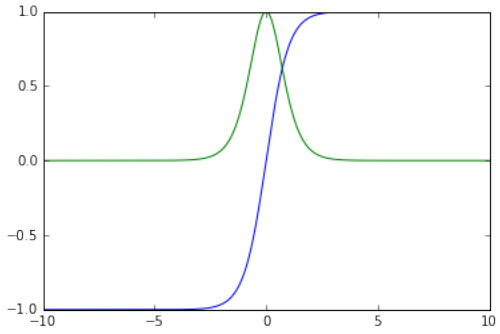

FIGURE 5 – Reprèsentation de la fonction tanh (blue) et de sa derivée (vert)

(12)

```
1  def tanh(z, prime=False):
2      if prime:
3          return 1.0 - np.square(tanh(z))
4      else:
5          return np.tanh(z)
```

Listing 4 – Définition de la fonction tanh et de sa derivée

La fonction Softplus est quant à elle beaucoup moins utilisée mais semblait intéressante au vue de sa simplicité. Elle est définie de la manière suivante :

$$\begin{aligned} f(x) &= ln(1 + e^x) \\ f'(x) &= \frac{1}{1 + e^{-x}} \end{aligned} \tag{13}$$

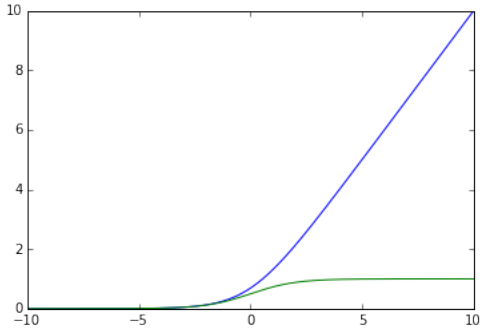

FIGURE 6 – Representation de la fonction softplus (blue) et de sa derivee (vert)

```
1  def softplus(z, prime=False):
2      if prime:
3          return sigmoid(z)
4      else:
5          return np.log(1 + np.exp(np.clip(z, -50, 50)))
```

Listing 5 – Definition de la fonction *softplus* et de sa derivee

3.5 Fonctions de coût

Comme énnoncé précédemment, nous offrons aussi à l'utilisateur la possibilité de choisir parmis différentes fonctions de coûts, soit la fonction que le réseau cherche à minimiser en ajustant les poids et seuils des connections.

quadratic est une fonction d'erreur quadratique moyenne. Cete fonction est la plus simple et est definie de la maniere suivante :

$$\begin{aligned} C(w,b) &= \frac{1}{2 \cdot d^{(L)}} \sum_{j}^{d^{(L)}} (\hat{\mathbf{y}} - \mathbf{y})^2 \\ \mathbf{C}'(w,b) &= \frac{1}{d^{(L)}} \cdot (\hat{\mathbf{y}} - \mathbf{y}) \end{aligned} \tag{14}$$

cross-entropy est la seconde option et est implementée dans le but de limiter le phénomène de *learning slowdown* en évitant la saturation des neurones. Cette fonction permet au réseau de corriger ses poids plus efficacement dans le cas d'une erreur importante. Elle ne peut être seulement utilisée avec une fonction d'activation *sigmoid* et est construite pour annuler la multiplication par $\sigma'(z)$ dans l'expression de $\boldsymbol{\delta}^{(L)}$.

$$\begin{aligned} C(w,b) &= -\frac{1}{d^{(L)}} \sum_{j}^{d^{(L)}} (\mathbf{y} \ln(\hat{\mathbf{y}}) + (1 - \mathbf{y}) \ln(1 - \hat{\mathbf{y}})) \\ \mathbf{C}'(w,b) &= \frac{1}{d^{(L)}} \cdot (\hat{\mathbf{y}} - \mathbf{y}) \end{aligned} \tag{15}$$

NOTE : On remarque ici que les dérivées des deux fonctions sont identiques. En revanche, une condition dans le code est ajoutée pour supprimer la multiplication par $\sigma'(z)$.

