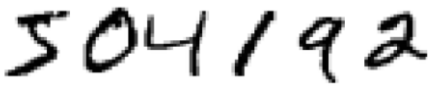
Atelier : Introduction aux réseaux de neurones

Le système visuel humain est l'une des merveilles du monde. Considérez la séquence suivante de chiffres manuscrits :



La plupart des gens reconnaissent sans effort ces chiffres comme 504192. Cette facilité est trompeuse. Dans chaque hémisphère de notre cerveau, les humains ont un cortex visuel primaire, également connu sous le nom de 1, contenant 140 millions de neurones, avec des dizaines de milliards de connexions entre eux. Et pourtant, la vision humaine n'implique pas seulement $V1$, mais toute une série de cortex visuels - $V2$,$V3$,$V4$ et $V5$ - réalisant un traitement d'image de plus en plus complexe. Nous portons dans nos têtes un superordinateur, réglé par l'évolution sur des centaines de millions d'années, et superbement adapté pour comprendre le monde visuel. Reconnaître les chiffres manuscrits n'est pas facile. Au contraire, nous, les humains, sommes prodigieusement, étonnamment doués pour donner un sens à ce que nos yeux nous montrent. Mais presque tout ce travail est fait inconsciemment. Et donc nous ne comprenons généralement pas à quel point nos systèmes visuels résolvent un problème difficile. La difficulté de la reconnaissance visuelle des formes devient évidente si vous essayez d'écrire un programme informatique pour reconnaître des chiffres comme ceux ci-dessus. Ce qui semble facile quand nous le faisons nous-mêmes devient soudain extrêmement difficile. Des intuitions simples sur la façon dont nous reconnaissons les formes - "un 9 a une boucle en haut et un trait vertical en bas à droite" - s'avèrent pas si simples à exprimer algorithmiquement. Lorsque vous essayez de préciser de telles règles, vous vous perdez rapidement dans un bourbier d'exceptions, de mises en garde et de cas particuliers. Cela semble désespéré. Les réseaux de neurones abordent le problème d'une manière différente. L'idée est de prendre un grand nombre de chiffres manuscrits, appelés exemples d'apprentissage :

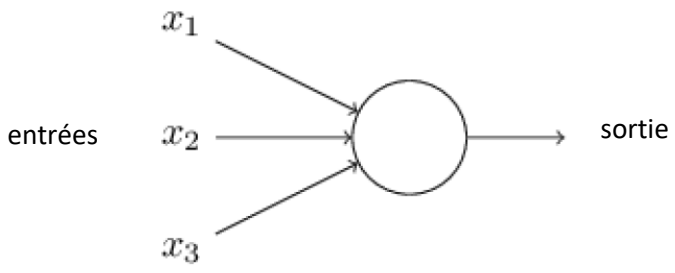
puis développer un système qui peut apprendre de ces exemples de formation. En d'autres termes, le réseau de neurones utilise les exemples pour déduire automatiquement des règles de reconnaissance des chiffres manuscrits. De plus, en augmentant le nombre d'exemples de formation, le réseau peut en apprendre davantage sur l'écriture manuscrite et ainsi améliorer sa précision. Ainsi, bien que je n'aie montré que 100 chiffres d'apprentissage ci-dessus, nous pourrions peut-être créer un meilleur outil de reconnaissance d'écriture manuscrite en utilisant des milliers, voire des millions ou des milliards d'exemples d'apprentissage. Dans ce chapitre, nous allons écrire un programme informatique mettant en œuvre un réseau de neurones qui apprend à reconnaître les chiffres manuscrits. Le programme ne fait que 74 lignes et n'utilise aucune bibliothèque spéciale de réseaux neuronaux. Mais ce programme court peut reconnaître les chiffres avec une précision de plus de 96 %, sans intervention humaine. De plus, dans les chapitres suivants, nous développerons des idées qui peuvent améliorer la précision à plus de 99 %. En fait, les meilleurs réseaux de neurones commerciaux sont maintenant si

