Mini-Rapport

Exploration de la notion de méta-apprentissage

Dans quelle mesure un système apprenant peut prendre conscience de ses performances et altérer son comportement ?

Yann Boniface, Alain Dutech, Nicolas Rougier
Matthieu Zimmer

15 mai 2012

1 Introduction

Notre intérêt s'est tourné vers l'article [?] et ses 2 types de réseaux proposés. Dans un premier temps, nous avons cherché à reproduire et expliquer les résultats donnés, et ensuite à des solutions pour tirer profit des paris réalisés.
Nous nous sommes également penchés sur [?] dont nous avons reproduit les expériences, mais leurs enjeux nous semblent encore vagues.

2 Dupliquer le premier réseau

2.1 Les bases

En premier lieu, rappelons la structure des réseaux et les résultats de l'article :

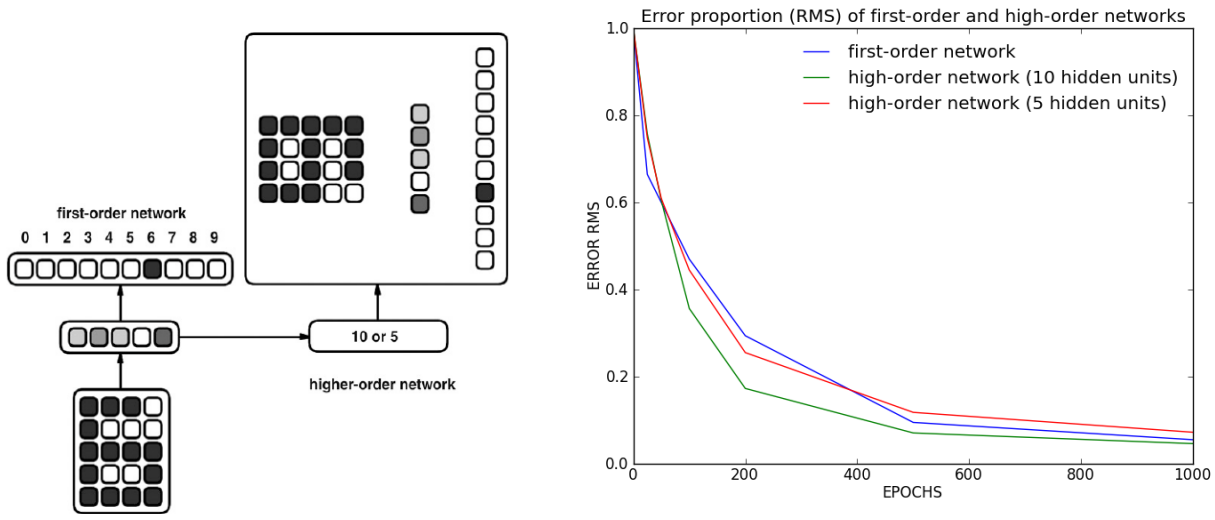

FIGURE 1 – [?] Architecture connexionniste avec méta-représentations

La formule RMS utilisée à une époque e est la suivante :

$$rms\ proportion_e = \frac{rms_e = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(o_{i,e} - d_i)^2}}{max(rms_{e'}),\ \forall e' \in epochs}$$

$$with \begin{cases} n : number\ of\ neurons\ on\ the\ output\ layer \\ o_{i,e} : value\ obtained\ for\ the\ i^{th}\ neuron\ at\ the\ e^{th}\ epoch \\ d_i : value\ desired\ for\ the\ i^{th}\ neuron \end{cases}$$

Nous avons décomposé les erreurs pour mieux comprendre le fonctionnement de l'architecture.