3.6 Eviter le sur-ajustement

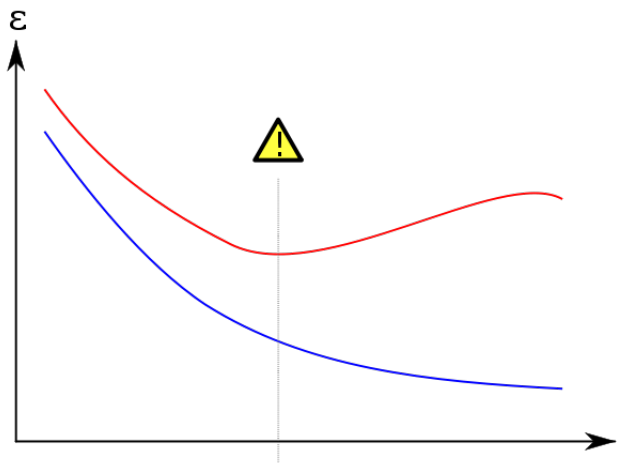

FIGURE 7 – erreurs sur le set d'entrainement(bleu) et de validation(rouge)

Le sur-ajustement (ou *overfitting*) survient lorsque le réseau n'est plus en train d'apprendre du dataset mais est en train de le mémoriser. Ceci se traduit par un apprentissage trop spécifique aux données, le réseau est donc en train de modéliser le *bruit*. Ceci survient lorsque le modèle (nombre de neurones et couches) est plus complexe que la fonction à modéliser. Ce sur-apprentissage à des conséquences néfastes sur les performances du réseau au moment de la généralisation, lorsqu'il est confronté à de nouvelles données. Il est donc impératif de le détecter et de le minimiser.

Validation croisée : Pour détecter le sur-ajustement, 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.

Detection : Le sur-ajustement peut etre aisement détecté lorsque l'erreur sur le set de validation commence à augmenter tandis que l'erreur sur le set d'entrainement continue de décroitre.

Early stopping est une méthode de limitation du sur-ajustement. Elle consiste à monitorer les erreurs au cours de l'entrainement du réseau et de l'interrompre sous certaines conditions(seuil, moyenne glissante, ...).

Les fonctions de régularisation sont une autre méthode permettant de limiter le sur-ajustement. Celles-ci ont pour approche de *pénaliser* la minimisation de la fonction d'erreur dans le but d'améliorer la capacité de généralisation du réseau.

Ici, sont implementées deux des méthodes les plus répendues : régularisation *L1* et *L2*. Toute deux visent à limiter la magnitude des poids dans l'ensemble du réseau. Redéfinissons la fonction de coût de la manière suivante :

$$C(\mathbf{w}, \mathbf{b}) = C_0(\mathbf{w}, \mathbf{b}) + \Omega(\mathbf{w}) \tag{16}$$

Où $\Omega(w)$ est l'expression générale de la fonction de régularization.

On peut maintenant definir $\Omega(\mathbf{w})$ pour chaque type de régularization :

$$\begin{aligned} \Omega_{L1}(\mathbf{w}) &= \frac{\lambda}{2 \cdot d^{(l)}} \cdot \sum |\mathbf{w}^{(l)}| \\ \mathbf{\Omega}'_{L1}(\mathbf{w}) &= \frac{\lambda}{d^{(l)}} \cdot sign\left(\mathbf{w}^{(l)}\right) \end{aligned} \tag{17}$$

$$\begin{aligned} \Omega_{L2}(\mathbf{w}) &= \frac{\lambda}{2 \cdot d^{(l)}} \cdot \sum \left(\mathbf{w}^{(l)}\right)^2 \\ \mathbf{\Omega}'_{L2}(\mathbf{w}) &= \frac{\lambda}{d^{(l)}} \cdot \mathbf{w}^{(l)} \end{aligned} \tag{18}$$

NOTE : La régularisation *Tikhonov* est une forme generalisée de ces approches et permet une plus grande flexibilité.

