# Projet Python - Reseau de Neurones

April 29, 2020

## 1 Introduction

### 1.1 Objectif et précisions

L'objectif de cette étude est de coder un réseau de neurones à 2 couches (2-Layers NN). Ce réseau doit nous permettre de résoudre un problème de classification supervisé multiclasses : pour chaque image, notre modèle doit assigner un label parmi dix (avion, voiture, oiseau etc..)

Voici quelques précisions importantes concernant la réalisation de ce projet :

- Il est basé sur l'**Assignment 1) Q4** du cours CS231n de Stanford : "Deep Learning for image recognition". Voici le lien si vous souhaitez consulter directement l'énoncé https://cs231n.github.io/assignments2020/assignment1/
- Nous utiliserons le jeu de données CIFAR-10 que nous importerons depuis le module TensorFlow. Ce jeu de données étant particulièrement volumineux, nous allons devoir utiliser Google Colab pour faire tourner notre code sur des serveurs et utiliser leurs CPUs et GPUs. Colab permet d'utiliser gratuitement les GPU Tesla K80 (pour plus d'infos sur Colab -> https://medium.com/deep-learning-turkey/google-colab-free-gpu-tutorial-e113627b9f5d)
- Le code que j'ai réalisé se trouve sur 2 fichiers : `two_layer_net.ipynb` (qui contient l'énoncé et les questions de l'exercice) et `cs231n/classifiers/neural_net.py`.

La partie codée par mes soins est facilement identifiable car elle se trouve à chaque fois entre les deux inscriptions ci-dessous :

**START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)**

MON CODE

**END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)**

### 1.2 Description du jeu de données

Après ces quelques détails décrivons brièvement notre jeu de données : CiFAR-10 contient 60000 images de taille 32x32 en couleur. Ces images sont réparties en 10 classes. On a donc 6000 images par classes.

Voici un aperçu de ces 10 classes avec 10 images prises au hasard pour chaque classe :

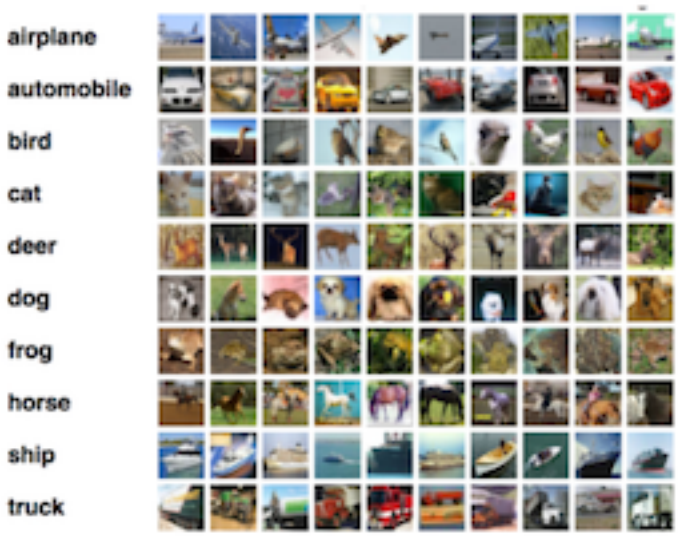


Fig 1 - Aperçu du jeu de données CIFAR-10

### 1.3 Pré-traitement

Les images sont d'abord normalisées : on leur soustrait l'image moyenne. L'image moyenne consiste à prendre le pixel moyen sur les 3 dimensions de toutes les images du jeu de données.

Ensuite, on transforme nos images de dimension 32 x 32 x 3 en ligne comme illustré ci-dessous :