performants qu'ils sont utilisés par les banques pour traiter les chèques et par les bureaux de poste pour reconnaître les adresses. Nous nous concentrons sur la reconnaissance de l'écriture manuscrite car c'est un excellent problème prototype pour l'apprentissage des réseaux de neurones en général. En tant que prototype, il atteint un point idéal : c'est un défi - ce n'est pas une mince affaire de reconnaître des chiffres manuscrits - mais ce n'est pas si difficile que d'exiger une solution extrêmement compliquée ou une énorme puissance de calcul. De plus, c'est un excellent moyen de développer des techniques plus avancées, telles que l'apprentissage en profondeur. Ainsi, tout au long du livre, nous reviendrons à plusieurs reprises sur le problème de la reconnaissance de l'écriture manuscrite. Plus loin dans le livre, nous discuterons de la manière dont ces idées peuvent être appliquées à d'autres problèmes de vision par ordinateur, ainsi qu'à la parole, au traitement du langage naturel et à d'autres domaines.

Bien sûr, si le but du chapitre était seulement d'écrire un programme informatique pour reconnaître les chiffres manuscrits, alors le chapitre serait beaucoup plus court ! Mais en cours de route, nous développerons de nombreuses idées clés sur les réseaux de neurones, y compris deux types importants de neurones artificiels (le perceptron et le neurone sigmoïde) et l'algorithme d'apprentissage standard pour les réseaux de neurones, connu sous le nom de descente de gradient stochastique. Tout au long, je me concentre sur l'explication des raisons pour lesquelles les choses sont faites comme elles le sont et sur la construction de votre intuition de réseaux neuronaux. Cela nécessite une discussion plus longue que si je présentais simplement les mécanismes de base de ce qui se passe, mais cela en vaut la peine pour la compréhension plus profonde que vous atteindrez. Parmi les avantages, à la fin du chapitre, nous serons en mesure de comprendre ce qu'est l'apprentissage en profondeur et pourquoi c'est important.

Perceptrons

Qu'est-ce qu'un réseau de neurones ? Pour commencer, je vais expliquer un type de neurone artificiel appelé perceptron. Les perceptrons ont été développés dans les années 1950 et 1960 par le scientifique Frank Rosenblatt, inspiré des travaux antérieurs de Warren McCulloch et Walter Pitts. Aujourd'hui, il est plus courant d'utiliser d'autres modèles de neurones artificiels, et dans de nombreux travaux modernes sur les réseaux de neurones, le principal modèle de neurone utilisé est celui appelé neurone sigmoïde. Nous allons bientôt aborder les neurones sigmoïdes. Mais pour comprendre pourquoi les neurones sigmoïdes sont définis comme ils le sont, cela vaut la peine de prendre le temps de comprendre d'abord les perceptrons. Alors comment fonctionnent les perceptrons ? Un perceptron prend plusieurs entrées binaires, $x_1, x_2, \ldots$ et produit une seule sortie binaire :



Dans l'exemple illustré, le perceptron a trois entrées, x_1, x_2, x_3 En général, il pourrait avoir plus ou moins d'entrées. Rosenblatt a proposé une règle simple pour calculer la sortie. Il a introduit des poids, $w_1, w_2, \ldots$ des nombres réels exprimant l'importance des entrées respectives dans la sortie. La sortie du neurone, 0 ou 1, est déterminée par le fait que la somme pondérée $\sum w_i x_i$est inférieure ou supérieure à une valeur seuil. Tout comme les poids, le seuil est un nombre réel qui est un paramètre du neurone. Pour le dire en termes algébriques plus précis :

