Deep Learning

Paul Lehaut

October 7, 2025

Contents

1 Bases du Machine Learning **3**
1.1 Types d'apprentissage . . . 3
1.2 Apprentissage Supervisé Paramétrique . . . 3

2 Premier Résau de Neurones **3**
2.1 Perceptron Multicouche (MLP) . . . 3
2.2 Optimisation . . . 4
2.3 Hyperparamètres Clés de l'Optimisation . . . 4

3 Backpropagation **5**
3.1 Formalisation . . . 5

4 Résaux de Neurones Convolutifs (CNN): Achitectures LeNet et AlexNet **5**
4.1 Principe de la Convolution . . . 6
4.2 Autres Couches Importantes . . . 6
4.3 LeNet: Le Premier CNN . . . 7
4.4 AlexNet: La Révolution de 2012 . . . 7
4.4.1 L'intérêt des Couches Conv2d, ReLU et MaxPool2d . . . 10
4.4.2 L'intérêt des Couches Entièrements Connectées . . . 10

5 A Retenir Des Quizzs **11**
5.1 Quizz 1 . . . 11

1 Bases du Machine Learning

Le Machine Learning est une discipline qui vise à concevoir des algorithmes généraux applicables à de nombreux problèmes et ce en partant uniquement des données.

Le data engineering consiste en l'étude approfondie des données et à les rendre exploitables. Il s'agit d'une étape cruciale qui concentre une grande partie du travail dans les problèmes réelles.

1.1 Types d'apprentissage

Selon les scénarios, il existe différents types d'apprentissage:

- L'apprentissage supervisé: les données sont composées à la fois des données d'entrées et des sorties attendues. Le but est d'implémenter un algorithme capable de généraliser proprement pour de nouvelles entrées.
- L'apprentissage non supervisé: les données sont fournies sans les sorties attendues, il s'agit alors de trouver des structures au sein des données (typiquement à l'aide de méthodes de clustering).
- Il en existe encore d'autres type comme le reinforcement learning qui permet à un algorithme d'interagir avec son environnement.

Il faut également faire la distinction entre deux types de méthodes:

- La méthode paramétrique: elle définit un ensemble de fonctions pour lesquelles il convient de trouver les meilleurs paramètres.
- La méthode non paramétrique: elle dépend directement des données.

1.2 Apprentissage Supervisé Paramétrique

Soit X un espace d'entrée et Y un espace de sortie, on va chercher à apprendre une fonction de prédiction: $f : X \longrightarrow Y$ paramétrée par un paramètre θ. On va donc chercher les, voire les, paramètre θ qui maximise la performance de la fonction f.

Formelement, on définit une fonction de coût $l(y', y)$, qui mesure l'erreur entre la prédiction y' et la valeur attendue y, ainsi qu'une fonction de risque: $R(f) = \mathbb{E}_{(x,y)}(l(f(x), y))$.

Malheureusement, on ne connaît jamais la distribution réelle des données mais plutôt simplement un échantillon Z. On définit alors le risque empirique:

$$R_Z(f) = \frac{1}{|Z|} \sum_{(x,y)\in Z} l(f(x), y).$$

L'apprentissage consiste alors à minimiser une fonction de perte définie à partir de ce risque et, éventuellement, d'autres termes.

Pour éviter qu'un modèle over/underfit les données, il est souvent judicieux de séparer les données en trois sets: un pour l'entraînement (train set), un pour la validation du modèle (validation set) et enfin un pour estimer la performance rélle (test set).

2 Premier Résau de Neurones

2.1 Perceptron Multicouche (MLP)

Un (MLP) est la composition de couches linéaires ($x \mapsto Wx + b$ avec W le poids et b le biais) et non-linéaires. Ces architectures sont caractérisées par leurs hyperparamètres (taille des couches, type de non-linéarité).

Théorème: Cybenko

Un MLP à 2 couches peut approximer n'importe quelle fonction continue sur un compact avec une précision arbitraire.

Définition: Fonction de perte

On définit la perte quadratique moyenne (MSE) adaptée à la classification multi-classe via codage one-hot par:

$$L_{MSE}(\theta, D) = \frac{1}{N}\sum_{i=1}^{N} ||f_\theta(x_i) - y_i||^2 \quad \text{où D correspond au dataset.}$$

Cette fonction ne cherche qu'à réduire l'erreur sur les données d'entraînement, il y a donc un risque d'overfiting, on peut donc ajouter une pénalité sur la taille des poids, on définit alors le coefficient de régularisation $\lambda > 0$, la fonction à minimiser devient:

$$L(\theta, D) = L_{MSE}(\theta, D) + \lambda||\theta||^2$$

plus λ est grand plus le résau est contraint de rester simple.