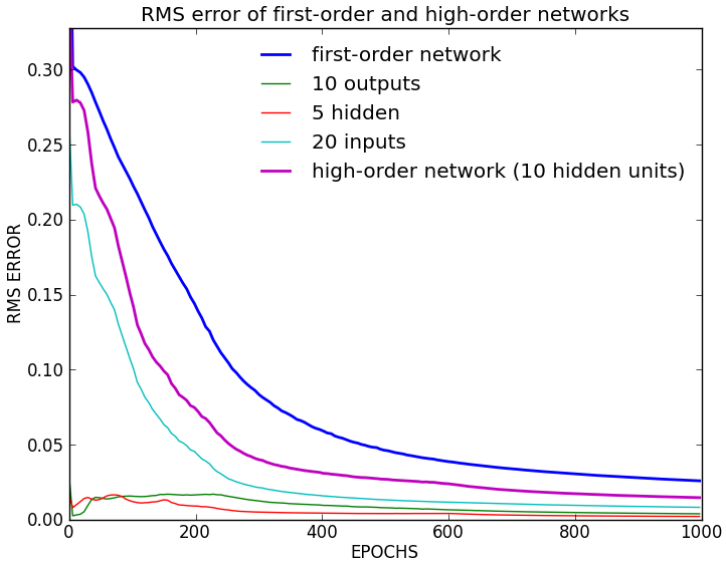

FIGURE 2 – **Erreur RMS sans proportion**. Le réseau supérieur à 5 unités cachées n'est plus considéré, et celui à 10 est découpé en 3 courbes pour représenter les 3 couches à reproduire. La courbe violette est donc la somme des 3 courbes des couches.

On peut en conclure 2 choses :

- la couche cachée et la couche de sortie ne posent aucun problèmes d'apprentissage
- les performances du second réseau dépendent principalement de sa capacité à reproduire les entrées

Il reste une question en suspend :

Pourquoi le second réseau apprendrait-il plus rapidement que le premier ?

Nous n'avons pas de réponse définitive. Cependant, nous pouvons nous intéresser à l'erreur de classification.

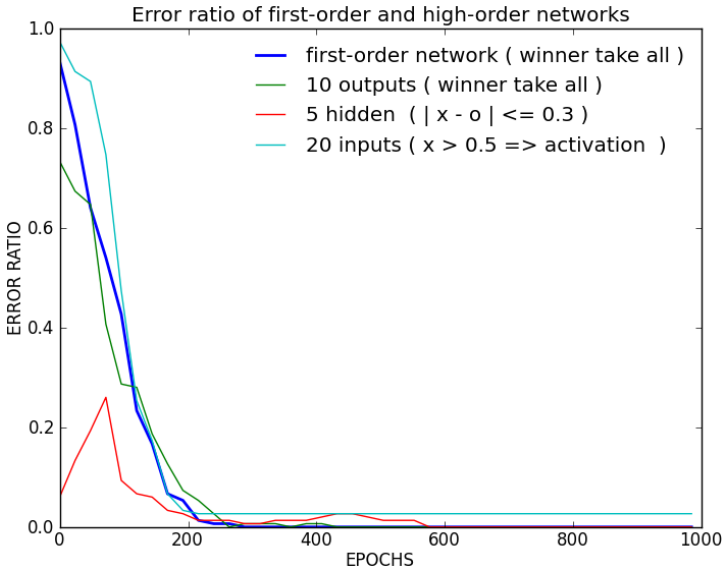

FIGURE 3 – **Erreur de classification**. Le réseau supérieur à 5 unités cachées n'est plus considéré, et celui à 10 est découpé en 3 courbes pour représenter les 3 couches à reproduire.

Il est alors beaucoup moins flagrant que le second réseau apprend mieux que le premier. Évidemment, parler de classification de la premier couche (courbe 20 inputs) et de la couche cachée (courbe 5 hidden) est relativement dérisoire.
La question de la rapidité d'apprentissage reste donc ouverte.

2.2 Les rouages

Notre attention s'est également portée sur les mécanismes qui permettaient la réalisation de cette architecture. Nous avons alors pu remarquer que les neurones de la couche cachée du premier réseau se stabilisaient très rapidement (autour de la $50^{\text{ième}}$ époque en moyenne), le tout permettant au second réseau d'avoir des entrées très peu variables, favorisant et permettant donc son apprentissage.