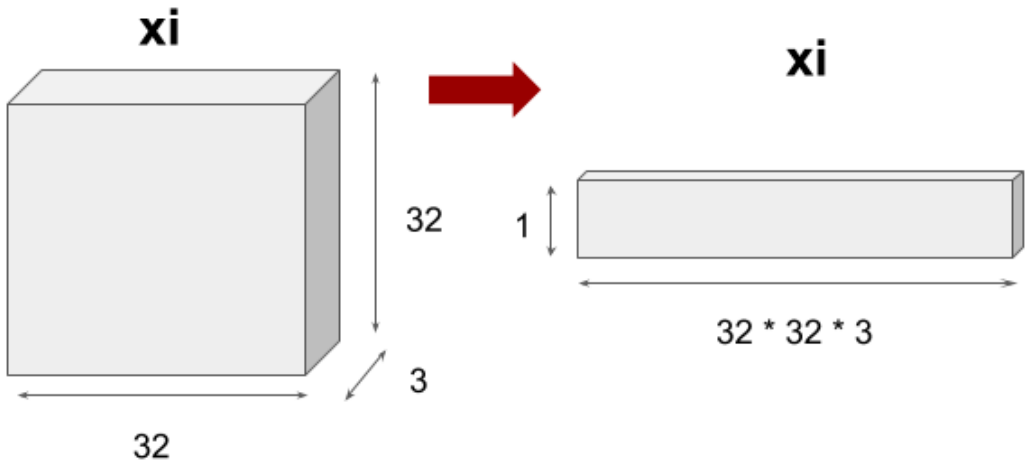


Fig 2 - Transformation de l'image en ligne

Enfin, on sépare notre jeu de données initial en 6 jeux de données comme indiqué ci-dessous :

- X_train (49000, 3072), y_train (49000,1)
- X_val (1000,3072), y_val (1000, 1)
- X_test (1000, 3072), y_test (1000, 1)

## 2 2-Layers Neural Network

L'objectif de cette étude est de réaliser un réseau de neurones à 2 couches. Notre réseau de neurones va prendre en entrée une image et devra prédire en sortie le label associé à l'image.

Un réseau de neurones à 2 couches peut être représenté de la manière suivante :

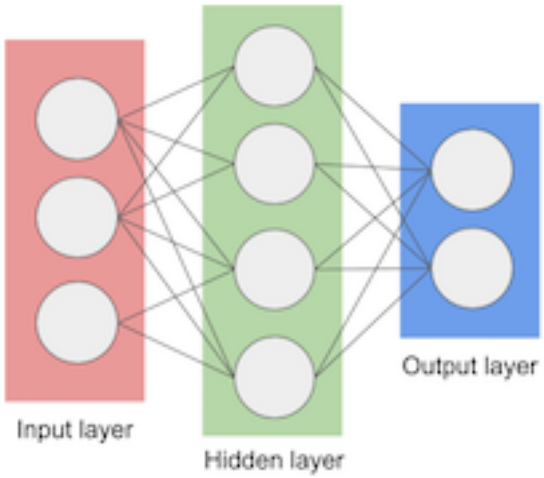


Fig 3 - 2-layers NN

On note qu'un réseau de neurone à 2 couches en contient en réalité 3 si l'on considère l'entrée. La convention veut qu'on omette celle-ci lorsque l'on désigne le réseau de neurones. Aussi, dans notre exemple le nombre de neurones est différent de celui de l'image. (Par exemple nous avons 10 neurones pour la dernières couches et non pas 2 comme sur l'exemple).

**Principe général :**

Un réseau de neurone est simplement une composition de fonctions linéaires et non-linéaires. Les fonctions non-linéaires sont appelées activations. Il en existe plusieurs, dans notre exemple nous prendrons la fonction **ReLU** : $x \mapsto max(x, 0)$. Notre réseau de neurones correspond à la fonction f :

$$f = W_2.max(0, W_1.x + b_1) + b_2$$

.

où $x$ correspond à nos images en entrée, $(W_1, b_1)$ désignent les poids et le biais de la couche 1, $(W_2, b_2)$ les poids et le biais de la couche 2. On notera $s_1 = W_1.x + b_1$ par la suite. Une fois $f$ réalisée notre réseau calcule la fonction de coût (loss function). Dans notre exemple nous prenons la fonction de coût softmax qui pour une observation, se traduit par :

$$L_i = -log(\frac{e^{s_{yi}}}{\sum_j e^{s_j}})$$

On précise que $s$ correspond aux scores d'une observation ($s = w.x + b$). $s_{yi}$ correspond au score de la vraie classe et $j$ parcourt tous les scores (10 au total). L'idée est de trouver le minimum de

cette fonction de coût de l'ensemble des observation pour obtenir la meilleure précision dans nos prédictions. En d'autres termes, il faut minimiser :

$$L = \frac{1}{N} \sum_i L_i(w) + \alpha R(w)$$

avec $\alpha$ le coefficient de pénalisation (regularization rate), N le nombre d'observations et $R(w)$ la fonction de pénalisation (regularization). Cette dernière permet d'éviter le surapprentissage. On prendra la pénalisation $L2$ dans notre exemple.

