Reconnaissance d'images par réseau de neurones

Travail de maturité de Nicolas Schmid

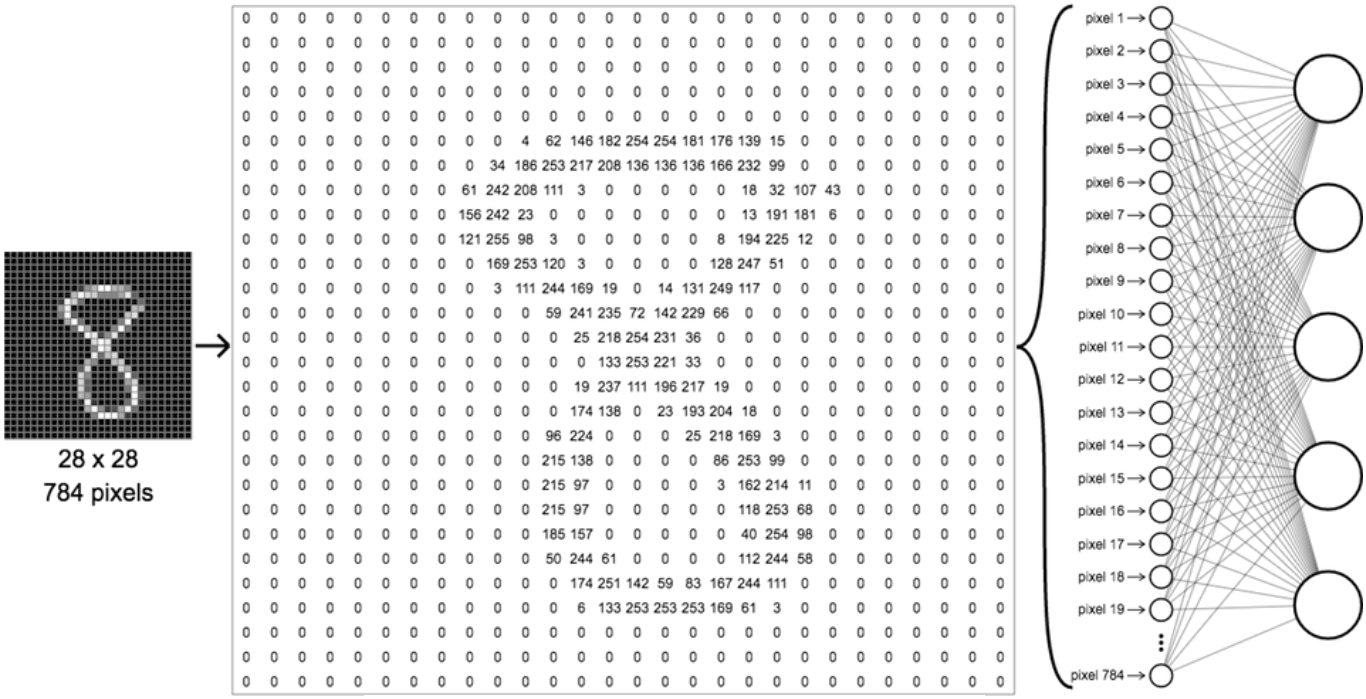


Figure 1 https://ml4a.github.io/ml4a/fr/neural_networks/

Lycée cantonal de Porrentruy, novembre 2018
Maître responsable : Didier Müller
Expert : Damien Dobler
Discipline : Informatique

Table des matières

1. Introduction

La reconnaissance d'image est de plus en plus utilisée de nos jours, que ce soit par exemple pour déverrouiller son téléphone ou encore pour faire rouler des voitures autonomes. Dans ce travail de maturité, nous allons nous intéresser à ce qu'est la reconnaissance d'image et comment cela fonctionne, puis nous verrons de manière plus approfondie comment créer un programme de reconnaissance d'image.

2. Fonctionnement d'un réseau de neurones linéaire

2.1 La reconnaissance d'images

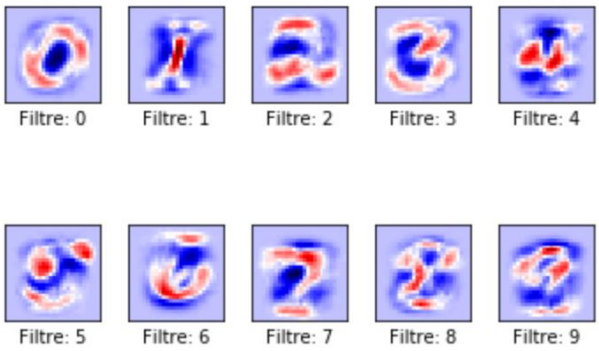
Prenons tout d'abord un cas très simple, où l'on demande à notre programme de reconnaissance d'image de différencier les chiffres de 0 à 9 écrits à la main.

Il est facile pour nous, humains, de différencier ces trois chiffres, respectivement : trois, cinq et huit. Cela semble évident car nous avons appris à analyser les informations que notre rétine envoie à notre cerveau dès notre plus jeune âge, et nous pouvons même reconnaître des images bien plus complexes comme par exemple des visages. Mais pour apprendre à un ordinateur à reconnaître des images, il va nous falloir comprendre de manière plus approfondie ce qu'est réellement la reconnaissance d'image. Prenons ces trois autres chiffres écrits à la main :

La valeur des pixels des images ci-dessus est différente de la valeur des pixels des images présentées plus haut, alors qu'elles représentent les mêmes chiffres. Il existe effectivement un nombre immense de manières d'écrire un trois ou un cinq. Mais d'une certaine façon, notre cerveau parvient à interpréter ces différentes combinaisons de pixels comme étant le même chiffre et de reconnaître d'autres images comme étant des chiffres différents.
De la même manière, un ordinateur devra calculer une combinaison des pixels d'une image pour reconnaître ce qu'elle représente. Pour cela, imaginons que les pixels d'une image en noir et blanc ne sont en réalité que des nombres rationnels compris entre 0 et 1. Un pixel noir est représenté par le chiffre 1 et un pixel blanc par 0. Un pixel gris foncé aura par exemple une

valeur de 0.88. Une des manières les plus simples pour déterminer de quel chiffre il s'agit est d'appliquer des filtres sur cette image.

2.2 Les filtres

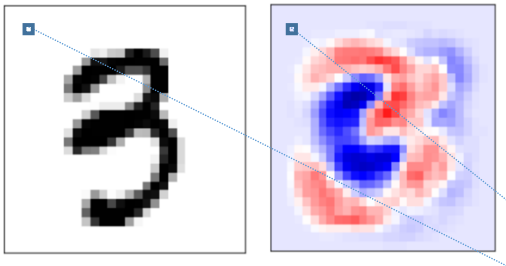


Nous avons ici 10 filtres, un pour chaque chiffre, qui représentent là où devraient se trouver les pixels les plus foncés de l'image (en rouge) et là où ils devraient être le plus clair possible (en bleu), pour que l'image corresponde à tel ou tel chiffre. Par exemple pour le zéro, on constate qu'il y a un cercle rouge rempli de bleu, ce qui signifie qu'il faut, pour qu'une image corresponde à un zéro, que les pixels situés dans la périphérie soient noirs, et que ceux du centre soient blancs. Ces 10 filtres ci-dessus ont été calculés de manière à ce qu'ils différencient de la manière la plus exacte possible les chiffres de 0 à 9.

Voici comment l'ordinateur va procéder pour définir à quel chiffre correspond une image : il va appliquer chaque filtre l'un après l'autre sur notre image, et le filtre donnant le meilleur résultat (celui pour lequel la partie rouge du filtre correspond le mieux aux pixels noirs de l'image) correspondra en général au chiffre représenté sur l'image. Par exemple, si on applique chacun des 10 filtres à une image représentant un 3, ce sera probablement le filtre 3 qui donnera le meilleur résultat. Le problème consiste donc à calculer le résultat de chaque filtre sur notre image.

Imaginons que les filtres soient eux aussi juste une grille de chiffres. Plus un pixel du filtre est rouge, plus le nombre correspondant est positif. Plus il est bleu, plus il est négatif. Pour calculer le résultat d'un filtre sur notre image, il suffit de multiplier la valeur du premier pixel de l'image par le premier pixel du filtre, la valeur du deuxième pixel de l'image par le deuxième pixel du filtre et ainsi de suite. Il faut ensuite faire la somme de ces produits et on obtient le résultat. Cette opération est répétée pour chaque filtre et le filtre ayant donné le plus grand résultat correspondra généralement au chiffre représenté sur l'image. Effectivement, si les pixels noirs d'une image, avec une valeur approchant 1, sont multipliés par des nombres positifs (les pixels rouges du filtre) et que les pixels multipliés par des nombres négatifs (les pixels bleus du filtre) sont blancs, donc proches de zéro, alors le résultat donné par ce filtre sera maximal.

De manière plus formelle, pour une image de 28×28 pixels (comme représenté ci-dessous), voici à quoi correspondra le résultat d'un filtre appliqué sur une image :



$$resultat = \sum_{k=1}^{784} p_k \cdot f_k$$

$$resultat = p_1 \cdot f_1 + p_2 \cdot f_2 + \cdots + p_{59} \cdot f_{59} + \cdots + p_{783} \cdot f_{783} + p_{784} \cdot f_{784}$$

p_n représente un pixel de l'image et f_n le pixel du filtre qui va multiplier ce pixel de l'image. Pour éviter d'avoir à écrire ces longues additions il est également possible de mettre tous ces nombres dans des matrices P et F.

$$P = (p_1 \quad p_2 \quad \cdots \quad p_{783} \quad p_{784}) \quad \text{et} \quad F = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{783} \\ f_{784} \end{pmatrix}$$

En réalité, additionner les produits des p_n et f_n revient à faire le produit matriciel entre P et F. On peut donc écrire :

$$resultat = P \cdot F$$

En l'écrivant ainsi, cela nous évite d'écrire cette énorme somme. De plus, les produits matriciels sont souvent bien optimisés, et demandent donc moins de temps de calcul que si nous donnions à notre programme toutes les opérations à faire l'une après l'autre. Ces optimisations viennent surtout du fait que les opérations matricielles peuvent être effectuées sur la carte graphique de l'ordinateur, qui permet de faire énormément de calculs simples, mais simultanément.*[1] Pour plus de cohérence, nous allons désormais appeler ces résultats des sorties (ou output en anglais).

Remarque : pour calculer les 10 différentes sorties pour chacun des 10 filtres, il n'est pas nécessaire de faire 10 opérations. Il suffit de mettre tous les filtres dans la même matrice de taille 784×10. De cette manière, le produit entre P et F donnera une matrice de 1×10 contenant les 10 sorties y_n.

$$(p_1, \quad p_2, \quad \cdots \quad p_{783}, \quad p_{784}) \cdot \begin{pmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,9} & f_{1,10} \\ f_{2,1} & f_{2,2} & \cdots & f_{2,9} & f_{2,10} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ f_{783,1} & f_{783,2} & \cdots & f_{783,9} & f_{783,10} \\ f_{784,1} & f_{784,2} & \cdots & f_{784,9} & f_{784,10} \end{pmatrix} = (y_1, \quad y_2, \quad \cdots \quad y_9, \quad y_{10})$$

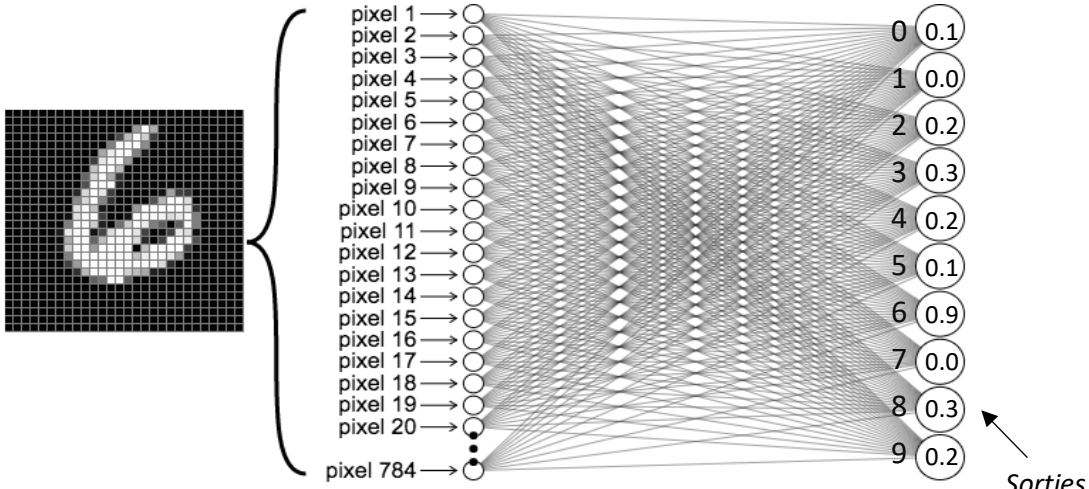
De cette manière, le calculs de l'application des 10 filtres sur une image s'écrit simplement :

$$Y = P \cdot F$$

Cela facilite grandement l'écriture mathématique et réduit donc également la longueur du code nécessaire pour décrire ces opérations.

*[1] « Les réseaux de neurones nécessitent de nombreuses opérations matricielles, qui sont accélérées d'un facteur dix lorsqu'elles sont distribuées sur les milliers de cœurs de GPU. » https://blog.octo.com/classification-dimages-les-reseaux-de-neurones-convolutifs-en-toute-simplicite/

Voici une autre manière de visualiser ce que l'on vient de faire :

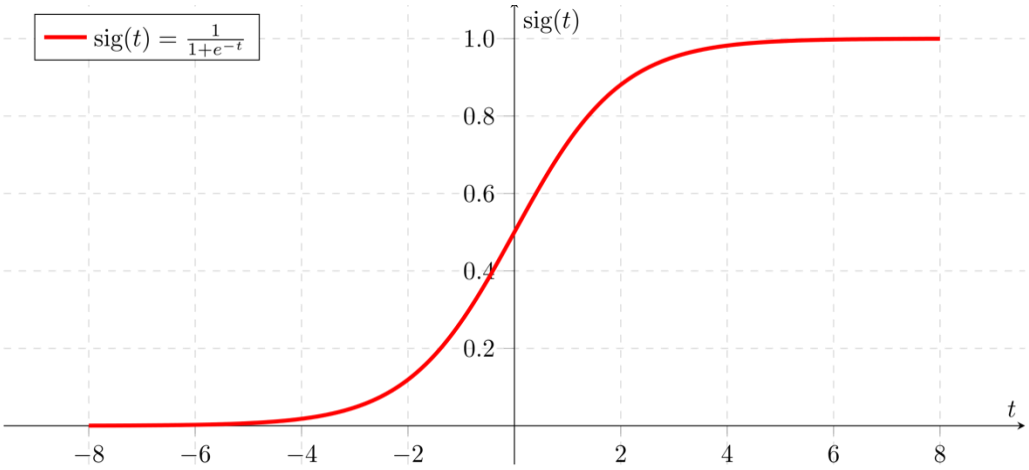


Lien de l'image : https://ml4a.github.io/ml4a/looking_inside_neural_nets/

Chaque trait reliant un pixel à une sortie est un nombre positif ou négatif multipliant la valeur d'un pixel. On va l'appeler « weight » (poids en anglais). Chaque sortie est reliée à tous les 784 pixels qui composent l'image, comme on peut le voir dans l'illustration ci-dessus.
Pour définir la valeur d'une sortie, on prend la somme des produits entre les pixels de l'image et les weight les reliant à cette sortie. Pour l'image ci-dessus, le résultat est le plus grand pour la sortie correspondant au chiffre 6. C'est donc le chiffre 6 qui a le plus de chances d'être représenté sur l'image.

2.3 La fonction sigmoïde

Dans la figure ci-dessus, les sorties affichées (dans les cercles) sont toutes comprises entre 0 et 1. Ce n'est pas un hasard. Cela vient du fait qu'après avoir multiplié et additionné tous ces pixels et ces weights ensemble, les résultats obtenus sont encore évalués par la fonction sigmoïde :



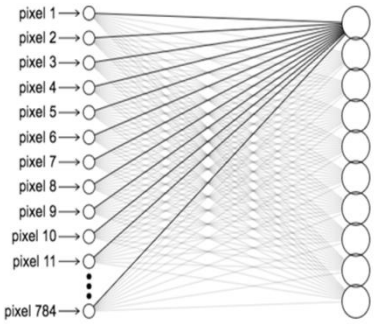
Cette fonction transforme tous les nombres compris dans $\mathbb{R}$ dans l'intervalle]0 ; 1[
Ainsi, si un résultat est très grand, la sortie sera proche de 1 et s'il est très négatif, alors la sortie s'approchera de 0. Donc les sorties vont désormais être définies par :

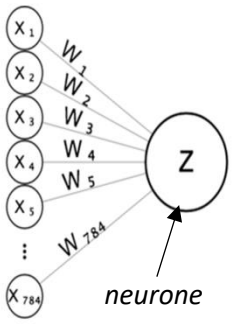
$$Y = sigmoïde(I \cdot F)$$

Remarque : *on considère qu'appliquer la fonction sigmoïde à une matrice revient à appliquer la fonction sigmoïde à chaque élément de la matrice séparément.*

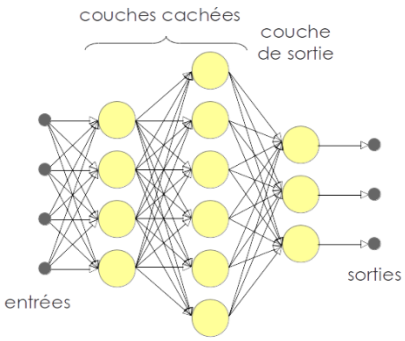
Vous vous demandez peut-être à quoi sert cette fonction sigmoïde. Le but de la fonction sigmoïde est de faire que tous les résultats obtenus se situent entre 0 et 1. Une sortie approchant 0 signifie que le filtre ne correspond pas à l'image évaluée et plus on s'approche de 1, plus il y a de chances que le filtre corresponde à l'image affichée. Une sortie valant plus de 1 ou moins de 0 n'aurait donc aucun sens.
De plus, les données que l'on fournira par la suite à l'ordinateur pour qu'il apprenne à différencier les images sont des vecteurs contenant des 0 et un 1 pour la bonne sortie. Notre but sera de faire que les sorties calculée ressemblent aux données fournies à l'ordinateur. En utilisant la fonction sigmoïde, les nombres que l'on fait passer à travers celle-ci doivent soit être très grand ou soit très petit pour se rapprocher de 1 ou 0, mais ils ne doivent pas être des valeurs exactes. Les sorties calculées correspondront donc mieux aux données fournies. En soit notre réseau pourrait fonctionner sans la fonction sigmoïde, mais il fonctionne mieux avec.

2.4 Les réseaux de neurones





Et avec tout ça, nous n'avons toujours pas parlé de réseau de neurones !
En réalité, un neurone est plus ou moins ce qu'on a appelé jusqu'ici une sortie, c'est le résultat de la somme des produits entre les neurones précédents (ici les pixels) et les synapses (=les weights) reliées à celui-ci. Un neurone est juste un « truc » qui contient un nombre. Plusieurs de ces neurones mis ensemble forment alors un réseau de neurones. On peut imaginer que pour résoudre des problèmes plus complexes comme par exemple reconnaître des visages, un simple réseau avec une seule couche de neurones agissant comme un filtre ne suffira pas. Il faut alors un réseau plus complexe avec plus de couches de neurones pour mieux reconnaître les détails et les petits motifs. Nous reparlerons de cela dans la partie sur les réseaux de neurones à convolution (au chapitre 6).



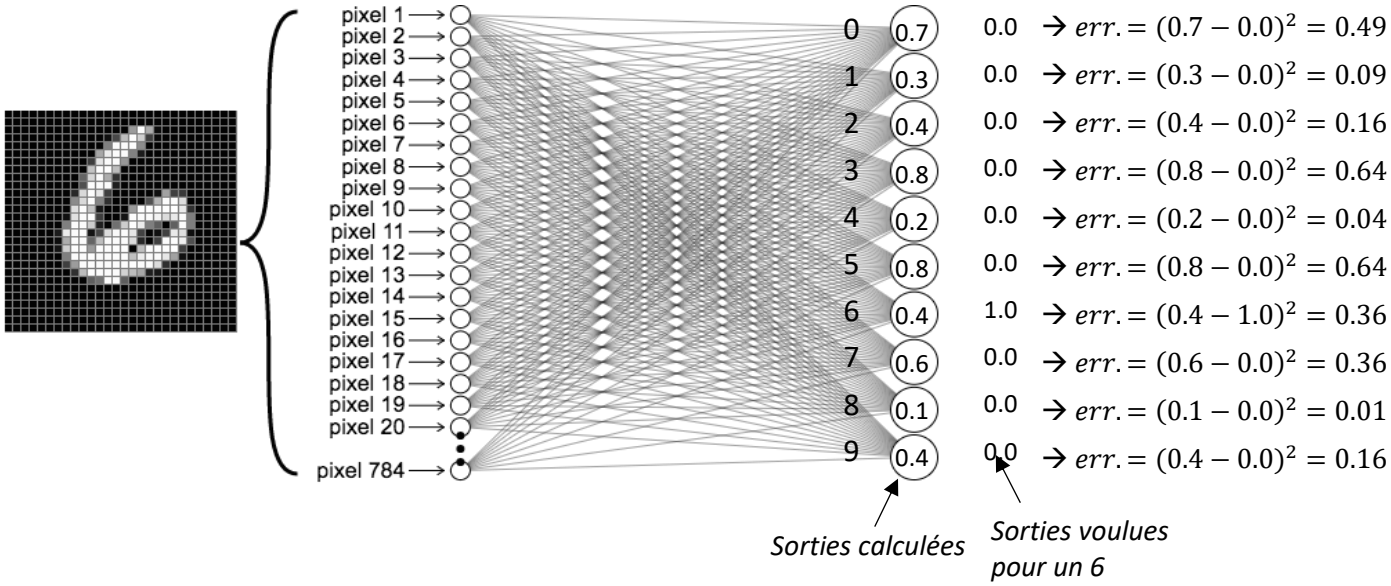
3. Phase d'apprentissage d'un réseau de neurones

Nous savons désormais comment calculer ce que représente une image, à partir de la valeur des pixels d'une image et des filtres, aussi appelés weights. Il suffit de faire le produit matriciel entre les weights et les pixels de l'image, de passer le tout à travers la fonction sigmoïde, puis de prendre l'élément le plus grand dans la matrice contenant les sorties, qui correspond généralement au chiffre représenté sur l'image.
Cependant, nous n'avons pas encore vu comment ces weights étaient choisis. En réalité, trouver ces weights est la partie la plus ardue d'un programme de reconnaissance d'image et correspond à la phase d'apprentissage pour le réseau de neurones. En effet, pour différencier

ces 10 chiffres, il y a 10 filtres de 784 paramètres. cela nous donne 7840 paramètres à ajuster pour que notre réseau soit capable de différencier ces 10 chiffres. Régler tous ces paramètres à la main est alors impensable. Il nous faut donc un algorithme qui nous permette de régler ces différents paramètres afin de rendre notre réseau le plus performant possible. Pour cela il nous faut tout d'abord une valeur qui détermine si notre réseau est performant ou non. Il faut donc calculer l'erreur de notre réseau.