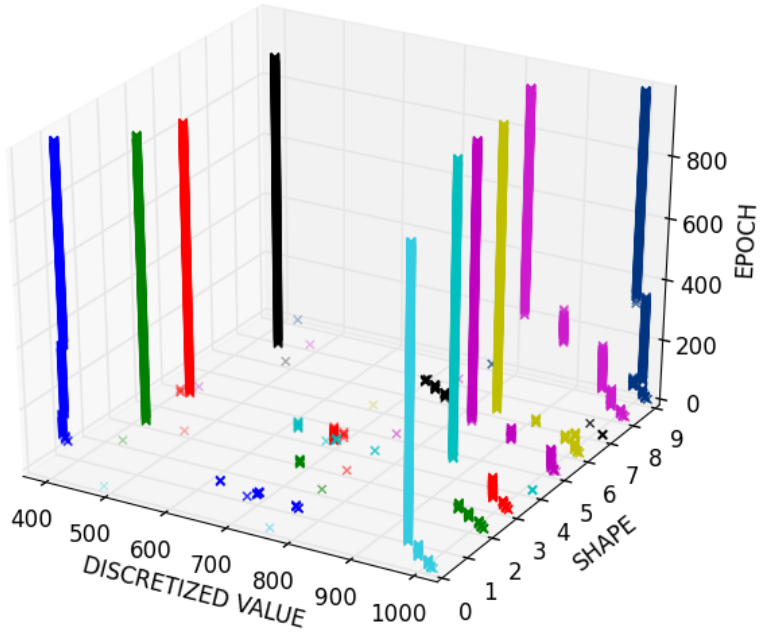

FIGURE 4 – Valeurs discrétisés de la couche cachée du premier réseau. Chaque couleurs représentant un des 10 chiffres en entrée. Les courbes deviennent rapidement stables.

2.3 Ré-haussement

Pour revenir à l'importance des entrées, nous avons réalisé que le second réseau n'était capable de dupliquer le premier qu'uniquement parce que ces entrées sont triviales. Ainsi nous avons augmenté le nombre d'entrées en passant sur des chiffres manuscrits [**?**], tout en augmentant proportionnellement le nombre de neurones des couches cachées :

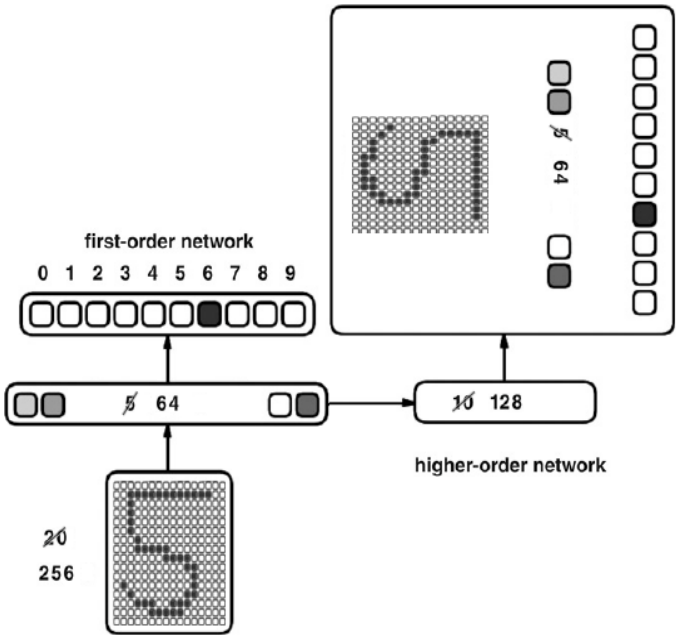

FIGURE 5 – Architecture avec méta-représentation pour chiffres manuscrits

Les performances du second réseau se sont alors écroulées :

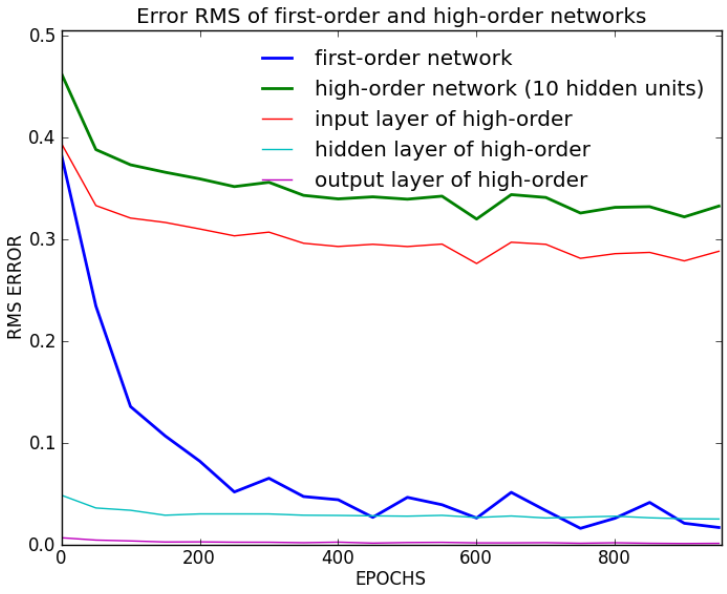

FIGURE 6 – **Erreur RMS sans proportion** de l'architecture sur des chiffres manuscrits. L'erreur du second réseau est toujours divisé en 3.

Il n'est alors plus capable de reproduire la couche d'entrée (qui a un poids très important).

Tentons maintenant d'expliquer cette chute :

La première chose à comprendre est que l'augmentation du nombre de neurone n'a rien à voir avec cette perte. En effet, nous avons fait les simulations avec les chiffres triviales (i.e. seulement 10 formes différentes) agrandit en 16×16 et tout se passe alors très bien.