2.2 Optimisation

Même avec un modèle simple, on obtient un problème d'optimisation non convexe, on utilise donc la descente de gradient.
Le principe de la descente gradient est le suivant: à chaque itération, on déplace le paramètre (ici θ) dans la direction opposée au gradient de la fonction de perte: $\theta \leftarrow \theta - \nu\nabla_\theta L(\theta, D)$ où ν correspond au taux d'apprentissage.
Cette technique nécessite toutefois de calculer le gradient sur toutes les données ce qui est couteux sur les grands datasets.

Pour contourner ce problème, on utilise la méthode de descente gradient stochastique (SGD) qui consiste à ne pas calculer le gradient sur l'ensemble des données mais à partir d'un échantillon.
Il en existe deux variantes:

- SGD pur: à chaque itération on prend un seul exemple aléatoire: $\theta \leftarrow \theta - \nu\nabla_\theta l(f_\theta(x_i), y_i)$.
- Mini-batch SGD: à chaque itération on prend un petit lot de données (batch) de taille B:

$$\theta \leftarrow \theta - \nu\frac{1}{B}\sum_{i=1}^{B} \nabla_\theta l(f(x_i), y_i).$$

  Cette version est plus simple en pratique car plus stable que le SGD pur et plus efficace que le gradient complet (elle est également bien adapté au GPU).

2.3 Hyperparamètres Clés de l'Optimisation

Les hyperparamètres clés sont:

- Le taux d'apprentissage: c'est le paramètre ν qui détermine la taille des pas dans l'espace des paramètres, lorsqu'il est trop grand le modèle diverge et lorsqu'il est trop petit l'apprentissage peut être très lent voire bloqué dans un minimum local. Souvent on choisit au départ $\nu \in [10^{-3}, 10^{-2}]$.
- La taille des batch B: une trop petite taille permet de mieux explorer le paysage de la perte mais augmente l'importance du bruit, lorsqu'elle est trop grande le modèle risque d'overfit. En pratique on choisit des batch de 32 ou 64 pour les petits modèles.
- Le calendrier du taux d'apprentissage: le taux d'apprentissage n'est souvent pas constant, on le fait varier de diverses façons (division toutes les x epochs, décroissance en forme de cosinus, augmentation au fur et à mesure etc). L'objectif est d'éviter de se retrouver coincé dans des minima locaux tout en assurant la convergence en fin d'entraînement.
- Le nombre d'époques (Epochs): l'algorithme voit l'ensemble des données d'entraînement en une epoch, lorsque le nombre d'epoch est trop grand le modèle risque d'overfit.

3 Backpropagation

La backpropagation se base essentiellement sur la règle de dérivation en chaîne. Chaque couche est vue comme pouvant faire deux choses:

- Forward: Calculer sa sortie en fonction de son entrée et de ses paramètres.
- Backward: Calculer les gradients de la perte par rapport à ses paramètres et à son entrée.

L'algorithme se fait en deux étapes:

- Forward pass: On fait passer les données à travers le réseau pour obtenir la prédiction.
- Backward pass: On fait revenir les gradients depuis la sortie jusqu'à l'entrée.

A la fin on a les gradients de la perte L par rapport à tous les paramètres du résau, ce qui permet de le mettre à jour avec la descente de gradient.
Par exemple pour un résau de trois couches: $x \to h_1 \to h_2 \to y$ on calcule en forward h_1, h_2 et y, puis en backward on remonte: $\frac{\partial L}{\partial y}$, puis $\frac{\partial L}{\partial h_2}$, ensuite $\frac{\partial L}{\partial h_1}$, et enfin les dérivées par rapport aux poids.

3.1 Formalisation

Considérons un résau à K couches, chaque couche f_k prend en entrée un vecteur x_{k-1} et des paramètres θ_k pour produire une sortie:

$$x_k = f_k(x_{k-1}, \theta_k)$$

la sortie finale du résaux est $y = x_K$.