3.7 Entrainement

L'entrainement du réseau est un processus gourmand en calculs et demandant beaucoup de ressources. Il est donc important de bien l'optimiser. Dans ce but, nous sommes passés par plusieurs méthodes avant d'arriver à la plus optimale.

batch gradient descent consiste à calculer le gradient en une seule itération sur l'ensemble des échantillons d'entrainement avant de pouvoir actualiser les poids. Etant donné que l'entrainement nécessite un grand nombre d'echantillons pour donner des resultats satisfaisant, cette approche impose des calculs sur des matrices de **grande** dimensions.

stochastic gradient descent : contrairement à la méthode précédente, le gradient est ici calculé sur un seul échantillon selectionné aléatoirement (d'où son nom). Ceci permet de réduire considérablement le coût des calculs et est prouvé de converger vers la même valeur que le *batch gradient descent* en montrant que l'espérance mathématique des deux versions du gradient sont égales.

Mini-batch stochastic gradient descent est la méthode implémentée dans le réseau. Cette version est une généralisation des deux approches précédentes. Au lieu de selectionner un seul élément aléatoirement, le set d'entrainement est divisé en sous ensembles de taille N. Il est intéréssant de noter que si $N = 1$, on obtient un *stochastic gradient descent* standard et si N est égal à la taille du set d'entrainement, on obtient un *batch gradient descent*.

```
1 In [1]: x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2 In [2]: N = 2
3 In [3]: print zip(*[iter(x)]*N)
4 [(1, 2), (3, 4), (5, 6), (7, 8)]
```

Listing 6 – division du set **x** en sous-ensembles de taille N

Par manque de temps, la rétro-propagation implémentée permet seulement de calculer le gradient sur un échantillon unique (un vecteur) et non une matrice des N échantillons. Il faut donc sommer les gradients sur tous les éléments de la mini-batch : $\nabla_w C = \sum_i^N \nabla_w C_i$ (Idem pour $\nabla_b C$).

Une impléméntation 'entièrement' matricielle permettrait d'améliorer encore les performances en exploitant pleinement les capacités de *numpy* (le module de calcul utilisé). Il est aussi intéressant de noter que, si plusieurs CPU sont disponibles, les calculs seront alors **distribués**.

```
1 for x, y in mini_batch:
2     # Sum all the gradients over the mini-batch
3     self.feedforward(x)
4     nabla_bC_i, nabla_wC_i = self.backpropagation(y)
5     nabla_bC = np.add(nabla_bC, nabla_bC_i)
6     nabla_wC = np.add(nabla_wC, nabla_wC_i)
```

