



# Table des matières

<b>LE MACHINE/DEEP LEARNING .....</b>	<b>3</b>
<i>Introduction .....</i>	3
<i>De l'humain au deep learning.....</i>	3
<i>Introduction aux réseaux de neurone.....</i>	4
<i>Définition.....</i>	4
<i>Principe .....</i>	5
<b>LES DIFFERENTS RESEAUX DE NEURONE CLASSIQUE.....</b>	<b>6</b>
LE PERCEPTRON SIMPLE (PS).....	6
<i>Définition.....</i>	6
<i>Signification de chaque mot .....</i>	7
<i>Caractéristique.....</i>	7
<i>Fonctionnement mathématique .....</i>	8
<i>Les fonctions de calcul de W.....</i>	8
LE PERCEPTRON MULTICOUCHE (MULTI LAYER NETWORK) .....	9
<i>Définition.....</i>	9
<i>Caractéristique.....</i>	10
<i>Fonctionnement mathématique .....</i>	10
RESEAUX DE NEURONES CONVOLUTIFS (CNN) .....	12
<i>Fonctionnement.....</i>	14
RESEAUX DE NEURONES RECURRENTS (RNN).....	16
<i>Définition.....</i>	16
<i>Caractéristique.....</i>	16
<i>Fonctionnement mathématique .....</i>	16
RESEAU HYBRIDE (MELANGE DE CNN & RNN) .....	17
<i>Définition.....</i>	17
<i>Caractéristique.....</i>	17
<i>Fonctionnement mathématique .....</i>	17
<b>ARCHITECTURES D'APPRENTISSAGE EN PROFONDEUR RECENTES .....</b>	<b>18</b>
RESEAU RESIDUEL (RESNETS) .....	19
<i>Définition.....</i>	19
<i>Caractéristique.....</i>	19
<i>Fonctionnement mathématique .....</i>	19
RESEAU ROUTIER (HIGHWAYNETS).....	20
<i>Définition.....</i>	20
<i>Caractéristique.....</i>	20
<i>Fonctionnement mathématique .....</i>	20
COMPARAISON DES DIFFERENTS MODEL SELON LEURS CAS D'USAGE .....	21
<b>LES DIFFERENTES FONCTIONS UTILISEES .....</b>	<b>22</b>
INTRODUCTION .....	22
OPTIMISEUR.....	22
<i>Algorithme qui permet de minimiser la fonction de perte pondérée. ....</i>	22
<i>Dropout.....</i>	22
<i>Fonctions de perte dans les réseaux de neurones .....</i>	23
<i>Activation.....</i>	24
LES METRIQUES.....	25
<i>Métriques pour évaluer les algorithmes d'apprentissage automatique en Python .....</i>	25
<i>Categorical_accuracy :.....</i>	25
<i>Sparse_categorical_accuracy : .....</i>	25
<i>Confrontation.....</i>	25

<i>Précision binaire:</i> .....	25
<i>Exactitude catégorique :</i> .....	26
<i>Précision catégorique clairsemée :</i> .....	26
<i>Précision catégorique maximale :</i> .....	26
<b>EXPERIMENTATION .....</b>	<b>27</b>
<i>Introduction .....</i>	27
<i>Pre-processing.....</i>	27
<i>Transformation de la donnée en fonction du model .....</i>	28
<b>NOTRE PROGRAMME .....</b>	<b>30</b>
<b>NOS RESULTATS .....</b>	<b>31</b>
INTRODUCTION .....	31
PERCEPTRON SIMPLE .....	32
PERCEPTRON MULTI-COUCHE.....	33
RNN.....	34
CNN.....	35
RESNETS.....	36
HIGHWAYNETS .....	37

# Le machine/deep Learning

## Introduction

Avec l'avènement de l'intelligence artificielle, l'essor des objets connectés et l'augmentation des capacités technologiques, nombreuses recherches convergent à simuler le comportement humain afin de reproduire ses capacités cognitives dans les outils qui nous entourent.

Dans ce contexte le Deep Learning, littéralement "l'apprentissage profond", est devenu l'un des axes de recherche les plus explorés. Alors qu'est-ce donc que le Deep Learning, qu'est-ce qu'un réseau de neurones, quel lien avec notre cerveau humain, comment cela fonctionne et surtout pour quelles applications ?

C'est dans cette approche qu'on a nous proposé un projet de machine/deep learning afin de comprendre tout le paradigme derrière les réseaux de neurone.

## De l'humain au deep learning

Pour comprendre le Deep Learning et surtout les réseaux de neurones, il ne suffit pas de s'intéresser aux mathématiques et à la technologie.

Comprendre le Deep Learning, c'est avant tout remonter à l'élément central de notre vision du monde : l'être humain. L'Homme a en effet toujours cherché à reproduire notre façon d'être en refaisant une simulation de lui-même, de quelque façon que ce soit.

Simuler l'humain, c'est chercher à reproduire les différentes capacités qu'il utilise au quotidien : sociales, comportementales, éthiques, physiques... Dans le lot, celle qui renferme le plus de mystère et qui sans doute revêt le plus d'attrait pour la recherche appliquée est la capacité cognitive. C'est elle qui nous permet de connaître, mémoriser, raisonner, apprendre ou encore parler.

Afin de comprendre cette approche de l'humain, nous avons étudié l'an dernier différents algorithmes tels que la recherche locale naïve, les algorithmes génétiques et enfin l'approche au niveau des réseaux de neurone. Ce qui nous a amené au machine/deep learning.