3.1 Calculer l'erreur du réseau de neurones

Pour calculer cette erreur, on donne à notre ordinateur une image de chiffre écrit à la main et les 10 sorties correspondante à cette image, c'est-à-dire un 1 pour la bonne sortie et des 0 pour les 9 autres. On calcule les 10 sorties avec les weights et les pixels de l'image puis on compare ces sorties calculées avec celle que l'on aurait dû obtenir. Le but va donc être de réduire la différence qu'il y a entre les sorties calculées et les sorties correctes.



Pour calculer l'erreur, il faut donc de faire la différence entre les valeurs que l'on a obtenues et les valeurs souhaitées. Cependant, certaines de ces différences sont négatives, il faut pourtant qu'elles s'ajoutent à l'erreur. Pour rendre toutes les erreurs positives, il suffit de les mettre au carré. Il pourrait sembler plus logique de prendre simplement la valeur absolue de chaque erreur à la place de la mettre au carré, mais cela poserait un problème par la suite car la valeur absolue n'est pas dérivable partout, alors qu'une fonction $f(x) = x^2$ est très facilement dérivable. De plus, il est ainsi plus flagrant lorsqu'une sortie ne correspond pas du tout à la valeur attendue, étant donné que l'erreur est élevée au carré. L'erreur totale pour une image est la somme de l'erreur pour chaque sortie.
La notation la plus simple pour décrire cette erreur est une soustraction de matrices entre la matrice des sorties obtenues et la matrice des résultats attendus.

Sorties calculées : $Y = (y_1 \quad y_2 \quad \cdots \quad y_9 \quad y_{10})$
Valeurs souhaitées : $S = (s_1 \quad s_2 \quad \cdots \quad s_9 \quad s_{10})$
Erreurs : $E = (Y - S)^2$

Remarque : *par souci de simplifier l'écriture, on considère qu'élever une matrice au carré revient à élever chaque élément de la matrice au carré séparément.*[*2]

$$erreur\ totale = \sum_{i=1}^{n} e_i$$ → *somme des éléments contenus dans la matrice* E

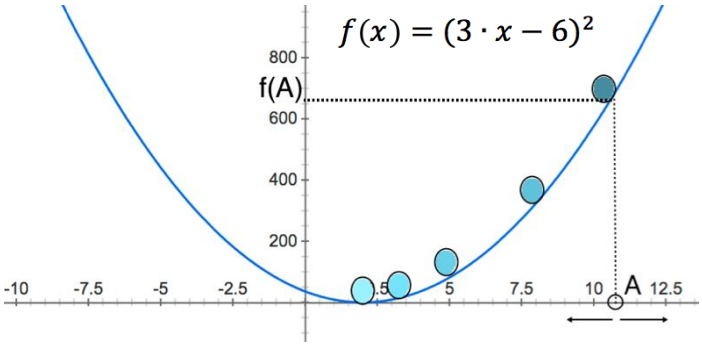
Il existe d'autres manières de définir l'erreur (qu'on appelle aussi généralement le coût) d'un réseau de neurones, mais celle-ci, qu'on appelle l'erreur quadratique, est la plus simple et est assez intuitive.

3.2 Minimisation de l'erreur du réseau de neurones

Pour que notre réseau de neurones soit le plus performant, il faut donc minimiser cette erreur. Minimiser l'erreur du réseau de neurones correspondra à ajuster les paramètres du réseau.

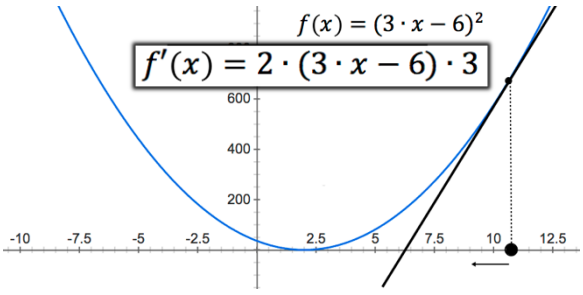
3.2.1 Minimisation d'une fonction simple

Imaginons que l'on ait une fonction simple telle que $f(x) = (3 \cdot x - 6)^2$ et que nous souhaitions minimiser cette fonction, c'est-à-dire trouver le point le plus bas dans le graphe de cette fonction.



Prenons au hasard le point $(A; f(A))$. Afin de minimiser notre fonction, il va falloir aller vers la gauche. C'est en quelque sorte comme si nous posions une balle sur la fonction et que nous la laissions rouler afin qu'elle atteigne le point le plus bas. Mais comment notre balle pourrait-elle savoir dans quelle direction aller ? Eh bien grâce à la pente de la fonction ! Si la pente est positive, comme au point $(A; f(A))$, la balle va aller vers la gauche, donc vers une abscisse plus petite. En revanche si la pente est négative, comme par exemple au point $(-5; f(-5))$, la balle ira vers la droite, vers une abscisse plus grande. Donc dans les deux cas, la direction pour minimiser la fonction correspond à l'opposé de la pente de la fonction. Et il se trouve que la pente d'une fonction n'est autre que sa dérivée.

Pour calculer le minima de cette fonction, il suffit donc de partir d'un point quelconque sur l'axe des abscisses, de calculer la dérivée en ce point, puis de soustraire cette dérivée à ce point. Ceci donnera une nouvelle abscisse. Une fois ceci accompli, il faut alors répéter l'opération depuis le début. Nous allons donc partir d'une abscisse x_i et définir $x_{i+1} = x_i -$



[*2] Je l'écris de cette manière car c'est ainsi que ça apparaîtra dans mes codes en python. Pour être plus correct d'un point de vue mathématique, on peut définir $X = Y - S$ et $erreur\ totale = X \cdot {}^{t}X$. Pour la suite de ce TM, lorsque que je noterai une matrice sera élevée au carré, ce sera en réalité chaque élément élevé au carré.

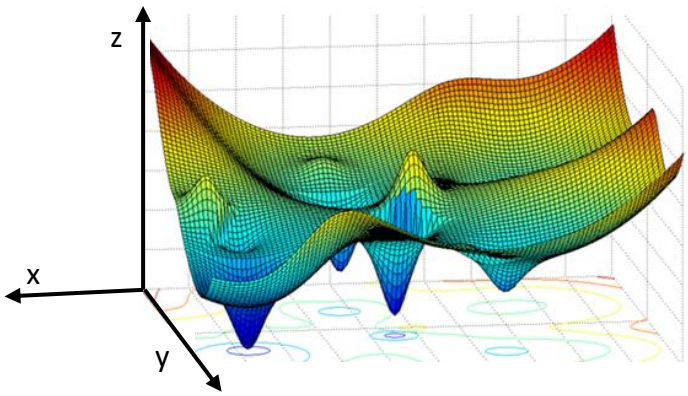
$f'(x_i)$. Cette action sera répétée jusqu'à ce qu'on arrive à un niveau d'exactitude convenable, lorsque la position de la « balle » ne variera quasiment plus.
Cependant, il arrive parfois que la pente de la fonction soit très élevée, et au lieu que notre suite de x_i ne convergent vers un point, celle-ci va diverger. Par exemple, pour la fonction présentée à la page précédente, $f'(11) = 162$ et donc si on part de $x_1 = 11$, alors $x_2 = 11 - 162 = -151$ et $x_3 = -151 - f'(-151) = 2603$ et ainsi de suite. On peut constater que nous nous sommes dirigés dans la bonne direction, mais beaucoup trop vite ! Il faut donc diminuer la vitesse à laquelle nous descendons la pente pour éviter de remonter encore plus haut de l'autre côté. Pour cela, il suffit de diminuer notre vitesse en la multipliant par un nombre compris entre 0 et 1.
Reprenons par exemple $x_1 = 11$, mais cette fois, prenons $x_{i+1} = x_i - f'(x_i) \cdot 0.05$*[3]

- $x_1 = 11$
- $x_2 = 11 - 162 \cdot 0.05 = 2.9$
- $x_3 = 2.9 - f'(2.9) \cdot 0.05 = 0.81$
- $x_4 = 0.81 - f'(0.81) \cdot 0.05 = 1.881$
- $x_5 = 1.881 - f'(1.881) \cdot 0.05 = 1.9881$
- $x_6 = 1.9881 - f'(1.9881) \cdot 0.05 = 1.99881$
- $x_7 = 1.99881 - f'(1.99881) \cdot 0.05 = 1.999881$

Nous pouvons constater que désormais, cette suite converge plus ou moins vers 2, et gagne beaucoup en précision d'une étape à l'autre.

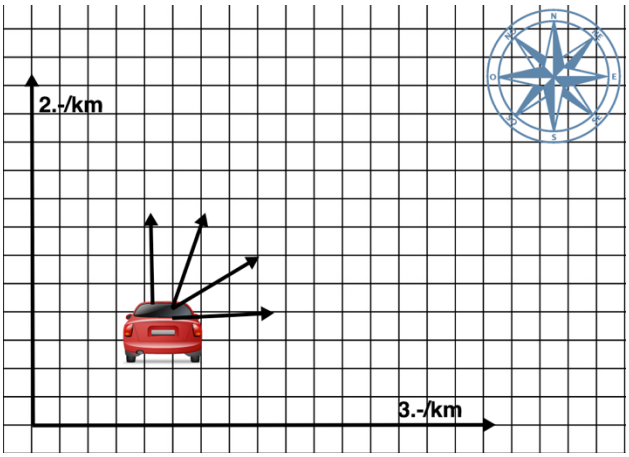
3.2.2 Le gradient d'une fonction

Nous avons vu comment minimiser une fonction simple avec un seul paramètre. Maintenant, compliquons un peu la chose. Imaginons que nous ayons une fonction à 2 variables et que nous souhaitions la minimiser.



Il y a donc 2 variables : *x* et *y* et une sortie : *z*
Cette fois, notre but est d'aller à un point de cette fonction où la cote est la plus petite possible. On peut à nouveau s'imaginer une balle qu'on placerait à un point quelconque de la fonction et qui roulerait vers son point le plus bas. En revanche, la direction que devrait emprunter la balle n'est plus seulement droite ou gauche, mais un vecteur direction à 2 dimensions, qui va dire où la balle est censée aller en *x* et en *y* afin que la valeur en *z* diminue le plus rapidement possible. Ce vecteur est l'opposé de ce qu'on appelle le gradient d'une fonction. Le gradient d'une fonction correspond en quelque sorte à sa pente. Il nous faut donc calculer ce gradient.

[3] À noter que le facteur 0.05 n'a pas été choisi totalement au hasard. J'ai essayé différents facteurs, pas trop grands, de manière à ce que la suite ne diverge pas, mais pas trop petit afin de se rapprocher assez rapidement de la solution. Il y a surement un moyen de calculer le facteur optimal, mais en général, en reconnaissance d'image, ce facteur est choisi de manière empirique, en testant différents facteurs et gardant celui qui donne les meilleurs résultats.



Pour mieux se représenter ce qu'est ce gradient, imaginons un chauffeur qui roule sur une grande plaine et qui serait payé 2.- pour chaque kilomètre qu'il effectuerait en direction du nord et 3.- pour chaque kilomètre qu'il effectuerait en direction de l'est. Il serait normal pour lui de se demander quelle direction emprunter afin de gagner le plus d'argent pour un certain nombre de kilomètres. Imaginons qu'il roule sur une distance de 10 kilomètres : s'il fait ses 10 kilomètres en direction du nord, il gagnera 20.- et s'il fait ses 10 kilomètres en direction de l'est, il gagnera 30.- mais s'il fait ses 10 kilomètres en direction du nord-est, il gagnera 35.4 francs. Donc la bonne direction à prendre se trouve donc entre le nord et l'est. Mais il semble logique que cette direction aille légèrement plus vers l'est que vers le nord, étant donné qu'il gagne plus en allant vers l'est. Pour maximiser l'argent gagné, il faut qu'un petit changement de position sur notre plan crée le plus grand changement possible sur notre argent. Il faut donc que $\frac{\delta Argent}{\delta position}$ soit le plus grand possible. Mais $\delta position = \begin{pmatrix}\delta x \\ \delta y\end{pmatrix}$ et donc la direction où aller pour qu'un petit changement de position crée le plus grand changement d'argent est aussi un vecteur de 2 dimensions :

$$meilleure\ direction = \begin{pmatrix}\frac{\delta Argent}{\delta x} \\ \frac{\delta Argent}{\delta y}\end{pmatrix}$$

Et c'est cela que l'on appelle un gradient. C'est un vecteur composé des dérivées partielles de chaque variable de la fonction. On peut le calculer pour notre fonction :

$$Argent(x,y) = 3x + 2y \qquad \Longrightarrow \quad \frac{\delta Argent}{\delta x} = 3 \qquad \frac{\delta Argent}{\delta y} = 2$$

Voici donc le gradient de cette fonction : $gradient = \begin{pmatrix}3 \\ 2\end{pmatrix}$

Cela représente la direction que devrait emprunter notre chauffeur pour gagner le plus d'argent. Donc pour 3 kilomètres vers l'est, notre chauffeur devra faire 2 kilomètres vers le nord et ainsi, pour 10 kilomètres, il gagnera 36.1 francs, ce qui est la somme maximale qu'il puisse gagner.
Ce que nous venons de réaliser est valable avec 2 variables mais aussi avec 3,4,10 ou même 7840 variables, mais se représenter mentalement une fonction avec autant de variables est quasiment impossible pour nous. En revanche calculer les dérivées partielles d'une fonction à 7840 variables, un ordinateur y arrive normalement sans trop de soucis.

3.2.3 Minimisation de l'erreur pour une image

Mais reprenons notre réseau de neurones. Afin de l'entraîner, nous souhaitions minimiser l'erreur de notre réseau qui en réalité est une fonction compliquée avec 7840 variables. C'est une fonction compliquée, mais ça reste juste une fonction que l'on peut minimiser en calculant son gradient. Petit rappel : nous avions défini l'erreur comme ceci :

Erreurs : $E = (Y - S)^2$ $\quad erreur\ totale = \sum_{i=1}^{n} e_i$ → *somme des éléments de la matrice* E

Nous avons vu au chapitre 2.2 comment calculer la matrice Y contenant les sorties du réseau de neurones. Il suffit de faire le produit matriciel entre la matrice contenant les pixels de l'image et la matrice contenant les filtres, puis de passer le tout à travers la fonction sigmoïde.

$$erreur\ totale = sum((sigmoïde(P \cdot W) - S)^2)$$

- *P = matrice contenant les pixels de taille 1×784*
- *W = matrice contenant les weights (=filtres) de taille 784×10*
- *S = une matrice de 1×10 contenant des 0 et un 1 pour le bon chiffre*
- *sum(vecteur) est une fonction qui additionne tous les éléments d'un vecteur entre eux*[4]*

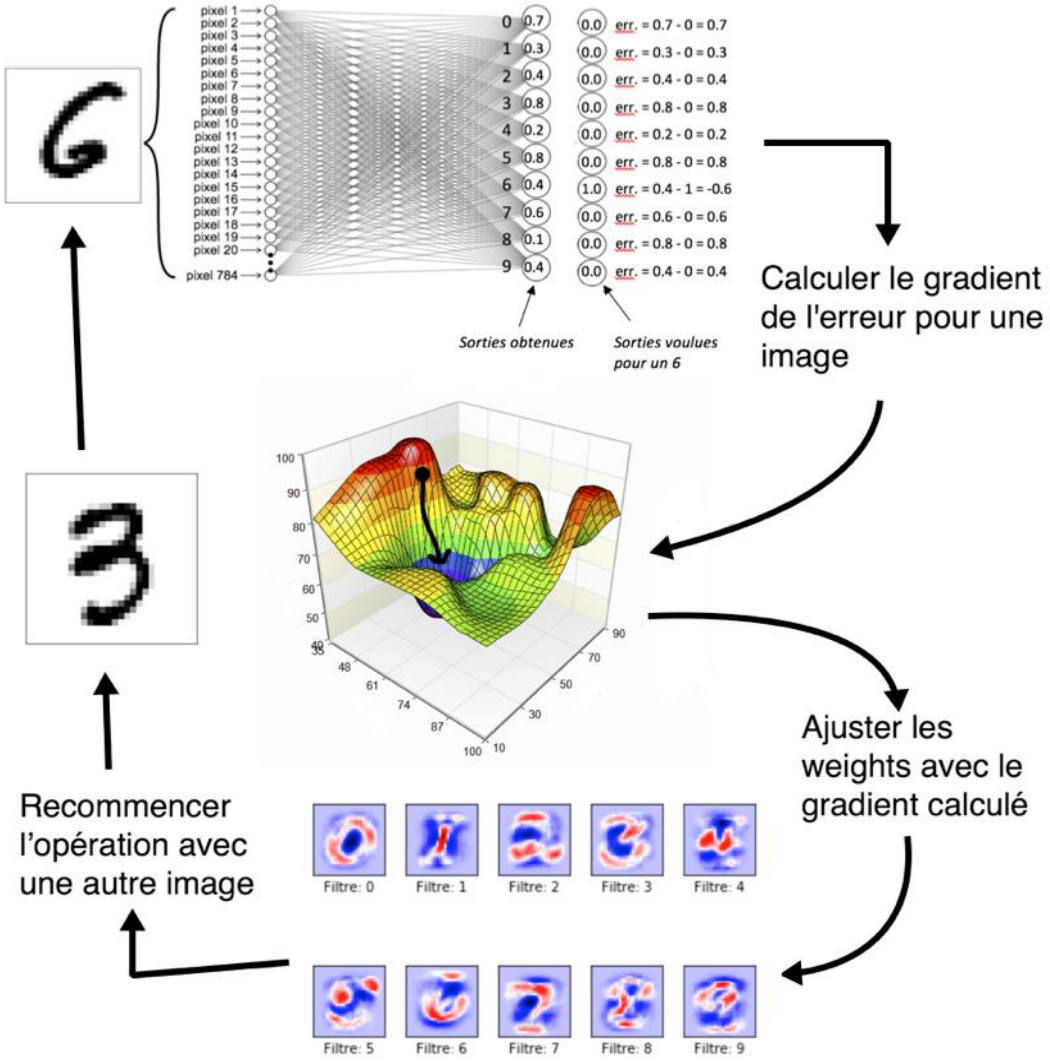
Attention ! Il faut bien noter que P et $sorties\ attendues$ ne sont pas les variables de la fonction d'erreur, ils sont comme le 3 et le -6 dans $f(x) = (3 \cdot x - 6)^2$. Les seules variables de cette fonction d'erreur sont contenues dans la matrice W. → On va pouvoir faire varier l'erreur en changeant la valeur des weights contenues dans W. On ne va en aucun cas faire varier la valeur des pixels des images pour entrainer le réseau.

3.2.4 Ajustement des weights du réseau de neurones

Pour que notre réseau apprenne à reconnaître des images, nous allons donc lui fournir des milliers d'images ainsi que les sorties que devraient nous donner ces images. Pour minimiser l'erreur de notre réseau, nous allons procéder de la même manière que nous avons vu précédemment. Afin de minimiser la fonction $f(x) = (3 \cdot x - 6)^2$ nous avions défini que $x_{i+1} = x_i - f'(x_i) \cdot 0.05$. Pour minimiser l'erreur du réseau de neurones, c'est quasiment la même chose. La fonction qui décrit l'erreur de notre réseau est :
$erreur(W) = sum((sidmoïde(P \cdot W) - S)^2)$. Nous allons poser que $W_{i+1} = W_i - Gradient(W_i) \cdot 0.05$ *[5]. Ainsi, à chaque étape, W_{i+1} sera légèrement plus correct que W_i et c'est donc ainsi que l'on va minimiser l'erreur du réseau de neurones.
L'ordinateur va calculer le gradient de l'erreur pour chaque image, ajuster les weigths grâce à ce gradient, puis répéter cette opération aussi souvent que possible.
Pour chaque image que nous allons donner à notre réseau de neurones, celui-ci va ajuster ses weights w_i contenus dans la matrice W, et ainsi devenir de plus en plus performant, minimisant l'erreur à chaque étape.

*[4] *Je choisi de l'écrire de cette manière car c'est ainsi que ça s'écrit en Python.*
*[5] Ce facteur est choisi de manière arbitraire, il faudra le changer en fonction du problème à résoudre.

Donc derrière cette chose mystérieuse au premier abord qu'est l'apprentissage par ordinateur, ne se cache en réalité que de l'optimisation de fonction. Tout ce qu'il faut est donc d'être capable de calculer le gradient de l'erreur de notre réseau de neurones.



4. Calcul du gradient

Nous allons maintenant voir comment calculer le gradient de l'erreur d'un réseau de neurones. Pour simplifier le calcul du gradient, nous allons prendre un réseau de neurones sans la fonction sigmoïde. C'est-à-dire que les dix différentes sorties seront simplement calculées comme nous l'avons vu au point 2.2. Voici donc l'erreur du réseau que nous tenterons de minimiser : $erreur = sum((P \cdot W - S)^2)$

4.1 Calcul du gradient d'une fonction simple

Pour commencer, nous allons considérer une fonction avec 5 variables dont nous allons calculer le gradient. Je rappelle que le gradient correspond à un tableau contenant les dérivées partielles de chaque variable.

Prenons $f(w_1, w_2, w_3, w_4, w_5) = (aw_1 + bw_2 + cw_3 + dw_4 + ew_5 - s)^2 \qquad f: \mathbb{R}^5 \rightarrow \mathbb{R}$

La dérivée partielle de w_1 est donc : $\frac{\delta f}{\delta w_1} = 2(aw_1 + bw_2 + cw_3 + dw_4 + ew_5 - s) \cdot a$
La dérivée partielle de w_2 est donc : $\frac{\delta f}{\delta w_2} = 2(aw_1 + bw_2 + cw_3 + dw_4 + ew_5 - s) \cdot b$
La dérivée partielle de w_3 est donc : $\frac{\delta f}{\delta w_3} = 2(aw_1 + bw_2 + cw_3 + dw_4 + ew_5 - s) \cdot c$
On peut constater que ces dérivées ne sont pas très compliquées et très semblables l'une à l'autre. Si on pose $k = 2(aw_1 + bw_2 + cw_3 + dw_4 + ew_5 - s)$
$\frac{\delta f}{\delta w_1} = k \cdot a \quad \frac{\delta f}{\delta w_2} = k \cdot b \quad \frac{\delta f}{\delta w_3} = k \cdot c \quad \frac{\delta f}{\delta w_4} = k \cdot d \quad \frac{\delta f}{\delta w_5} = k \cdot e$

Écrit plus simplement : $Gradient = \begin{pmatrix} \frac{\delta f}{\delta w_1} \\ \frac{\delta f}{\delta w_2} \\ \frac{\delta f}{\delta w_3} \\ \frac{\delta f}{\delta w_4} \\ \frac{\delta f}{\delta w_5} \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \\ d \\ e \end{pmatrix} \cdot k$ $\Longrightarrow$ Il nous faut donc calculer k

$$k = 2(aw_1 + bw_2 + cw_3 + dw_4 + ew_5 - s) = 2 \cdot ((a, \quad b, \quad c, \quad d, \quad e) \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{pmatrix} - s)$$

On pose que $P = (a, \quad b, \quad c, \quad d, \quad e)$ et $W = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{pmatrix} \Longrightarrow k = 2 \cdot (P \cdot W - s)$

Voilà donc à quoi correspond le gradient :

$$Gradient = \begin{pmatrix} a \\ b \\ c \\ d \\ e \end{pmatrix} \cdot k = {}^tP \cdot \big(2 \cdot (P \cdot W - s)\big)$$

Donc le gradient de $f(W) = (P \cdot W - s)^2$ correspond à : ${}^tP \cdot 2\,(P \cdot W - s)$

4.2 Calcul du gradient de l'erreur d'une sortie du réseau

Nous allons maintenant calculer le gradient d'une fonction à 784 variables.
Prenons : $f(w_1, w_2, \dots, w_{783}, w_{784}) = (w_1p_1 + w_2p_2 + \cdots + w_{783}p_{783} + w_{784}p_{784} - s)^2$
Cette fonction correspond à l'erreur de l'application d'un des 10 filtres sur une image de 28x28 pixels. Ce gradient nous permettra d'ajuster les weights reliés à une des 10 sorties du réseau.