Listing 7 – sommation N des gradients

3.8 Ajustement des hyper-paramètres

dataset	out	σ	cost	Ω	η	λ	epoch	batch	hidden	tr(%)	va(%)	test(%)	t(s)
50	18	s	C	L2	0.01	0.001	5	10	150	59	53	60	11
50	18	s	C	L2	0.05	0.001	5	10	150	38	34	43	10
50	18	s	C	L2	0.1	0.001	5	10	150	20	15	24	10
50	18	s	C	L2	0.5	0.001	5	10	150	9	6	16	10
50	18	s	C	L2	0.5	0.001	20	10	20	3	8	15	10
50	18	s	C	L2	0.5	0.001	30	10	0	0	3	12	20
40	18	s	C	L2	0.5	0.001	30	10	0	0	2	14	20
60	18	s	C	L2	0.5	0.001	30	10	0	0	6	12	20
40	9	s	C	L2	0.5	0.001	30	10	0	1	1	7	20
50	9	s	C	L2	0.5	0.001	30	10	0	2	1	7	20
60	9	s	C	L2	0.5	0.001	30	10	0	5	1	9	20
40	18	t	Q	L2	0.01	0.001	30	10	150	12	10	21	40
50	18	t	Q	L2	0.01	0.001	30	10	150	11	8	20	50
60	18	t	Q	L2	0.01	0.001	30	10	150	10	13	21	53
40	9	t	Q	L2	0.01	0.001	30	10	50	5	1	10	15
50	9	t	Q	L2	0.01	0.001	30	10	50	5	1	12	15
60	9	t	Q	L2	0.01	0.001	30	10	50	4	1	11	15
40	18	sp	Q	L2	0.1	0.001	30	10	50	10	6	16	20
50	18	sp	Q	L2	0.1	0.0001	30	10	50	7	6	15	20
60	18	sp	Q	L2	0.1	0.0001	30	10	50	7	6	13	20
40	9	sp	Q	L2	0.01	0.0001	30	10	50	5	1	9	20
50	9	sp	Q	L2	0.01	0.001	30	10	50	5	1	9	20
60	9	sp	Q	L2	0.01	0.001	30	10	50	6	2	9	20

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

Pour obtenir la meilleur solution nous avons chercher en premier le *learning rate* qui donnait la plus petite erreur sur le training set, ensuite nous avons ajusté le *regularization rate* avec un faible nombre d'epochs et pour finir nous avons ajusté le nombre de neurone sur une seule couche cachée. Les valeurs de sortie 18 et 9 représentent respectivement avec et sans classification Homme/Femme. C représente la méthode de *cross-entropy*, Q la méthode *quadratic* et s, t, sp sont respectivement les fonctions sigmoid, tanh et softplus.

3.9 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(voir annexe C).

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

D'autre part, la conception du réseau permetterai, avec peu d'effort, de coupler plusieurs objects *Network* en un *super* réseau. Ceci pourrai être fait en implémentant les méthodes *__add__()* et *__getslice__()* pour respectivement concaténer et couper des objects *Network*.

N'ayant pas eu besoin de ces opérations, ces méthodes n'ont cependant pas été implémentées.

4 Interface graphique

L'interface graphique permet de choisir de definir les parametres du réseaux pour afficher le réseau ainsi que les courbes d'erreur et de cout.

Menu L'application posséde un menu permettant de charger un fichier de configuration et de sauvegarder les paramètres actuel du réseau ou de faire un 'reset' du réseau.

Paramètres d'activation Sur cette couche de l'application on peut retrouver les fonctions d'activations, de regularisations, de couts. Il est aussi possible de régler les taux d'apprentissages et de regularisations. Il faut dans un premier temps charger les données, ensuite il faut choisir les paramètres sur lesquels on veut jouer et appuyer sur play afin de les mettre à jour. Pour finir il faut cliquer sur training.

FIGURE 8 – Affichage des paramètres d'activation

Choix des couches La seconde couche permet de choisir les 3 types de set de base de données pour la couche d'entrée (40,50,60 features), le nombre de couches cachées et le nombres de sortie du réseau. Il est aussi possible de choisir de détecter le sexe de la personne ainsi que de choisir un facteur multiplicatif pour la création des neurones.

Choix du nombre de neurone Cette troisième couche permet de choisir un nombre de neurone illimité pour chacune des couches du réseau.

Affichage du réseau Cette dernière couche permet de naviguer entre 2 'Tab' afin de visualiser le réseau ou les erreurs et les couts de validation et d'entrainement (voir 13 en annexe D).

L'affichage des couts et des erreurs ce fait pendant que le réseau est entrain de de s'entrainer(voir 14 en annexe D).

4.1 Navigation

Par défault le résau est configuré avec un taux d'apprentissage de 0.5, une fonction d'activation SoftPlus, un taux de régularisation de 0.01 et une fonction de cout de type "cross-entropy". La couche d'entrée possède par default un un dataset de 40 multiplié par 15, une seule couche cachée avec 50 neurones et 9 sorties. Ensuite les boutons démarrer et pause permettent de respectivement de démarrer l'entrainement du réseau ou de le stopper.