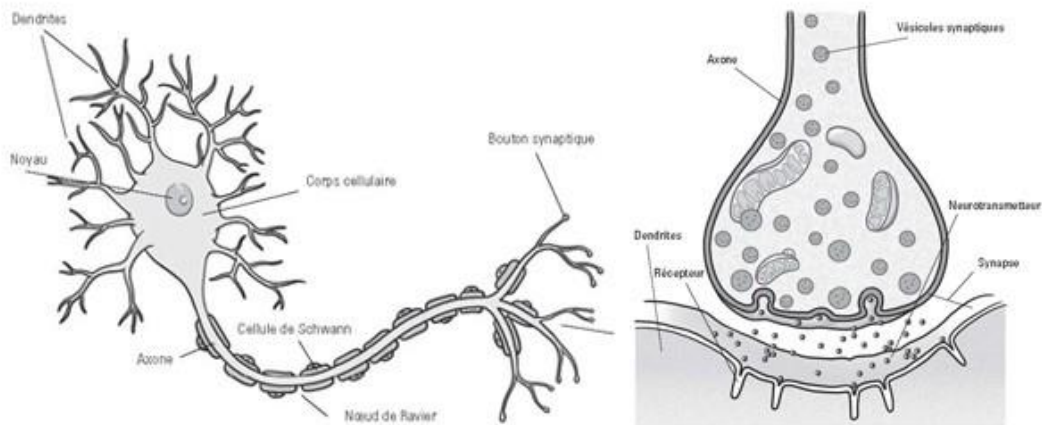
Son objectif est précisément de simuler le cerveau humain par des procédés informatiques. Imaginez un ordinateur qui puisse stocker autant d'informations que notre mémoire et qui sache décider et agir en même temps ! Imaginez un Jarvis d'Iron Man ! (je tease VIDAL)

Plonger dans le Deep Learning, c'est donner corps à ces concepts de science-fiction à travers les réseaux de neurone, se rendre compte que les avancées sont tangibles et les résultats déjà impressionnants. Les recherches des géants du secteur comme Facebook, Google ou encore Apple dans ce domaine nous le prouvent chaque année. C'est par cette approche que nous allons donc parler des réseaux de neurone.

## Introduction aux réseaux de neurone

Un neurone se compose d'un corps cellulaire, d'un axone qui représente le lien de transmission des signaux et d'une synapse qui permet le déclenchement d'un potentiel d'action dans le neurone pour activer une communication avec un autre neurone. Il faut savoir que la force d'un réseau de neurones réside dans la communication de ses neurones à travers des signaux électriques qu'on nomme "influx nerveux". Ces signaux se caractérisent par des fréquences qui jouent un rôle important au niveau de la propagation des signaux dans le réseau en question.

Ci-dessous une représentation d'un neurone issu du centre de recherche ICM



Les réseaux de neurones sont actuellement très utilisés dans de nombreux domaines industriels ou de la vie de tous les jours : reconnaissance d'images, d'écriture de sons. Analyse de cours boursiers, classifications, ils font fonctionner des sondes spatiales ou des robots sur Mars...

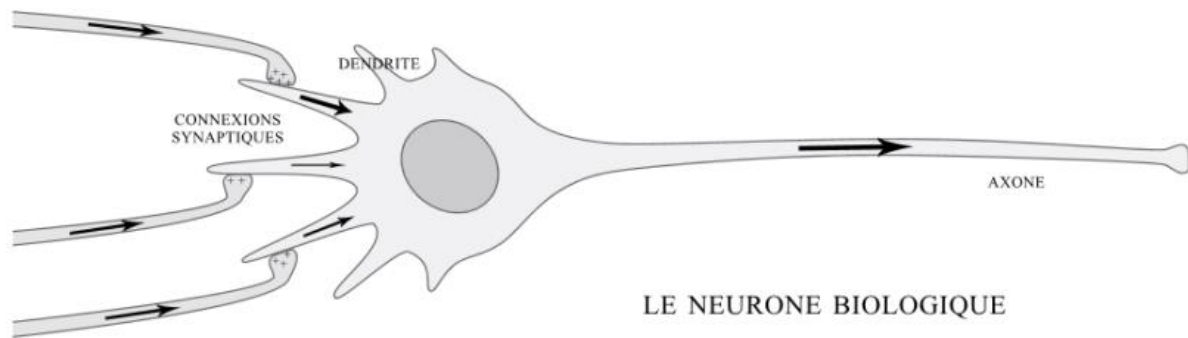
Un neurone biologique peut être représenté ainsi :

- La fonction d'un neurone est de transmettre l'influx nerveux, un signal électrique, mais il ne s'agit pas d'une simple liaison passive. Structuellement le neurone est relié, d'une part à un ensemble de dendrites et d'autre part à un axone.

## Définition

Les dendrites constituent les entrées et servent à accumuler de la charge électrique au niveau du neurone jusqu'à atteindre un certain seuil. C'est alors que le neurone transmet un signal via l'axone qui agit comme sortie du neurone.

L'expérience montre de plus que les dendrites ne sont pas toutes équivalentes, certaines contribuent plus fortement que d'autre à atteindre le seuil, on associe donc les dendrites à un poids.



Les réseaux de neurone en informatique sont inspirés des neurones biologiques.

## Principe

Dans chaque réseau de neurone, on doit apprendre aux réseaux les données ce qu'on lui envoie comme type de données en fonction de la sortie que l'on souhaite, cette phase est la phase d'apprentissage.

Une fois que la phase d'apprentissage est optimale pour un cas général, on peut utiliser le réseau pour différents tests sur les mêmes types de donnée semblable à celle utilisée en apprentissage.