Soit une fonction différentiable de perte L pour la sortie y du résau, le backward pass a pour objectif de calculer, pour tout paramètre θ_k, la dérivée $\frac{\partial L}{\partial \theta_k}$, on peut alors mettre à jour la valeur de ce paramètre:

$$\theta_k \leftarrow \theta_k - \alpha \frac{\partial L}{\partial \theta_k}(\theta_k) \quad \text{avec } \alpha \text{ le taux d'apprentissage.}$$

Pour effectuer ce calcule on calcule récurcivement $\frac{\partial L}{\partial \theta_k}$ à partir de la dernière couche:

$$\frac{\partial L}{\partial \theta_k}(\theta_k) = \frac{\partial L}{\partial x_k}(x_k) \frac{\partial f_k}{\partial \theta_k}(x_{k-1}, \theta_k)$$

or, par hypothèse, L est différentiable donc on peut calculer directement $\frac{\partial L}{\partial x_K}$ puis:

$$\frac{\partial L}{\partial x_{k-1}}(x_{k-1}) = \frac{\partial L}{\partial x_k}(x_k) \frac{\partial x_k}{\partial x_{k-1}} = \frac{\partial L}{\partial x_k}(x_k) \frac{\partial f_k}{\partial x_{k-1}}(x_{k-1}\theta_k).$$

4 Résaux de Neurones Convolutifs (CNN): Achitectures LeNet et AlexNet

Il existe une différence fondamentale entre les CNN et les MLP, en effet un MLP considère chaque entrée comme indépendante des autres. Or, cette interprétation des données peut être problématique, dans une image par exemple les pixels proches sont corrélés, on utilise alors un CNN pour conserver cette cohérence spatiale en exploitant des opérations locales appelées convolutions.

4.1 Principe de la Convolution

Une couche de convolution apprend un ensemble de filtres (ou noyaux) qui détectent des motifs locaux dans les données d'entrée.

Définition:

Soit une entrée $X \in \mathbb{R}^{W \times H \times C_{in}}$ où W et H sont la largeur et la hauteur, C_{in} correspond au nombre de canaux (par exemple trois pour du RGB).

Un noyau ou filtre $K \in \mathbb{R}^{S \times S \times C_{in} \times C_{out}}$ (pour l'exemple typique d'un filtre en carré) où S est la taille du filtre et C_{out} le nombre de canaux de sortie.

La sortie de la couche de convolution est alors: $Y_i = K_i * X$ pour $i \in [|1, C_{out}|]$ avec $*$ la convolution en 2D.

Chaque filtre K_i balaie l'image, effectue des multiplications locales, et produit une carte d'activation (feature map) qui indique où le motif est présent.

Pour des données brutes (sans padding), la taille de sortie est $W_{out} = W - S + 1$, $H_{out} = H - S + 1$, avec un padding (ajout de zéro aux limites des données donc autour de l'image par exemple), on conserve la taille: $W_{out} = W$ et $H_{out} = H$.

On peut également implémenter un pas s (stride) afin d'échantilloner moins souvent et donc réduire la dimension de sortie: $W = \left\lfloor \frac{W-S}{s} + 1 \right\rfloor$.

En pratique, la plupart des CNN modernes utilisent des filtres 3×3, un padding de 1 et un stride de 1 (voire 2).

L'intérêt de la convolution est de diminuer significativement le nombre de paramètres du modèle. Par exemple une couche linéaire classique reliant tous les pixels entre eux aurait:

$$(W \times H \times C_{in})(W_{out} \times H_{out} \times C_{out})$$

paramètres ce qui devient vite énorme. Tandis que une convolution ne dépend que d'un petit voisinage et donc possède uniquement: $S^2 \times C_{in} \times C_{out}$ paramètres.

Une convolution permet donc de réduire significativement le poids tout en exploitant la structure spatiale.

4.2 Autres Couches Importantes

Une architecture de CNN manipule de nombreux tenseurs de différentes dimensions, des opérations pour changer ces dimensions peuvent donc s'avérer utiles, il sera par ailleurs de les interpréter comme des couches à part entières de l'architecture bien qu'elles ne puissent présenter aucun paramètre à entraîner.

On considère donc des couches de:

- mise en forme (reshaping) qui changent la structure des tenseurs,
- pooling qui réduisent la taille spatiale des cartes caractéristiques pour diminuer le coût de calcul et augmenter la robustesse aux petites translations tout en conservant les activations les plus significatives. Il en existe deux types principaux:

  - le Max Pooling qui garde la valeur maximale dans chaque région

  - le Average Pooling qui fait la moyenne par région.