5 Discussions

5.1 Problemes rencontrés

Lors de l'entrainement du réseau, nous avons d'abord été confronté à de très mauvaises performances. Nous avons initialement pensé que notre pré-traitement était inefficace voir destructeur. En analysant les performances plus en détail, nous nous sommes aperçu que le réseau était moins performant que si nous lui faisions faire des prédictions aléatoires $\epsilon_{\mathbf{x}} \geq \left(1 - \frac{1}{9} = 88.9\%\right)$. Ce qui est impossible et suggère que notre évaluation du coût était fausse.

En effet, nous avions commis une erreur dans la génération de nos vecteurs de sortie. Lorsque la prédiction était comparée à la sortie attendue, celle-ci était décalée de 1 comme suit :

$$\left(\hat{\mathbf{y}}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right) \neq \left(\mathbf{y}_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right) \tag{19}$$

5.2 Conclusion et pistes d'amélioration

Comme expliqué précédement, du à un manque de temps, nous avons été contrain de couper certaines optimisations notemment sur l'implémentation matricielle du *Mini-batch SGD*.

En revanche, malgré cela, on observe tout de même de bonnes performances. En utilisant comme référence une classification de 9 digits en combinaison avec une classification Homme/Femme (soit une sortie de 18 neurones) du *TiDigits* dataset, on obtient un temps d'entrainement de 10 secondes sur un *i5 dual-core, 2.2GHz* temps qui tombe à moins de 2 secondes sur un *i7 quad-core, 4GHz*.

On remarque aussi une bonne capacité de généralisation avec une erreur d'environs 10-15% sur le set de test après l'apprentissage.

En observant attentivement les matrices de confusions en figure 12 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 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'ajouter la possibilité de choisir une fonctions d'activation différente pour la couche de sortie ou encore l'implémentation de techniques permettant de determiner *automatiquement* les hyper-paramèters du réseau.