$W_n = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_{784} \end{pmatrix}$ et $P = (p_1, \quad p_2, \quad p_3, \quad \cdots \quad p_{784})$ et $s_n = sortie\ attendue$

Rien ne change par rapport à avant ! On peut effectuer exactement le même raisonnement que pour 5 variables. Donc le gradient est facile à calculer :

$$Gradient_n = {}^tP \cdot \big(2 \cdot (P \cdot W_n - s_n)\big)$$

4.3 Calcul du gradient de l'erreur d'une image

Cela correspond au gradient de l'erreur sur une sortie. Or, il y a 10 sorties. Nous avions définit l'erreur totale: $erreur\ totale = e_1 + e_2 + \cdots + e_{10}$
On pose que W_1 est la matrice des weights pour la première sortie, W_2 est la matrice des weights pour la deuxième sortie etc.

$$W_1 = \begin{pmatrix} w_{1,1} \\ w_{2,1} \\ w_{3,1} \\ \vdots \\ w_{784,1} \end{pmatrix} \quad W_2 = \begin{pmatrix} w_{1,2} \\ w_{2,2} \\ w_{3,2} \\ \vdots \\ w_{784,2} \end{pmatrix} \quad \cdots$$

En résumé nous avons donc:

$$e_1 = (P \cdot W_1 - s_1)^2$$
$$e_2 = (P \cdot W_2 - s_2)^2$$
$$\cdots$$
$$e_{10} = (P \cdot W_{10} - s_{10})^2$$

Comme on peut le constater, l'erreur totale correspond la somme des e_i pour $1 \leq i \leq 10$. Mais comme e_1 ne dépend pas des mêmes variables W_1 que e_2 ou e_3, cela signifie qu'on peut calculer e_1, e_2, e_3, e_4, etc. indépendamment les uns des autres.
Par exemple si on prend la dérivée partielle de $x_1$dans $f(x_1, x_2) = 3x_1 + 6x_2 + 8$
On constate que $\frac{\delta f}{\delta x_1}$ ne dépend pas de x_2. La dérivée partielle de x_1 serait la même si on avait : $f(x_1, x_2) = 3x_1 - 4x_2 + 17 \Rightarrow$ les dérivées partielles ne dépendent pas l'une de l'autre.
On peut donc calculer les gradients pour l'erreur de chaque sortie séparément*[6] :

$$grad_1 = 2 \cdot {}^tP \cdot (P \cdot W_1 - s_1)$$
$$grad_2 = 2 \cdot {}^tP \cdot (P \cdot W_2 - s_2)$$
$$\cdots$$
$$grad_{10} = 2 \cdot {}^tP \cdot (P \cdot W_{10} - s_{10})$$

Mais à la place de faire ces 10 opérations séparément, on peut tout réunir dans une seule matrice. Il suffit de poser :

$$W = (W_1, \quad W_2, \quad \cdots \quad W_9, \quad W_{10}) \qquad Grad = (grad_1, \quad grad_2, \quad \cdots \quad grad_9, \quad grad_{10})$$
$$s = (s_1 \quad s_2 \quad \cdots \quad s_9 \quad s_{10})$$

De cette manière, on peut facilement calculer le gradient pour les dix sorties à la fois :

$$Grad = 2 \cdot {}^tP \cdot (P \cdot W - s)$$

$Grad$ correspond au gradient de $erreur\ totale = sum((P \cdot W - s)^2)$ *[7] C'est donc le gradient que nous cherchions à calculer depuis le début, et nous savons désormais le calculer.

*[6] Cela n'est valable que pour cette forme particulière de fonction.

*[7] Il faut comprendre qu'ici, ce n'est pas la matrice $\mathrm{M} = (\mathrm{P} \cdot \mathrm{W} - \mathrm{s})$ qui est élevée au carré, mais bien chaque élément de la matrice séparément. J'ai fait ce choix d'écriture car il est moins encombrant et de plus, c'est ainsi que ça s'écrit en python.

Attention ! il faut bien penser que W_1, W_2 ou encore $grad_1$ sont eux-mêmes de taille 784×1. Donc si on voulait représenter les matrices W et $Grad$ telle qu'elles sont réellement voici ce que ça donnerait :

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,9} & w_{1,10} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,9} & w_{2,10} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{783,1} & w_{783,2} & \cdots & w_{783,9} & w_{783,10} \\ w_{784,1} & w_{784,2} & \cdots & w_{784,9} & w_{784,10} \end{pmatrix} Grad = \begin{pmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,9} & g_{1,10} \\ g_{2,1} & g_{2,2} & \cdots & g_{2,9} & g_{2,10} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ g_{783,1} & g_{783,2} & \cdots & g_{783,9} & g_{783,10} \\ g_{784,1} & g_{784,2} & \cdots & g_{784,9} & g_{784,10} \end{pmatrix}$$

Nous avons donc réussi à calculer le gradient de l'erreur sans la fonction sigmoïde. Voici le gradient pour l'erreur avec la fonction sigmoïde :

$$Grad = 2 \cdot {}^tP \cdot [sigmoïde(P \cdot W) - s] \cdot sigmoïde(P \cdot W) \cdot [1 - sigmoïde(P \cdot W)]$$

C'est certes un peu plus compliqué que sans la fonction sigmoïde, mais dans l'idée, ça reste la même chose. Le développement pour calculer ce gradient se trouve en (Annexe 1). Mais pour la suite, afin d'éviter de rendre les choses trop compliquées, nous allons rester sur la définition du gradient de l'erreur sur une image comme étant :

$$Grad = 2 \cdot {}^tP \cdot (P \cdot W - s)$$

4.4 Ajustement des variables du réseau de neurones

Pour que notre programme apprenne, il faut lui donner plein d'images afin qu'il calcule le gradient de l'erreur pour chaque image puis modifie les weights en leur soustrayant le gradient calculé $\Longrightarrow W_{i+1} = W_i - Gradient(W_i) \cdot 0.05$
On sait désormais que :

$$Gradient(W) = 2 \cdot {}^tP \cdot (P \cdot W - s)$$

On peut donc définir W_{i+1}:

$$W_{i+1} = W_i - 2 \cdot {}^tP \cdot (P \cdot W - s) \cdot 0.05$$

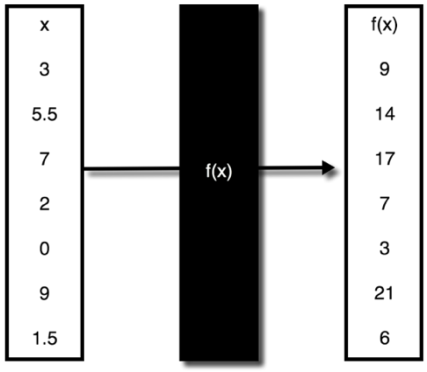
Nous avons donc quasiment toutes les clés en main pour créer notre premier programme de reconnaissance d'images.

5. Un peu de code…

Nous allons maintenant nous intéresser à mettre tout ce que nous venons d'apprendre sous la forme d'un programme. Nous allons commencer par des programmes courts et simples, puis progressivement ajouter de la complexité, jusqu'à arriver à un programme qui reconnaît les chiffres de 0 à 9.

5.1 Ajustement des paramètres d'une fonction simple

Prenons un problème que vous avez sans doute déjà rencontré : on nous donne des nombres x_i et les sorties $f(x_i)$ auxquels ils correspondent. Le but est de déterminer la fonction qui lie ces nombres à leur sortie respective.



Partant du principe que nous savons que nous avons affaire une fonction de type : $f(x) = ax + b$ alors il n'est pas si difficile de créer un programme qui trouve les valeurs de a et b en connaissant la valeur de $f(x_i)$ pour chaque entrée x_i.
Pour cela, il faut tout d'abord définir l'erreur que nous voulons minimiser. Ici l'erreur correspond à : $erreur = (ax + b - f(x))^2$. Or, pour minimiser l'erreur nous avons vu précédemment qu'il suffit de prendre le gradient de l'erreur et de le soustraire à nos deux paramètres que sont a et b. On peut calculer la dérivée partielle $\frac{\delta erreur}{\delta a} = 2\big(ax + b - f(x)\big) \cdot x$ et la dérivée partielle $\frac{\delta erreur}{\delta b} = 2\big(ax + b - f(x)\big)$.
Tout ce qu'il nous faut pour minimiser l'erreur, c'est calculer le gradient de l'erreur pour chaque couple $(x, f(x))$ puis soustraire ce gradient, multiplié par un nombre compris entre 0 et 1, aux paramètres a et b. De cette manière a et b vont se rapprocher des valeurs 2 et 3, car en l'occurrence $f(x) = 2x + 3$.

Essayons de mettre tous cela sous la forme d'un programme :

$a = 0, b = 0$ (on initialise les paramètres a et b)
$liste\ entrées = [3,\ 5.5,\ 7,\ 2,\ 0,\ 9,\ 1.5]$ et $liste\ sorties = [9,\ 14,\ 17,\ 7,\ 3,\ 21,\ 6]$

Pour i variant de 0 à 300 : ← *nombre de fois que nous allons calculer le gradient*

$indice\ =$ i modulo la longueur de la liste d'entrée
De cette manière on va prendre plusieurs fois chaque élément des listes.

$entrée\ =\ liste\ entrées_{indice}$ → *par exemple liste entrée₃ est le quatrième élément de la liste si on commence à numéroter les éléments de la liste à partir de 0*

$sortie\ =\ liste\ sorties_{indice}$ → *par exemple liste sortie₃ = 7 si on commence à numéroter les éléments de la liste à partir de 0*

On sait que l'erreur correspond à $(ax + b - f(x))^2$ ⇒ *nous pouvons donc calculer :*
$\delta erreur \delta a = 2(a \cdot entrée + b - sortie) \cdot entrée$
$\delta erreur \delta b = 2(a \cdot entrée + b - sortie)$
→ *On prend la dérivée partielle pour chaque paramètre.*

$a := a - \delta erreur \delta a \cdot 0.05$
$b := b - \delta erreur \delta b \cdot 0.05$
→ *On modifie ensuite ces paramètres grâce aux dérivées partielles multipliées par un petit nombre (compris entre 0 et 1) afin d'éviter que nos paramètres divergent.*

Une fois ces opérations effectuées, on revient au début de la boucle en incrémentant i de 1.

Si on demande ensuite à notre programme quelles sont les valeurs de a et b après avoir exécuté ces calculs, il en sortira des valeurs très proches 2 et 3. Voici le programme en python :

In [ ]:
```
a=0
b=0
vitesse=0.05

liste_entree=[3,5.5,7,2,0,9,1.5]
liste_sortie = [9,14,17,7,3,21,6]
longeur_listes=len(liste_entree)

for i in range(300):
    indice = i%longeur_listes
    entree = liste_entree[indice]
    sortie = liste_sortie[indice]
    #erreur = (a*entree+b-sortie)**2
    derreur_da = 2*(a*entree+b-sortie)*entree
    derreur_db = 2*(a*entree+b-sortie)
    a-= derreur_da*vitesse
    b-= derreur_db*vitesse

    if i%20==0:
        print("Predictions  a : {a:.5f}  b : {b:.5f}".format(a=a, b=b))
```

Et voici ce que ce programme affiche :

```
Predictions  a : -1.87110  b : 2.39302
Predictions  a : 3.55120  b : 3.45606
Predictions  a : 2.07349  b : 2.92507
Predictions  a : 1.97848  b : 3.02438
Predictions  a : 2.00849  b : 2.99395
Predictions  a : 2.00025  b : 3.00216
Predictions  a : 2.00019  b : 2.99942
Predictions  a : 1.99892  b : 2.99983
Predictions  a : 2.00043  b : 3.00013
Predictions  a : 2.00002  b : 2.99998
Predictions  a : 1.99999  b : 3.00001
Predictions  a : 2.00000  b : 3.00000
```

5.2 Calcul du gradient avec des matrices

Écrire comme ci-dessus un programme avec plus de 7000 paramètres prendrait énormément de temps, puisque chaque paramètre nécessite un certain nombre de lignes de code. Il nous faut donc être un peu plus malin. L'astuce consiste à mettre les paramètres dans des matrices et ainsi écrire tous les calculs à effectuer en seulement quelques opérations matricielles.

Prenons comme exercice une fonction $f(x_1, \dots, x_5) = 2x_1 - x_2 + 4x_3 + 5x_4 - 3x_5$ dont l'ordinateur n'a pas connaissance des paramètres 2, -1, 4, 5 et -3. Il connaît uniquement les entrées x_1, x_2, x_3, x_4, x_5 et la sortie à laquelle ils correspondent. Nous allons créer un programme qui va retrouver ces paramètres uniquement grâce aux entrées et aux sorties correspondantes que nous lui fournirons. Il faut d'abord définir l'erreur de notre fonction :

$$erreur = (ax_1 + bx_2 + cx_3 + dx_4 + ex_5 - sortie)^2$$

Pour ajuster les paramètres a, b, c, d, e il faut calculer le gradient de l'erreur grâce aux entrées fournies et à leur sortie correspondante. Le calcul du gradient revient à calculer la dérivée partielle de chaque paramètre a, b, c, d, e.

$$\frac{\delta erreur}{\delta a} = 2 \cdot (ax_1 + bx_2 + cx_3 + dx_4 + ex_5 - sortie) \cdot x_1$$
$$\frac{\delta erreur}{\delta b} = 2 \cdot (ax_1 + bx_2 + cx_3 + dx_4 + ex_5 - sortie) \cdot x_2$$
$$\dots$$

Pour simplifier l'écriture : $t = 2 \cdot (ax_1 + bx_2 + cx_3 + dx_4 + ex_5 - sortie)$

$\Longrightarrow \frac{\delta erreur}{\delta a} = t \cdot x_1 \quad \frac{\delta erreur}{\delta b} = t \cdot x_2 \quad \frac{\delta erreur}{\delta c} = t \cdot x_3 \ \cdots$

On définit les matrices :

$$Gradient = \begin{pmatrix} \frac{\delta erreur}{\delta a} & \frac{\delta erreur}{\delta b} & \frac{\delta erreur}{\delta c} & \frac{\delta erreur}{\delta d} & \frac{\delta erreur}{\delta e} \end{pmatrix}$$
$$W = \begin{pmatrix} a & b & c & d & e \end{pmatrix} \qquad {}^tX = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{pmatrix}$$

De cette manière on peut calculer le gradient beaucoup plus simplement :

$$t = 2 \cdot (W \cdot X - sortie) \qquad Gradient = t \cdot {}^tX$$

Il suffit alors de calculer le gradient pour chaque groupe d'entrées et leur sortie respective, puis d'ajuster ainsi nos paramètres, contenus dans la matrice W, en leur soustrayant le gradient, multiplié par un facteur compris entre 0 et 1. Ainsi, voilà à quoi devrait ressembler notre programme :

$W = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ *ici on initialise les paramètres à 0*
$P = \begin{pmatrix} 2 & -1 & 4 & 5 & -3 \end{pmatrix}$

Pour i variant de 0 à 700 :

Dans un premier temps nous allons donc d'abord créer ces entrées et les sorties auxquelles elles correspondent. Afin d'éviter d'écrire toutes les entrées et toutes les sorties à la main, nous allons simplement prendre comme entrées des nombres aléatoires et comme sortie la somme de ces nombres multipliés par les paramètres p.

$X = \begin{pmatrix} random \\ random \\ random \\ random \\ random \end{pmatrix}$ → *On donne une valeur aléatoire aux entrées.*

$sortie = P \cdot X$ → *On calcule la sortie correspondante aux entrées aléatoires.*

À ce stade, les entrées et la sortie ont été générées, la suite du programme va tenter de retrouver les paramètres uniquement grâce aux entrées et à la sortie.

$t = 2 \cdot (W \cdot X - sortie)$

$Gradient = {}^tX \cdot t$ → *On calcule le gradient grâce aux entrées X et aux sorties.*

$W \coloneqq W - Gradient \cdot 0.1$ → *On multiplie le gradient par un nombre compris entre 0 et 1 pour éviter que les paramètres ne divergent.*

→ *Cette opération est répétée 700 fois (on repart en haut de la boucle à chaque fois et ainsi, les variables contenues dans W deviennent de plus en plus proches de ce qu'elles devraient être, c'est-à-dire qu'elles se rapprochent de la valeur de P.*

Pour écrire ce programme en python, il a fallu faire appel à un module externe nommé numpy. Celui-ci nous permet de faire des produits matriciels :

In [ ]:
```
import numpy as np

parametres=np.array([[2,-1,4,5,-3]])
nbr_parametres=len(parametres[0])

weights= np.zeros((1,nbr_parametres))
#ici on initialise les weights
vitesse = 0.1

for i in range(700):
    #On définit la sortie avec les entrées aléatoires
    entree = np.random.rand(nbr_parametres,1)
    sortie=np.dot(parametres,entree)

    #Phase d'apprentissage -> on ajuste les weights
    w=2*(np.dot(weights,entree)-sortie)
    gradient = w*entree.T
    weights -= gradient*vitesse

    if i%50==0:
        print("Prediction", weights[0].round(3))
```

Et voici ce que ce programme affiche :

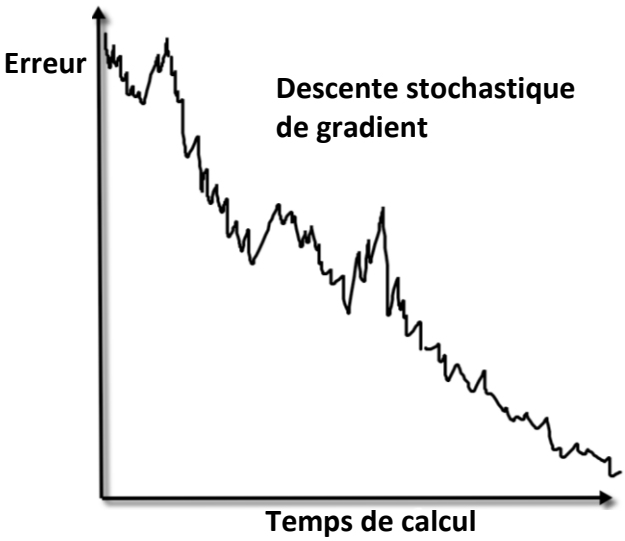
```
Prediction [ 1.964  0.025  2.208  2.496 -1.191]
Prediction [ 2.245 -0.12   3.325  4.015 -2.082]
Prediction [ 2.136 -0.594  3.638  4.665 -2.713]
Prediction [ 2.064 -0.877  3.882  4.875 -2.928]
Prediction [ 2.026 -0.939  3.941  4.963 -2.943]
Prediction [ 2.003 -0.974  3.968  4.985 -2.982]
Prediction [ 2.004 -0.997  3.988  4.992 -2.996]
Prediction [ 2.002 -0.998  3.994  4.997 -2.997]
Prediction [ 2.001 -0.999  3.998  4.999 -2.998]
Prediction [ 2.001 -1.     3.998  4.999 -2.999]
Prediction [ 2.    -1.     3.999  5.    -3.   ]
Prediction [ 2. -1.  4.  5. -3.]
```

5.3 Descente stochastique de gradient

Jusque-ici, dans ces programmes, on calcule seulement le gradient pour un groupe d'entrées et une sortie à la fois. Le gradient va donc minimiser l'erreur sur la fonction pour ce groupe d'entrée et cette sortie, mais pas pour les autres entrées et les autres sorties. On ne calcule donc pas la manière la plus directe de minimiser l'erreur. Cependant, en additionnant tous ces gradients calculés, on parvient quand même à optimiser notre fonction, car généralement un gradient calculé sur un seul groupe d'entrée va minimiser l'erreur plus souvent qu'il ne va l'augmenter grâce à un phénomène statistique.

Pourtant, il est possible de calculer ce gradient pour plusieurs groupes d'entrées et de sorties à la fois. Si on le voulait, on pourrait même calculer directement le gradient pour toutes les entrées et les sorties à la fois, et ainsi connaître le moyen le plus direct de minimiser l'erreur.

Cependant, faire ce calcule pourrait s'avérer un peu trop dur pour notre ordinateur. Alors, pour lui simplifier la tâche, on va lui demander de calculer le gradient pour quelques groupes d'entrées et de sorties, puis pour quelques autres, etc. On serait alors plus précis que si on prenait un seul groupe d'entrées et une sortie à la fois, mais moins que si on les prenait tous. En soit, en calculant ainsi le gradient sur un batch d'images et non sur la totalité des images, on n'emprunte pas le chemin le plus direct pour réduire l'erreur de notre réseau de neurones, mais il est ainsi plus facile de le calculer et au final l'erreur diminuera plus rapidement. C'est comme si pour descendre une montagne, nous avions l'option entre descendre très prudemment en empruntant exactement le chemin le plus direct, ou alors d'emprunter un chemin chaotique en zigzag mais de le descendre en courant.



Cette manière de minimiser l'erreur est appelée « descente stochastique de gradient ». Le mot « stochastique » signifie que l'on s'approche de manière hasardeuse du point ou notre erreur est la plus basse.

Ces groupes d'entrées et ces sorties desquels on va calculer le gradient sont appelés « batch », ce qui signifie bêtement « groupe » en anglais. Le nombre d'éléments dans ces batchs est généralement choisis assez arbitrairement et dépend du problème à résoudre, ou encore de la puissance de l'ordinateur. Moins l'ordinateur est puissant, plus ce nombre doit être petit.

5.4 Calcul du gradient sur un batch

Nous allons donc essayer de faire un programme similaire au précédent, mais cette fois en calculant le gradient pour un batch entier en même temps. On crée d'abord nos i groupes de n entrées aléatoires $x_{n,i}$ et leur sorties correspondantes en définissant que $sortie_i = 2x_{1,i} - 5x_{2,i} + 6x_{3,i} + 7x_{4,i} - 3x_{5,i} + 2x_{6,i}$. Il est possible de mettre toutes les entrées dans une matrice et toutes les sorties dans une autre matrice. J'ai choisi ici de créer 600 groupes d'entrées et 600 sorties de manière totalement arbitraire.