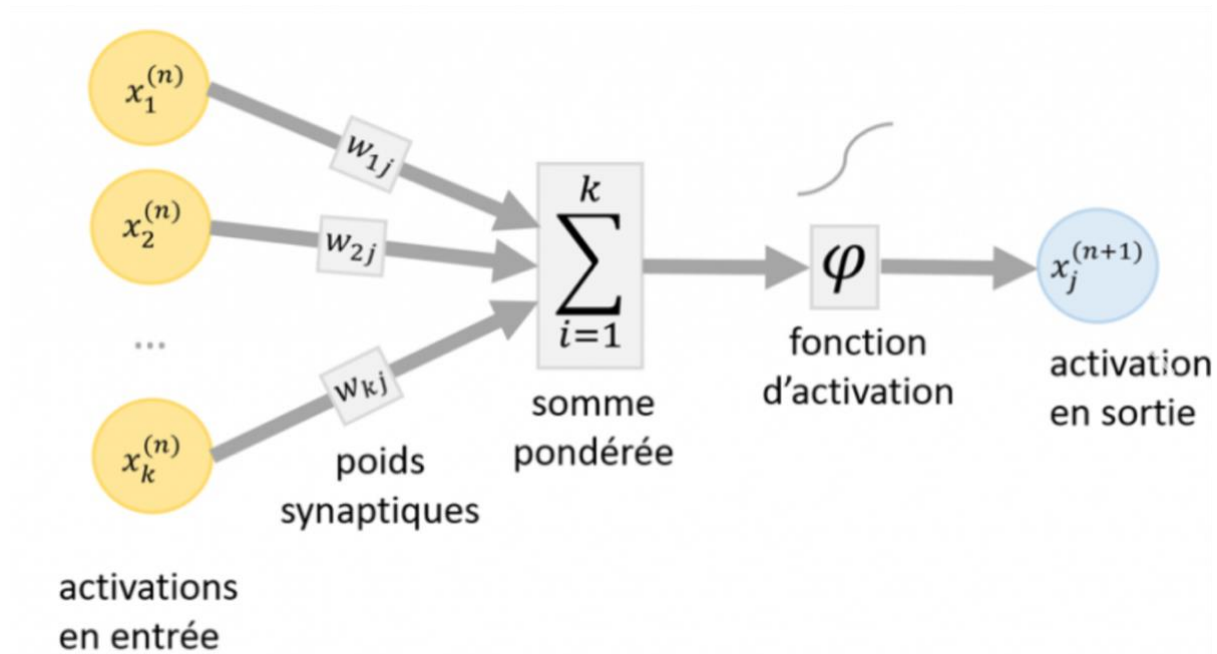
Il existe plusieurs types de réseau de neurones inspiré des neurones biologiques que l'on vous présente par la suite.

# Les différents réseaux de neurone Classique

## Le perceptron Simple (PS)

Le perceptron est le premier réseau de neurone qu'on a mis en place dans ce projet.

### Définition



Le perceptron est un algorithme d'apprentissage supervisé de classification dit *classifieurs binaires* (c'est-à-dire séparant deux classes).

Il a été inventé en 1957 par Frank Rosenblatt au laboratoire d'aéronautique de l'université Cornell.

C'est un modèle inspiré des théories cognitives de Friedrich Hayek et de Donald Hebb. Il s'agit d'un neurone formel muni d'une règle d'apprentissage qui permet de déterminer automatiquement les poids synaptiques de manière à séparer un problème d'apprentissage supervisé.

## Signification de chaque mot

**Algorithme** : le perceptron est une suite d'opérations et de calcul = la somme des entrées, leur pondération, la vérification d'une condition et la production d'un résultat d'activation.

**Apprentissage** : l'algorithme doit être "entraîné", c'est à dire qu'en fonction d'une prédiction voulue, le poids des différentes entrées va évoluer et il faudra trouver une valeur optimale pour chacune.

**Supervisé** : l'algorithme trouve les valeurs optimales de ses poids à partir d'une base de données d'exemples dont on connaît déjà la prédiction. Par exemple on a une base de données de photos de banane et on "règle" notre algorithme jusqu'à ce que chaque photo (ou presque) soit classé comme *banane*.

**Classification** : l'algorithme permet de prédire une caractéristique en sortie et cette caractéristique sert à *classer* les différentes entrées entre elles. Par exemple, trouver toutes les bananes dans un panel de photos de fruits.

**Binaire** : l'algorithme sépare un ensemble de valeurs d'entrée en seulement deux classes différentes.

**Linéaire** : l'algorithme sépare un ensemble de valeurs de manière linéaire. Prenons l'exemple d'une feuille de papier où vous auriez tracé une diagonale *droite*. Cette diagonale est une séparation linéaire de la feuille de papier. Si vous aviez tracé un rond au milieu de la feuille, la séparation serait considérée comme non linéaire.

## Caractéristique

Un perceptron possède :

- n entrée(s) X.
- n poids W.
- Une fonction de Somme  $\Sigma$ .
- Une fonction d'activation j (ou fonction de transfert).
- Une sortie Y.



## Fonctionnement mathématique

Chaque entrées  $X_i$

- Chaque liaison entre une entrée  $X_i$  et la fonction somme possède un poids  $W_i$
- La fonction somme est la suivante :

$$Z = f(W_1 * X_1 + W_2 * X_2 + \dots + W_i * X_i)$$

Une fois cette somme calculée, on applique une fonction d'activation selon le type de procédé que l'on veut :

**Classification :**

- Si  $z > 0 \Rightarrow Y = 1$
- Si  $z < 0 \Rightarrow Y = 0$

**Régression :** une fonction affine

## Les fonctions de calcul de W

Classification : Perceptron Learning Algorithm	$W \leftarrow W + \alpha Y^k X^k$
Classification : Règle de Rosenblatt	$W \leftarrow W + \alpha (Y^k - g(X^k)) X^k$
Régression linéaire : Pseudo inverse	$W = ((X^T X)^{-1} X^T) Y$

On applique une fonction d'activation/transfert (voir partie sur les fonctions d'activation) sur la somme totale pour définir sa valeur.

$$S = b + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + w_n \cdot x_n = b + \sum_{i=1}^n w_i \cdot x_i$$