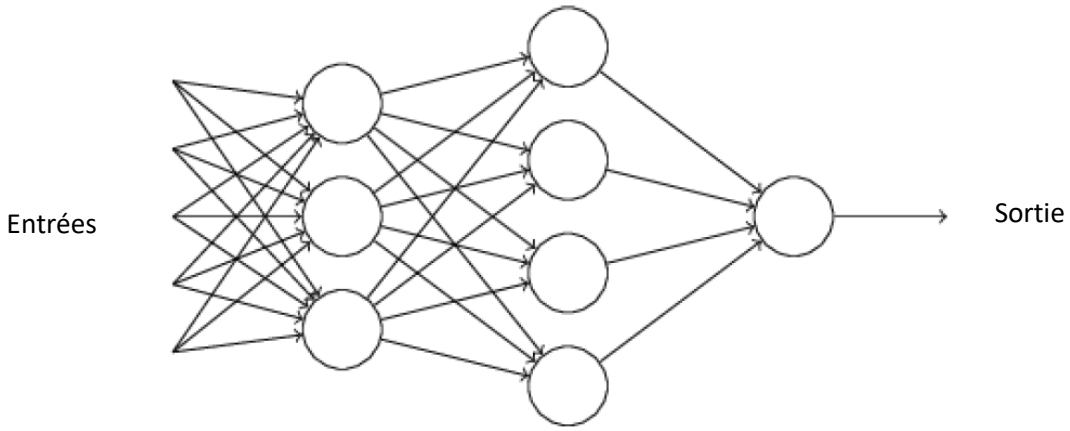
$$sortie = \begin{cases} 0\ si\ \sum w_i x_i \leq seuil \\ 1\ si\ \sum w_i x_i > seuil \end{cases}$$

Voilà comment fonctionne un perceptron ! C'est le modèle mathématique de base. Une façon de penser au perceptron est que c'est un appareil qui prend des décisions en pesant des preuves. Laissez-moi vous donner un exemple. Ce n'est pas un exemple très réaliste, mais c'est facile à comprendre, et nous arriverons bientôt à des exemples plus réalistes. Supposons que le week-end approche et que vous ayez entendu dire qu'il allait y avoir un festival du fromage dans votre ville. Vous aimez le fromage et vous décidez d'aller ou non au festival. Vous pouvez prendre votre décision en pesant trois facteurs :

1- Le temps est-il clément ?
2- Votre copain ou copine souhaite vous accompagner ?
3- Le festival est-il à proximité des transports en commun ? (Vous ne possédez pas de voiture).

Nous pouvons représenter ces trois facteurs par les variables binaires correspondantes x_1, x_2, x_3. Par exemple, nous aurions $x_1 = 1$ s'il fait beau, et $x_1 = 0$ s'il fait mauvais. De même, $x_2 = 1$ si votre petit ami ou votre petite amie veut y aller, et $x_2 = 0$ sinon. Et de même encore pour $x_3 = 1$ et les transports en commun. Maintenant, supposons que vous adorez le fromage, à tel point que vous êtes heureux d'aller au festival même si votre petit ami ou votre petite amie n'est pas intéressé et que le festival est difficile d'accès. Mais peut-être que vous détestez vraiment le mauvais temps, et il n'y a aucune chance que vous alliez au festival s'il fait mauvais. Vous pouvez utiliser des perceptrons pour modéliser ce type de prise de décision. Une façon de le faire est de choisir un poids $w_1 = 6$ pour la météo, et $w_2 = 2$ et $w_3 = 2$ pour les autres conditions. La plus grande valeur de 1 indique que la météo compte beaucoup pour vous, bien plus que si votre petit ami ou votre petite amie vous rejoint, ou la proximité des transports en commun. Enfin, supposons que vous choisissiez un seuil de 5 pour le perceptron. Avec ces choix, le perceptron met en œuvre le modèle décisionnel souhaité,