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,5} & x_{1,6} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,5} & x_{2,6} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{599,1} & x_{599,2} & \cdots & x_{599,5} & x_{599,6} \\ x_{600,1} & x_{600,2} & \cdots & x_{600,5} & x_{600,6} \end{pmatrix} \qquad Y = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{599} \\ s_{600} \end{pmatrix}$$

Un batch de X et un batch de Y de 70 éléments correspondraient alors à :

$$BatchX = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,5} & x_{1,6} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,5} & x_{2,6} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{70,1} & x_{70,2} & \cdots & x_{70,5} & x_{70,6} \end{pmatrix} \qquad BatchY = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{70} \end{pmatrix} \qquad Variables = \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix}$$

On a calculé précédemment le gradient pour un groupe d'entrées et une sortie :

$$Gradient_i = {}^tX_i \cdot 2 \cdot (X_i \cdot Variables - sortie_i)$$

Le calcul du gradient pour un batch d'entrée et un batch de sortie correspond à :

$$Gradient = {}^tBatchX \cdot 2(BatchX \cdot Variables - BatchY)$$

Le gradient calculé sur un Batch correspond à la somme des gradients pour chaque groupe X_i d'entrée contenus dans ce batch. Si on a par exemple un batch de 70 images, alors le gradient sera 70 fois plus grand. Afin que les valeurs contenues dans le gradient ne dépendent pas de la longueur du batch choisi, il suffit de diviser le gradient par la longueur du batch.

Voici à quoi ressemble un programme qui calcule le gradient sur un batch d'image en une fois :

$Paramètres = (2 \quad -5 \quad 6 \quad 7 \quad -3 \quad 2)$
→*Le but de ce programme est de retrouver ces paramètres.*
$Longueur\ d'un\ Batch = 70$

$$X = \begin{pmatrix} random_{1,1} & random_{1,2} & \cdots & random_{1,5} & random_{1,6} \\ random_{2,1} & random_{2,2} & \cdots & random_{2,5} & random_{2,6} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ random_{600,1} & random_{600,2} & \cdots & random_{600,5} & random_{600,6} \end{pmatrix} \quad W = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

→*On donne une valeur aléatoire aux entrées et on initialise les variables à 0.*
$Y = X \cdot {}^tParamètres$ → *On calcule alors les sorties correspondantes aux entrées aléatoires.*

Pour i variant de 0 à 10:

Il n'y a pas besoin de faire autant de calculs que dans les programmes précédents vu qu'ici le gradient est beaucoup plus précis car il sera calculé sur un batch entier.

→ *On définit un $BatchX$ et un $BatchY$ qui sont un groupe d'éléments contenus dans X et Y. En l'occurrence, on prend les 70 premières lignes des matrices dans X et Y puis les 70 suivants etc. jusqu'à arriver au bout de la liste puis on recommence au début.*

$Gradient = {}^tBatchX \cdot 2(BatchX \cdot Variables - BatchY)$ → *On calcule le gradient pour ces groupes d'entrées et leur sortie correspondante.*

$W \coloneqq W - \frac{Gradient}{Longueur\ d'un\ Batch} \cdot 0.1$ → *On modifie les variables en leur soustrayant le gradient, multiplié par un nombre en 0 et 1 et divisé par la longueur du Batch pour éviter que les variables ne divergent.*

→ *Cette opération est répétée 10 fois (on repart en haut de la boucle à chaque fois et ainsi, les variables contenues dans W deviennent de plus en plus proche de ce qu'elles devraient être, c'est-à-dire qu'elles se rapprochent de la valeur des paramètres.*

Voici le programme en python :

In [16]:
```python
import numpy as np
#d'abord on crée les entrées et les sorties...
parametres = np.array([[2,-5,6,7,-3,2]])
nbr_parametres = len(parametres[0])
entrees = (np.random.rand(600,nbr_parametres)-0.5)*20
sorties = np.dot(entrees, parametres.T)

def nextbatch(liste_entree, liste_sortie,n):
    global a
    #cette fonction renvoie des batchs
    a = (a+n)%(len(liste_entree)-n)
    return np.array(liste_entree[a:a+n]), np.array(liste_sortie[a:a+n])
def learn_batch(x_batch, y_batch):
    global variables
    #On calcule tous les gradients du batch en même temps
    derreur_dx = np.dot(x_batch.T,2*(np.dot(x_batch,variables)-y_batch))
    variables -= (derreur_dx/len(x_batch))*0.02

taille_batch = 70
a = (-taille_batch)
variables = np.zeros(( nbr_parametres,1))
for i in range(10):
    #apprentissage
    x_batch, y_batch = nextbatch(entrees, sorties, taille_batch)
    learn_batch(x_batch, y_batch)
    #affichage
    print("\n y =", end=" ")
    for indice,valeur in enumerate(variables[:,0]):
        print("+ {:1.2f}*x{}".format(valeur, indice+1), end=" ")
```

Voilà ce que le programme affiche :

```
y = + 1.04*x1 + -5.33*x2 + 5.63*x3 + 7.99*x4 + -2.46*x5 + 1.53*x6
y = + 2.33*x1 + -4.45*x2 + 6.08*x3 + 6.27*x4 + -3.47*x5 + 2.39*x6
y = + 1.64*x1 + -5.07*x2 + 5.61*x3 + 7.41*x4 + -2.59*x5 + 1.68*x6
y = + 1.95*x1 + -4.96*x2 + 6.09*x3 + 7.15*x4 + -3.03*x5 + 2.08*x6
y = + 1.99*x1 + -4.96*x2 + 6.03*x3 + 6.97*x4 + -3.04*x5 + 1.96*x6
y = + 2.02*x1 + -5.02*x2 + 6.00*x3 + 6.99*x4 + -2.99*x5 + 2.01*x6
y = + 1.99*x1 + -4.99*x2 + 5.99*x3 + 7.01*x4 + -3.00*x5 + 2.00*x6
y = + 2.01*x1 + -5.00*x2 + 6.00*x3 + 7.00*x4 + -3.00*x5 + 2.00*x6
```

5.5 Programme de reconnaissance d'image

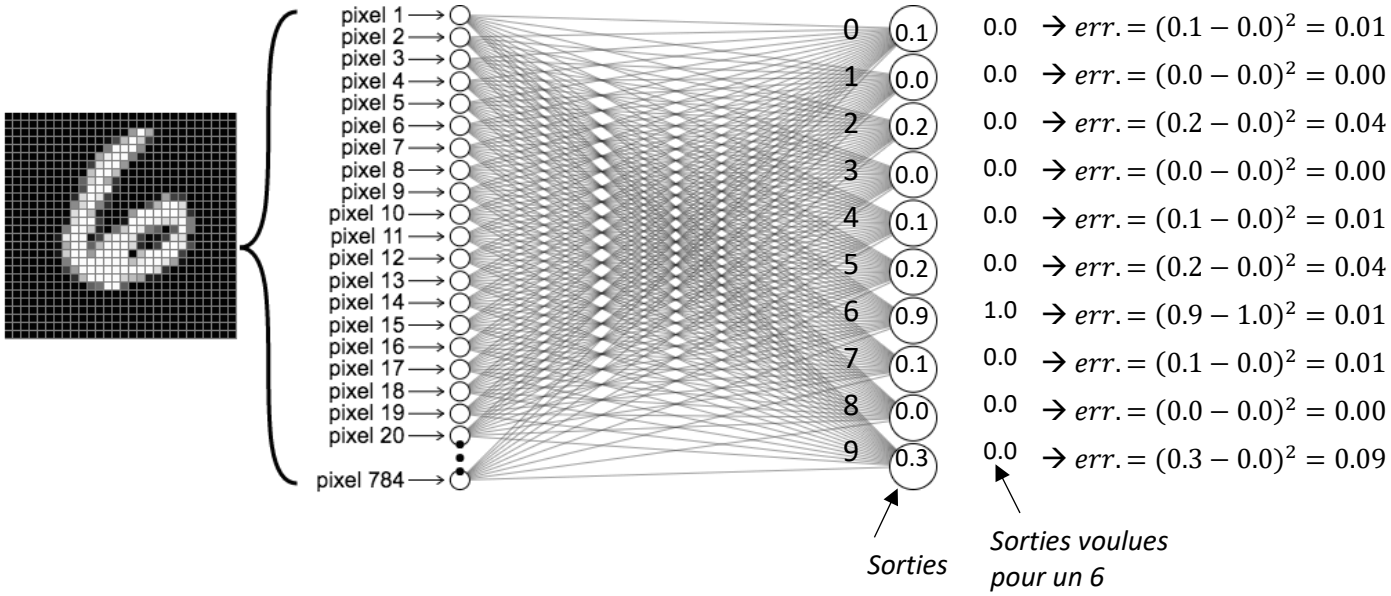
Nous avons désormais les clés en main pour créer un programme de reconnaissance d'images basique.

5.5.1 MNIST

La base de données MNIST (pour Mixed National Institute of Standards and Technology), est une base de données de chiffres écrits à la main. Elle contient 70'000 images de 28x28 pixels séparés en deux groupes : L'un servant à l'apprentissage du réseau de neurones et l'autre groupe servant à tester par la suite l'efficacité de notre programme sur de nouvelles images. Toutes les images ont déjà été attribuées au chiffre correspondant, donc la base de donnée est prête à être utilisée. La base de données MNIST est en quelque sorte le point de départ pour toute personne qui souhaite se lancer dans la reconnaissance d'image, car ce sont des images de petite taille et relativement faciles à différencier.

Nous allons donc dans un premier temps créer un programme qui va calculer le gradient sur des batchs d'images de la base de données d'apprentissage, puis ajuster des paramètres (=weights). Une fois le programme entraîné, nous allons tester notre programme sur des images qu'il n'a jamais vues, contenues dans la base de données de teste, pour ainsi voir à quel point notre programme est capable de différentier de nouvelles images.

5.5.2 Calcul du gradient d'un batch d'images de MNIST



Un programme qui différencie les chiffres de 0 à 9, a 10 sorties (une pour chaque chiffre), contrairement aux programmes que nous avons vu jusqu'ici qui n'avaient qu'une seule sortie. Pour calculer l'erreur, on définit que chaque sortie ne correspondant pas à l'image évaluée doit se rapprocher le plus possible de 0 et la sortie qui correspond à l'image en question doit se rapprocher de 1. Cette manière de procéder est appelée *one hot encoding*.

Le gradient sera calculé sur un batch de 100 images en une seule fois (choix arbitraire).

$$BatchX = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,783} & x_{1,784} \\ x_{2,1} & x_{2,2} & \dots & x_{2,783} & x_{2,784} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{99,1} & x_{99,2} & \dots & x_{99,783} & x_{99,784} \\ x_{100,1} & x_{100,2} & \cdots & x_{100,783} & x_{100,784} \end{pmatrix} \quad BatchY = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,9} & s_{1,10} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,9} & s_{2,10} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ s_{99,1} & s_{99,2} & \cdots & s_{99,9} & s_{99,10} \\ s_{100,1} & s_{100,2} & \cdots & s_{100,9} & s_{100,10} \end{pmatrix}$$

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,9} & w_{1,10} \\ w_{2,1} & w_{2,2} & \dots & w_{2,9} & w_{2,10} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{783,1} & w_{783,2} & \dots & w_{783,9} & w_{783,10} \\ w_{784,1} & w_{784,2} & \cdots & w_{784,9} & w_{784,10} \end{pmatrix}$$

$$E = (BatchX \cdot W - BatchY)^2$$

L'erreur E est une matrice de taille 100×10 (une erreur pour chaque sortie et de chaque image du batch) et donc il faut minimiser la somme de tous les éléments de cette matrice.

Étonnamment, le calcul du gradient de l'erreur des 10 sorties reste le même que dans les programmes ne calculant le gradient pour une seule sortie à la fois :

$$Gradient = {}^{t}BatchX \cdot 2(BatchX \cdot W - BatchY)$$

La seule chose qui change par rapport au calcul d'une seule sortie est la taille des matrices*[8], mais la forme du calcul reste la même.

Une fois le gradient calculé, il suffit de multiplier celui-ci par un nombre compris entre 0 et 1, puis de le soustraire aux weights. Il ne reste ensuite plus qu'à répéter l'opération un grand nombre de fois. On peut donc s'attaquer au code du programme de reconnaissance d'images :

Il faut d'abord importer les images de MNIST et les mettre sous forme d'une matrice avec des valeurs variant de 0 (pixel blanc) à 1 (pixel noir).

$$Data\ images = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,783} & x_{1,784} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,783} & x_{2,784} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0.1 & 0.9 & \cdots & 0.8 & 0.0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{59999,1} & x_{59999,2} & \cdots & x_{59999,783} & x_{59999,784} \\ x_{60000,1} & x_{60000,2} & \cdots & x_{60000,783} & x_{60000,784} \end{pmatrix}$$

Puis il faut importer les sorties correspondantes aux images de MNIST et les mettre sous la forme d'une matrice contenant des 1 pour les bonnes sorties et des 0 pour les mauvaise (one hot encoding).

$$Data\ sorties = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,9} & s_{1,10} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,9} & x_{2,10} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ s_{59999,1} & s_{59999,2} & \cdots & s_{59999,9} & s_{59999,10} \\ s_{60000,1} & s_{60000,2} & \cdots & s_{60000,9} & s_{60000,10} \end{pmatrix}$$

Il faut ensuite initialiser les variables contenues dans W (on les met arbitrairement à 0).

Pour i variant de 0 à 2000 :

On définit un batch de 100 images et de 100 sorties correspondante. On prend d'abord les images de 1 à 100 pour le premier batch, puis de 100 à 200 etc. :

$$BatchX = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,784} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,784} \\ \vdots & \vdots & \ddots & \vdots \\ x_{99,1} & x_{99,2} & \cdots & x_{99,784} \\ x_{100,1} & x_{100,2} & \cdots & x_{100,784} \end{pmatrix} BatchY = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,10} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,10} \\ \vdots & \vdots & \ddots & \vdots \\ s_{99,1} & s_{99,2} & \cdots & s_{99,10} \\ s_{100,1} & s_{100,2} & \cdots & s_{100,10} \end{pmatrix}$$

*[8] Si on calcule le gradient pour une seule des 10 sorties de l'image, voici la taille des différentes matrices :
Gradient : 784x1 ; W : 784x1 ; BatchX : 100x784 ; BatchY : 100x1
En revanche, si on calcule le gradient pour les 10 sorties en une seule fois, voici la taille des matrices :
Gradient : 784x10 ; W : 784x10 ; BatchX : 100x784 ; BatchY : 100x10

$$Gradient = {}^{t}BatchX \cdot 2(BatchX \cdot W - BatchY)$$
$$W \coloneqq W - \frac{Gradient \cdot 0.02}{longeur\ Batch}$$

Il faut recommencer au début de la boucle et répéter cette opération 2000 fois. De cette manière, les variables contenues dans W vont se rapprocher d'une valeur qui donne le moins d'erreur possible sur l'ensemble des images, et donc vont être en mesure de prédire ce qui se trouve sur une image de chiffre écrit à la main que le programme n'a encore jamais vue.

Pour vérifier que notre programme soit bien entraîné, on peut le tester sur 15 images qu'il n'a jamais vues que l'on va prendre aléatoirement dans les images test de MNIST.

Pour i variant de 0 à 15 :

On prend une image au hasard dans les images test de MNIST.
$Image = (i_1, \quad i_2, \quad \cdots \quad i_{783}, \quad i_{784})$

On affiche cette image → pixel blanc pour 0, pixel noir pour un 1, pixel gris pour 0.68.

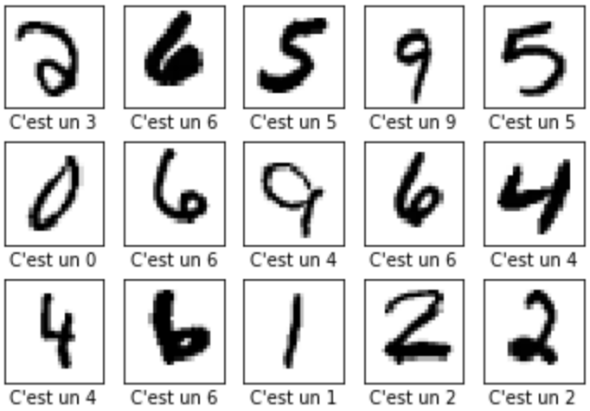
On calcule les 10 sorties pour chacun des chiffres.
$S = (s_0, \quad s_1, \quad \cdots \quad s_8, \quad s_9) = Image \cdot W$

On regarde dans la matrice des sorties laquelle est la plus grande et on affiche son indice. Par exemple si la sortie s_2 est la plus grande, alors on affiche 2.

Voici finalement notre programme de reconnaissance d'image en python:

```
import numpy as np
import matplotlib.pyplot as plt #ce module nous aide à afficher les images plus simplement
from random import choice
import tensorflow.examples.tutorials.mnist.input_data as input_data
data =input_data.read_data_sets("/Users/hannoschmid/Documents/Nicolas/data/MNIST", one_hot=True)

entrees=data.train.images #d'abord on définit les entrées et les sorties...
sorties= data.train.labels
taille_batch = 100
a = (-taille_batch)
variables = np.zeros((784,10)) #il y a 784 variables pour chaque chiffre de 1 à 10

for i in range(2000):
    #creer le batch
    a = (a+taille_batch)%(len(entrees)-taille_batch)
    x_batch, y_batch = np.array(entrees[a:a+taille_batch]), np.array(sorties[a:a+taille_batch])
    #apprendre le batch
    derreur_dx = 2*np.dot(x_batch.T,(np.dot(x_batch,variables)-y_batch))
    variables -= (derreur_dx/len(x_batch))*0.02

images_a_tester, cases = plt.subplots(3, 5)#on affiche les variables sur deux lignes de 5 colonne
for case in cases.flat:
    image = choice(data.test.images) #on teste notre programme sur des images qu'il n'a jamais vues
    sortie_calculée= np.argmax(np.dot(image,variables))
    case.set_xlabel("C'est un " + str(sortie_calculée))
    case.imshow(image.reshape(28,28),cmap='binary')
    case.set_xticks([])
    case.set_yticks([])
plt.show()
```

Et voici ce qu'il affiche :



Je rappelle que le programme n'avait encore jamais vu ces 15 images ci-dessus. Pourtant, dans la majorité des cas, il a réussi à les classer au bon endroit. Notre programme a donc appris à reconnaître les chiffres de 0 à 9 écrits à la main !
Bon, il faut reconnaître qu'il ne parvient pas à les classer au bon endroit. Dans certains cas, c'est car les chiffres sont mal écrits, mais dans d'autres cas cela vient du fait que ce programme est loin d'être parfait. Si on le teste sur les 10'000 images test de MNIST on arrive à une précision d'environ 85%.

5.6 Le défaut du programme…

Alors d'où vient le problème dans notre programme ? On pourrait imaginer que le réseau de neurones n'est pas assez entraîné et que si à la place de calculer 2000 fois le gradient, on le calculait 10'000 fois, alors le programme deviendrait plus performant. On pourrait aussi par exemple augmenter ou diminuer la taille du batch ou encore la grandeur du facteur par lequel on multiplie le gradient, afin de voir si cela pourrait changer quelque chose. J'ai moi-même testé différentes configurations*[9] et j'en suis arrivé à cette conclusion : On pourra l'entraîner tant qu'on veut, le programme restera bloqué à une précision d'environ 85% et la taille du batch ne changera pas grand-chose non plus, si ce n'est le temps de calcul nécessaire.
Une chose en revanche qui modifie un peu ce résultat est de calculer le gradient de l'erreur avec la fonction sigmoïde.

Dans le programme précédent, nous avons définit l'erreur du réseau de neurone :

$$E = (BatchX \cdot W - BatchY)^2$$

Pourtant, comme nous l'avons vu précédemment, il serait plus judicieux d'utiliser la fonction sigmoïde pour calculer la sortie du réseau de neurones:

$$E = (sigmoïde(BatchX \cdot W) - BatchY)^2$$

Pour ceux qui ont eu le courage de lire ce passage, nous avons vu que le gradient de l'erreur avec la fonction sigmoïde est le suivant*[10] :

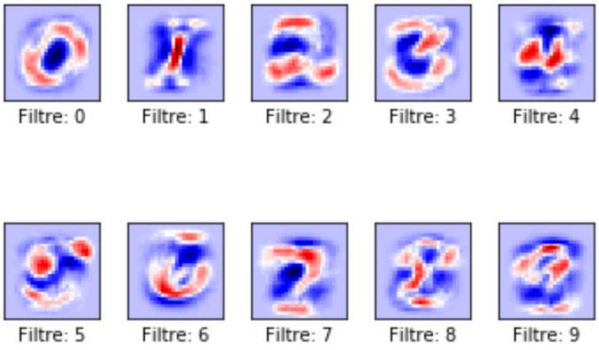
*[9] Les différents taux de réussites sont en annexe (annexe 2.2).
*[10] En réalité on a vu comment calculer le gradient pour l'erreur sur une image et non un batch d'image, mais le calcul reste le même.

$$Gradient = 2 \cdot {}^{t}BatchX \cdot [sigmoïde(BatchX \cdot W) - BatchY] \cdot sigmoïde(BatchX \cdot W) \cdot [1 - sigmoïde(BatchX \cdot W)]$$

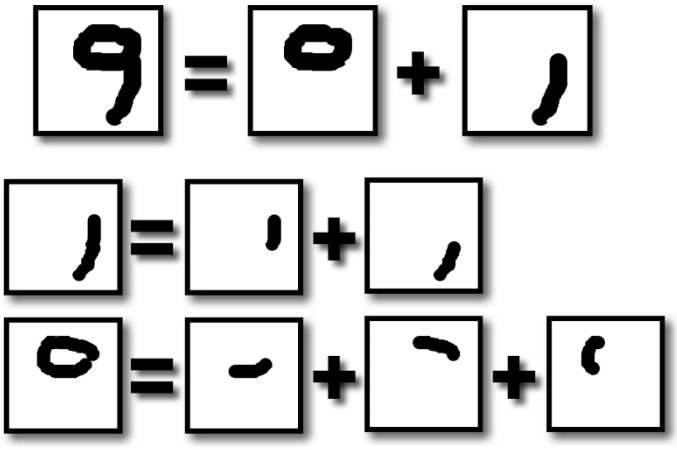
Il suffit donc, dans notre programme, de calculer le gradient comme présenté ci-dessus et le tour est joué. En faisant ceci, et en entraînant assez le réseau de neurones, avec une taille de batch et un facteur multipliant le gradient adéquat, on arrive à une précision d'environ 91%*[11].
C'est donc déjà une bonne amélioration par rapport au programme précédent, mais ce n'est pas parfait du tout, le programme échoue à reconnaître des chiffres qui sont pourtant faciles à reconnaître mais qui sont écrits d'une manière inhabituelle. Il a de la peine par exemple lorsque les chiffres sont un peu décalés sur le côté, écrits plus petits, ou avec quelques fioritures.

6. Les réseaux de neurones à convolution (ou CNN)

En réalité le problème vient de la construction du programme lui-même. Jusqu'ici on a vu comment fonctionne un programme qui différencie les chiffres écrits à la main en appliquant 10 filtres (=weights) sur une image et en regardant quel filtre donnait le meilleur résultat. Voilà à quoi ressemblent les filtres sur un modèle entrainé :



Cette façon de procéder est cependant limitée car si un chiffre écrit à la main est légèrement décalé sur la droite ou sur la gauche, il se peut que les pixels du dessin ne correspondent plus du tout avec les pixels du filtre et le programme risque donc de donner une mauvaise réponse. Il nous faut donc trouver une autre manière de reconnaître des images.