Pour minimiser la fonction $L$, on peut penser à calculer numériquement la dérivée de la fonction de coût par rapport aux poids W. Cette opération se révèle coûteuse en mémoire et en temps de calcul dans la pratique. On utilisera tout de même le calcul du gradient numérique pour vérifier notre gradient analytique. En effet, nous allons calculer notre gradient de manière analytique en utilisant la chain rule que nous décrirons dans la partie backpropagation.

## 3 Forward pass

Voici une illustration (computational graph) de notre réseau de neurones :

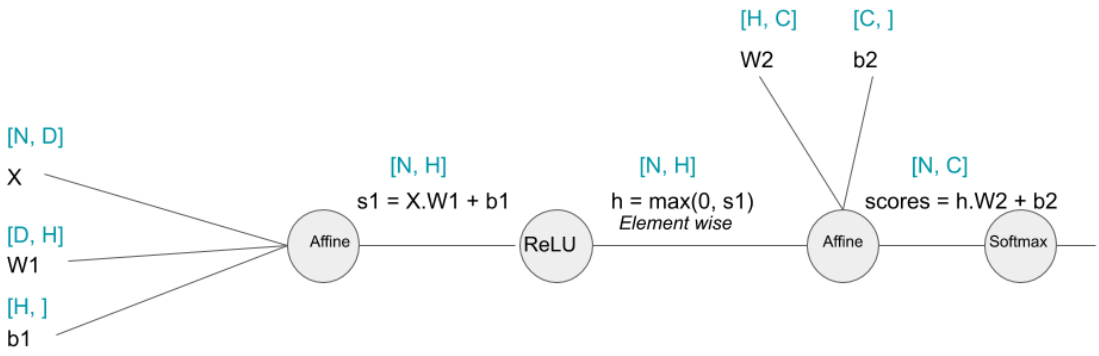


Fig 4 - Forward pass - computational graph

On précise qu'ici C désigne le nombre de classe (et non pas la constante de normalisation évoquée plus haut)

On commence par calculer les scores comme indiqué sur le computational graph. Cela correspond à la partie "Forward pass: compute scores" du fichier `two_layer_net.ipynb`. On vérifie que nos scores sont les bons. Ensuite, dans la section suivante "Forward pass: compute loss", on calcul la fonction de coût softmax. En pratique on peut rencontrer des problèmes de stabilités numériques avec cette fonction. Les exponentielles peuvent devenir très grandes. Pour éviter ce problème, on va utiliser l'astuce de normalisation ci dessous. On vient multiplier le numérateur et le dénominateur par une constante C :

$$\frac{Ce^{s_{yi}}}{C(\sum_j e^{s_j})} = \frac{e^{s_{yi}+logC}}{\sum_j e^{s_j+logC}}$$

Un choix classique revient à prendre $logC = -max_j(s_j)$

Une fois cette pré-opération effectuée, on obtient la matrice des probabilités d'appartenance à chaque classe pour nos N observations comme illustrée ci-dessous :

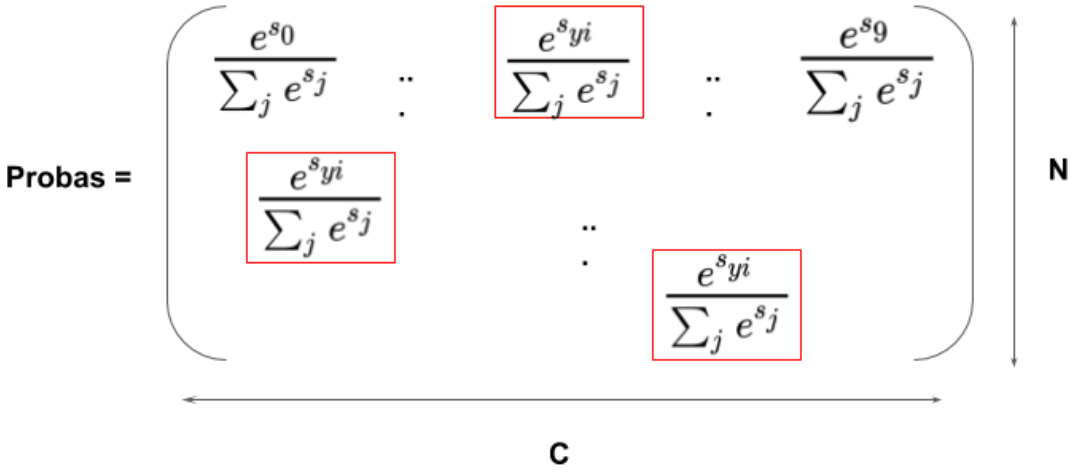


Fig 5 - Matrice des probabilités d'appartenance

On rappelle que la fonction de coût pour une observation est :