Cette chute provient du fait que les entrées proposées sont très nombreuses et différentes. Il s'opère ainsi une première étape de classification dans la couche cachée du premier réseau. Ainsi différentes formes peuvent être représentées par la même couche cachée. Sauf que le second réseau n'ayant en entrée que la version factorisé (la couche cachée), il lui est impossible de retrouver les formes initiales : il est entraîné pour la même entrée, à différentes sorties. Il est donc naturel que les courbes soient désavantageuses, sauf que lorsqu'on regarde les formes, on n'est pas si loin de la réalité.

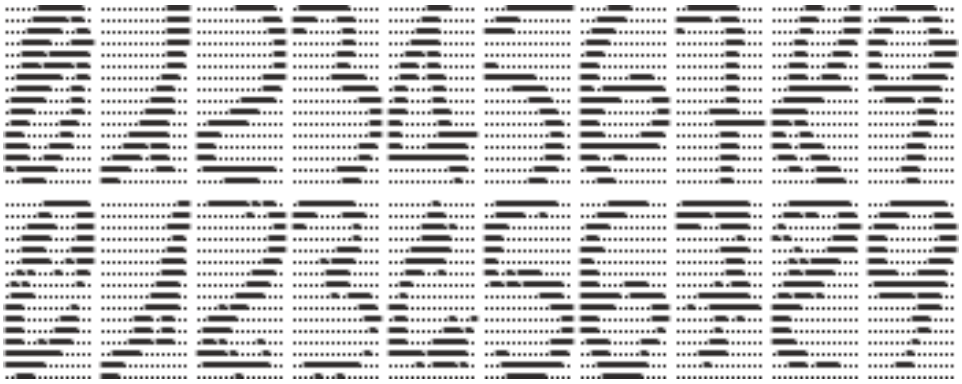

FIGURE 7 – En première ligne, 10 entrées aléatoires. En seconde, première partie de la sortie du second réseau

Si l'on est optimiste, on peut même considérer que ceux sont des représentations personnelles de l'agent pour chaque chiffre différents qu'il a vu jusqu'alors.

2.4 Représentations

Enfin, nous avons remarqué qu'en bloquant l'apprentissage entre la couche cachée et les entrées du premier réseau, puis en changeant de tâche, le réseau était capable de réapprendre la nouvelle tâche, ce qui prouve bien la présence d'une représentation des entrées dans la couche cachée.

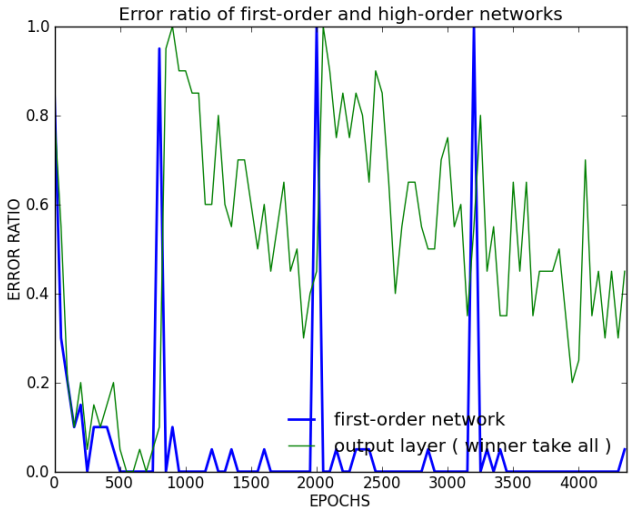

FIGURE 8 – **Erreur de classification** de l'architecture sur des chiffres manuscrits. Apprentissage bloqué à l'époque 800. Changement de tâche aux époques : 800, 2000, 3200

2.5 Remarque

Il faut cependant remarquer qu'un simple perceptron est suffisant pour réaliser la tâche du premier réseau (même sur les chiffres manuscrits), et donc, qu'il est possible que dans le cas d'un problème non linéairement séparable cette architecture soit invalidée.