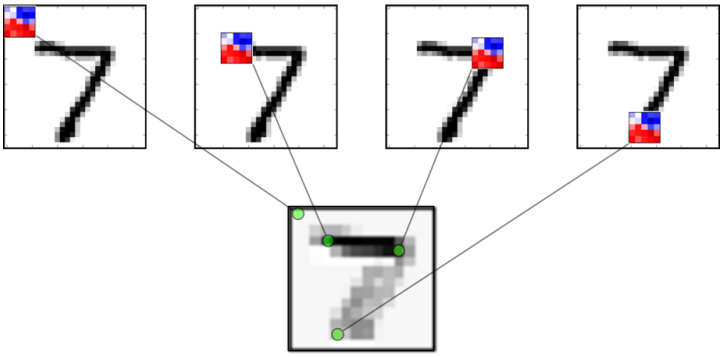


En y réfléchissant bien, pour voir qu'un chiffre écrit à la main corresponde par exemple à un 9, un humain ne va pas vérifier qu'il y ait des pixels allumés à tel ou tel endroit. Pour nous, une manière simple de définir le symbole 9 est : un trait vertical surmonté d'une boucle. Reste à définir ce qu'est une boucle et un trait vertical, mais en soi, ce que cela montre, c'est qu'on s'occupe d'abord des petits détails, puis on associe ces petits détails pour finalement reconnaître l'image en général. Ceci n'est pas valable que pour les chiffres, mais pour n'importe quelle image ! Par exemple on reconnaît un visage à partir de petites parties de visages (nez, bouche, yeux) qui combinées ensemble forment un visage. Ce que l'on peut en conclure c'est qu'il ne faut pas

*[11] Le code du programme de reconnaissance d'image avec la fonction sigmoïde est en annexe 2.1.
Les différents taux de réussite avec différentes taille de batch etc. sont en annexe 2.3

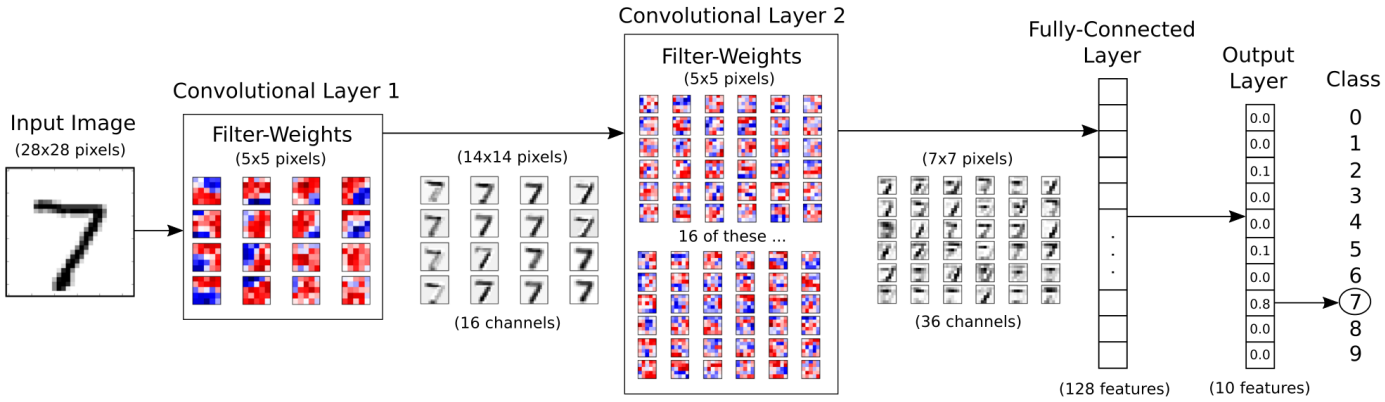
considérer tous les pixels d'une image d'un coup en appliquant un grand filtre sur toute l'image, mais plutôt se focaliser sur des petites parties de l'image, puis d'assembler ces parties. C'est ce que nous proposent de faire les réseaux de neurones à convolution.

Contrairement au réseau linéaire que nous avons vu précédemment, ce réseau à convolution ne va pas appliquer des filtres sur l'ensemble de l'image à la fois, mais de petits filtres de 5x5 pixels qu'il va bouger sur l'ensemble de l'image. Tout d'abord, il va appliquer le filtre dans le



coins supérieur gauche de l'image, c'est-à-dire qu'il va faire le produit entre les 5x5 pixels du coin supérieur gauche de l'image initiale et les valeurs contenues dans le filtre. La somme de ces produit va donner la valeur du pixel supérieur gauche de l'image de sortie. Le filtre est ensuite déplacé d'un pixel vers la gauche et calcule ainsi le pixel suivant. Cette opération est répétée jusqu'à ce qu'il arrive au bout de la ligne, puis il recommence le tout en descendant le filtre d'un pixel, puis le déplace d'un pixel vers la droite etc. Dans l'image ci-dessus, le filtre crée donc une nouvelle image dans laquelle les parties horizontales de l'image sont mises en valeur.

Voici comment un réseau de neurones à convolution évalue une image de A à Z:



En réalité, tout ce qu'il fait, c'est appliquer successivement des filtres comme nous l'avons vu ci-dessus. Cependant, il n'applique pas qu'un seul filtre par image, mais en applique par exemple 16 sur l'image initiale (première convolution). Chacun de ces filtres va créer une nouvelle image, et va donc mettre en valeur un détail différent de l'image, par exemple le premier filtre va mettre en valeur les lignes horizontales, un autre va mettre en valeur les diagonales etc. Sur chacune des 16 images créées sont à nouveau appliqué 36 filtres (deuxième convolution). Il y a donc $16 \cdot 36 = 576$ filtres qui sont appliqués. Les 36 premiers filtres sont appliqués sur la première image, les 36 suivants sur la deuxième image et ainsi de suite jusqu'à la l'image 16. Cela crée donc 36 groupes de 16 images, et ces groupes de 16 images sont encore additionnés entre eux (on additionne la valeur des pixels des 16 images entre-elles pour n'en faire plus qu'une), diminuant ainsi le nombre d'images créées à 36. Vous imaginez que créer tant d'images peut demander énormément de mémoire et de capacité de calcul. Pour diminuer cet effet, la résolution des images est réduite à chaque convolution. En

général on prend la moyenne de 4 pixels voisin pour créer le nouveau pixel de l'image compressée. Ces 36 images obtenues après deux convolutions de l'image initiale mettent donc en valeur des détails de l'image. Pour reconnaître l'image, il faut encore combiner tous ces détails ensemble.
Pour cela, après avoir calculé ces 36 images, le programme va les mettre dans un grand vecteur de dimension 1764 contenant tous les pixels des 36 images de résolution 7x7 pixel ($7 \cdot 7 \cdot 36 = 1764$). Ce vecteur est multiplié par une matrice de taille 1764x128, ce qui crée un vecteur de 128 éléments, puis ce vecteur est à nouveau multiplié par une matrice de taille 128x10 pour enfin donner les dix différentes sorties pour chacun des chiffres de 0 à 9. Ces deux dernières étapes permettent aux réseau d'associer différentes combinaisons de neurones allumés, obtenus par les convolutions, à un certain chiffre et donc de reconnaître des images.
Vous imaginez que calculer le gradient d'un tel système n'est pas tout simple, c'est pourquoi, pour créer mon propre réseau de neurones à convolution*[12], j'ai utilisé un module nommé tensorflow, qui a été développé par Google, et qui aide notamment les programmeurs à créer des réseaux de neurones très complexes. Avec ce module, il suffit de décrire les opération du modèle, et tensorflow s'occupe de minimiser le coût de l'erreur en en calculant le gradient et ajustant les weights qui composent les filtres.

Le programme décrit dans ce chapitre est capable de reconnaître de nouvelles images de MNIST avec une précision de plus de 98%.

7. Récit de la création de mon TM

J'ai décidé de faire ce TM en reconnaissance d'image après avoir visionné une série de vidéos très intéressantes de 3blue1brown, expliquant le fonctionnement de la reconnaissance d'image avec quelques détails mathématiques. J'avais commencé à programmer 2-3 mois auparavant et je n'y connaissais pas grand-chose, que ce soit en informatique ou en programmation, mais cela me fascinait de voir ce qu'il était possible de faire avec quelques petites lignes de code. J'ai donc voulu reproduire ce que j'avais vu en vidéo, en créant moi-même un programme de reconnaissance d'images, mais j'ai d'emblée été confronté à quelques problèmes. Je devais faire du calcul matriciel et importer des images de MNIST et pour cela, je devais importer des modules externes tel que tensorflow, numpy, ou encore matplotlib. Or, je ne savais pas comment installer des modules en python, et il m'a fallu plusieurs semaines avant que j'y parvienne.
J'ai alors essayé tout seul de trouver comment créer et entraîner un réseau de neurones à partir de la théorie que j'avais apprise en vidéo, mais cela n'a pas donné de résultats très concluants.

Après quelques échecs, j'ai cherché sur le web des programmes similaires à celui que je souhaitais créer afin de m'en inspirer pour écrire le mien, mais je ne trouvais que des programmes utilisant tensorflow et non des programmes avec toutes les opérations mathématiques, faites de A à Z, comme j'avais l'intention de faire personnellement. Je me suis donc résolu à utiliser tensorflow, en copiant d'abord d'autres programmes. J'ai copié un simple classificateur d'images MNIST comme celui du point 5.5.2, puis je l'ai progressivement

*[12] Le programme en annexe 2.5 correspond plus ou moins à ce que j'ai décrit dans ce chapitre.

modifié, en ayant l'avantage de pouvoir vérifier après chaque modifications, si celui-ci fonctionnait toujours ou s'il affichait une erreur, j'ai ainsi pu comprendre comment fonctionnait ce programme.
À force de le modifier, j'ai finalement créé mon propre programme qui reconnaissait des chiffres que j'écrivais moi-même avec ma souris sur mon écran. J'ai dû d'ailleurs pour cela créer une interface graphique, chose que je ne maîtrisais pas vraiment.

J'avais donc un programme de reconnaissance d'images simple (une seule couche de neurones), qui reconnaissait les chiffres de 0 à 9 et qui pouvait reconnaître ce que je dessinais avec ma souris, mais j'avais envie qu'il reconnaisse d'autres choses que des chiffres.
J'ai donc créé un programme dans lequel je pouvais dessiner des symboles de 28x28 pixels et les associer à une sortie (un mot, un chiffre ou une lettre). On pouvait par exemple dessiner 10 triangles, et dire au programme que c'étaient des triangles, puis 10 carrés, et finalement entraîner le réseau avec les données que l'on venait de créer. Le problème était que toutes ces données créées s'effaçaient lorsque je fermais le programme. J'ai donc amélioré le programme, en lui ajoutant une fonction de sauvegarde des données dans un fichier texte, afin de pouvoir les reprendre à la prochaine ouverture du programme. J'ai aussi créé des fonctionnalités qui me permettaient de supprimer telles ou telles données, puis j'ai ajouté une fonctionnalité qui permettait d'afficher les weights.

C'était donc devenu un très beau programme (très semblable à celui en annexe 2.4 mais avec tensorflow) mais je n'étais pas très satisfait car j'utilisais tensorflow pour entraîner le réseau de neurones, et je souhaitais comprendre tous les calculs derrière mon programme.
J'ai donc recommencé depuis la base, mais cette fois sans utiliser tensorflow. J'ai d'abord fait des programmes très simples, qui minimisaient des fonctions à 2 inconnues, puis 3, puis 10 etc. J'ai ainsi appris à tâtons comment utiliser python pour minimiser des fonctions linéaires.
Ensuite, il a fallu que j'utilise des matrices, un outil que je ne maitrisais pas encore vraiment, mais grâce à quelques tutoriels sur YouTube, j'ai mieux compris de quoi il s'agissait. J'ai donc augmenté la complexité de mes programmes, en refaisant les mêmes calculs, mais cette fois avec des matrices, puis à la place de faire une opération après l'autre, j'ai essayé d'utiliser des batchs. Bien sûr, ça ne fonctionnait presque jamais du premier coup, mais certaines fois, au contraire, le programme fonctionnait sans que je ne comprenne pourquoi. Par exemple lorsque j'ai voulu calculer le gradient pour un batch d'images, je ne savais pas comment procéder. J'ai alors utilisé les mêmes opérations pour calculer le gradient d'un batch d'images que celles que j'avais utilisées pour calculer le gradient d'une seule image, et par chance, les opérations nécessaires étaient les mêmes.

Mon objectif a donc été atteint lorsque j'ai réussi à appliquer ce que j'avais appris, en résolvant des petits problèmes faciles, sur les images de MNIST.
J'ai ensuite essayé d'ajouter la fonction sigmoïde à tout cela. Pour cela, j'ai dû à nouveau recommencer par des petits programmes basiques puis progressivement les modifier jusqu'à arriver à un programme de reconnaissance d'image.
Au bout d'un certain temps, j'ai finalement créé un programme fonctionnel et j'ai remarqué que j'obtenais les mêmes performances que le programme initial fait avec tensorflow, ce qui était tout de même assez satisfaisant. J'ai ensuite appliqué ce que j'avais appris au programme que j'avais créé initialement où je créais moi-même les symboles à reconnaître. Le programme ainsi obtenu est celui en annexe 2.4.

Cependant, je restais bloqué à une précision de 90% alors que dans la vidéo de 3blue1brown, il arrivait vers environ 98% de précision (avec un réseau de neurones à plusieurs couches) Je me suis donc renseigné et j'ai découvert les réseaux de neurones à convolution. Il y avait également des tutoriel pour en créer avec tensorflow. Je les ai donc suivis et j'ai alors réussi à créer un programme reconnaissant les chiffres de 0 à 9 avec une précision de plus de 98%.
J'ai à nouveau modifié le programme qui reconnaît les symboles qu'on lui dessine en lui implémentant ce nouvel algorithme d'apprentissage et le programme fonctionna étonnamment bien. J'ai constaté qu'il fallait plus de données pour entrainer correctement un CNN*[13], et que le temps de calcul pour lui faire apprendre était nettement plus long. J'ai donc fait de sorte que mon programme soit capable de sauvegarder les weights du réseau de neurones à la fermeture du programme, afin d'éviter d'avoir à les recalculer à chaque fois. Le programme obtenu est en annexe 2.5.
Là aussi j'ai recherché à faire un réseau à convolution de A à Z, sans tensorflow, mais je ne voyais vraiment pas comment calculer le gradient du réseau, et rien que construire le modèle sans tensorflow me paraissait difficile. J'ai donc dû me contenter de cela…

8. Conclusion

Ce qui a donc été le plus difficile pour moi dans la création de mon travail de maturité, a été de trouver des informations de qualité mais aussi de les comprendre, en ajoutant à cela que la majorité des informations étaient en anglais. En soit j'ai vraiment apprécié faire ce travail de maturité et je l'aurais probablement fait sans devoir faire un TM. C'est une bonne satisfaction personnelle d'avoir réussi à accomplir (plus ou moins) ce que je voulais, c'est-à-dire faire de la reconnaissance d'image.

Pour l'écrit de mon travail de maturité, j'ai essayé de rassembler les informations que j'ai lues et vues par-ci par-là, mais aussi ce que j'ai appris par moi-même, pour créer un contenu aussi pédagogique que possible, afin d'aider des futures personnes qui s'intéresseraient à la reconnaissance d'image. Je me suis basé sur un problème à résoudre (reconnaître une image) et j'ai peu à peu donné les outils nécessaires pour le faire, en fournissant d'abord des problèmes relativement simples, avec des exemples concrets et des petits bouts de code, pour parvenir graduellement à résoudre le problème initial. J'ai en quelque sorte essayé de créer le document qui aurait été idéal pour moi lorsque j'ai commencé à m'intéresser à la reconnaissance d'image.
En expliquant ainsi tout ce que j'ai fait de la manière qui me semble la plus claire possible, cela m'a permis d'éclaircir également mes propres idées.

*[13] Un CNN a beaucoup plus de paramètres que le réseau linéaire créé au chapitre 5. Il faut donc énormément de données pour l'entraîner, car s'il possède beaucoup moins d'images que de paramètres, il est possible qu'il ajuste ses paramètres que pour certains cas spécifique et non qu'il trouve une sorte de solution global pour différentier les images. On peut faire une analogie avec une personne à qui on demande d'apprendre à résoudre des problème de logique. Si on lui en donne beaucoup, elle va apprendre à raisonner logiquement et être capable de résoudre de nouveaux problèmes, alors que si on lui en donne qu'une dizaine, elle va simplement apprendre par cœur les réponses.

Annexes :

Annexe 1 - Calcul du gradient de l'erreur avec la fonction sigmoïde

Ce chapitre est un peu compliqué et non indispensable à la compréhension globale du thème, c'est pourquoi je l'ai mis en annexe.

Annexe 1.1 Dérivée de la fonction sigmoïde

Nous avons jusqu'ici défini le gradient de : $erreur = sum((P \cdot W - sorties\ attendues)^2)$
Nous allons maintenant calculer le gradient de l'erreur avec la fonction sigmoïde. Nous allons donc calculer le gradient de $erreur = sum((sigmoïde(P \cdot W) - sorties\ attendues)^2)$

La fonction $sigmoïde(x) = \frac{1}{1+e^{(-x)}}$
Cette fonction à comme point particulier que :

$$sigmoïde'(x) = sigmoïde(x)(1 - sigmoïde(x))$$

Donc par exemple si $f(x) = \big(sigmoïde(3x)\big)^2$

$$f'(x) = 2 \cdot sigmoïde(3x) \cdot sigmoïde(3x)\big(1 - sigmoïde(3x)\big) \cdot 3$$

Annexe 1.2 - Calcul du gradient d'une fonction simple avec sigmoïde

Si on prend $f(w_1, w_2, w_3) = (sigmoïde(aw_1 + bw_2 + cw_3) - s)^2$

$$\frac{\delta f}{\delta w_1} = 2 \cdot (sigmoïde(aw_1 + bx_2 + cw_3) - s) \cdot sigmoïde(aw_1 + bw_2 + cw_3)\big(1 - sigmoïde(aw_1 + bw_2 + cw_3)\big) \cdot a$$
$$\frac{\delta f}{\delta w_2} = 2 \cdot (sigmoïde(aw_1 + bx_2 + cw_3) - s) \cdot sigmoïde(aw_1 + bw_2 + cw_3)\big(1 - sigmoïde(aw_1 + bw_2 + cw_3)\big) \cdot b$$
$$\dots$$

On peut poser que :

$$k = 2 \cdot (sigmoïde(aw_1 + bw_2 + cw_3) - s) \cdot sigmoïde(aw_1 + bw_2 + cw_3) \cdot \big(1 - sigmoïde(aw_1 + bw_2 + cw_3)\big)$$

$$\text{On pose : } P = (a, \quad b, \quad c,) \qquad W = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

$$\frac{\delta f}{\delta w_1} = a \cdot k \quad \frac{\delta f}{\delta w_2} = b \cdot k \quad \frac{\delta f}{\delta w_3} = c \cdot k \qquad \rightarrow \qquad Grad = \begin{pmatrix} \frac{\delta f}{\delta w_1} \\ \frac{\delta f}{\delta w_2} \\ \frac{\delta f}{\delta w_3} \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \cdot k = {}^tP \cdot k$$

Pour la suite, j'ai fait un choix de notation pour différentier le produit matriciel du produit scalaire : Lorsque j'écris $P \times W$ cela correspond au produit matriciel entre P et W alors que si j'écris $P \cdot W$ cela correspond au produit de chaque élément d'un certain indice dans la matrice P avec l'élément ayant le même indice dans la matrice W, ou le produit d'un scalaire avec

chaque élément de la matrice → Lorsque j'écris $P \cdot W$, il faut que les deux matrices aient la même taille, ou alors que P ou W soient un scalaire.

Ex : $\begin{pmatrix}1 & 2\\3 & 4\\5 & 6\end{pmatrix} \times \begin{pmatrix}1 & 2\\3 & 4\end{pmatrix} = \begin{pmatrix}7 & 10\\15 & 22\\23 & 34\end{pmatrix}$ $\quad$ $\begin{pmatrix}1 & 2\\3 & 4\\5 & 6\end{pmatrix} \cdot \begin{pmatrix}1 & 4\\3 & 2\\5 & 3\end{pmatrix} = \begin{pmatrix}1 & 8\\9 & 8\\25 & 18\end{pmatrix}$

On peut écrire k sous la forme de produits de matrices :

$$k = 2 \cdot (sigmoïde(aw_1 + bw_2 + cw_3) - s) \cdot sigmoïde(aw_1 + bw_2 + cw_3) \cdot \big(1 - sigmoïde(aw_1 + bw_2 + cw_3)\big)$$

$$k = 2 \cdot \left[sigmoïde\left((a, \quad b, \quad c) \times \begin{pmatrix}w_1\\w_2\\w_3\end{pmatrix}\right) - s\right] \cdot sigmoïde\left((a, \quad b, \quad c) \times \begin{pmatrix}w_1\\w_2\\w_3\end{pmatrix}\right) \cdot \left[1 - sigmoïde\left((a, \quad b, \quad c) \times \begin{pmatrix}w_1\\w_2\\w_3\end{pmatrix}\right)\right]$$

$$k = 2 \cdot [sigmoïde(P \times W) - s] \cdot sigmoïde(P \times W) \cdot [1 - sigmoïde(P \times W)]$$

Je rappelle que le gradient vaut :

$$Grad = {}^tP \cdot k$$

On remplace k par ce qu'on a calculé plus haut :

$$Grad = 2 \cdot {}^tP \times [sigmoïde(P \times W) - s] \cdot sigmoïde(P \times W) \cdot [1 - sigmoïde(P \times W)]$$

Ceci est valable pour 3 variables, mais également pour 784 variables.

On remarque dans ce calcul que le gradient sur l'erreur d'une sortie correspond à la valeur des pixels de l'image ${}^tP$ multipliés par un certain facteur dépendant de l'erreur sur la sortie. Le gradient sur toutes les images n'est donc qu'une sorte de mélange de toutes les images avec lesquelles on a entraîné le modèle. Cela peut expliquer pourquoi ce programme sera mauvais pour reconnaître des images placés à différents endroits, ou de taille différentes, car un mélange des valeurs des pixels de chiffres placés à différents endroits ne va pas correspondre à ce chiffre, mais à une sorte de gribouillage insensées.