A Influences du pré-traitement sur les performances du réseau

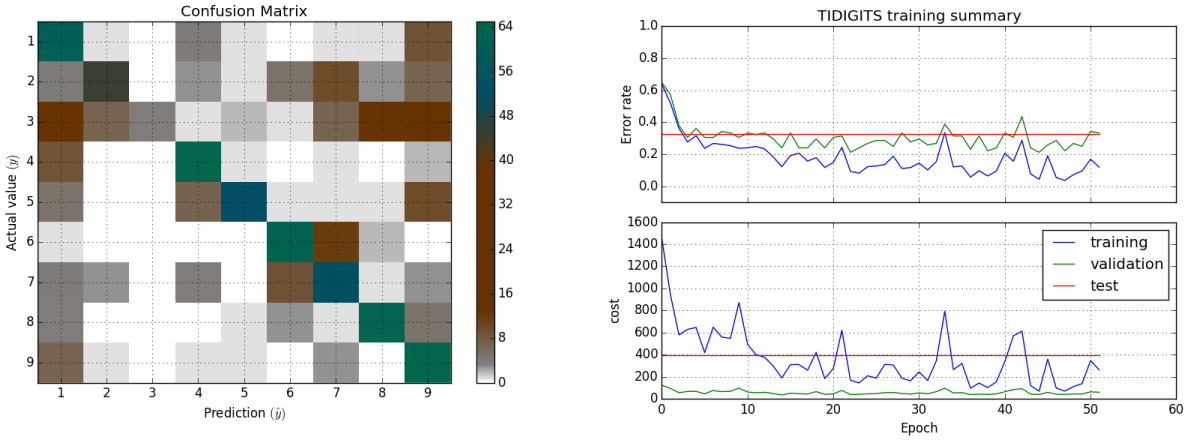

FIGURE 9 – Performances sans pré-traitement

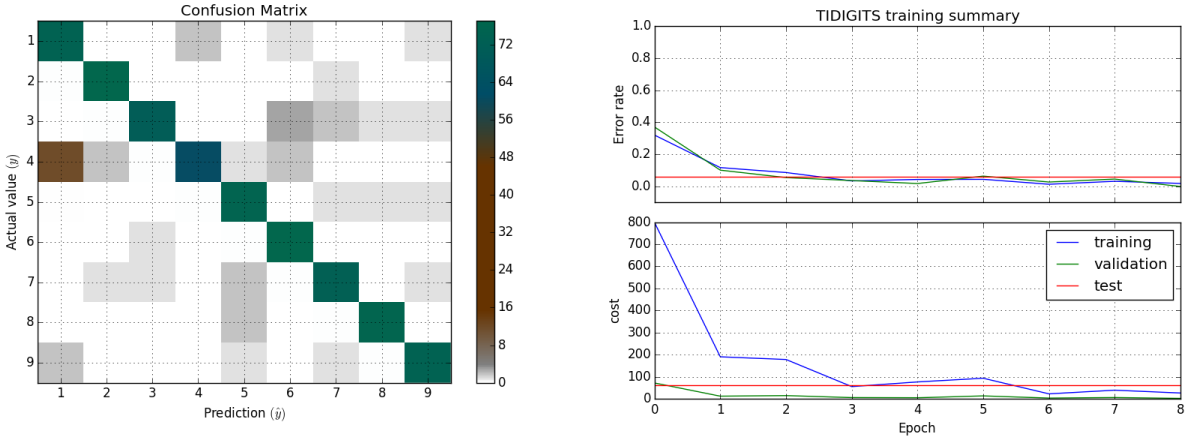

FIGURE 10 – Performances avec pré-traitement

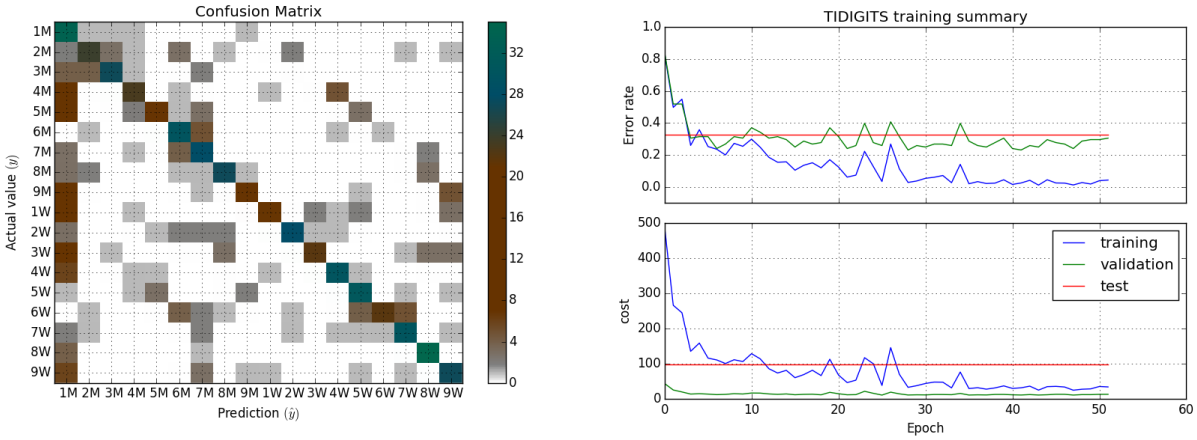

FIGURE 11 – Performances sans pré-traitement

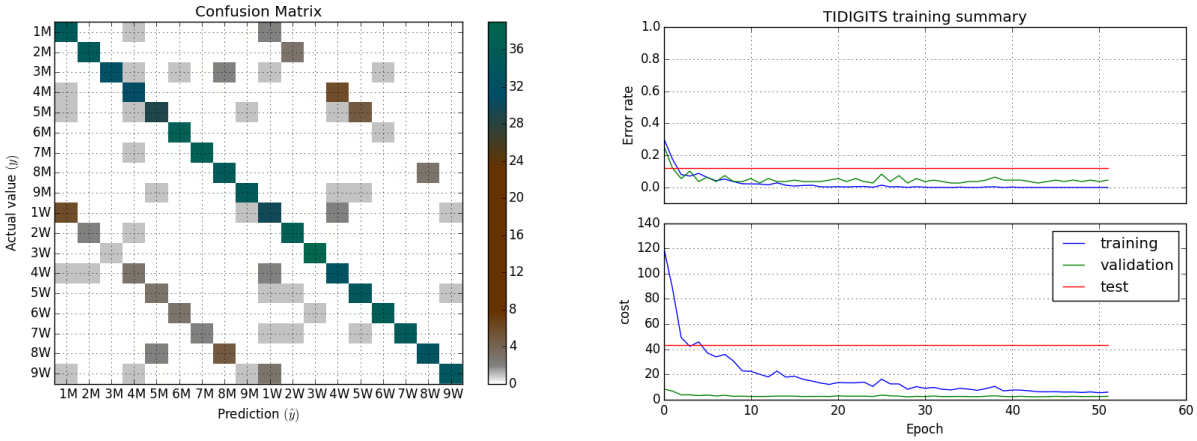

FIGURE 12 – Performances avec pré-traitement

B sauvegarde/chargement d'une configuration

```
In [1]: import network as n
In [2]: my_net = n.Network([3, 4, 2], activation='sigmoid',
                   cost='cross-entropy', learning_rate=0.5)
In [3]: print my_net
Neural Network      : [3, 4, 2]
Activation function : sigmoid
Cost function       : cross-entropy
Regularization func : none
learning rate       : 0.5
Regularization rate : 0.1

L0  * * *
L1 * * * *
L2   * *
In [4]: my_net.save('foo.save')
In [5]: my_net = n.Network([8, 4, 4, 2], activation='tanh',
   ...:                    cost='quadratic', learning_rate=0.02,
   ...:                    regularization='L2', lambda_ = 0.001)
In [6]: print my_net
Neural Network      : [8, 4, 4, 2]
Activation function : tanh
Cost function       : quadratic
Regularization func : L2
learning rate       : 0.02
Regularization rate : 0.001

L0 * * * * * * * * *
L1     * * * *
L2     * * * *
L3       * *
In [7]: my_net.load('foo.save')
In [8]: print my_net
Neural Network      : [3, 4, 2]
Activation function : sigmoid
Cost function       : cross-entropy