  On détaille le fonctionnement de tels fonctions juste après.
- interpolation (upsampling) qui augmentent la taille spatiale

Exemple pour le pooling:

On s'intéresse à:

$$\begin{pmatrix} 1 & 3 & 2 & 0 & 1 & 2 \\ 4 & 6 & 5 & 1 & 3 & 1 \\ 7 & 2 & 8 & 2 & 0 & 4 \\ 3 & 5 & 0 & 6 & 7 & 3 \\ 2 & 1 & 9 & 8 & 2 & 4 \\ 1 & 0 & 3 & 5 & 2 & 1 \end{pmatrix}$$

à laquelle on applique MaxPool2d(kernel_size=2, stride=2), on obtient donc:

$$\begin{pmatrix} 6 & 5 & 3 \\ 7 & 8 & 7 \\ 2 & 9 & 4 \end{pmatrix}.$$

4.3 LeNet: Le Premier CNN

Proposé par Yann LeCun dans les années 90, il est utilisé pour reconnaîtres les chiffres manuscrits sur les chèques bancaires.

Sa structure générale est la suivante:

- Convolution $->$ ReLU $->$ pooling
- Convolution $->$ ReLU $->$ pooling
- Fully Connected (dense) $->$ ReLU
- Fully Connected $->$ sortie (10 classes)

Ce modèle possède peux de paramètres (quelques centaines de milliers), il s'agit du premier CNN montrant qu'on poouvait apprendre directement à partir de pixels pour faire de la reconnaissance.

4.4 AlexNet: La Révolution de 2012

Avant 2012 les approches de vision par ordinateur reposaient surtout sur des descripteurs manuels. Puis arrivent Alex Krizhevsky, Ilya Sutskever et Geoffrey Hinton qui publient AlexNet un résau de deep learning qui écrase la concurrence (réduisant par deux la marge d'erreur).

Cette révolution s'organise autour de cinq points clés:

- un résau très profond pour l'époque (8 couches)
- une utilisation massive du GPU pour l'entraînement
- la fonction d'activation ReLU plus facile à entraîner que le sigmoid
- la technique du dropout
- et une data augmentation

Le dropout est une technique de régularisation qui consiste à 'éteindre' aléatoirement un certain pourcentage des neurones à chaque étape d'entraînement. L'objectif de cette méthode est d'empêcher le résaux de devenir trop dépendant d'un petit ensemble de neurones pour avoir une meilleur généralisation.

Par exemple supposons qu'on est en entrée le vecteur d'activation $x = (2, 8, -3, 11, 5)$ et pour la couche un dropout de 50%, alors on génère selon une loi de Bernoulli(0.5): $m = (1, 0, 1, 0, 1)$, alors l'entrée devient: $x' = x \odot m = (2, -3, 5)$.

Néanmoins, quand le résau est utilisé pour faire des prédictions, tous les neurones sont activés mais les sorties sont réduites par la probabilité de maintient p: $x_{test} = px_{train}$.

On va désormais détailler l'architecture d'AlexNet. Pour ce faire on considère une taille d'entrée classique 224x224x3 ($H \times W \times C_{in}$). L'architecture est à retrouver dans AlexNet.py.

La première couche conv1:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 3,\ C_{out} = 96,\ S = 11,\ stride = 4,\ padding = 2$ activation de ReLU
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{224+2\cdot 2-11}{4} \right\rfloor + 1 = 55 = H_{out}$ la sortie totale est donc: $55^2 \times 96$.

- Nombre de paramètres, poids et biais: poids= $96 \times 3 \times 11^2 = 34848$, chaque filtre a un biais scalaire unique donc 96 biais, soit 34 944 paramètres au total.
- Il s'agit ici d'une réduction forte de la dimension spatiale dès le départ pour minimiser le coût de calcul, le filtre large permet d'observer des motifs de grande échelle.

La couche pool1:

- Type: MaxPool2d
- Paramètres de la fonction: $kernel = 3,\ stride = 2$ pas de padding.
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{55-3}{2} \right\rfloor + 1 = 27 = H_{out}$ la sortie totale est donc: $27^2 \times 96$.
- Nombre de paramètres, poids et biais: pas de paramètres.
- L'objectif des couches de pooling est ici de réduire la résolution et les invariances locales aux translations.

La seconde couche conv2:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 96,\ C_{out} = 256,\ S = 5,\ stride = 1,\ padding = 2$ activation de ReLU
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{27+2^2-5}{1} \right\rfloor + 1 = 27 = H_{out}$ la sortie totale est donc: $27^2 \times 96$.
- Nombre de paramètres, poids et biais: poids= $256 \times 96 \times 5^2 = 614400$, chaque filtre a un biais scalaire unique donc 256 biais, soit 614 656 paramètres au total.