Annexe 1.3 - Calcul du gradient de l'erreur d'une image

On peut donc calculer les 10 gradients pour chacune des sorties :

$$grad_1 = 2 \cdot {}^tP \times [sigmoïde(P \times W_1) - s_1] \cdot sigmoïde(P \times W_1) \cdot [1 - sigmoïde(P \times W_1)]$$
$$grad_2 = 2 \cdot {}^tP \times [sigmoïde(P \times W_2) - s_2] \cdot sigmoïde(P \times W_2) \cdot [1 - sigmoïde(P \times W_2)]$$
$$\dots$$
$$grad_{10} = 2 \cdot {}^tP \times [sigmoïde(P \times W_{10}) - s_{10}] \cdot sigmoïde(P \times W_{10}) \cdot [1 - sigmoïde(P \times W_{10})]$$

Ensuite comme nous l'avions fait sans la fonction sigmoïde, nous allons calculer les 10 sorties d'une image en une seule fois, en définissant que :

$$W = (W_1 \quad W_2 \quad \cdots \quad W_9 \quad W_{10}) \qquad Grad = (grad_1 \quad grad_2 \quad \cdots \quad grad_9 \quad grad_{10})$$

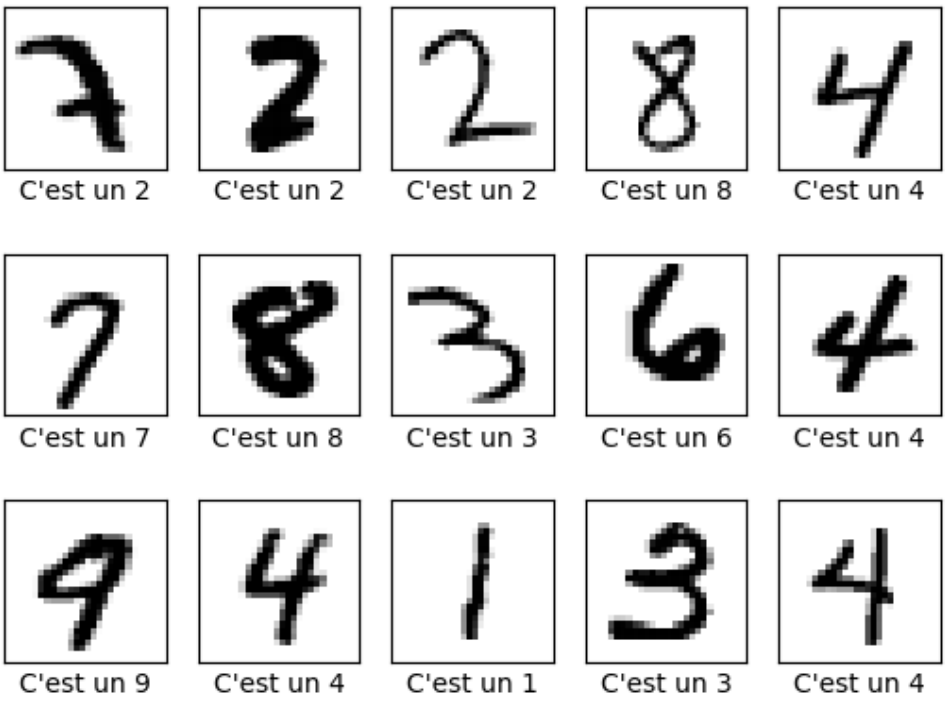
$$Grad = 2 \cdot {}^tP \times [sigmoïde(P \times W) - s] \cdot sigmoïde(P \times W) \cdot [1 - sigmoïde(P \times W)]$$

Annexe 2 – Programmes en python

Annexe 2.1 – Reconnaissance d'image avec la fonction sigmoïde

```python
import numpy as np
import matplotlib.pyplot as plt #ce module nous aide à afficher les images plus simplement
from random import choice
import tensorflow.examples.tutorials.mnist.input_data as input_data
data =input_data.read_data_sets("data/MNIST", one_hot=True)

def sigmoid(x):
    return 1/(1+np.exp(-x))
def der_sigmoid(x):
    return sigmoid(x)*(1-sigmoid(x))

entrees=data.train.images #d'abord on définit les entrées et les sorties...
sorties= data.train.labels
taille_batch = 100
a = (-taille_batch)
variables = np.zeros((784,10)) #il y a 784 variables pour chaque chiffre de 1 à 10

for i in range(8000):
    a = (a+taille_batch)%(len(entrees)-taille_batch)
    x_batch, y_batch = np.array(entrees[a:a+taille_batch]), np.array(sorties[a:a+taille_batch])
    derreur_dx = np.dot(2*x_batch.T,(sigmoid(np.dot(x_batch,variables))-y_batch)\
        *sigmoid(np.dot(x_batch,variables))*(1-sigmoid(np.dot(x_batch,variables))))
    variables -= (derreur_dx/len(x_batch))*0.5

images_a_tester, cases = plt.subplots(3, 5)#on affiche les variables sur deux lignes de 5 colonne
for case in cases.flat:
    image = choice(data.test.images) #on teste notre programme sur des images qu'il n'a jamais vues
    sortie_calculée= np.argmax(np.dot(image,variables))
    case.set_xlabel("C'est un " + str(sortie_calculée))
    case.imshow(image.reshape(28,28),cmap='binary')
    case.set_xticks([])
    case.set_yticks([])
plt.show()
```

Voici ce que ce programme affiche :



Annexe 2.2 – Statistiques du taux de réussite pour un réseau linéaire simple

```python
import numpy as np
from random import choice
import tensorflow.examples.tutorials.mnist.input_data as input_data
data =input_data.read_data_sets("data/MNIST", one_hot=True)


def apprendre(nbrit,batch):
    entrees=data.train.images #d'abord on définit les entrées et les sorties...
    sorties= data.train.labels
    taille_batch = batch
    a = (-taille_batch)
    variables = np.zeros((784,10)) #il y a 784 variables pour chaque chiffre de 1 à 10


    for i in range(nbrit):
        #creer le batch
        a = (a+taille_batch)%(len(entrees)-taille_batch)
        x_batch, y_batch = np.array(entrees[a:a+taille_batch]), np.array(sorties[a:a+taille_batch])
        #apprendre le batch
        derreur_dx = 2*np.dot(x_batch.T,(np.dot(x_batch,variables)-y_batch))
        variables -= (derreur_dx/len(x_batch))*0.005

    data.test.cls = np.array([label.argmax() for label in data.test.labels])
    a=0
    for x in range(10000):
        sortie = (np.dot(variables.T,data.test.images[x])).argmax()
        if sortie==data.test.cls[x]:
            a+=1
    print("Taux de réussite:",a/100,"%")

for b in [20,50,100,200]:
    for x in [50,100,200,500,2000,8000,15000]:
        print("long. batch:",b,"- Pour",x,"iteration --> ",end="")
        apprendre(x,b)
```

Voici ce que ce programme affiche :

long. batch: 20 - Pour 50 iteration --> Taux de réussite: 76.25 %
long. batch: 20 - Pour 100 iterations --> Taux de réussite: 80.53 %
long. batch: 20 - Pour 200 iterations --> Taux de réussite: 80.43 %
long. batch: 20 - Pour 500 iterations --> Taux de réussite: 84.8 %
long. batch: 20 - Pour 2000 iterations --> Taux de réussite: 85.36 %
long. batch: 20 - Pour 8000 iterations --> Taux de réussite: 84.57 %
long. batch: 20 - Pour 15000 iterations --> Taux de réussite: 85.32 %
long. batch: 50 - Pour 50 iterations --> Taux de réussite: 78.49 %
long. batch: 50 - Pour 100 iterations --> Taux de réussite: 80.08 %
long. batch: 50 - Pour 200 iterations --> Taux de réussite: 83.69 %
long. batch: 50 - Pour 500 iterations --> Taux de réussite: 85.34 %
long. batch: 50 - Pour 2000 iterations --> Taux de réussite: 85.46 %
long. batch: 50 - Pour 8000 iterations --> Taux de réussite: 85.15 %
long. batch: 50 - Pour 15000 iterations --> Taux de réussite: 85.28 %
long. batch: 100 - Pour 50 iterations --> Taux de réussite: 77.92 %
long. batch: 100 - Pour 100 iterations --> Taux de réussite: 82.28 %
long. batch: 100 - Pour 200 iterations --> Taux de réussite: 84.55 %
long. batch: 100 - Pour 500 iterations --> Taux de réussite: 84.66 %
long. batch: 100 - Pour 2000 iterations --> Taux de réussite: 85.72 %
long. batch: 100 - Pour 8000 iterations --> Taux de réussite: 85.77 %
long. batch: 100 - Pour 15000 iterations --> Taux de réussite: 85.17 %
long. batch: 200 - Pour 50 iterations --> Taux de réussite: 80.27 %

long. batch: 200 - Pour 100 iterations --> Taux de réussite: 82.95 %
long. batch: 200 - Pour 200 iterations --> Taux de réussite: 84.13 %
long. batch: 200 - Pour 500 iterations --> Taux de réussite: 85.22 %
long. batch: 200 - Pour 2000 iterations --> Taux de réussite: 85.7 %
long. batch: 200 - Pour 8000 iterations --> Taux de réussite: 85.42 %
long. batch: 200 - Pour 15000 iterations --> Taux de réussite: 85.72 %

Annexe 2.3 – Statistiques du taux de réussite avec la fonction sigmoïde

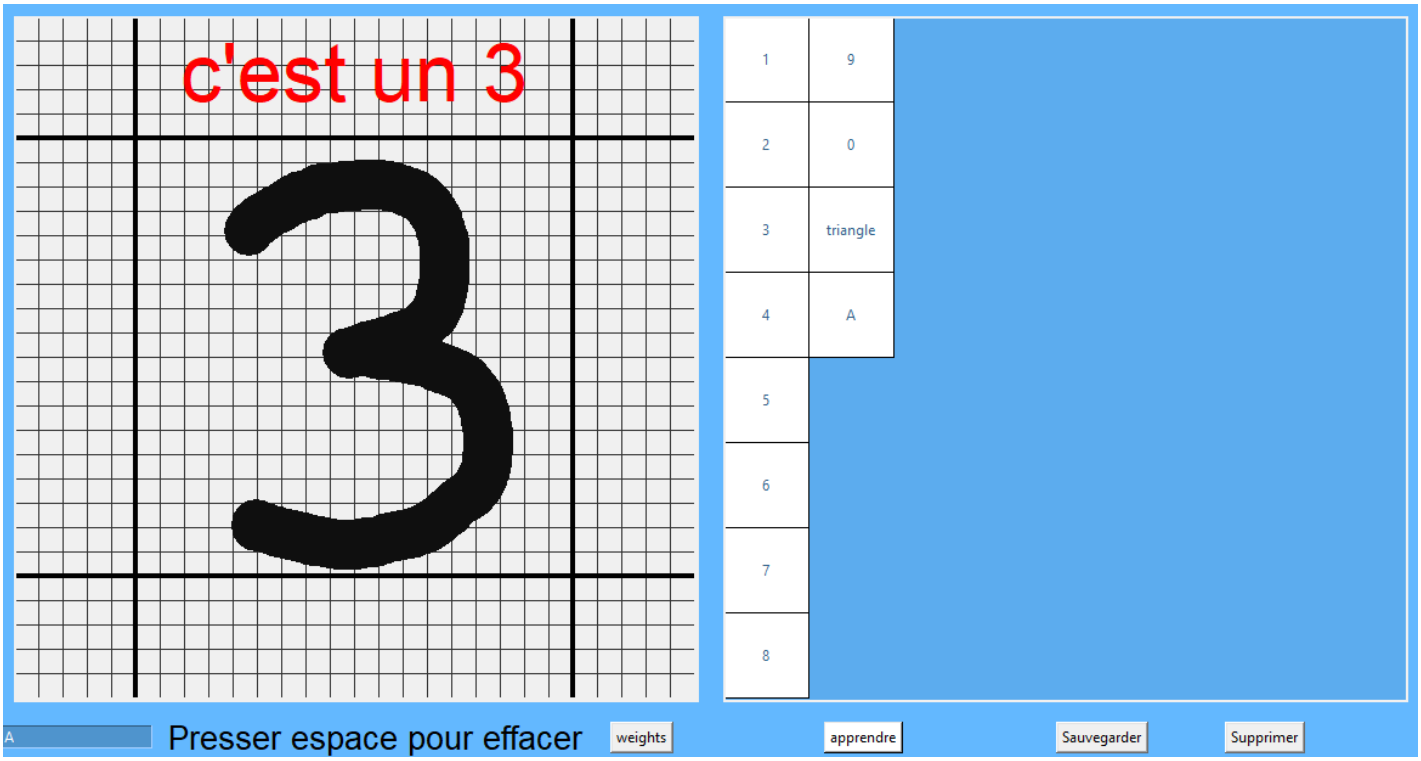
```python
import numpy as np
from random import choice
import tensorflow.examples.tutorials.mnist.input_data as input_data
data =input_data.read_data_sets("data/MNIST", one_hot=True)

def sigmoid(x):
    return 1/(1+np.exp(-x))
def der_sigmoid(x):
    return sigmoid(x)*(1-sigmoid(x))

def apprendre(nbrit,batch):
    entrees=data.train.images #d'abord on définit les entrées et les sorties...
    sorties= data.train.labels
    taille_batch = batch
    a = (-taille_batch)
    variables = np.zeros((784,10)) #il y a 784 variables pour chaque chiffre de 1 à 10

    for i in range(nbrit):
        a = (a+taille_batch)%(len(entrees)-taille_batch)
        x_batch, y_batch = np.array(entrees[a:a+taille_batch]), np.array(sorties[a:a+taille_batch])
        derreur_dx = np.dot(2*x_batch.T,(sigmoid(np.dot(x_batch,variables))-y_batch)\
            *sigmoid(np.dot(x_batch,variables))*(1-sigmoid(np.dot(x_batch,variables))))
        variables -= (derreur_dx/len(x_batch))*0.5

    data.test.cls = np.array([label.argmax() for label in data.test.labels])
    a=0
    for x in range(10000):
        sortie = (np.dot(variables.T,data.test.images[x])).argmax()
        if sortie==data.test.cls[x]:
            a+=1
    print("Taux de réussite:",a/100,"%")

for b in [20,50,100,150]:
    for x in [50,100,200,500,2000,6000,10000]:
        print("long. batch:",b,"- Pour",x,"iteration --> ",end="")
        apprendre(x,b)
```

Voici ce que le programme affiche :

long. batch: 20 - Pour 50 iterations --> Taux de réussite: 71.77 %
long. batch: 20 - Pour 100 iterations --> Taux de réussite: 83.67 %
long. batch: 20 - Pour 200 iterations --> Taux de réussite: 84.59 %
long. batch: 20 - Pour 500 iterations --> Taux de réussite: 88.42 %
long. batch: 20 - Pour 2000 iterations --> Taux de réussite: 90.03 %
long. batch: 20 - Pour 6000 iterations --> Taux de réussite: 90.94 %
long. batch: 20 - Pour 10000 iterations --> Taux de réussite: 91.49 %
long. batch: 50 - Pour 50 iterations --> Taux de réussite: 82.7 %
long. batch: 50 - Pour 100 iterations --> Taux de réussite: 84.98 %
long. batch: 50 - Pour 200 iterations --> Taux de réussite: 88.0 %
long. batch: 50 - Pour 500 iterations --> Taux de réussite: 89.17 %

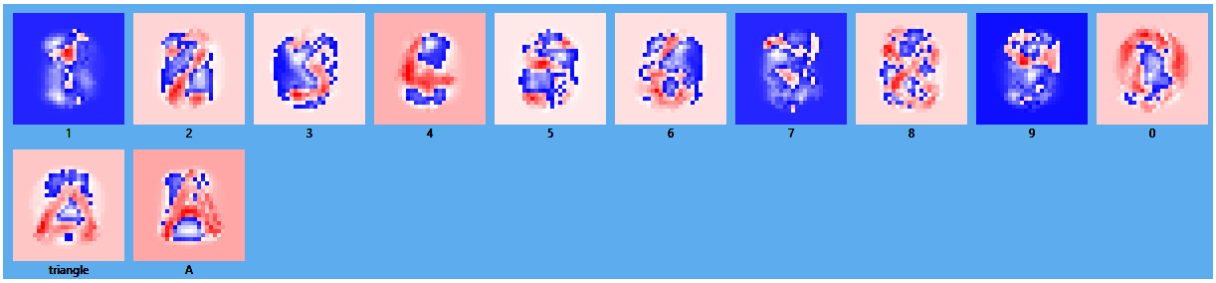
long. batch: 50 - Pour 2000 iterations --> Taux de réussite: 90.89 %
long. batch: 50 - Pour 6000 iterations --> Taux de réussite: 91.38 %
long. batch: 50 - Pour 10000 iterations --> Taux de réussite: 91.27 %
long. batch: 100 - Pour 50 iterations --> Taux de réussite: 82.71 %
long. batch: 100 - Pour 100 iterations --> Taux de réussite: 86.62 %
long. batch: 100 - Pour 200 iterations --> Taux de réussite: 88.47 %
long. batch: 100 - Pour 500 iterations --> Taux de réussite: 89.47 %
long. batch: 100 - Pour 2000 iterations --> Taux de réussite: 91.06 %
long. batch: 100 - Pour 6000 iterations --> Taux de réussite: 91.56 %
long. batch: 100 - Pour 10000 iterations --> Taux de réussite: 91.65 %
long. batch: 150 - Pour 50 iterations --> Taux de réussite: 82.25 %
long. batch: 150 - Pour 100 iterations --> Taux de réussite: 86.86 %
long. batch: 150 - Pour 200 iterations --> Taux de réussite: 88.45 %
long. batch: 150 - Pour 500 iterations --> Taux de réussite: 89.9 %
long. batch: 150 - Pour 2000 iterations --> Taux de réussite: 90.83 %
long. batch: 150 - Pour 6000 iterations --> Taux de réussite: 91.5 %
long. batch: 150 - Pour 10000 iterations --> Taux de réussite: 91.68 %

Annexe 2.4 – Reconnaissance de dessins avec un réseau linéaire simple



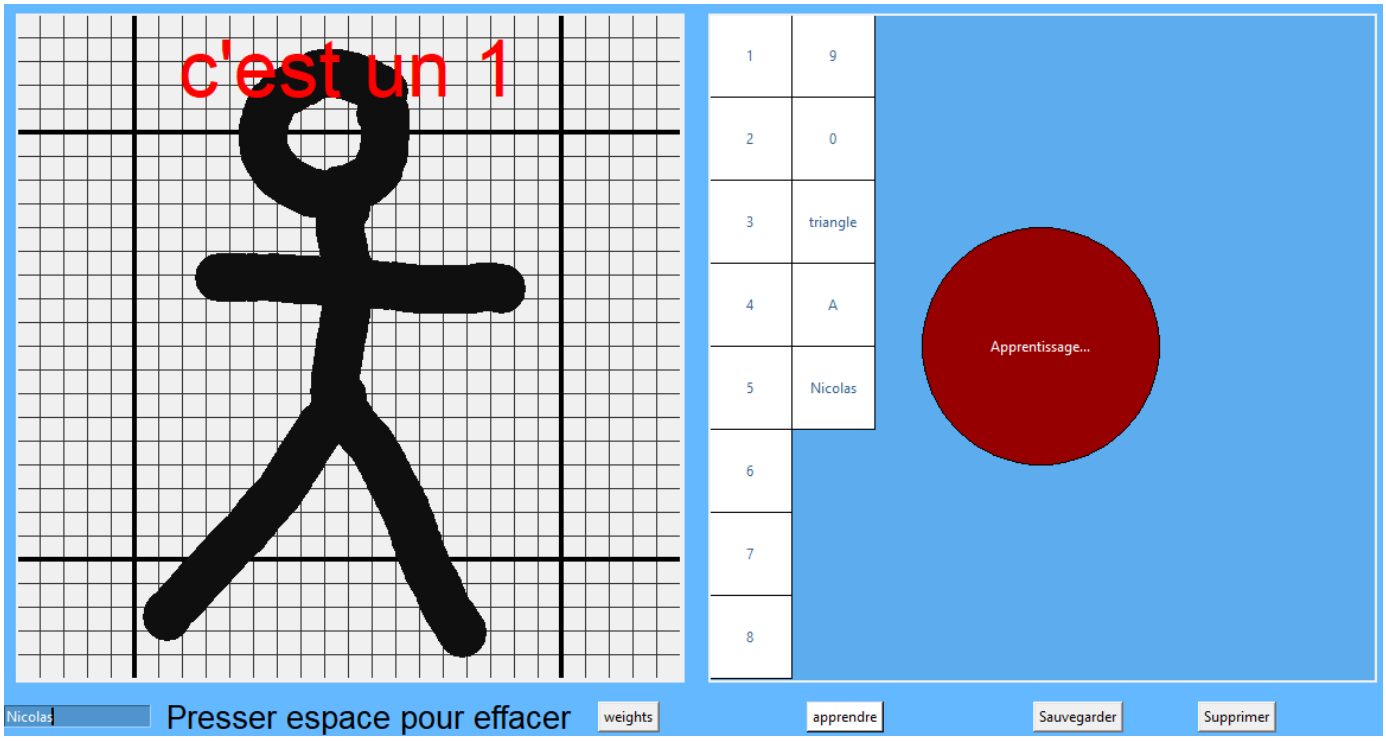
Ce programme prend en entrée le dessin dessiné sur la partie gauche de l'image, calcule ensuite de quoi il s'agit puis affiche en rouge le résultat qu'il pense être le bon dans le coin supérieur gauche. Il va choisir une des 12 réponses affichées dans les case sur la partie gauche de l'image. Il suffit de presser la touche « espace » pour effacer un dessin.

Il calcule le résultat en appliquant des filtres que l'on peut afficher en cliquant sur le bouton « weights » (en bas, au milieu).



Il va appliquer les filtres 10 fois par seconde sur le dessin et donc actualiser le résultat affiché.

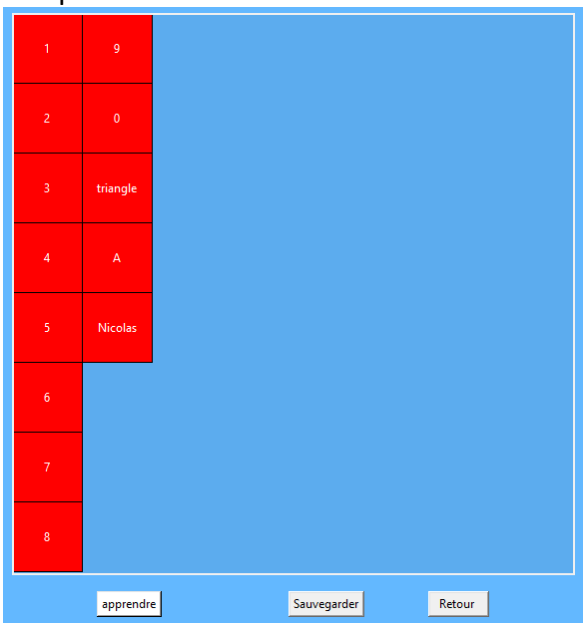
Pour calculer ces filtres, le programme va calculer le gradient sur l'ensemble des images contenues dans un fichier texte (sous forme de 0 et de 1).



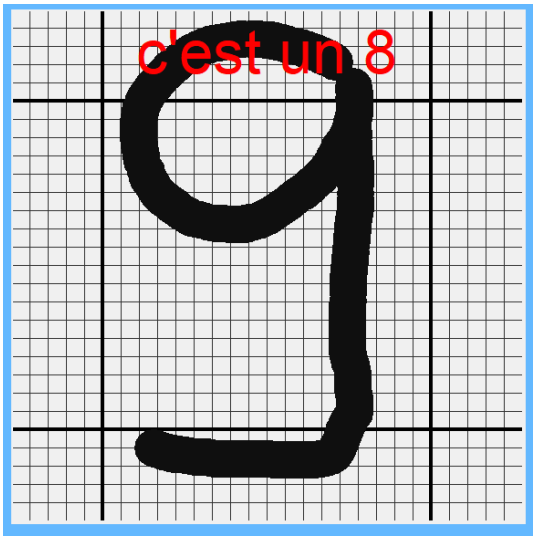
Ces images ont également été créées sur ce programme. Pour en ajouter de nouvelles, il suffit d'entrer le nom que l'on souhaite donner à l'image dans la zone prévue à cet effet (en bas à gauche) puis de taper la touche « Enter » et un nouveau bouton apparaîtra dans la partie de droite avec le nom entré. Il faut ensuite faire un dessin (ici un bonhomme) et de l'attribuer à la nouvelle sortie (« Nicolas ») en lui cliquant dessus. Afin que le programme reconnaisse le dessin par la suite, il est préférable de faire plusieurs dessins (ici des dessins de bonhomme). Ainsi, l'ordinateur pourra mieux comprendre les caractéristiques qui constituent ce dessin.
Il ne reste plus qu'à cliquer sur le bouton apprendre (au milieu en bas) et un cercle rouge nous informera que notre programme est en train d'ajuster les paramètres (=apprendre) avec les dessins que nous lui avons fourni.