sortie 1 chaque fois que le temps est bon, et 0 chaque fois que le temps est mauvais. Peu importe que votre petit ami ou votre petite amie veuille y aller ou que les transports en commun soient à proximité. En faisant varier les poids et le seuil, on peut obtenir différents modèles de prise de décision. Par exemple, supposons que nous choisissions plutôt un seuil de 3. Ensuite, le perceptron déciderait que vous devriez aller au festival chaque fois qu'il fait beau ou lorsque le festival est à proximité des transports en commun et que votre petit ami ou votre petite amie est prêt à vous rejoindre. En d'autres termes, ce serait un modèle différent de prise de décision. Laisser tomber le seuil signifie que vous êtes plus disposé à aller au festival. Évidemment, le perceptron n'est pas un modèle complet de prise de décision humaine ! Mais ce que l'exemple illustre, c'est comment un perceptron peut peser différents types de preuves afin de prendre des décisions. Et il devrait sembler plausible qu'un réseau complexe de perceptrons puisse prendre des décisions assez subtiles :

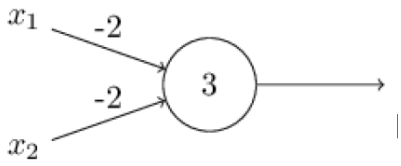


Dans ce réseau, la première colonne de perceptrons - ce que nous appellerons la première couche de perceptrons - prend trois décisions très simples, en pesant les preuves d'entrée. Qu'en est-il des perceptrons de la deuxième couche ? Chacun de ces perceptrons prend une décision en pesant les résultats de la première couche de prise de décision. De cette façon, un perceptron de la deuxième couche peut prendre une décision à un niveau plus complexe et plus abstrait que les perceptrons de la première couche. Et des décisions encore plus complexes peuvent être prises par le perceptron de la troisième couche. De cette façon, un réseau de perceptrons à plusieurs couches peut s'engager dans une prise de décision sophistiquée. Incidemment, lorsque j'ai défini les perceptrons, j'ai dit qu'un perceptron n'avait qu'une seule sortie. Dans le réseau au-dessus, les perceptrons ressemblent à

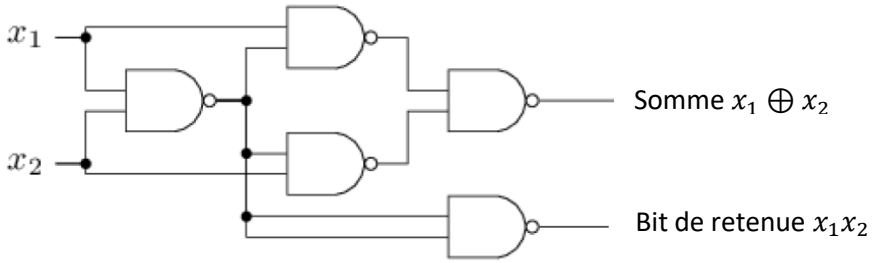
ils ont plusieurs sorties. En fait, ils sont toujours à sortie unique. Les flèches de sortie multiples sont simplement un moyen utile d'indiquer que la sortie d'un perceptron est utilisée comme entrée de plusieurs autres perceptrons. C'est moins lourd que de dessiner une seule ligne de sortie qui se divise ensuite. Simplifions la façon dont nous décrivons les perceptrons. La condition $\Sigma wjxjj{>}threshold$ est lourde, et on peut faire deux changements de notation pour la simplifier. Le premier changement consiste à écrire $\Sigma wjxjj$ sous la forme d'un produit scalaire, $\langle w|x\rangle = \sum w_i x_i$, où w et x sont des vecteurs dont les composants sont respectivement les poids et les entrées. Le deuxième changement consiste à déplacer le seuil de l'autre côté de l'inégalité et à le remplacer par ce que l'on appelle le biais du perceptron, $b = -seuil$. En utilisant le biais au lieu du seuil, la règle du perceptron peut être réécrite :

$$sortie = \begin{cases} 0 \; si \; \langle w|x\rangle + b \leq 0 \\ 1 \; si \; \langle w|x\rangle + b > 0 \end{cases}$$