La couche pool2:

- Type: MaxPool2d
- Paramètres de la fonction: $kernel = 3,\ stride = 2$ pas de padding.
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{27-3}{2} \right\rfloor + 1 = 13 = H_{out}$ la sortie totale est donc: $13^2 \times 96$.
- Nombre de paramètres, poids et biais: pas de paramètres.

La troisième couche conv3:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 256,\ C_{out} = 384,\ S = 3,\ stride = 1,\ padding = 1$ activation de ReLU
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{13+2-3}{1} \right\rfloor + 1 = 13 = H_{out}$ la sortie totale est donc: $13^2 \times 384$.
- Nombre de paramètres, poids et biais: poids= $384 \times 256 \times 3^2 = 884736$, chaque filtre a un biais scalaire unique donc 384 biais, soit 885 120 paramètres au total.

La quatrième couche con4:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 384,\ C_{out} = 384,\ S = 3,\ stride = 1,\ padding = 1$ activation de ReLU
- Taille de sortie (spatial): la taille de sortie reste $13^2 \times 384$
- Nombre de paramètres, poids et biais: poids= $384 \times 384 \times 3^2 = 1327104$, chaque filtre a un biais scalaire unique donc 384 biais, soit 1 327 488 paramètres au total.

La cinquième couche conv5:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 384,\ C_{out} = 256,\ S = 3,\ stride = 1,\ padding = 1$ activation de ReLU
- Taille de sortie (spatial): la taille de sortie est $13^2 \times 256$.
- Nombre de paramètres, poids et biais: poids= $256 \times 384 \times 3^2 = 884736$, chaque filtre a un biais scalaire unique donc 256 biais, soit 884 992 paramètres au total.

On peut observer dans les couches précédentes un enchaînement de petits filtres (3×3) capturant les motifs locaux et combinatoire.
La couche pool3:

- Type: MaxPool2d
- Paramètres de la fonction: $kernel = 3,\ stride = 2$ pas de padding.
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{13-3}{2} \right\rfloor + 1 = 6 = H_{out}$ la sortie totale est donc: $6^2 \times 256$.
- On flatten ensuite pour la couche fully connected: vecteur de dimension $256 \times 6^2 = 9216$.

On passe désormais aux couches dites fully connected.
La couche fc1, précédée d'une couche de dropout pour éviter le surapprentissage:

- Type: dense.
- Paramètres: $C_{in} = 9216,\ C_{out} = 4096$ puis activation de ReLU.
- Nombre de paramètres poids, et biais: Poids= 4096×9216, biais=4096 donc 37 752 832 paramètres au total.

La seconde couche fc2, précédée d'une couche de dropout pour éviter le surapprentissage:

- Type: dense.
- Paramètres: $C_{in} = 4096,\ C_{out} = 4096$ puis activation de ReLU.
- Nombre de paramètres poids, et biais: Poids= 4096×4096, biais=4096 donc 16 781 312 paramètres au total.

Et enfin la dernière couche et la sortie du programme fc3:

- Type: dense.
- Paramètres: $C_{in} = 4096,\ C_{out}$ =NbClasses puis activation de ReLU.
- Nombre de paramètres poids, et biais: Poids= 4096×NbClasses, biais=NbClasses donc 4 097 000 paramètres au total pour 1000 classes.

En 2012 il était en effet courant d'enchaîner les couches fully connected pour transformer les caractéristiques spatiales en représentations globales et classifier (comme nous allons le voir c'est très coûteux en paramètres et mémoire).
Le résaux compte donc un total d'environ 62.4M paramètres (il est intéressant de constater que 94% des paramètres se trouvent dans les couches fc dont 60% dans la première).

4.4.1 L'intérêt des Couches Conv2d, ReLU et MaxPool2d

On détaille tout d'abord chacune de ces fonctions:

- La fonction Conv2d (Convolution 2D): Son but est d'extraire les motifs locaux d'une image (bords, textures, formes, motifs répétitifs, etc), c'est à ce moment que le résaux repère et analyse les structures visuelles.

  Le principe des convolutions est détaillé au début de cette section.

  L'intérêt d'utiliser ces filtres est ici de les appliquer partout dans l'image afin de reconnaître un motif quelque soit son emplacement.

- La fonction ReLU (Rectified Linear Unit): C'est une fonction d'activation qui a pour objectif d'introduire de la non linéarité dans le résau. Sans cette fonction même les CNN profonds seraient équivalents à une seule convolution linéaire. La ReLU s'est imposée car il s'agit d'une activation simple, rapide et efficace.