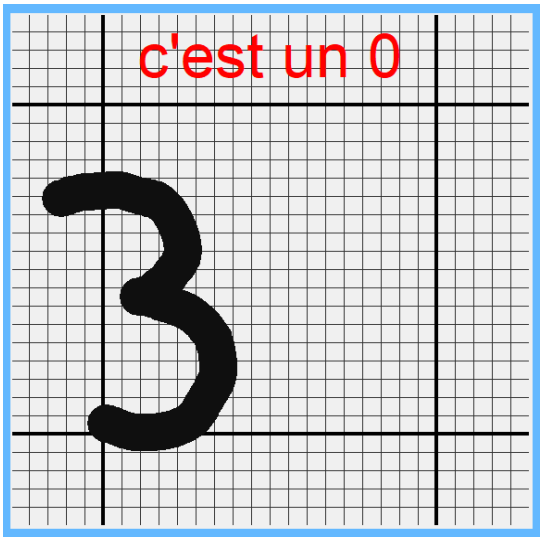
L'ordinateur aura alors pris connaisance de ce nouveau dessin et le reconnaîtra lorsqu'il recevra un nouveau dessin avec des caractéristiques similaires.

Si nous souhaitons supprimer des dessins, il est possible de cliquer sur le bouton supprimer (en bas à droit). Les boutons vont alors s'afficher en rouge et il suffit de cliquer dessus pour qu'ils disparaissent et les dessins auxquels ils étaient associés sont alors supprimés.
Il suffit ensuite de cliquer sur retour (le nom s'est changé au moment où l'on clique sur supprimer) afin de continuer à utiliser le programme normalement.
Avant de fermer le programme, si l'on souhaite que les modifications effectuées soient sauvegardées et réapparraissent donc à la prochaine ouverture du programme, alors il faut cliquer sur le bouton sauvegarder.



Ce programme ne marche pourtant pas à la perfection. Il suffit que le dessin soit plus grand ou décalé par rapport aux dessins « habituels » et l'ordinateur ne pravient plus du tout à reconnaître de quoi il s'agit. Le problème vient du fait que le programme attribue trop d'importance à la position des pixels de l'image et non à l'association des pixels entre eux. Avec un réseau de neurones si simple, il est difficle de faire un meilleur résultat. C'est pourquoi j'ai utilisé par la suite des réseaux de neurones à convolution, qui eux prennent plus en compte les différentes formes qui composent les images.





Voici le code de ce programme en python (pour que ce code fonctionne, il faut qu'il y ai un fichier texte nommé dessins.txt dans le même dossier que le code) :

```
1.  import numpy as np
2.  from copy import deepcopy
3.  from random import *
4.  try:
5.      from tkinter import *
6.  except:
7.      from Tkinter import *
8.
9.  def sigmoid(x):
10.     return 1/(1+np.exp(-x))
11.
12. def der_sigmoid(x):
13.     return sigmoid(x)*(1-sigmoid(x))
14.
15. def dessiner(event):
16.     global matrice,evaluerimg
17.     X = event.x
18.     Y = event.y
19.     if X<0:
20.         X=0
21.     if X>Largeur:
22.         X=Largeur
23.     if Y<0:
24.         Y=0
25.     if Y>Hauteur:
26.         Y=Hauteur
27.     evaluerimg=True
28.     matrice[Y*28//Hauteur][X*28//Largeur]=1
29.     Cadre.create_oval(X-TailleRond,Y-TailleRond,\
30. X+TailleRond,Y+TailleRond,width=0,fill="grey6")
31.
32. def effacer_dessin (t=0):
33.     global matrice,evaluerimg
34.     evaluerimg=False
35.     Cadre.delete(ALL)
36.     matrice=[]
37.     for y in range(28):
38.         ligne = []
39.         for x in range(28):
40.             ligne.append(0)
41.         matrice.append(ligne)
42.     for x in range(27):
43.         Cadre.create_line(x*20+20,0,x*20+20,560, fill="grey20")
44.         Cadre.create_line(0,x*20+20,560,x*20+20, fill="grey20")
45.     Cadre.create_line(0,4*20+20,560,4*20+20, fill="black",width=4)
46.     Cadre.create_line(0,22*20+20,560,22*20+20, fill="black",width=4)
47.     Cadre.create_line(4*20+20,0,4*20+20,560, fill="black",width=4)
48.     Cadre.create_line(22*20+20,0,22*20+20,560, fill="black",width=4)
49.
50. def compiler():
51.     listeimage=[]
52.     matricefin=deepcopy(matrice)
53.     for y in range(len(matrice)):
54.         for x in range(len(matrice)):
55.             a=0
56.             if matrice[y][x]!=1:
57.                 if matrice[y][(x+1)%28]==1 or matrice[y][(x-1)%28]==1:
58.                     a+=0.6
59.                 if matrice[(y+1)%28][x]==1 or matrice[(y-1)%28][x]==1:
60.                     a+=0.6
61.                 if matrice[y][(x+2)%28]==1 or matrice[y][(x-2)%28]==1:
62.                     a+=0.1
63.                 if matrice[(y+2)%28][x]==1 or matrice[(y-2)%28][x]==1:
64.                     a+=0.1
65.                 if matrice[(y+1)%28][(x+2)%28]==1 or \
66.  matrice[(y+1)%28][(x-2)%28]==1:
```

```
                    a+=0.03
                if matrice[(y-1)%28][(x+2)%28]==1 or \
matrice[(y-1)%28][(x-2)%28]==1:
                    a+=0.03
                if matrice[(y+2)%28][(x+1)%28]==1 or\
 matrice[(y-2)%28][(x+1)%28]==1:
                    a+=0.03
                if matrice[(y+2)%28][(x-1)%28]==1 or \
matrice[(y-2)%28][(x-1)%28]==1:
                    a+=0.03
                if a>1:
                    a=1
                if a < 0.15:
                    a=0
            else:
                a=1
            matricefin[y][x]=a
    for y in matricefin:
        for x in y:
            listeimage.append(x)
    return listeimage

def recuperer(): #recuperer les anciens dessins dans un fichier texte
    global noms,dessins,association,idees,vects
    fichier = open("dessins.txt", 'r')
    texte = fichier.read()
    try:
        texte2=texte.split(",")
        texte2.remove(texte2[-1])
        nbras=int(texte2[0])
        idees=int(texte2[1])
        association = texte2[2:nbras+2]
        noms = texte2[nbras+2:nbras+idees+2]
         texte3=texte2[nbras+idees+2:]
         i=0
         for z in range(nbras):
            lis=[]
            for y in range(784):
                try:
                    lis.append(float(texte3[i]))
                except:
                    lis.append(0.0)
                i+=1
            dessins.append(lis)
         vects=[]
         for name in association:
            vect=[]
            for a in range(idees):
                if noms.index(name)==a:
                    vect.append(1)
                else:
                    vect.append(0)
            vects.append(vect)
        vects=np.array(vects)
    except:
        print("fichier vide")
    fichier.close()

def quitter():  #enregistrer les nouveaux dessins dans un fichier texte
    fichier = open("dessins.txt", 'w')
    fichier.write(str(len(association)))
    fichier.write(",")
    fichier.write(str(len(noms)))
    fichier.write(",")
    for x in association:
        fichier.write(x)
```

```
133.         fichier.write(",")
134.     for x in noms:
135.         fichier.write(x)
136.         fichier.write(",")
137.     for x in dessins:
138.         for y in x:
139.             fichier.write(str(y))
140.             fichier.write(",")
141.     fichier.close()
142.
143. def boutons():
144.     couleur1 = "white"
145.     couleur2 = "SteelBlue4"
146.     if supprime:
147.         couleur1 = "red"
148.         couleur2 = "white"
149.     Bou.delete(ALL)
150.     for x in range(len(noms)):
151.         Bou.create_rectangle((x//8)*70,(x%8)*70,(x//8)*70+70,\
152. (x%8)*70+70,fill=couleur1)
153.         Bou.create_text((x//8)*70+35,(x%8)*70+35,text=str(noms[x]),fill=couleur2)
154.
155. def presse_sur_bouton(event):
156.     #lorsqu'on clique sur un bouton pour dire de quel symbole il s'agit
157.     global dessins,association,vects,noms,idees
158.     X = event.x
159.     Y = event.y
160.     cx = (X//70)%8
161.     cy = (Y//70)
162.     try:
163.         n = noms[cy+8*cx]
164.         if supprime:
165.             copieass = deepcopy(association)
166.             copiedess = deepcopy(dessins)
167.             for i in range(len(association)):
168.                 if association[i]==n:
169.                     copieass.remove(association[i])
170.                     copiedess.remove(dessins[i])
171.             association = deepcopy(copieass)
172.             dessins = deepcopy(copiedess)
173.             noms.remove(n)
174.             boutons()
175.         else:
176.             a=compiler()
177.             dessins.append(a)
178.             association.append(n)
179.             effacer_dessin()
180.         idees=len(noms)
181.         vects=[]
182.         for name in association:
183.             vect=[]
184.             for a in range(idees):
185.                 if noms.index(name)==a:
186.                     vect.append(1)
187.                 else:
188.                     vect.append(0)
189.             vects.append(vect)
190.         vects=np.array(vects)
191.     except:
192.         print("pas de bouton")
193.
194. def valider(a=0):
195.     #creer de nouveaux boutons --> nouvelles association ex: triangle, a, 5, etc.
196.     global noms
197.     nom=nom_ecrit.get()
198.     noms.append(nom)
```

```
199.     boutons()
200.
201. def active_supprimer ():
202.     #supprimer des boutons
203.     global supprime
204.     if supprime:
205.         supprime = False
206.         Suppr.config(text="Supprimer")
207.     else:
208.         supprime = True
209.         Suppr.config(text="  Retour   ")
210.     boutons()
211.
212. def apprendre(xt,yt):
213.     #ajuster les weights par rapport aux dessins
214.     global weights,taille_input,taille_output
215.     taille_batch=100
216.     taille_input=np.size(xt[0])
217.     taille_output=np.size(yt[0])
218.     weights =np.zeros((taille_input,taille_output))
219.     for i in range(1001):
220.         lot_x=np.array(xt)
221.         lot_y =np.array(yt)
222.         learn_batch(lot_x,lot_y)
223.
224. def learn_batch(x_batch,y_batch):
225.     global weights
226.     wt=np.zeros((taille_input,taille_output))
227.     bt=np.zeros((taille_output,1))
228.     for x,y in zip(x_batch,y_batch):
229.         #cost= (sigmoid(np.dot(x,weights))-y)**2
230.         dcost_db = \
231. 2*(sigmoid(np.dot(x,weights))-y)*(der_sigmoid(np.dot(x,weights)))
232.         wt+=np.outer(x,dcost_db)
233.     weights -= wt/len(x_batch)*vitesse
234.
235. def active_apprendre():
236.     #gérer les actions qu'il se passe lorsqu'on clique sur apprendre
237.     cercle_appr= Bou.create_oval(Largeur/2-100,Hauteur/2+100,\
238.         Largeur/2+100,Hauteur/2-100,fill=rgb((150,0,0)))
239.     text_appr= Bou.create_text(Largeur/2,Hauteur/2,\
240.         text="Apprentissage...",fill = "white")
241.     Bou.update()
242.     apprendre(xt=dessins,yt=vects)
243.     Bou.delete(cercle_appr,text_appr)
244.     effacer_dessin()
245.     evaluer_dessin ()
246.
247. def evaluer_dessin ():
248.     #deviner le symbole dessiné avec la souris 10 fois par seconde
249.     global result
250.     try:
251.         Cadre.delete(result)
252.     except:
253.         pass
254.     if True:
255.         a = compiler()
256.         w = weights
257.         y=np.dot(a,w)
258.         sortie = y.argmax()
259.         if evaluerimg:
260.             result=Cadre.create_text(Largeur/2,(Hauteur/15)+10,\
261.                 text="c'est un "+str(noms[sortie]),font=('', '50'),fill="red")
262.     else:
263.         pass
264.     fenetre.after(100,evaluer_dessin)
```

```
265.
266. def plot_weights():
267.     #afficher les weights
268.     try:
269.         w = weights.T
270.         pix = 4 #largeur d'un pixel
271.         space = 9 #espacement entre les images
272.         fen = Toplevel()
273.         fen.title("weights")
274.         nbr_weights = len(w)
275.         if nbr_weights>10:
276.             nbr_weights = 10
277.         canv = Canvas(fen,width=(nbr_weights)*(28)*pix+space*nbr_weights+7,\
278.             height=(len(w)//10 +1)*(32)*pix+3 + space*(len(w)//10 +1),\
279. bg ='SteelBlue2')
280.         for i,img in enumerate(w):
281.             originex=3+space+(28*pix+space)*(i%10)
282.             originey=2+space+(32*pix+space)*(i//10)
283.             mi = np.min(img)
284.             mx = np.max(img)
285.             esp = (mx-mi)/508 #code rgb est de 0 à 255
286.             #-> pour rouge et bleu -> 510 - les bornes
287.             for y in range(28):
288.                 for x in range(28):
289.                     coordx=originex+x*pix
290.                     coordy=originey+y*pix
291.                     val=img[y*28+x]
292.                     RGB=int((val-mi)//esp)
293.                     if RGB<=254:
294.                         coul = (255-RGB,255-RGB,255)
295.                     if RGB>254:
296.                         RGB-=254
297.                         coul=(255,255-RGB,255-RGB)
298.                     canv.create_rectangle(coordx,coordy,coordx+pix,coordy+pix,\
299.                         fill=rgb(coul),width=0)
300.                     canv.create_text(originex+14*pix,originey+30*pix,\
301. text=str(noms[i]))
302.         canv.pack()
303.         fen.mainloop()
304.     except:
305.         effacer_dessin ()
306.         Cadre.create_text(Largeur/2,(Hauteur/15)+10,text="d'abord apprendre!",\
307.             font=('', '40'),fill="red")
308.
309. def rgb(code_rgb):
310.     #renvoie une couleur rgb compréhensible par tkinter à partir des données r,g,b
311.     #--> c'est pour afficher les weights
312.     r = code_rgb[0]
313.     g = code_rgb[1]
314.     b = code_rgb[2]
315.     coul = '#%02x%02x%02x' % (r, g, b)
316.     return coul
317.
318. dessins=[]
319. association=[]
320. vects=[]
321. noms = []
322. idees=[]
323. supprime = False
324. Largeur = 560
325. Hauteur = 560
326. TailleRond = 20
327. vitesse=0.6 #j'ai choisi ça après quelques essais
328.
329. fenetre = Tk()
330. fenetre.title("dessiner")
```

```
331. fenetre.configure(background='SteelBlue1')
332. nom_ecrit= StringVar()
333. 
334. Cadre = Canvas(fenetre,width=Largeur,height=Hauteur,bg ="grey94")
335. Cadre.bind('<B1-Motion>',dessiner)
336. Cadre.bind('<Button-1>',dessiner)
337. Cadre.grid(row=1,rowspan=3,columnspan=6,padx=10,pady=10)
338. 
339. Bou = Canvas(fenetre,width=Largeur,height=Hauteur,bg ='SteelBlue2')
340. Bou.grid(row=1,column=6,columnspan=6,padx=10,pady=10)
341. Bou.bind('<Button-1>',presse_sur_bouton)
342. 
343. app = Button(fenetre,text="apprendre",bg="white",command=active_apprendre)
344. app.grid(row=5,column=7)
345. 
346. effacer=Label(fenetre,text="Presser espace pour effacer",\
347.     font=("","20"),bg='SteelBlue1',)
348. effacer.grid(row=5,column=3)
349. 
350. Quitte=Button(fenetre,text="Sauvegarder",command=quitter)
351. Quitte.grid(row=5,column=9)
352. 
353. Suppr =Button(fenetre,text="Supprimer",command=active_supprimer)
354. Suppr.grid(row=5,column=10)
355. 
356. We =Button(fenetre,text="weights",command=plot_weights)
357. We.grid(row=5,column=4)
358. 
359. Champ = Entry(fenetre, textvariable= nom_ecrit, fg='white',bg='SteelBlue3')
360. Champ.grid(row=5,column=0,columnspan=3)
361. Champ.bind('<Return>', valider)
362. fenetre.bind("<space>",effacer_dessin)
363. 
364. recuperer()
365. boutons()
366. effacer_dessin()
367. text_item=Cadre.create_text(Largeur/2,9*Hauteur/10,\
368.     text="Il faut d'abord apprendre\n        avant d'évaluer",font=('', '30') )
369. bbox = Cadre.bbox(text_item)
370. rect_item = Cadre.create_rectangle(bbox, outline="black", fill="grey94")
371. Cadre.tag_raise(text_item,rect_item)
372. fenetre.mainloop()
373. 
```

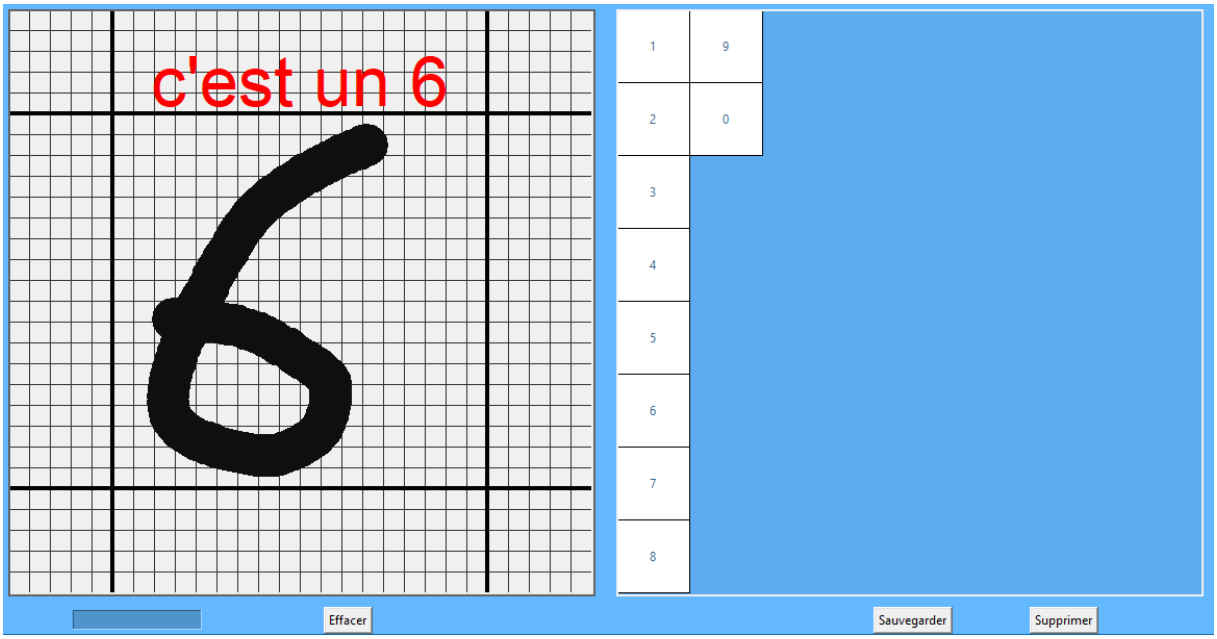
Annexe 2.5 – Reconnaissance de dessins avec un réseau de neurones à convolution

Les réseaux de neurones à convolution demandent plus de temps de calcul qu'un simple réseau de neurone linéaire. Entraîner le programme peut donc nécessiter une dizaine de minutes de calculs, c'est pourquoi j'ai créé un programme qui retient ce qu'il a appris (les weights) dans un fichier. Le réseau sera donc déjà entraîné à la prochaine ouverture du programme. En réalité j'ai séparé le tout en deux programmes, l'un servant à apprendre (ajuster les wights) et l'autre servant à créer des images (des dessins) et à évaluer de nouveaux dessins.

Voici ce qu'affiche le programme servant à entraîner le réseau :

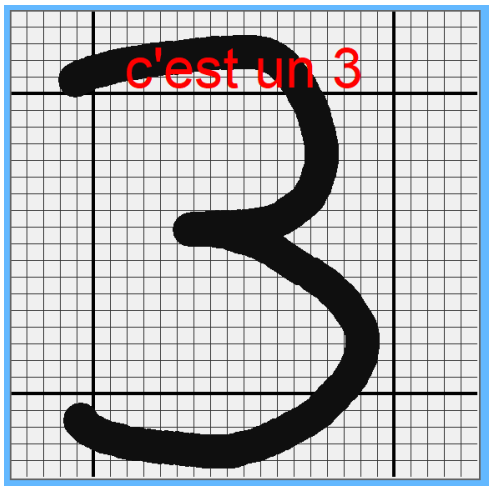
```
model restauré
apprentissage en cours pour 100 Iterations...
                    |100%|
####################taux sur le learning set: 100.0 %
[Finished in 148.8s]
```

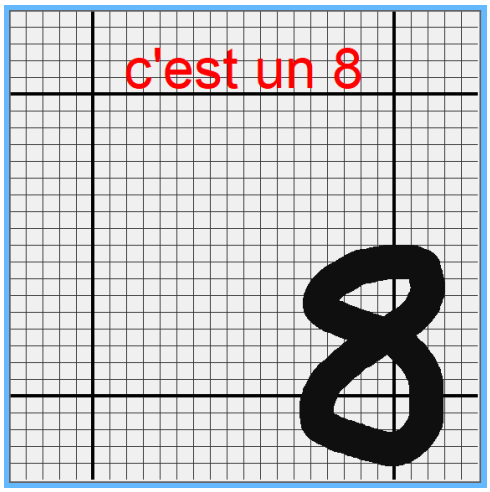
Ce programme va donc entraîner le réseau de neurones avec les dessins que j'ai créés à l'aide du second programme ci-dessous.



Ce programme fonctionne de la même manière que le programme 2.4. Il suffit de faire un dessin puis de cliquer dans la partie de droite sur ce à quoi correspond le dessin et le programme fera ainsi l'association entre les deux.

Pour faire reconnaître les chiffres à ce programme, j'ai dessiné plus d'une centaine de fois chaque chiffre, en variant les tailles, les styles et les positions d'écriture des chiffres. J'aurais bien sûr pu faire de même avec d'autres symboles, comme des lettres, des formes etc.
Ce programme fonctionne bien mieux que le précédent et reconnaît donc tous les chiffres dans n'importe quels cas (à moins qu'ils soient écrits de manière très inhabituelle).