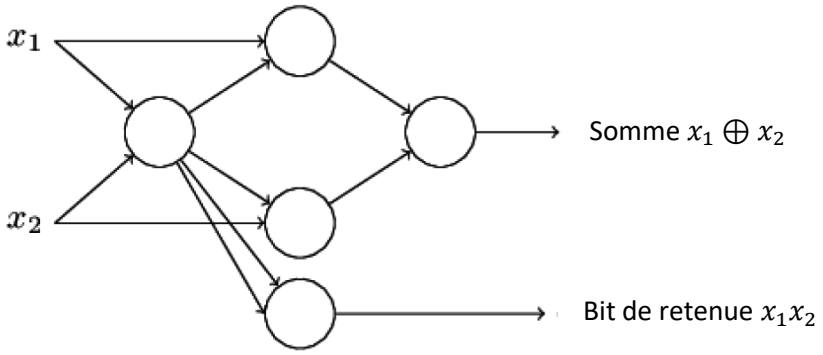
Vous pouvez considérer le biais comme une mesure de la facilité avec laquelle le perceptron produit un 1. Ou, pour le dire en termes plus biologiques, le biais est une mesure de la facilité avec laquelle le perceptron se déclenche. Pour un perceptron avec un très gros biais, il est extrêmement facile pour le perceptron de produire un 1. Mais si le biais est très négatif, il est alors difficile pour le perceptron de produire un 1. Évidemment, l'introduction du biais n'est qu'un petit changement dans comment nous décrivons les perceptrons, mais nous verrons plus tard que cela conduit à d'autres simplifications notationnelles. Pour cette raison, nous n'utiliserons pas le seuil, nous utiliserons toujours le biais. J'ai décrit les perceptrons comme une méthode d'évaluation des preuves pour prendre des décisions. Une autre façon d'utiliser les perceptrons est de calculer les fonctions logiques élémentaires que nous considérons généralement comme le calcul sous-jacent, des fonctions telles que AND, OR et NOR. Par exemple, supposons que nous ayons un perceptron avec deux entrées, chacune avec un poids -2, et un biais global de 3. Voici notre perceptron :



On voit alors que l'entrée 00 produit la sortie 1, puisque (−2)∗0+(−2)∗0+3=3 est positif. Des calculs similaires montrent que les entrées 01 et 10 produisent la sortie 1. Mais l'entrée 11 produit la sortie 0, puisque (−2)∗1+(−2)∗1+3=−1 est négatif. Et donc notre perceptron implémente une porte $NAND$! L'exemple $NAND$ montre que nous pouvons utiliser des perceptrons pour calculer des fonctions logiques simples. En fait, nous pouvons utiliser des réseaux de perceptrons pour calculer n'importe quelle fonction logique. La raison en est que la porte $NAND$ est universelle pour le calcul, c'est-à-dire que nous pouvons construire n'importe quel calcul à partir de portes $NAND$. Par exemple, nous pouvons utiliser des portes $NAND$ pour construire un circuit qui ajoute deux bits, $x1$ et $x2$. Cela nécessite de calculer la somme au niveau du bit, $x_1 \oplus x_2$, ainsi qu'un bit de retenue qui est mis à 1 lorsque $x_1$et x_2 sont tous deux égaux à 1, c'est-à-dire que le bit de retenue n'est que le produit au niveau du bit x_1x_2 :

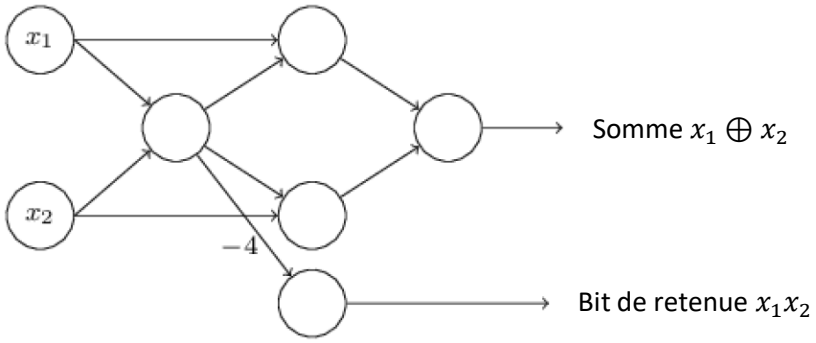


Pour obtenir un réseau de perceptrons équivalent, nous remplaçons toutes les portes $NAND$ par des perceptrons à deux entrées, chacune avec un poids -2 et un biais global de 3. Voici le réseau résultant.