Regularization func : none
learning rate       : 0.5
Regularization rate : 0.1

L0  * * *
L1 * * * *
L2   * *
```

Listing 8 – Sauvegarde et chargement interactif d'une configuration

C Inspection du réseau

```
In [1]: import network as n
In [2]: my_net = n.Network((1, 2))
In [3]: my_net.load('foo.save')
In [4]: print my_net.struct
[3, 4, 2]

In [15]: print my_net.weights[0]
[[ 0.12737432  0.97734051 -0.56505148]
 [ 0.9090921   0.19178132 -0.15200818]
 [-0.07138651 -0.59903432 -0.78958921]
 [-0.55362229  0.51319807  0.2935551 ]]

In [16]: print my_net.weights[1]
[[-0.69519106  0.15539354 -0.46978019 -0.88703575]
 [-0.44994898 -0.57113777 -0.3736959  -0.00245537]]

In [17]: cat foo.save
# vim: set ft=yaml:
activation: sigmoid
biases:
- - [-0.920441751728251]
  - [-0.36244765002601287]
  - [-0.2926753984327689]
  - [-0.12034180777419923]
- - [1.1212748305651405]
  - [-0.565286303186786]
cost: cross-entropy
eta: 0.5
lambda: 0.1
regularization: none
struct: [3, 4, 2]
weights:
- - [0.12737431801447707, 0.9773405101355827, -0.5650514778232817]
  - [0.90909209744093, 0.19178132268834805, -0.1520081800115934]
  - [-0.07138650509286246, -0.5990343151487274, -0.7895892103519402]
  - [-0.5536222909987745, 0.5131980654305697, 0.29355509901743354]
- - [-0.6951910582048451, 0.1553935350379862, -0.4697801922017521,
      -0.8870357534796379]
  - [-0.44994897980803156, -0.5711377657299319, -0.3736959026341852,
      -0.002455368584577519]
```