$$L_i = -log(\frac{e^{s_{yi}}}{\sum_j e^{s_j}})$$

On remarque donc que la fonction de coût ne prend en compte que la probabilité de la vraie classe. On sélectionne donc uniquement ces probabilités auxquelles on applique la fonction log. Enfin, on somme chaque $Li$ puis on divise par N pour obtenir la fonction de coût totale. On n'oublie pas le terme de pénalisation à la fin

**Note :** Il existe un moyen simple de tester la fonction de coût softmax. En effet, si on prend $W$ approximativement égal à 0 alors les scores valent 0 et $L = -log(\frac{1}{C})$ où C étant le nombre de classes.

## 4 Backward pass

On s'attaque maintenant à la phase de Backward propagation. J'ai fait le choix de particulièrement détailler cette partie afin de justifier mathématiquement la réalisation du code. On rappelle que l'idée est de minimiser la fonction de coût en la dérivant. Plus précisement, on va effectuer une descente de gradient pour obtenir la fonction de coût minimale. On veut calculer :

$$\Delta_w L = \frac{1}{N} \sum_i \Delta_w L_i + \alpha \Delta_w R(w)$$

Notons que l'on peut réécrire la fonction de coût de la manière suivante :

$$L_i = -log(\frac{e^{s_{yi}}}{\sum_j e^{s_j}}) = -s_{yi} + log(\sum_j e^{s_j})$$

Voici un aperçu de la phase de backpropagation et des gradients à calculer :

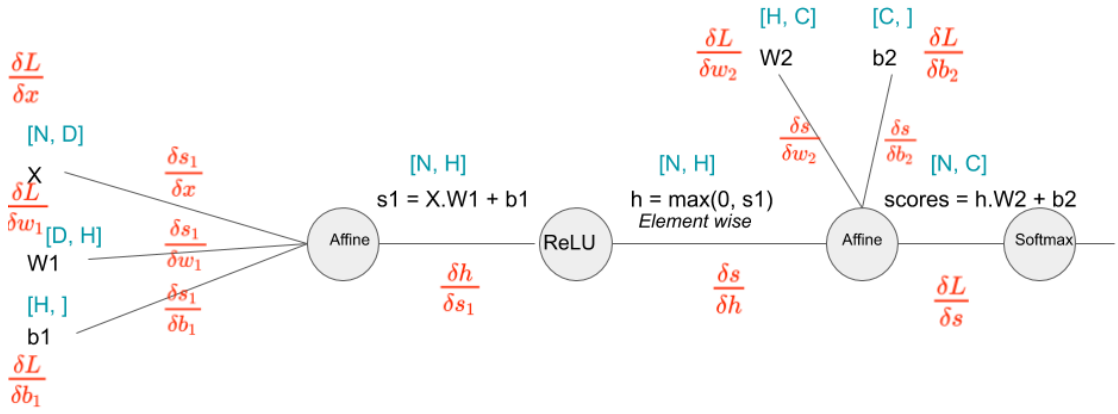


Fig 6- Backward pass - computational graph

### 4.1 Gradient dW2

$$\Delta_{w_2} L = \frac{1}{N} \sum_i \Delta_{w_2} L_i + \alpha * 2 * w_2$$

or $\Delta_{w_2} L_i = \frac{\delta L_i}{\delta s} \frac{\delta s}{\delta w_2}$

$$\frac{\delta L_i}{\delta s} = \begin{cases} \frac{\delta L_i}{\delta s_{y_i}} = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} - 1 \\ \frac{\delta L_i}{\delta s_{s_j}} = \frac{e^{s_j}}{\sum_l e^{s_l}} \end{cases} \tag{1}$$

On remarque que $\frac{\delta L}{\delta s}$ correspond à la matrice des probabilités d'appartenance de chaque classe pour chaque observation à un détail près : on enlève 1 aux probabilités des vrais classes. On la note 'margin2' dans le code.

et $\frac{\delta s}{\delta w_2} = h$ donc $\Delta_{w_2} L_i = \frac{\delta L_i}{\delta s} h$