Le code de ce programme est séparé en 2 parties. Voici la première partie, servant à entraîner le réseau de neurones :

```
1.  import numpy as np
2.  from random import shuffle
3.  import tensorflow as tf
4.  import sys
5.
6.  def new_weights(shape):
7.      return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
8.
9.  def new_biases(length):
10.     return tf.Variable(tf.constant(0.05, shape=[length]))
11.
12. def new_conv_layer(input,num_input_channels,filter_size,\
13. num_filters,use_pooling=True):
14.     shape = [filter_size, filter_size, num_input_channels, num_filters]
15.     weights = new_weights(shape=shape)
16.     biases = new_biases(length=num_filters)
17.     layer = tf.nn.conv2d(input=input,filter=weights,strides=[1, 1, 1, 1],\
18. padding='SAME')
19.     layer += biases
20.     if use_pooling:
21.         layer = tf.nn.max_pool(value=layer,ksize=[1, 2, 2, 1],\
22. strides=[1, 2, 2, 1],padding='SAME')
23.     layer = tf.nn.relu(layer)
24.     return layer, weights
25.
26. def flatten_layer(layer):
27.     layer_shape = layer.get_shape()
28.     num_features = layer_shape[1:4].num_elements()
29.     layer_flat = tf.reshape(layer, [-1, num_features])
30.     return layer_flat, num_features
31.
32. def new_fc_layer(input,num_inputs,num_outputs,use_relu=True):
33.     weights = new_weights(shape=[num_inputs, num_outputs])
34.     biases = new_biases(length=num_outputs)
35.     layer = tf.matmul(input, weights) + biases
36.     if use_relu:
37.         layer = tf.nn.relu(layer)
38.     return layer
39.
40. def creer_model():
41.     global saver,x,x_image,y_true,y_true_cls,layer_conv1, weights_conv1,\
42. layer_conv2, weights_conv2,layer_flat, num_features,layer_fc1,layer_fc2,\
43. y_pred,y_pred_cls,cross_entropy,cost,optimizer,correct_prediction,accuracy,session
44.     tf.reset_default_graph()
45.     x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name='x')
46.     x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
47.     y_true = tf.placeholder(tf.float32, shape=[None, num_classes], name='y_true')
48.     y_true_cls = tf.argmax(y_true, axis=1)
49.     layer_conv1, weights_conv1 = \
50.         new_conv_layer(input=x_image,
51.                        num_input_channels=num_channels,
52.                        filter_size=filter_size1,
53.                        num_filters=num_filters1,
54.                        use_pooling=True)
55.     layer_conv2, weights_conv2 = \
56.         new_conv_layer(input=layer_conv1,
57.                        num_input_channels=num_filters1,
58.                        filter_size=filter_size2,
59.                        num_filters=num_filters2,
60.                        use_pooling=True)
61.     layer_flat, num_features = flatten_layer(layer_conv2)
62.
```

```
    layer_fc1 = new_fc_layer(input=layer_flat,
                             num_inputs=num_features,
                             num_outputs=fc_size,
                             use_relu=True)
    layer_fc2 = new_fc_layer(input=layer_fc1,
                             num_inputs=fc_size,
                             num_outputs=num_classes,
                             use_relu=False)
    y_pred = tf.nn.softmax(layer_fc2)
    y_pred_cls = tf.argmax(y_pred, axis=1)
    cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits=layer_fc2,
                                                               labels=y_true)
    cost = tf.reduce_mean(cross_entropy)
    optimizer = tf.train.AdamOptimizer(learning_rate=0.0001).minimize(cost)
    correct_prediction = tf.equal(y_pred_cls, y_true_cls)
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    session = tf.Session()
    session.run(tf.global_variables_initializer())
    saver=tf.train.Saver()

def optimize(num_iterations):
    print("apprentissage en cours pour",num_iterations,"Iterations...")

    for i in range(num_iterations//5):
        print(" ",end="")
    print("|100%|")

    for i in range(num_iterations):
        ind=0
        indice_aleatoire = list(range(len(dessins)))
        shuffle(indice_aleatoire)
        dessins_apprend=[]
        vects_apprend=[]
        vetcs=np.array(vects)
        for iii in indice_aleatoire:
            dessins_apprend.append(dessins[iii])
            vects_apprend.append(vects[iii])
         for t in range(len(dessins)//train_batch_size):
            x_batch, y_true_batch = \
np.array(dessins_apprend[ind:ind+train_batch_size]),\
np.array(vects_apprend[ind:ind+train_batch_size])
            ind+=train_batch_size
            feed_dict_train = {x: x_batch,y_true: y_true_batch}
            session.run(optimizer, feed_dict=feed_dict_train)
         if i%5 ==0:
            sys.stdout.write("#")
            sys.stdout.flush()

def recuperer():
    global noms,dessins,association,nbrlabels,vects
    fichier = open("dataperso.txt", 'r')
    texte = fichier.read()
    try:
        texte2=texte.split(",")
        texte2.remove(texte2[-1])
        nbras=int(texte2[0])
        nbrlabels=int(texte2[1])
        association = texte2[2:nbras+2]
        noms = texte2[nbras+2:nbras+nbrlabels+2]
        texte3=texte2[nbras+nbrlabels+2:]
        i=0
        for z in range(nbras):
            lis=[]
            for y in range(784):
                try:
                    lis.append(float(texte3[i]))
```

```
                except:
                    lis.append(0.0)
                i+=1
            dessins.append(lis)
        vects=[]
        for name in association:
            vect=[]
            for a in range(nbrlabels):
                if noms.index(name)==a:
                    vect.append(1)
                else:
                    vect.append(0)
            vects.append(vect)
        vects=np.array(vects)
    except:
        print("fichier vide")
    fichier.close()

def print_accuracy():
    global dessins_faux
    nbr_juste=0
    dessins_faux=[]
    for i in range(len(dessins)):
        img=dessins[i]
        ass=association[i]
        pred = session.run(y_pred_cls,feed_dict={ x : [img]})
        if (pred+1)%10==int(ass):
            nbr_juste+=1
        else:
            dessins_faux.append(img)
    print("taux sur le learning set:",(nbr_juste/len(dessins))*100,"%")

dessins=[]
vects=[]
recuperer()
print(len(dessins))
img_size = 28
img_size_flat = img_size * img_size
img_shape = (img_size, img_size)
num_channels = 1
num_classes = 10
filter_size1 = 5
num_filters1 = 16
filter_size2 = 5
num_filters2 = 36
fc_size = 128
train_batch_size = 80

xt=dessins
yt = vects
taille_input=np.size(xt[0])
taille_output=np.size(yt[0])

creer_model()

try:
    saver.restore(session, "weights_entrainepasMNIST2/model.ckpt")
    print("model restauré")
except:
    pass

optimize(num_iterations=5)
save_path = saver.save(session, "weights_entrainepasMNIST2/model.ckpt")
print_accuracy()
```

Voici la deuxième partie ue code permettant de créer de nouveaux dessins et d'évaluer des dessins:

```
1.  import tensorflow as tf
2.  import numpy as np
3.  from tkinter import *
4.  from random import *
5.
6.  def new_weights(shape):
7.      return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
8.
9.  def new_biases(length):
10.     return tf.Variable(tf.constant(0.05, shape=[length]))
11.
12. def new_conv_layer(input,num_input_channels,filter_size,num_filters,use_pooling=True
    ):
13.     shape = [filter_size, filter_size, num_input_channels, num_filters]
14.     weights = new_weights(shape=shape)
15.     biases = new_biases(length=num_filters)
16.     layer = tf.nn.conv2d(input=input,filter=weights,strides=[1, 1, 1, 1],padding='SA
    ME')
17.     layer += biases
18.     if use_pooling:
19.         layer = tf.nn.max_pool(value=layer,ksize=[1, 2, 2, 1],strides=[1, 2, 2, 1],p
    adding='SAME')
20.     layer = tf.nn.relu(layer)
21.     return layer, weights
22.
23. def flatten_layer(layer):
24.     layer_shape = layer.get_shape()
25.     num_features = layer_shape[1:4].num_elements()
26.     layer_flat = tf.reshape(layer, [-1, num_features])
27.     return layer_flat, num_features
28.
29. def new_fc_layer(input,num_inputs,num_outputs,use_relu=True):
30.     weights = new_weights(shape=[num_inputs, num_outputs])
31.     biases = new_biases(length=num_outputs)
32.     layer = tf.matmul(input, weights) + biases
33.     if use_relu:
34.         layer = tf.nn.relu(layer)
35.     return layer
36.
37. def creer_model():
38.     global saver,x,x_image,y_true,y_true_cls,layer_conv1, weights_conv1,layer_conv2,
     weights_conv2,layer_flat, num_features,layer_fc1,layer_fc2,y_pred,y_pred_cls,cross_
    entropy,cost,optimizer,correct_prediction,accuracy,session
39.     tf.reset_default_graph()
40.     x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name='x')
41.     x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
42.     y_true = tf.placeholder(tf.float32, shape=[None, num_classes], name='y_true')
43.     y_true_cls = tf.argmax(y_true, axis=1)
44.     layer_conv1, weights_conv1 = \
45.         new_conv_layer(input=x_image,
46.                        num_input_channels=num_channels,
47.                        filter_size=filter_size1,
48.                        num_filters=num_filters1,
49.                        use_pooling=True)
50.     layer_conv2, weights_conv2 = \
51.         new_conv_layer(input=layer_conv1,
52.                        num_input_channels=num_filters1,
53.                        filter_size=filter_size2,
54.                        num_filters=num_filters2,
55.                        use_pooling=True)
56.     layer_flat, num_features = flatten_layer(layer_conv2)
57.
```

```
58.     layer_fc1 = new_fc_layer(input=layer_flat,
59.                              num_inputs=num_features,
60.                              num_outputs=fc_size,
61.                              use_relu=True)
62.     layer_fc2 = new_fc_layer(input=layer_fc1,
63.                              num_inputs=fc_size,
64.                              num_outputs=num_classes,
65.                              use_relu=False)
66.     y_pred = tf.nn.softmax(layer_fc2)
67.     y_pred_cls = tf.argmax(y_pred, axis=1)
68.
69.     session = tf.Session()
70.     session.run(tf.global_variables_initializer())
71.     saver=tf.train.Saver()
72.
73. def evaluer_dessin ():
74.     global matrice,matricefin,listeimage,img
75.     listeimage=[]
76.     for y in matrice:
77.         for x in y:
78.             listeimage.append(x)
79.     img = np.array(listeimage)
80.     afficher_resultat()
81.     fenetre.after(300,evaluer_dessin)
82.
83. def afficher_resultat():
84.     global result,mem,pred
85.     try:
86.         Cadre.delete(result)
87.     except:
88.         pass
89.     pred = session.run(y_pred_cls,feed_dict={ x : [img]})
90.     mem = pred
91.     if evaluerimg:
92.         result=Cadre.create_text(Largeur/2,Hauteur/8,text="c'est un "+str((pred[0]+1
    )%10),font=('', '50'),fill="red")
93.
94. def dessiner(event):
95.     global matrice,evaluerimg
96.     evaluerimg=True
97.     X = event.x
98.     Y = event.y
99.     if X<0:
100.         X=0
101.     if X>Largeur:
102.         X=Largeur
103.     if Y<0:
104.         Y=0
105.     if Y>Hauteur:
106.         Y=Hauteur
107.     try:
108.         matrice[Y*28//Hauteur][X*28//Largeur]=1
109.     except:
110.         pass
111.     Cadre.create_oval(X-TailleRond,Y-
    TailleRond,X+TailleRond,Y+TailleRond,width=0,fill="grey6")
112.
113. def effacer_dessin (t=0):
114.     global matrice,evaluerimg
115.     evaluerimg=False
116.     Cadre.delete(ALL)
117.     matrice=[]
118.     for y in range(28):
119.         ligne = []
120.         for x in range(28):
121.             ligne.append(0)
```

```
122.         matrice.append(ligne)
123.     for x in range(27):
124.         Cadre.create_line(x*20+20,0,x*20+20,560, fill="grey20")
125.         Cadre.create_line(0,x*20+20,560,x*20+20, fill="grey20")
126.     Cadre.create_line(0,4*20+20,560,4*20+20, fill="black",width=4)
127.     Cadre.create_line(0,22*20+20,560,22*20+20, fill="black",width=4)
128.     Cadre.create_line(4*20+20,0,4*20+20,560, fill="black",width=4)
129.     Cadre.create_line(22*20+20,0,22*20+20,560, fill="black",width=4)
130.
131. def compiler():
132.     listeimage=[]
133.     for y in matrice:
134.         for x in y:
135.             listeimage.append(x)
136.     return listeimage
137.
138. def quitter():
139.
140.     fichier = open("dataperso.txt", 'w')
141.     fichier.write(str(len(association)))
142.     fichier.write(",")
143.     fichier.write(str(len(noms)))
144.     fichier.write(",")
145.     for x in association:
146.         fichier.write(x)
147.         fichier.write(",")
148.     for x in noms:
149.         fichier.write(x)
150.         fichier.write(",")
151.     for x in dessins:
152.         for y in x:
153.             fichier.write(str(y))
154.             fichier.write(",")
155.     fichier.close()
156.
157. def recuperer():
158.     global noms,dessins,association,idees,vects
159.     fichier = open("dataperso.txt", 'r')
160.     texte = fichier.read()
161.     try:
162.         texte2=texte.split(",")
163.         texte2.remove(texte2[-1])
164.         nbras=int(texte2[0])
165.         idees=int(texte2[1])
166.         association = texte2[2:nbras+2]
167.         noms = texte2[nbras+2:nbras+idees+2]
168.         texte3=texte2[nbras+idees+2:]
169.
170.         i=0
171.         for z in range(nbras):
172.             lis=[]
173.             for y in range(784):
174.                 try:
175.                     lis.append(float(texte3[i]))
176.                 except:
177.                     lis.append(0.0)
178.                 i+=1
179.             dessins.append(lis)
180.
181.         vects=[]
182.         for name in association:
183.             vect=[]
184.             for a in range(idees):
185.                 if noms.index(name)==a:
186.                     vect.append(1)
187.                 else:
```

```
                    vect.append(0)
                vects.append(vect)
            vects=np.array(vects)

        except:
            print("fichier vide")
        fichier.close()

        liste_nbr_assos=[0,0,0,0,0,0,0,0,0,0]

        for n in range(10):
            for x in association:
                if int(x) ==int(n):
                    liste_nbr_assos[n]+=1
        print(liste_nbr_assos)

    def valider(a=0):
        global noms
        nom=nom_ecrit.get()
        noms.append(nom)
        boutons()

    def boutons():
        couleur1 = "white"
        couleur2 = "SteelBlue4"
        if supprime:
            couleur1 = "red"
            couleur2 = "white"
        Bou.delete(ALL)
        for x in range(len(noms)):
            Bou.create_rectangle((x//8)*70,(x%8)*70,(x//8)*70+70,(x%8)*70+70,fill=coule
    ur1)
            Bou.create_text((x//8)*70+35,(x%8)*70+35,text=str(noms[x]),fill=couleur2)

    def presse_sur_bouton(event):
        global dessins,association,vects,noms,idees
        X = event.x
        Y = event.y
        cx = (X//70)%8
        cy = (Y//70)
        try:
            n = noms[cy+8*cx]
            if supprime:
                copieass = deepcopy(association)
                copiedess = deepcopy(dessins)
                for i in range(len(association)):
                    if association[i]==n:
                        copieass.remove(association[i])
                        copiedess.remove(dessins[i])
                association = deepcopy(copieass)
                dessins = deepcopy(copiedess)
                noms.remove(n)
                boutons()
            else:
                a=compiler()
                dessins.append(a)
                association.append(n)
                effacer_dessin()
            idees=len(noms)
            vects=[]
            for name in association:
                vect=[]
                for a in range(idees):
                    if noms.index(name)==a:
                        vect.append(1)
                    else:
```

```
                    vect.append(0)
                vects.append(vect)
            vects=np.array(vects)
        except:
            print("pas de bouton")

def active_supprimer ():
    global supprime
    if supprime:
        supprime = False
        Suppr.config(text="Supprimer")
    else:
        supprime = True
        Suppr.config(text="  Retour   ")
    boutons()

img_size = 28
img_size_flat = img_size * img_size
num_channels = 1
num_classes = 10
filter_size1 = 5
num_filters1 = 16
filter_size2 = 5
num_filters2 = 36
fc_size = 128
train_batch_size = 64
test_batch_size=100

dessins=[]
association=[]
vects=[]
noms = []
idees=[]
supprime = False
Largeur = 560
Hauteur = 560
TailleRond = 20
vitesse=0.6 #j'ai choisi ça après quelques essais

fenetre = Tk()
fenetre.title("dessiner")
fenetre.configure(background='SteelBlue1')
nom_ecrit= StringVar()

Cadre = Canvas(fenetre,width=Largeur,height=Hauteur,bg ="grey94")
Cadre.bind('<B1-Motion>',dessiner)
Cadre.bind('<Button-1>',dessiner)
Cadre.grid(row=1,rowspan=3,columnspan=6,padx=10,pady=10)

Bou = Canvas(fenetre,width=Largeur,height=Hauteur,bg ='SteelBlue2')
Bou.grid(row=1,column=6,columnspan=6,padx=10,pady=10)
Bou.bind('<Button-1>',presse_sur_bouton)

effacer=Button(fenetre,text="Effacer",command=effacer_dessin)
effacer.grid(row=5,column=3)

Quitte=Button(fenetre,text="Sauvegarder",command=quitter)
Quitte.grid(row=5,column=9)

Suppr =Button(fenetre,text="Supprimer",command=active_supprimer)
Suppr.grid(row=5,column=10)

Champ = Entry(fenetre, textvariable= nom_ecrit, fg='white',bg='SteelBlue3')
Champ.grid(row=5,column=0,columnspan=3)
Champ.bind('<Return>', valider)

```

```
319. Cadre.bind("<Key>",effacer_dessin)
320. Cadre.focus_set()
321. 
322. recuperer()
323. boutons()
324. effacer_dessin()
325. creer_model()
326. try:
327.     saver.restore(session,"weights_entrainepasMNIST2/model.ckpt") #"/tmp/model.ckpt
    "
328.     print("model restauré")
329. except:
330.     pass
331. fenetre.after(2000,evaluer_dessin)
332. fenetre.mainloop()
```

Sources

La majorité de ce que j'ai écrit dans ce travail de maturité a été aquis par expérience personnelle et ne possède donc comme source que lui-même.

Voici la série de vidéos qui m'ont donné envie de faire ce TM :
Première vidéo : https://www.youtube.com/watch?v=aircAruvnKk
Deuxième vidéo : https://www.youtube.com/watch?v=IHZwWFHWa-w
Troisième vidéo : https://www.youtube.com/watch?v=Ilg3gGewQ5U
Quatrième vidéo : https://www.youtube.com/watch?v=tIeHLnjs5U8
Date de consultation : novembre 2017

Voici des vidéos de Hvass-Labs qui m'ont aidées à créer mes programmes avec tensorflow :

La première vidéo commente un tutoriel de tensorflow pour créer un réseau de neurones linéaire classifiant des images de MNIST:
https://www.youtube.com/watch?v=wuo4JdG3SvU
Elle est accompagnée d'un tutoriel partagé sur github : https://github.com/Hvass-Labs/TensorFlow-Tutorials/blob/master/01_Simple_Linear_Model.ipynb
Le tutoriel de tensorflow en question n'existe plus, mais il n'était pas très compréhensible…
Date de consultation : janvier 2018

La deuxième vidéo commente un tutoriel de tensorflow pour créer un réseau de neurones à convolution qui classifie des images de MNIST :
https://www.youtube.com/watch?v=HMcx-zY8JSg
Cette vidéo a également été très utile pour moi dans la compréhension de ce qu'était un réseau de neurones à convolution et son fonctionnement.
Elle est accompagné d'un tutoriel partagé sur github : https://github.com/Hvass-Labs/TensorFlow-Tutorials/blob/master/02_Convolutional_Neural_Network.ipynb
Je l'ai beaucoup servi pour créer le programme en annexe 2.5. J'ai également utilisé les images de ce tutoriel pour illustrer l'explication de ce qu'est un réseau à convolution. Je conseille ce tutoriel à tous ceux qui souhaitent créer un CNN.
Le tutoriel de tensorflow en question n'est plus en ligne.
Date de consultation : mars 2018

Voici la série de 12 vidéos qui m'ont fait comprendre comment créer un réseau de neurone de A à Z. C'est très péadgogoique et donc recommendable pour les débutants : https://www.youtube.com/playlist?list=PLxt59R_fWVzT9bDxA76AHm3ig0Gg9S3So
Date de consultation : février 2018

Voici des vidéos de Thibault Neveu qui m'ont aidées à comprendre le fonctionnement de tensorlow :
https://www.youtube.com/watch?v=O9yl9KKKoQI
https://www.youtube.com/watch?v=YeRBN2fFoIA
Date de consultation : avril 2018

Les bases pour comprendre ce qu'est un réseau de neurone :
https://medium.com/technology-invention-and-more/how-to-build-a-simple-neural-network-in-9-lines-of-python-code-cc8f23647ca1
Date de consultation : janvier 2018

Voici un cours assez complet sur les réseaux de neurones :
http://neuralnetworksanddeeplearning.com/chap1.html
Date de consultation : janvier 2018

C'est le lien pour le tutoriel de reconnaissance d'image fait par tensorflow, mais entre-temps, ce tutoriel a été remplacé par un autre…
https://www.tensorflow.org/get_started/mnist/beginners
Date de consultation : février 2018

Cours servant à comprendre le calcul du gradient :
http://mrmint.fr/gradient-descent-algorithm
Date de consultation : février 2018

Comment créer un réseau de neurones linéaire avec tensorflow pour différencier les images de MNIST :
https://www.oreilly.com/learning/not-another-mnist-tutorial-with-tensorflow
Date de consultation : janvier 2018

Guide utile pour comprendre ce qu'est un réseau de neurones à convolution :
https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/
Date de consultation : mars 2018

Ce site m'a aidé à trouver des réponses aux petits problème de programmation que j'ai pu recontrer, et à installer des modules tels que tensorflow sur mon ordinateur :
https://stackoverflow.com/
Date de consultation : à chaque fois que j'avais une question en programmation → très souvent.

Déclaration

Je déclare par la présente que j'ai réalisé ce travail de manière autonome et que je n'ai utilisé aucun autre moyen que ceux indiqués dans le texte. Tous les passages inspirés ou cités d'autres auteur-es sont dûment mentionnés comme tels. Je suis conscient que de fausses déclarations peuvent conduire le Lycée cantonal à déclarer le travail non recevable et m'exclure de la session d'examens à laquelle je suis inscrit.
Ce travail de maturité reflète mes opinions, il n'engage que moi-même, et non le professeur ni l'expert qui m'ont accompagné dans ce travail.

Lieu et date : …… Signature : …….

Autorisation

Le lycée cantonal requiert votre autorisation afin qu'un exemplaire de votre Travail de maturité soit mis à la disposition des étudiants du Lycée cantonal, par le biais de la médiathèque de l'école.

☐ Oui, j'accepte que mon Travail de maturité soit mis à la disposition des étudiants du Lycée cantonal.

☐ Non, je refuse que mon Travail de maturité soit mis à la disposition des étudiants du Lycée cantonal.

Lieu et date : …… Signature : …….