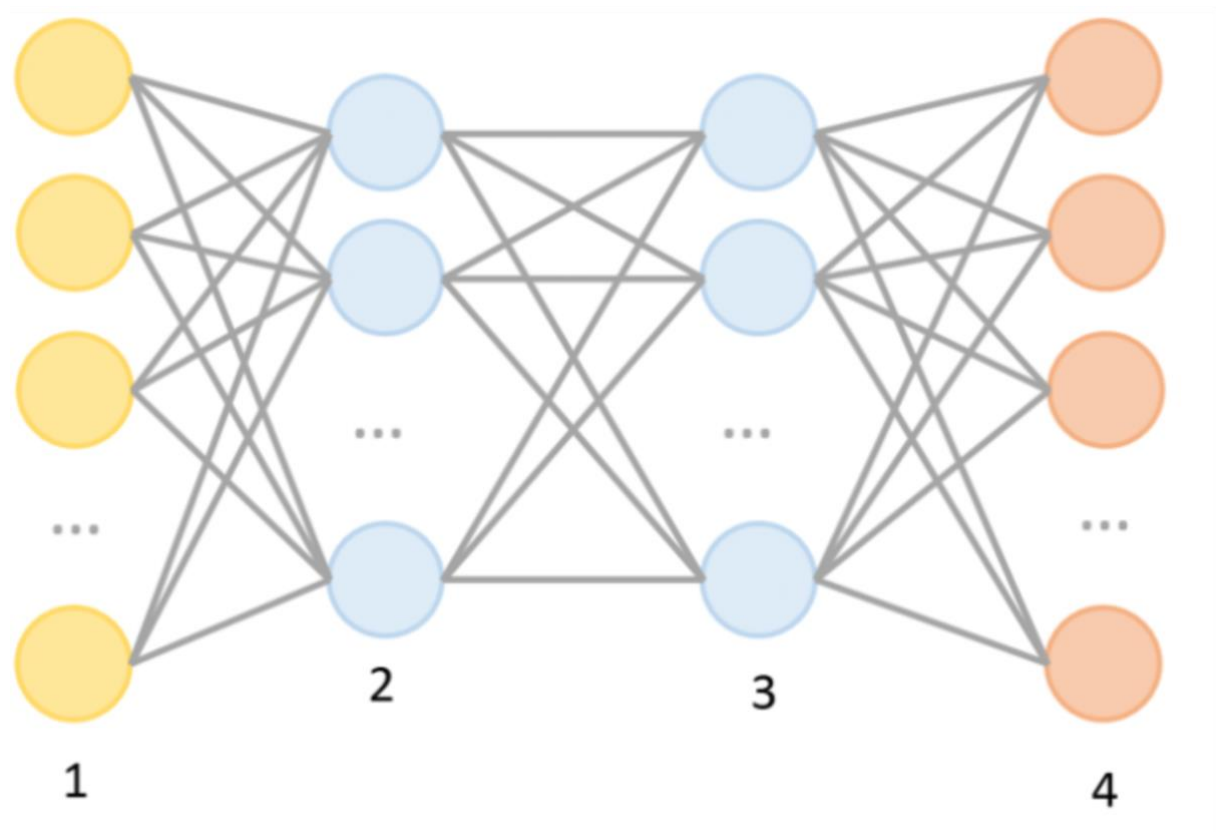
## Le perceptron Multicouche (Multi Layer Network)

### Problématique :

Le problème du Perceptron simple sont ses limitations que nous allons essayer de dépasser avec le Perceptron multicouche.

(CF : A RAJOUTER PLUS DE CHOSE)

### Définition



Le perceptron multicouche (multilayer perceptron MLP) est un type de réseau neuronal formel organisé en plusieurs couches au sein desquelles une information circule de la couche d'entrée vers la couche de sortie uniquement ; il s'agit donc d'un réseau à propagation directe (feedforward). Chaque couche est constituée d'un nombre variable de neurones, les neurones de la dernière couche (dite « de sortie ») étant les sorties du système global.

## Caractéristique

Le Perceptron multicouche basé sur le modèle du Perceptron Simple a plusieurs couches de neurones liées entre elles. Chaque couche a un ou plusieurs neurones.

Le Perceptron simple ne pouvait classer que des données séparées par un hyperplan.

Or nous avons d'autres données plus complexes séparables par des hyper-surfaces.

Nous avons donc travaillé avec un perceptron Multicouche qui se compose de :

- Une couche d'entrée.
- Une ou plusieurs couches cachées.
- Une couche de sortie.

## Fonctionnement mathématique

Première couche, couche d'entrée :

Votre ligne de données arrive en entrée du réseau, dans la première couche, dit couche d'entrée (cf figure X : numéro 1). Tous les neurones de la couche d'entrée vont ensuite dans chaque entrée de la première couche cachée.

1ère Couche cachée :

Vous pouvez calculer la sortie de tous les neurones de la première couche cachée en appliquant la formule de la somme avec biais :

$$S = b + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + w_n \cdot x_n = b + \sum_{i=1}^n w_i \cdot x_i$$

Puis en lui appliquant une fonction d'activation, dans cet exemple on a utilisé la fonction tanh :

$$x_j^l = \theta(s_j^l) = \text{Tanh}\left(\sum_{i=0}^{d^{l-1}} w_{ij}^l x_i^{l-1}\right)$$

Couche cachée et couche de sortie :

Avec les sorties de la première couche cachée, vous pouvez ensuite calculer les sorties de la 2ème couche cachée, puis la 3ème couche cachée, et ainsi de suite jusqu'à la sortie. Votre information s'est donc propagée dans l'ensemble du réseau !

## Recalcul de delta

Recalcule un Delta en fonction de l'erreur, ce delta permet par la suite de recalculer les poids

Par classification :

$$\delta_j^L = (1 - (x_j^L)^2) \times (x_j^L - y_j)$$

$$\delta_i^{l-1} = (1 - (x_i^{l-1})^2) \times \sum_{j=1}^{d^l} (w_{ij}^l \times \delta_j^l)$$

Par régression :

$$\delta_j^L = (x_j^L - y_j)$$

$$\delta_i^{l-1} = (1 - (x_i^{l-1})^2) \times \sum_{j=1}^{d^l} (w_{ij}^l \times \delta_j^l)$$

$$w_{ij}^l \leftarrow w_{ij}^l - \alpha x_i^{l-1} \delta_j^l$$

Les fonctions d'activation (ou fonction de transfert).

Il existe plusieurs fonctions d'activation dans un réseau, les fonctions d'activations sont énoncées dans **la partie XX** pour plus de précision.

Calcul des poids W

Il faut mettre à jour les poids afin d'adapter le modèle aux données.