### 4.2 Gradient db2

On s'intéresse cette fois ci au gradient du biais $b_2$ :

$$\frac{\delta L_i}{\delta b_2} = \frac{\delta L_i}{\delta s} \frac{\delta s}{\delta b_2} = \frac{\delta L_i}{\delta s} \mathbf{1}$$

où $\mathbf{1}$ est une matrice de 1 de dimension [N, ]

### 4.3 Gradient dW1

De même,

$$\Delta_{w_1} L = \frac{1}{N} \sum_i \Delta_{w_1} L_i + \alpha * 2 * w_1$$

A nouveau, on utilise la chain rule :

$$\Delta_{w_1} L = \frac{\delta L}{\delta w_1} = \frac{\delta L}{\delta s}\frac{\delta s}{\delta h} * \frac{\delta h}{\delta s_1}\frac{\delta s_1}{\delta w_1}$$

On précise qu'ici * désigne le produit terme à terme. On a déjà calculé $\frac{\delta L}{\delta s}$ à l'étape précédente et :

$$\frac{\delta s}{\delta h} = w_2$$
$$\frac{\delta h}{\delta s_1} = \frac{\delta(max(0, s_1))}{\delta s_1} = \mathbb{1}_{s_1 \geq 0}$$
$$\frac{\delta s_1}{\delta w_1} = x$$

Par conséquent,

$$\Delta_{w_1} L = \frac{\delta L}{\delta w_1} = \frac{\delta L}{\delta s} w_2 * \mathbb{1}_{s_1 \geq 0} x$$

### 4.4 Gradient db1

Toujours en utilisant la chaine rule :

$$\Delta_{b_1} L = \frac{\delta L}{\delta b_1} = \frac{\delta L}{\delta s}\frac{\delta s}{\delta h} * \frac{\delta h}{\delta s_1}\frac{\delta s_1}{\delta b_1}$$

On connaît déjà tous les termes sauf $\frac{\delta s_1}{\delta b_1} = \mathbf{1}$ un vecteur de 1 de dimension [N,]

$$\Delta_{w_1} L = \frac{\delta L}{\delta w_1} = \frac{\delta L}{\delta s} w_2 * \mathbb{1}_{s_1 \geq 0} \mathbf{1}$$

Enfin, on vérifie la valeur de nos gradients en les comparant avec les gradients numériques (fonction pré-implémentée dans l'exercice).

## 5 Entrainement du réseau de neurones

### 5.1 Descente de gradient stochastique

La section précédente nous a permis d'évaluer le gradient de la fonction de coût par rapport aux paramètres. On peut désormais réaliser la descente de gradient que l'on peut résumer de la manière suivante :

```
while True:
    dw = evaluate_gradient(loss, data, weights)
    w += - lambda * dw
```

où lambda désigne le taux d'apprentissage (learning rate).

On rappelle que :

$$\Delta_w L(w) = \frac{1}{N}\sum_i \Delta_w L_i(s_i, w) + \alpha \Delta_w R(w)$$

où $s_i = x_i.w + b$

Evaluer le gradient pour chaque $x_i$ peut s'avérer coûteux en calcul et en temps. En pratique, on utilise donc la descente de gradient stochastique : plutôt que de calculer la fonction de coût sur le jeu de données en entier, on constitue un échantillon à chaque itération qu'on appelle mini-batch. On évalue ainsi notre gradient non plus sur le jeu de données en entier mais sur un échantillon.

### 5.2 Premiers résultats

Une fois la descente de gradient implémentée, notre fonction de coût doit donc diminuer à chaque itération et notre modèle doit donc devenir de plus en plus performant. Le choix des hyperparamètres $\alpha$ (coefficient de pénalisation), $\lambda$ (taux d'apprentissage) et le nombre de neurones dans la première couche (hidden size H) est important pour obtenir de bonnes performances. Jetons d'abord un oeil à nos premiers résultats sans se soucier du choix de ces hyperparamètres :