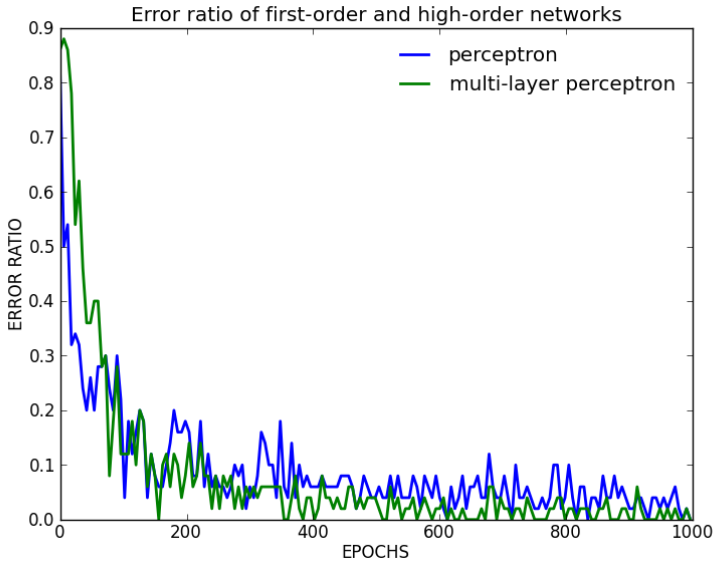

FIGURE 9 – **Erreur de classification** d'un perceptron et d'un perceptron multi-couches sur la base de chiffres manuscrits.

3 Parier sur le premier réseau

3.1 Les bases

Rappelons également la structure des réseaux et les résultats :

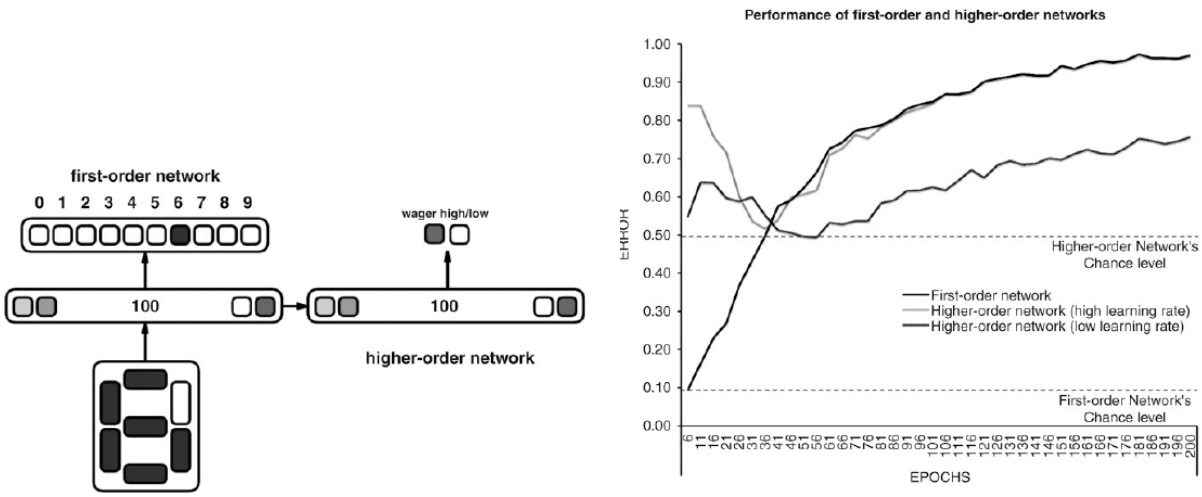

FIGURE 10 – [?] Architecture connexionniste avec paris

La première chose que nous avons faites à été d'améliorer les performances du second réseau en modifiant quelques paramètres (initialisation des poids sur [-1 ; -1], momentum à 0.5).

FIGURE 11 – **Performance de classification**. À gauche la base, à droite la version améliorée. On ne considère plus le réseau avec apprentissage faible, mais on a ajouté le taux de paris hauts.

Contrairement à celui de l'article, il ne se contentera plus simplement de parier haut à chaque coups (après 40 époques). Il aura une longueur d'avance sur le premier réseau sur toute la durée de l'apprentissage. On pourra donc tirer profit de cette avance.

3.2 Feedbacks

À partir de cette différence de performances, nous avons imaginé plusieurs architectures, qui améliorent plus ou moins les performances de reconnaissance du réseau sur des chiffres manuscrits :