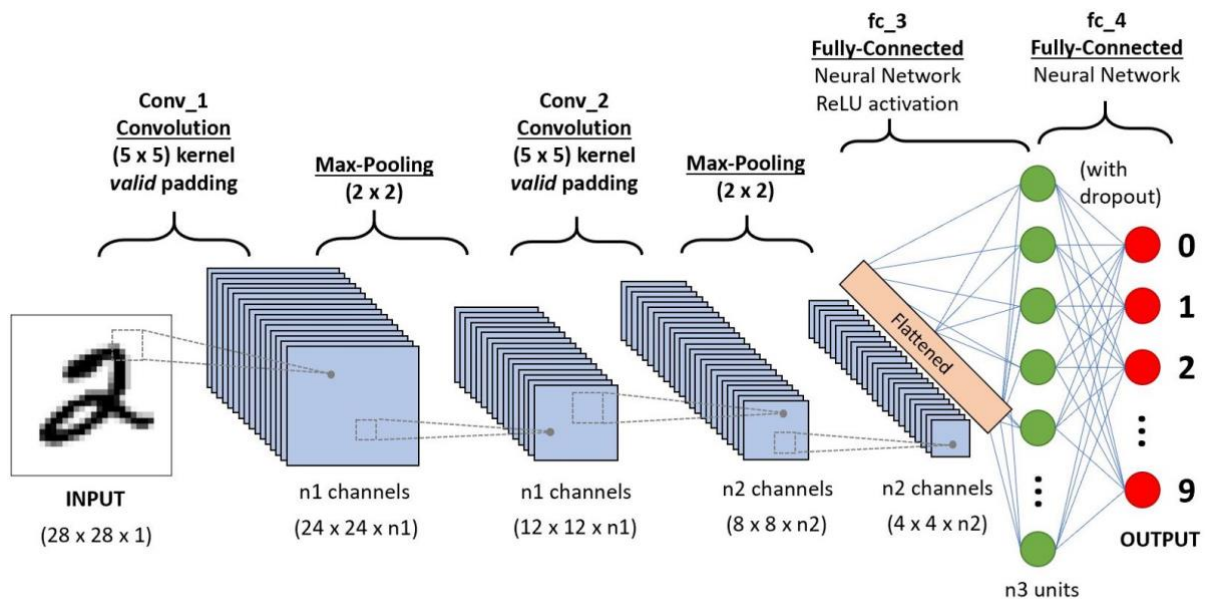
$$w_{ij}^l \leftarrow w_{ij}^l - \alpha x_i^{l-1} \delta_j^l$$

Durant la phase de retro-propagation, on part de la fin du réseau pour remonter vers les couches en amont du réseau en corrigeant les états et en mettant à jour les poids W de chaque réseau.

La descente de gradient permet d'évaluer l'impact d'un poids sur l'erreur et donc de s'améliorer.

Lorsque l'on regarde l'erreur commise par notre perceptron sur une propagation, on peut évaluer l'impact qu'a eu un poids en particulier sur cette erreur. Grâce à cette information, on peut également évaluer si augmenter ou diminuer ce poids va améliorer ou empirer notre erreur.

## Réseaux de neurones convolutifs (CNN)



Un réseau de neurones convolutifs (ConvNet / CNN) est un algorithme d'apprentissage en profondeur qui peut intégrer une image d'entrée, attribuer une importance (poids appris et biais) à divers aspects / objets de l'image et être capable de se différencier les uns des autres.

Le prétraitement requis dans un réseau ConvNet est beaucoup plus faible que d'autres algorithmes de classification.

Alors que dans les méthodes primitives, les filtres sont conçus à la main, avec une formation suffisante, le ConvNet est capable d'apprendre ces filtres/caractéristiques (cf **partie sur le prétraitement d'image**) .

L'architecture d'un ConvNet est analogue à celle du modèle de connectivité des neurones dans le cerveau humain et s'inspire de l'organisation du cortex visuel. Les neurones individuels ne répondent aux stimuli que dans une région restreinte du champ visuel appelée champ récepteur. Une collection de ces champs se chevauchent pour couvrir toute la zone visuelle.

Un ConvNet est capable de capturer avec succès les dépendances spatiales et temporelles dans une image grâce à l'application de filtres appropriés. L'architecture s'ajuste mieux au jeu de données d'image en raison de la réduction du nombre de paramètres impliqués et de la réutilisabilité des poids. En d'autres termes, le réseau peut être formé pour mieux comprendre la sophistication de l'image.

Le rôle de ConvNet est de réduire les images à une forme plus facile à traiter, sans perdre les caractéristiques essentielles à une bonne prédiction. Ceci est important lorsque nous devons concevoir une architecture qui soit non seulement bonne pour l'apprentissage des fonctionnalités, mais également évolutive pour des jeux de données volumineux.



## Fonctionnement

Le ConvNet décrit quatre opérations principales dans la Figure 3 ci-dessus :

- Convolution
- Non linéarité (ReLU)
- Regroupement ou sous-échantillonnage
- Classification (couche entièrement connectée)

Ces opérations sont les éléments de base de *chaque* réseau de neurones convolutionnels. Comprendre leur fonctionnement est donc une étape importante pour développer une compréhension solide de ConvNets. Nous essaierons de comprendre l'intuition de chacune de ces opérations ci-dessous.





## Réseaux de neurones récurrents (RNN)

Définition

Caractéristique

Fonctionnement mathématique

## Réseau hybride (mélange de CNN & RNN)

Définition

Caractéristique

Fonctionnement mathématique

## Architectures d'apprentissage en profondeur récentes

Les réseaux résiduels profonds sont apparus comme une famille d'architectures extrêmement profondes présentant une précision convaincante et de beaux comportements de convergence.

Chacune permet de former avec succès des réseaux profonds en surmontant les limites de la conception de réseau traditionnelle.

La première intuition lors de la conception d'un réseau profond peut consister simplement à empiler bon nombre des blocs de construction typiques tels que des couches convolutives ou entièrement connectées.