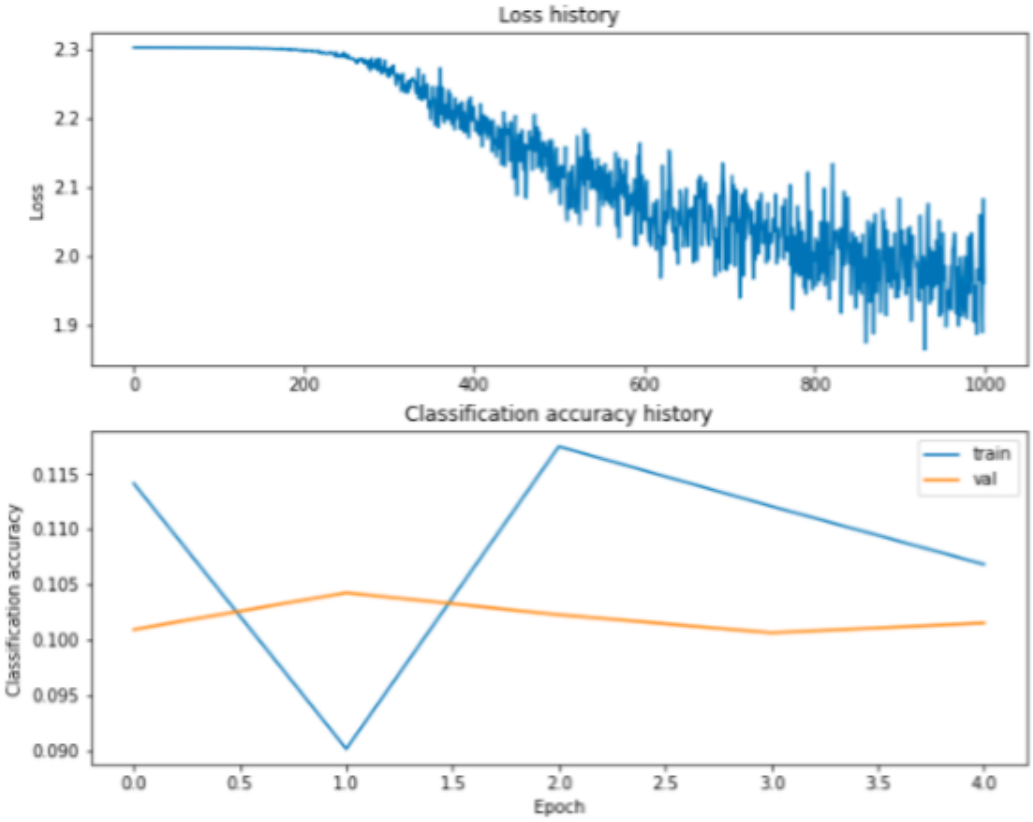


Fig 7, 8 - Premières performances du modèle

Comme indiqué dans l'énoncé, on observe que la fonction de coût diminue de façon quasi linéaire. Notre taux d'apprentissage est trop faible. De plus, les précisions d'entrainement (train acc) et de validation (val acc) sont très proches. On peut donc augmenter légèrement la complexité du modèle pour éviter ce léger sous apprentissage. On notera que les deux précisions sont très faibles, environ 10%.

On peut aussi observer les poids de la première couche :

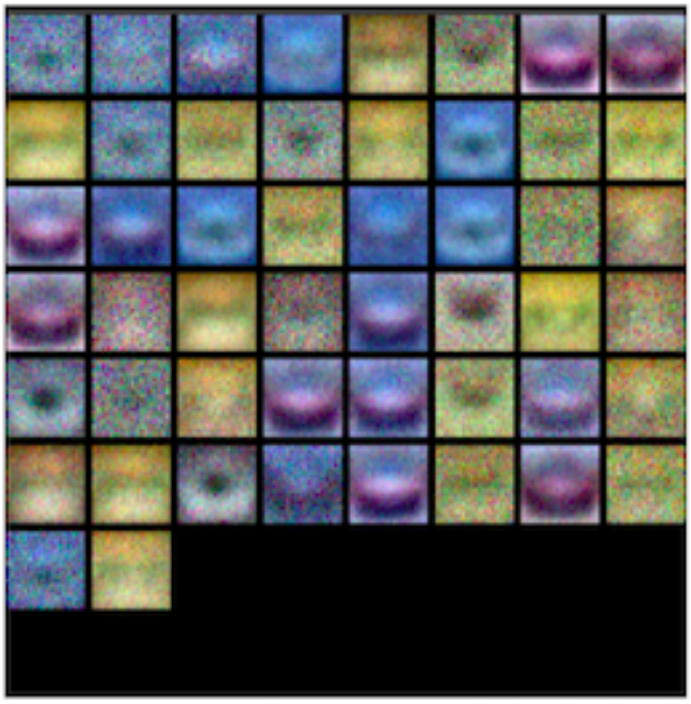

Fig 9- aperçu des poids de la première couche après apprentissage

### 5.3 Hyperparameters tuning :

Nous allons tenter de trouver les hyperparamètres qui vont offrir les meilleurs performances pour notre modèle. Attention, cette fois-ci le code à compléter se trouve directement sur le fichier `two_layer_net.ipynb`, section "Tune your hyperparameters". 3 hyperparamètres sont à considérer :