Listing 9 – Exemple d'inspection interactive d'un réseau de neurones

D Visuels de l'interface graphique

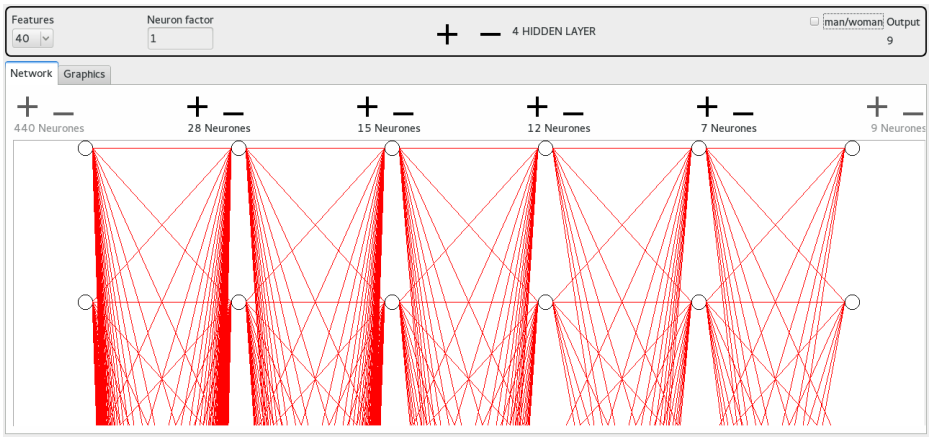

FIGURE 13 – Représentation visuelle du réseau

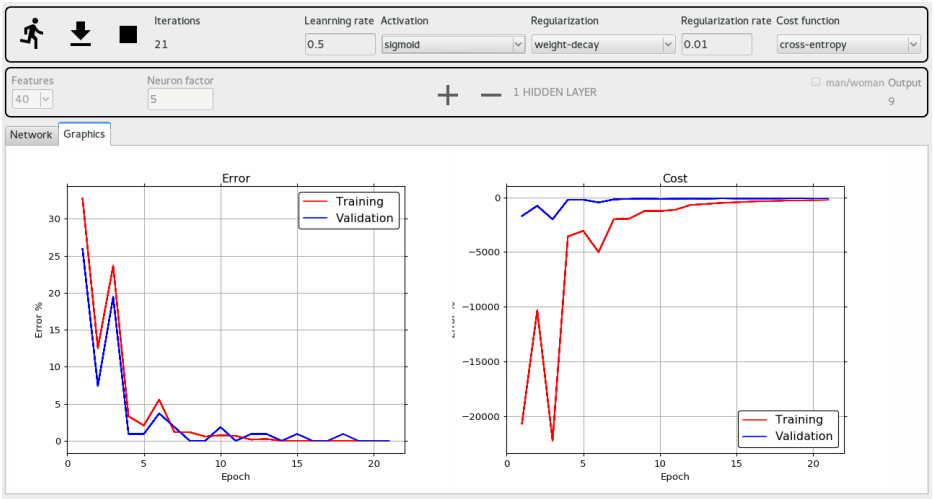

FIGURE 14 – Affichage des courbes d'erreur et de coût