Cela fonctionne à un point, mais les performances diminuent rapidement à mesure que le réseau traditionnel devient profond. Le problème découle de la manière dont les réseaux de neurones sont formés par rétropropagation. Lors de la formation d'un réseau, un signal de gradient doit être propagé vers l'arrière sur le réseau, de la couche supérieure à la couche inférieure, afin de garantir que le réseau se met à jour correctement. Avec un réseau traditionnel, ce gradient est légèrement diminué au fur et à mesure qu'il traverse chaque couche du réseau. Pour un réseau ne comportant que quelques couches, ce n'est pas un problème. Cependant, pour un réseau de plus d'une vingtaine de couches,

# Réseau résiduel (ResNets)

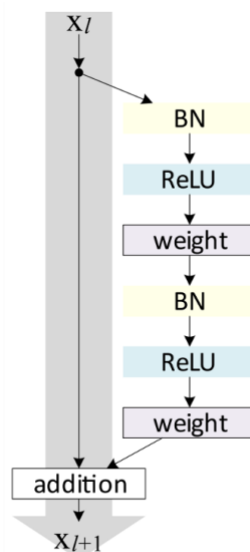
## Définition

Un réseau résiduel, ou ResNet, est une architecture de réseau de neurones qui résout le problème de la disparition de gradients de la manière la plus simple possible. Si vous rencontrez des difficultés pour renvoyer le signal de gradient en arrière, pourquoi ne pas fournir au réseau un raccourci à chaque couche pour que les choses se passent mieux ? Dans un réseau traditionnel, l'activation au niveau d'une couche est définie comme suit :  $y = f(x)$

Où  $f(x)$  est notre convolution, notre multiplication matricielle, notre normalisation par lots, etc. Lorsque le signal est renvoyé, le gradient doit toujours passer par  $f(x)$ , ce qui peut poser problème en raison des non-linéarités impliquées. Au lieu de cela, ResNet implémente à chaque couche :  $y = f(x) + x$

Le « $+ x$ » à la fin est le raccourci. Cela permet au dégradé de revenir directement en arrière. En empilant ces couches, le gradient pourrait théoriquement « sauter » sur toutes les couches intermédiaires et atteindre le fond sans être diminué.

Bien que ce soit l'intuition, la mise en œuvre réelle est un peu plus complexe. Dans la dernière incarnation de ResNets,  $f(x) + x$  prend la forme :



## Caractéristique

## Fonctionnement mathématique

## Réseau routier (HighwayNets)

### Définition

La deuxième architecture que je voudrais présenter est le réseau routier. Il s'appuie sur le ResNet de manière assez intuitive. Le réseau routier conserve les raccourcis introduits dans ResNet, mais les ajoute à un paramètre pouvant être appris pour déterminer dans quelle mesure chaque couche doit être une connexion à ignorer ou une connexion non linéaire. Les couches d'un réseau routier sont définies comme suit :

### Caractéristique

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W}_H) \cdot T(\mathbf{x}, \mathbf{W}_T) + \mathbf{x} \cdot (1 - T(\mathbf{x}, \mathbf{W}_T)). \quad (3)$$

Dans cette équation, nous pouvons voir un aperçu des deux types de couches précédentes :

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W}_H)$$

Reflète notre couche traditionnelle et  $\mathbf{y} = H(\mathbf{x}, \mathbf{W}_H) + \mathbf{x}$  reflète notre unité résiduelle.

La nouveauté est la fonction  $T(\mathbf{x}, \mathbf{W}_T)$ .

Cela permet au commutateur de déterminer dans quelle mesure les informations doivent être envoyées via le chemin primaire ou le chemin sauté. En utilisant  $T$  et  $(1-T)$  pour chacune des deux voies, l'activation doit toujours être égale à 1.

### Fonctionnement mathématique

## Comparaison des différents model selon leurs cas d'usage

Utilisation des MLP dans le monde séculier :

- Jeux de données tabulaires
- Problèmes de prédiction de classification
- Problèmes de prédiction de régression

Les réseaux de neurones convolutifs, ou CNN, ont été conçus pour mapper des données d'image à une variable de sortie.

Ils se sont avérés si efficaces qu'ils sont la méthode de choix pour tout type de problème de prédiction impliquant des données d'image en entrée. L'avantage d'utiliser les CNN est leur capacité à développer une représentation interne d'une image bidimensionnelle comme expliqué dans la partie XX. Cela permet au modèle d'apprendre la position et la mise à l'échelle de variantes de structure dans les données, ce qui est important lorsque vous travaillez avec des images.

Utilisation des CNN dans le monde séculier :

- Données d'image
- Problèmes de prédiction de classification
- Problèmes de prédiction de régression

Les RNN en général et les LSTM en particulier ont eu le plus de succès lorsqu'on travaille avec des séquences de mots et des paragraphes, généralement appelés traitements de langage naturel.

Cela inclut à la fois des séquences de texte et des séquences de langage parlé représentées sous forme de série chronologique. Ils sont également utilisés comme modèles génératifs nécessitant une sortie de séquence, non seulement avec du texte, mais également dans des applications telles que la génération d'écriture manuscrite.

Utilisation des RNN dans le monde séculier :

- Données texte
- Données de la parole
- Problèmes de prédiction de classification
- Problèmes de prédiction de régression
- Modèles génératifs

Afin de valider si ses théories étaient exactes nous avons fait plusieurs tests sur le dataset cifar10 sur les différents modèles afin d'analyser le comportement de chaque modèles en fonction des hyper-paramètres qu'on lui fournis.

# Les différentes fonctions utilisées

## Introduction

Suite aux présentations des différents modèles que nous avons utilisé, nous allons expliquer brièvement dans cette partie les différentes fonctions que l'on peut utiliser dans les modèles de machine/deep learning.

## Optimiseur

### Algorithme qui permet de minimiser la fonction de perte pondérée.

Descente de gradient stochastique (SGD) :

**Fonction sur keras :**

SGD (lr=int, momentum=int, decay=int, nesterov=False)

Momentum : Momentum prend en compte les gradients passés pour aplanir les étapes de la descente. Il peut être appliqué avec une descente par gradient en batch, une descente en gradient par mini-batch ou une descente par gradient stochastique.

batch\_size indique la taille du sous-ensemble de votre échantillon d'apprentissage (par exemple, 100 sur 1 000) qui sera utilisée pour former le réseau au cours de son processus d'apprentissage. Chaque lot entraîne le réseau dans un ordre successif, en tenant compte des poids mis à jour provenant de l'Appliance du lot précédent.

return\_sequence indique si une couche récurrente du réseau doit renvoyer la totalité de sa séquence de sortie (c'est-à-dire une séquence de vecteurs de dimension spécifique) à la couche suivante du réseau, ou tout simplement sa dernière sortie, qui est un vecteur unique de la même dimension. Cette valeur peut être utile pour les réseaux conformes à une architecture RNN.

batch\_input\_shape définit que la classification séquentielle du réseau de neurones peut accepter des données d'entrée de la taille de lot définie uniquement, limitant ainsi la création de tout vecteur de dimension variable. Il est largement utilisé dans les réseaux LSTM empilés.