- $\alpha$ : coefficient de pénalisation
- $\lambda$ : taux d'apprentissage
- hidden size H : le nombre de neurones dans la première couche

La procédure est la suivante : pour chaque paramètre, on va séléctionner aléatoirement des valeurs dans un interval. On commence par prendre un interval large et des valeurs dans une échelle logarithmique. Une fois qu'on a une meilleure idée de la valeur du paramètre on réduit l'intervalle et on réitère l'opération. Cette recherche aléatoire de paramètres optimaux permet de couvrir un éventail plus large de valeurs comme illustré sur la figure ci-dessous :

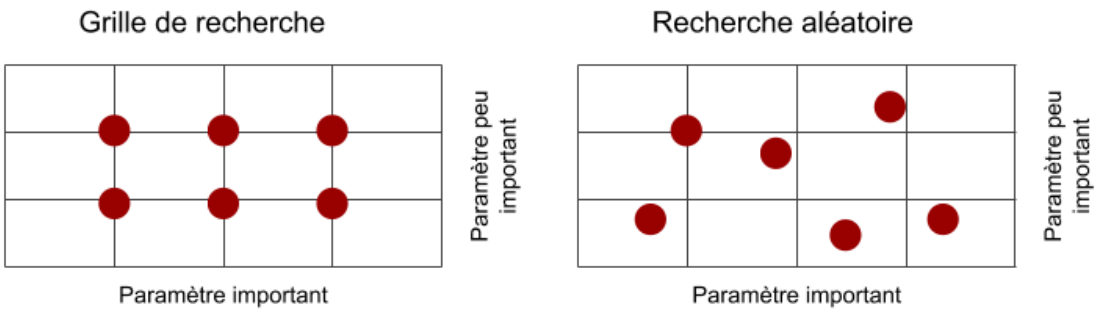


Fig 10 - Comparaison des 2 méthodes de gridSearch

On obtient finalement nos meilleurs hyperparamètres, c'est à dire ceux qui maximisent la précision de validation. On affiche à nouveau les poids de la première couche :

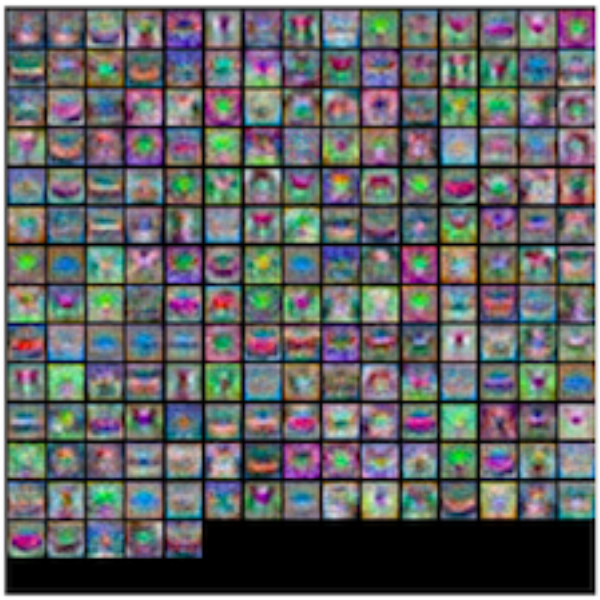

Fig 11 - aperçu des poids de la première couche après apprentissage et optimisation

On remarque certains contours orientés, beaucoup plus précis que nos précédents poids. On remarquera que l'on a augmenté la taille de la première couche.

Finalement, on observe les performances de notre modèle sur le jeu de données test : on obtient une précision de 11% ce qui est toujours très faible. Bien que notre modèle se soit très légèrement amélioré et que l'affichage des poids après modification des hyperparamètres soit encourageant, je n'ai malheureusement pas réussi à obtenir une précision de 50% comme indiqué dans l'énoncé.