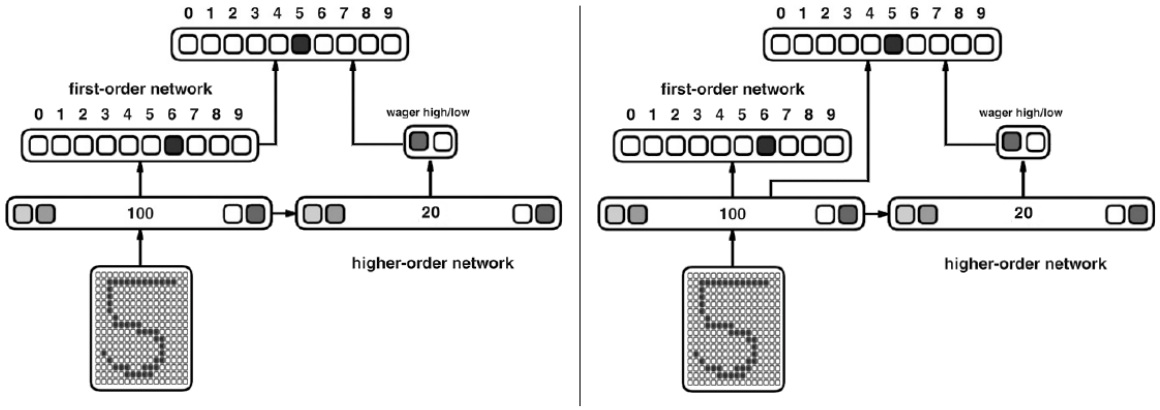

FIGURE 12 – Architecture avec 3$^{\text{ième}}$ réseau

Dans ces 2 architectures, nous nous contentons de connecter un 3$^{\text{ième}}$ réseau qui doit tirer des conclusions à partir d'informations sur les 2 premiers.

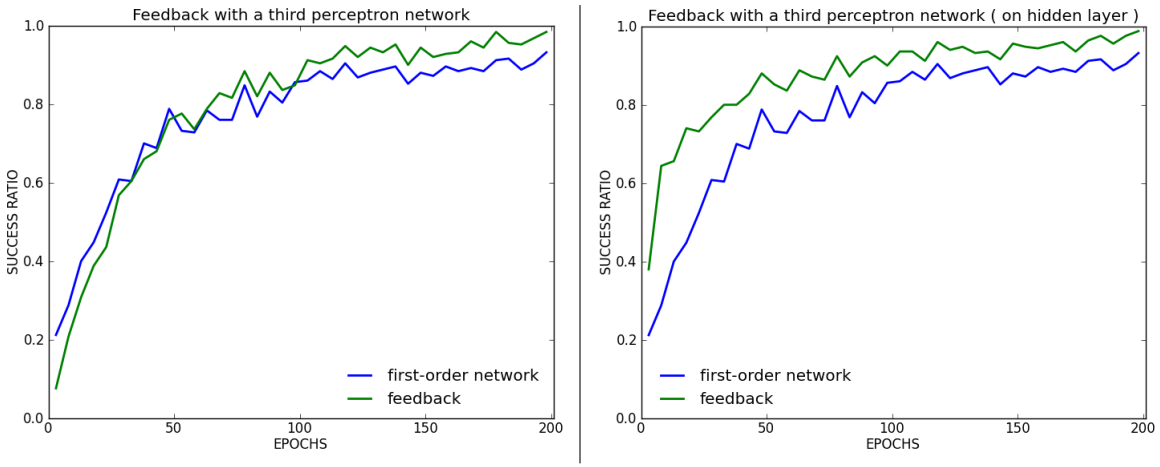

FIGURE 13 – **Performance de classification** des 2 architectures.

Les deux architectures apportent un gain. Il est plus accentué lorsque la dernière couche dispose de plus d'informations (i.e. connecté sur la couche cachée).

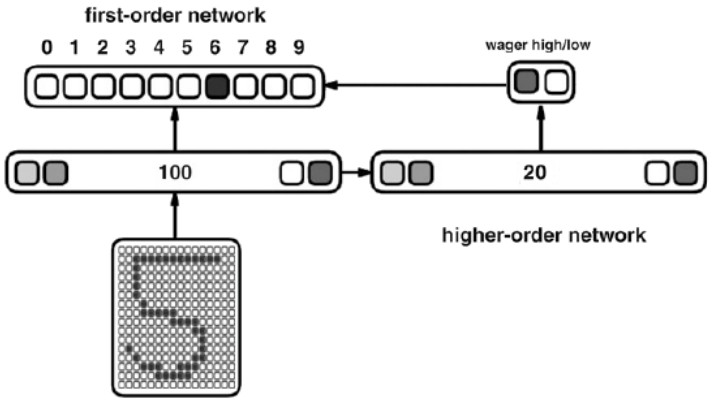

FIGURE 14 – Architecture par fusion

Ici, nous mélangeons un apprentissage par descente de gradient (sur le premier et second réseau) et un apprentissage perceptron (entre les 2 couches de sorties). L'algorithme peut être trouvé dans l'annexe A.