## Dropout

Dropout prend la sortie des activations de la couche précédente et définit de manière aléatoire une certaine fraction (taux d'abandon) des activations sur 0, en les annulant ou en les « supprimant ».

C'est une technique de régularisation courante utilisée pour prévenir les sur ajustements dans les réseaux de neurones. Le taux d'abandon est l'hyper paramètre ajustable qui est

ajusté pour mesurer les performances avec différentes valeurs. Il est généralement défini entre 0,2 et 0,5 (mais peut être défini de manière arbitraire).

Le décrochage n'est utilisé que pendant la formation ; Au moment du test, aucune activation n'est abandonnée, mais réduite par un facteur de taux d'abandon. Cela tient compte du nombre d'unités actives pendant la période de test par rapport à la durée de formation.

- Learning Rate (Lr) : Le taux d'apprentissage est un hyper-paramètre qui contrôle à quel point nous ajustons les poids de notre réseau en fonction du gradient de perte. Plus la valeur est basse, plus nous roulons lentement sur la pente descendante. Bien que cela puisse être une bonne idée (en utilisant un taux d'apprentissage faible) pour nous assurer que nous ne manquons aucun minimum local, cela pourrait également signifier que nous allons prendre beaucoup de temps pour converger - surtout si nous restons coincés dans une région de plateau. (mettre le schéma de la cuve ici )

$$\text{new\_weight} = \text{existing\_weight} - \text{learning\_rate} * \text{gradient}$$










## Fonctions de perte dans les réseaux de neurones

Les fonctions de pertes utilisées dans Keras :

- Mes: mean\_squared\_error
- Mae: mean\_absolute\_error
- mean\_absolute\_percentage\_error
- mean\_squared\_logarithmic\_error
- squared\_hinge
- hinge
- categorical\_hinge
- logcosh
- categorical\_crossentropy
- sparse\_categorical\_crossentropy
- binary\_crossentropy
- kullback\_leibler\_divergence
- poisson
- cosine\_proximity



## Activation

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

## Les métriques

### Métriques pour évaluer les algorithmes d'apprentissage automatique en Python

Les mesures que vous choisissez pour évaluer vos algorithmes d'apprentissage machine sont très importantes.

Le choix des métriques influence la manière dont la performance des algorithmes d'apprentissage automatique est mesurée et comparée. Ils influencent la manière dont vous pondérez l'importance des différentes caractéristiques dans les résultats et votre choix ultime de l'algorithme à choisir.

Quelle est la différence entre `categorical_accuracy` et `sparse_categorical_accuracy` in Keras ?

Pour l'exemple suivant :

```
def categorical_accuracy(y_true, y_pred):
    return K.cast(K.equal(K.argmax(y_true, axis=-1),
                           K.argmax(y_pred, axis=-1)),
                  K.floatx())

def sparse_categorical_accuracy(y_true, y_pred):
    return K.cast(K.equal(K.max(y_true, axis=-1),
                              K.cast(K.argmax(y_pred, axis=-1), K.floatx()))),
                  K.floatx())
```

#### **Categorical\_accuracy :**

Vérifie si l'index de la valeur vraie maximale est égal à l'index de la valeur prédite maximale.

#### **Sparse\_categorical\_accuracy :**

Vérifie si la valeur vraie maximale est égale à l'index de la valeur prédite maximale.

#### **Confrontation**

Donc, `categorical_accuracy` vous devez spécifier votre target ( y ) en tant que vecteur codé à une seule étape (par exemple, dans le cas de 3 classes, lorsqu'une vraie classe est une seconde classe, elle ydevrait l'être (0, 1, 0)). `sparse_categorical_accuracy` Vous devez uniquement fournir un entier de la vraie classe (dans cas de l'exemple précédent - ce serait 1 comme l'indexation des classes est basée sur).

#### **Précision binaire:**

```
def binary_accuracy ( y_true , y_pred ):  
    retourner K . moyenne ( K . égale ( y_true , K . rond ( y_pred ) ), axe = - 1 )  
K.round (y_pred) implique que le seuil est 0.5, tout ce qui est supérieur à 0.5 sera  
considéré comme correct.
```

## Exactitude catégorique :

```
def categorical_accuracy ( y_true , y_pred ):  
    retourner K . coulée ( K . égale ( K . argmax ( y_true , axe = - 1 ),  
K . argmax ( y_pred , axis = - 1 ) ),  
K . floatx () )  
K.argmax (y_true)  
prend la valeur la plus élevée pour être la prédiction et correspond au jeu de  
comparaison.
```

## Précision catégorique clairsemée :

```
def sparse_categorical_accuracy ( y_true , y_pred ):  
    retourner K . coulée ( K . égale ( K . max ( y_true , axe = - 1 ),  
K . coulée ( K . argmax ( y_pred , axe = - 1 ), K . floatx () ) ),  
K . floatx () )  
Peut-être une meilleure métrique que categorical_accuracy dans certains cas, en  
fonction de vos données.
```

## Précision catégorique maximale :

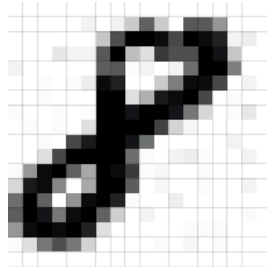
```
def top_k_categorical_accuracy ( y_true , y_pred , k = 5 ):  
    retourner K . moyenne ( K . in_top_k ( y_pred , K . argmax ( y_true , axe = - 1 ), k ), l'axe  
= - 1 )  
Le top-k est mesuré sur la précision de la prédiction correcte dans les prédictions du  
top-k. La plupart des articles exposeront l'efficacité des modèles en fonction de la  
précision du top 5.
```

# Expérimentation

## Introduction

Une image est une matrice de valeurs de pixels.