  Mathématiquement, elle correspond à la fonction: $\max(0, x)$, cette fonction possède donc une dérivée simple: $1_{R_+^*}$, elle est rapide à entraîner, elle introduit énormément de zéros donc permet une meilleure généralisation.

- La fonction MaxPool2d: Il s'agit d'une fonction de pooling (son fonctionnement est détaillé au début de cette section).

L'intérêt de ces couches est de construire une hiérarchie de représentations.

En effet, au cours de l'entraînement le résaux apprend automatiquement quels motifs sont utiles pour minimiser la fonction de perte. Cette apprentissage se déroule de la façon suivante:

- Initialement: Le poids de chacun des filtres est initilaisé aléatoirement, ils ne détectent rien a priori.
- Entraînement: Lorsqu'on calcule la perte (par exemple une erreur de classification) on fait ensuite un rétropropagation (backpropagation), les poids des filtres sont légèrement ajustés pour réduire l'erreur.
- Résultat: Certains filtres deviennent sensibles aux bords verticaux (par exemple $[-1,\ 0,\ 1]$), d'autre aux bords horizontaux, aux motifs à répétitions etc. Il est intéressant de constater que cette apprentissage permet au résau de découvrir par lui même les motifs qui maximisent la bonne classification.

L'intérêt d'enchaîner ces couches est de permettre à chaque couche de prendre comme entrée les cartees d'activation de la couche précédente, les couches supérieures repèrent donc les détails de bas-niveau (bords, textures, couleurs) puis les couches suivantes les motifs intermédiaires (parties d'objets) puis les concepts entiers (objets, visages) et ainsi de suite.

4.4.2 L'intérêt des Couches Entièrements Connectées

Ces couches forment la tête du résau, elles interprètent les caractéristiques extraites par les convolutions et prennent la décision finale de classification.

Remarque:

Dans la fonction forward(self, x) on remarque l'appele x = torch.flatten(x, 1) entre le passage des couches de convolutions aux couches entièrements connectées. En effet ces dernières prennent en entrée des vecteurs d'une seule dimension tandis que le résau de convolution renvoi des cartes caractéristiques de dimension (256, 6, 6), l'appele x = torch.flatten(x, 1) convertit donc x en un vecteur de taille 256 x 6 x 6 = 9 216.

La première couche fc1 opère la transformation affine:

$$y = Wx + b \ \text{ avec } W \in \mathcal{M}_{4096,9216}(\mathbb{R}) \ \text{ est la matrice de poids, } b \in \mathbb{R}^{4096} \ \text{ le vecteur de biais}$$

$x \in \mathbb{R}^{9216},\ y \in \mathbb{R}^{4096}$ sont les entrée et sortie respective de la couche.

Ainsi, chaque neurone de cette couche reçoit tous les éléments d'entrée, apprend à les combiner et peut ainsi en déduire un représentation de l'image.

Les fonctions dropout et ReLU sont détaillées précédement.

5 A Retenir Des Quizzs

5.1 Quizz 1

- Comment calculer le nombre de paramètres d'un résaux de neurones: pour chaque couche on a le calcule suivant: $y = Wx$ avec y la sortie en dimension k, x l'entrée de dimension d et donc W la matrice de paramètres dans $\mathcal{M}_{k,d}(\mathbb{R})$, cette couche a donc $k * d$ paramètres.
  Par exemple pour un résau en trois couches dont les deux premières couches contiennent respectivement 200 et 100 neurones, pour une entrée de dimension 10 et une sortie de dimension 1, on a:

$$\mathbb{R}^{10} \longrightarrow_{W_1} \mathbb{R}^{200} \longrightarrow_{W_2} \mathbb{R}^{100} \longrightarrow_{W_3} \mathbb{R}^{1} \quad \text{on a donc } 10 \cdot 200 + 200 \cdot 100 + 100 \cdot 1 \quad \text{paramètres.}$$

- Calcul de la mémoire nécessaire pour implémenter le forward/backward pass: pour un résau dont les paramètres sont stoqués sur M bits et dont les données à chaque couche pour un unique échantillon sont stoquées sur m bits, la mémoire nécessaire pour implémenter le forward pass sur un batch de taille B est

$$M + B \cdot m \cdot \text{nombre de couches.}$$

  Pour le backward pass, il faut également compter les paramètres du gradient qui sont d'environ un par paramètre du résau donc la mémoire nécessaire est:

$$M + B \cdot m \cdot \text{nombre de couches.}$$