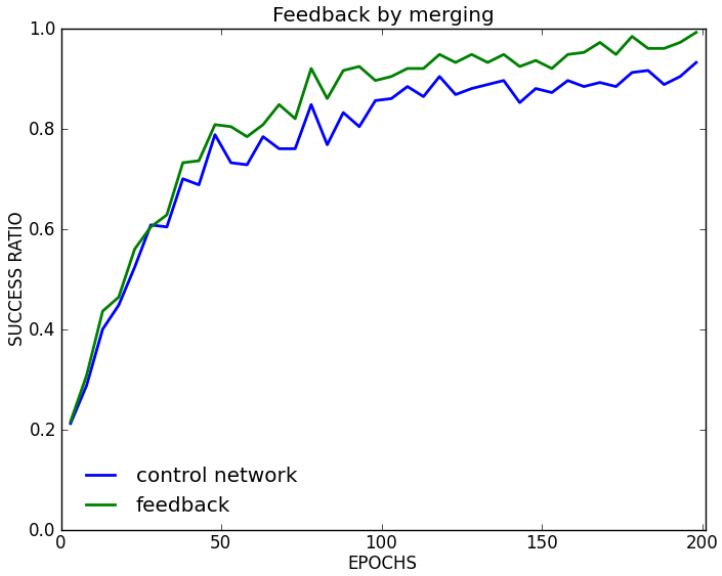

FIGURE 15 – **Performance de classification** de l'architecture fusion.

Le gain est aussi notable d'autant qu'il ne nécessite aucun ajout de neurone. Il est d'ailleurs assez similaire au gain de la première architecture, puisqu'elle est aussi basée sur les 2 couches de sorties des 2 réseaux (néanmoins, le début diffère, mais c'est le temps que les perceptron apprennent).

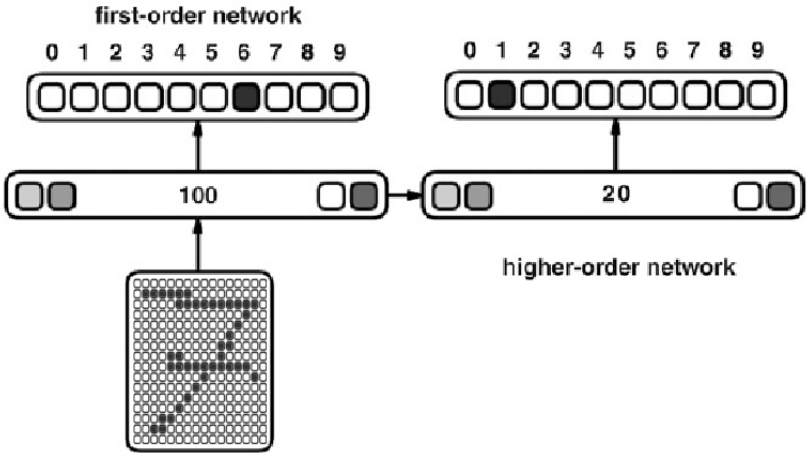

FIGURE 16 – Architecture par intuitions

Cette architecture est légèrement différente dans le sens où elle n'enregistre plus de pari mais l'indice du n$^{\text{ième}}$ neurone le plus actif contenant la bonne réponse. Exemple : le réseau supérieur sort 2 -> la réponse est le 3$^{\text{ième}}$ neurone le plus actif de la couche de sortie du premier réseau.

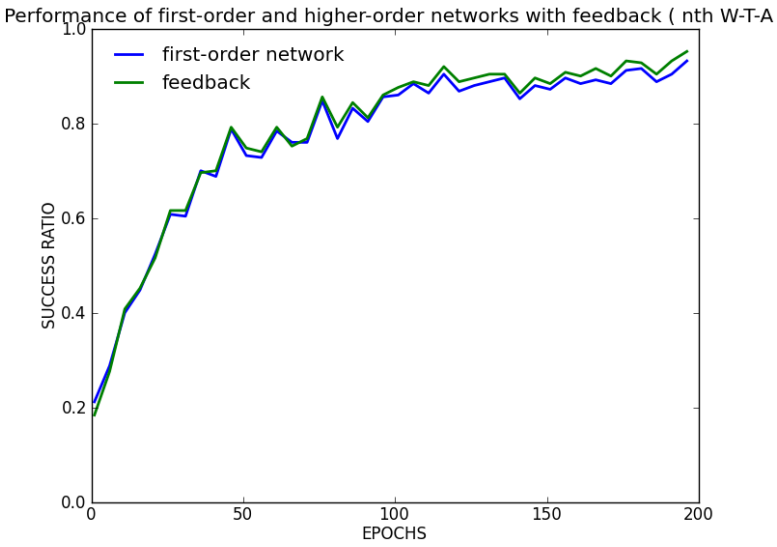

FIGURE 17 – **Performance de classification** de l'architecture intuition.