Essentiellement, chaque image peut être représentée sous la forme d'une matrice de valeurs de pixels.



**Canal** est un terme conventionnel utilisé pour désigner une certaine composante de l'image. Une image d'un appareil photo numérique standard aura trois canaux - rouge, vert et bleu - vous pouvez les imaginer sous la forme de trois matrices 2D superposées (une pour chaque couleur), chacune ayant des valeurs de pixels comprises entre 0 et 255.

Pour l'analyse de nos modèles, nous nous sommes penchés sur le dataset `cifar10`.

## Pre-processing

L'original des données de traitement par lots est une matrice de taille 10 000 x 3072.

Le nombre de colonnes, (10000), indique le nombre de données d'échantillon.

Comme indiqué dans l'ensemble de données CIFAR-10 / CIFAR-100, le vecteur de lignes (3072) représente une image couleur de 32 x 32 pixels.

Étant donné que notre projet utilise des modèles différents, pour les tâches de classification, le vecteur de ligne d'origine n'est pas approprié selon le modèle utilisé.

Dans le code du projet (cf aux différentes fonctions de pre-processing dans chaque *fichier.py* des modèles) nous avons utilisé une fonction nommée *preprocess\_cifar10*, elle représente la fonction mère de tous les modèle dans la classe *ModelManager.py*.

Cette fonction ensuite est surchargé (overrided) dans chaque classe de modèle.

Nous n'allons pas expliquer tous les prétraitements existants pour le dataset `cifar10` mais uniquement celle de *ModelManager.py*

## Transformation de la donnée en fonction du model

Pour insérer des données d'image dans un modèle CNN, la dimension du tenseur en entrée doit être soit (largeur x hauteur x num\_channel) ou (num\_channel x largeur x hauteur). Cela dépend de votre choix (consultez le tensorflow conv2d).

Modifier ce truc en fonction de nos modele (voir dans le code au niveau du preprocessing)

Le vecteur ligne d'une image a exactement le même nombre d'éléments si vous calculez  $32 * 32 * 3 == 3072$ .

Pour reformater le vecteur ligne sous la forme (largeur x hauteur x num\_channel), deux étapes sont nécessaires. La première étape consiste à utiliser la fonction **reshape**, et la deuxième étape consiste à utiliser la fonction **transpose** dans numpy.

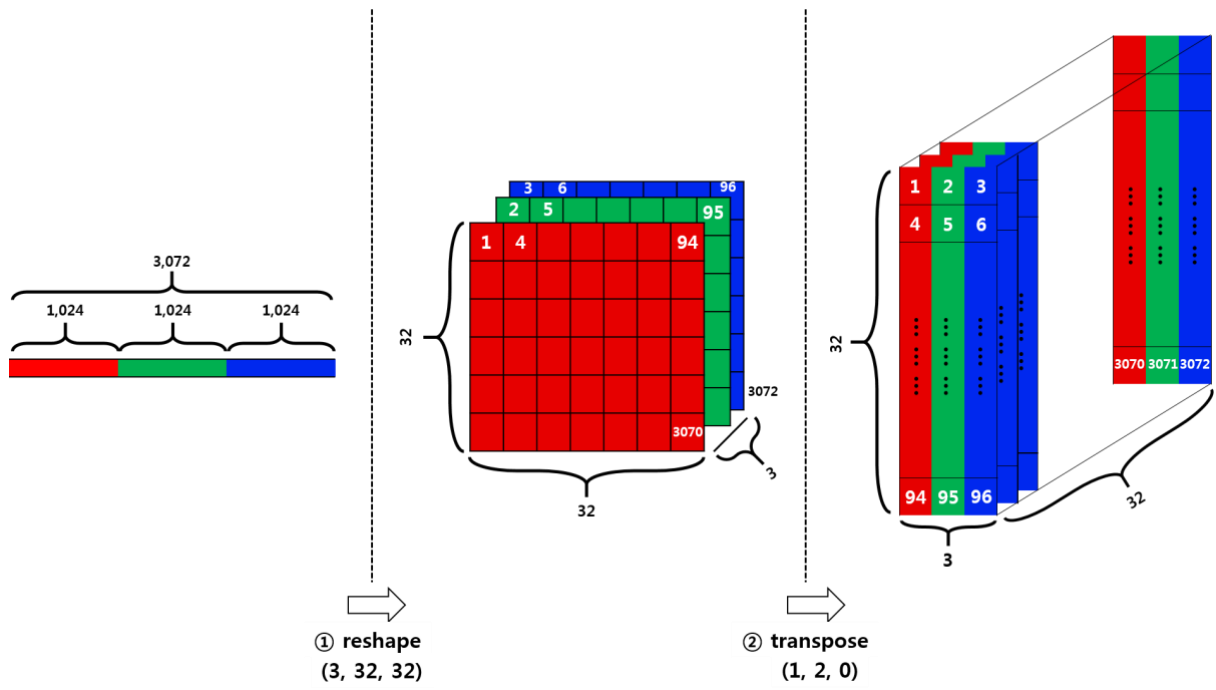
Par définition la fonction *reshapetransforme* un tableau en une nouvelle forme sans modifier ses données. Ici, la phrase sans changer ses données est une partie importante puisque vous ne voulez pas blesser les données. reshapeles opérations doivent être livrées en trois étapes plus détaillées. La direction suivante est décrite dans un concept logique.

Divisez le vecteur de rangée en 3 morceaux, chaque morceau désignant chaque canal de couleur.

- la matrice résultante a (3 x 1024) matrice, ce qui donne un total de (10 000 x 3 x 1024) tenseur.

Divisez les 3 pièces par 32 . 32 est la largeur et la hauteur d'une image.

- cela donne (3 x 32 x 32), ce qui donne un tenseur (10000 x 3 x 32 x 32) au total



## Notre programme

Faire un organigramme ici avec la main qui permet de taper sur chaque file\_model.py  
En mode comme ça :

Nos résultats

Introduction



## Perceptron Simple

## Perceptron Multi-couche

RNN

CNN

## Resnets