Dans ce cas malheureusement, les performances sont très peu améliorées.

Nous avons aussi essayé quelques modèles où le réseau supérieur servait de superviseur à l'apprentissage du premier réseau. Par exemple, s'il parie haut, l'apprentissage du premier réseau sera faible, sinon il sera accentué.

4 La suite

Ce que nous continuons d'étudier :

- validation sur des expériences plus complexes (qui ne peuvent être résolue directement par un perceptron)
- relation entre la taille de la couche cachée du premier réseau et le taux de paris avantageux
- approfondir les intérêts du second article [**?**]
- de nouvelles architectures axées méta-apprentissage où le réseau d'ordre supérieur contrôle le premier (taux d'apprentissage, momentum, entrées à approfondir, ...)

A Algorithme fusion gradient/perceptron

Ensemble d'instructions exécuté pour une époque et un tirage d'entrées/sorties à apprendre.
Cet algorithme se déroule en plusieurs temps :

a) On calcule la sortie du premier réseau en ignorant le second (valeur à 0)
b) On calcule la sortie du second réseau
c) On calcule la sortie du premier réseau avec les sorties de b)
d) Les performances sont calculées avec c)
e) Le second réseau apprend à parier sur le résultat a) (sans prendre compte de lui-même)
f) Le premier réseau apprend à discriminer en prennent compte de b)

$first\_order.calc\_hidden\_layer(samples.inputs)$
$high\_order.calc\_output\_layer(first\_order.hidden\_layer)$
$first\_order.calc\_output\_layer(first\_order.hidden\_layer,\ [0, ..., 0])$

$h\_output \leftarrow ampli(high\_order.output\_layer)$
$right\_houtput \leftarrow [0,\ 0]$
if $good\_answer(first\_order)$ **then**
    $right\_houtput[1] \leftarrow 1$
else
    $right\_houtput[0] \leftarrow 1$
end if
$first\_order.calc\_output\_layer(first\_order.hidden\_layer,\ h\_output)$

$calc\_stats()$

$high\_order.train(first\_order.hidden\_layer,\ right\_houtput)$
$first\_order.train(samples.inputs,\ samples.outputs,\ h\_output)$

Intéressons-nous justement à l'étape f) :

function $train(inputs,\ outputs,\ add)$
    for $i = 0 \rightarrow output\_neurons.length$ **do**
        $y_{output}[i] \leftarrow g'(output\_neurons[i].a) \times (outputs[i] - output\_neurons.state)$
    end for

    for $i = 0 \rightarrow hidden\_neurons.length$ **do**
        $w\_sum \leftarrow \sum_{j=0}^{output\_neurons.length} output\_neurons[j].weights[i] \times y_{output[j]}$
        $y_{hidden}[i] \leftarrow g'(hidden\_neurons[i].a) \times w\_sum$
    end for
    $update\_weights\_hidden\_layer(y_{hidden})$

for $i = 0 \rightarrow output\_neurons.length$ **do**
$output\_neurons[i].update\_weights\_gradient(y_{output}[i],\ hidden\_neurons,\ add)$
$output\_neurons[i].update\_weights\_perceptron(outputs[i],\ hidden\_neurons,\ add)$
end for
end function

La différence principale se situe sur le dernier for, avec les 2 méthodes de mise à jours des poids.

function $update\_weights\_gradient(error,\ intputs,\ add)$
$calc\_output(inputs + add)$

for $j = 0 \rightarrow inputs.length$ **do**
$dw \leftarrow weights[j] - last\_weights[j]$
$p \leftarrow error \times inputs[j]$
$weights[j] \leftarrow weights[j] + learning\_rate \times p + momentum \times dw$
end for
end function

function $update\_weights\_perceptron(goal,\ intputs,\ add)$
$calc\_output(inputs + add)$

for $j = inputs.length \rightarrow inputs.length + add.length$ **do**
$dw \leftarrow weights[j] - last\_weights[j]$
$p \leftarrow (goal - state) \times add[inputs.length - j]$
$weights[j] \leftarrow weights[j] + \frac{learning\_rate \times p + momentum \times dw}{add.length}$
end for
end function