Un aspect notable de ce réseau de perceptrons est que la sortie du perceptron le plus à gauche est utilisée deux fois comme entrée du perceptron le plus bas. Lorsque j'ai défini le modèle perceptron, je n'ai pas dit si ce genre de double sortie au même endroit était autorisé. En fait, cela n'a pas beaucoup d'importance. Si nous ne voulons pas autoriser ce genre de chose, alors il est possible de simplement fusionner les deux lignes, en une seule connexion avec un poids de -4 au lieu de deux connexions avec des poids de -2. (Si vous ne trouvez pas cela évident, vous devriez vous arrêter et vous prouver que

c'est équivalent.) Avec ce changement, le réseau se présente comme suit, avec tous les poids non marqués égaux à −2 , tous les biais égaux à 3, et un poids unique de -4, comme indiqué :



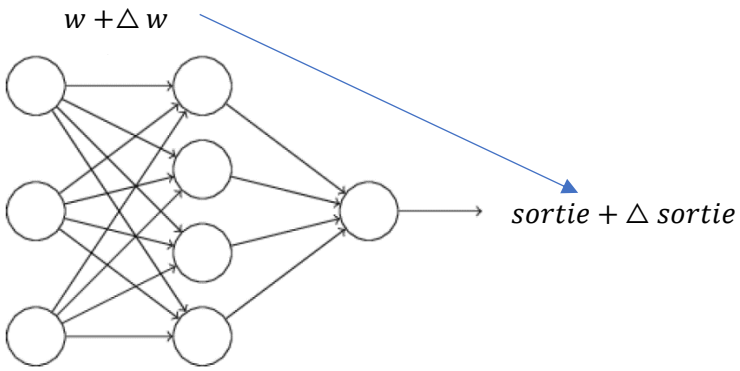
Cette notation pour les perceptrons d'entrée, dans laquelle nous avons une sortie, mais pas d'entrées, est un raccourci

Cela ne signifie pas réellement un perceptron sans entrées. Pour voir cela, supposons que nous ayons un perceptron sans entrées. Alors la somme pondérée $\sum w_i x_i$ serait toujours nulle, et donc le perceptron produirait 1 si $b > 0$, et 0 si $b \leq 0$. C'est-à-dire que le perceptron produirait simplement une valeur fixe, pas la valeur souhaitée (x_1, dans l'exemple ci-dessus). Il est préférable de considérer les perceptrons d'entrée comme n'étant pas du tout des perceptrons, mais plutôt des unités spéciales qui sont simplement définies pour sortir les valeurs souhaitées, $x_1, x_2, \ldots$ L'exemple de l'additionneur montre comment un réseau de perceptrons peut être utilisé pour simuler un circuit contenant de nombreuses portes. Et parce que les portes sont universelles pour le calcul, il s'ensuit que les perceptrons sont également universels pour le calcul. L'universalité computationnelle des perceptrons est à la fois rassurante et décevante. C'est rassurant car cela nous dit que les réseaux de perceptrons peuvent être aussi puissants que n'importe quel autre appareil informatique. Mais c'est aussi décevant, car cela donne l'impression que les perceptrons ne sont qu'un nouveau type de porte $NAND$. Ce n'est pas une grande nouvelle ! Cependant, la situation est meilleure que ce point de vue ne le suggère. Il s'avère que nous pouvons concevoir des algorithmes d'apprentissage qui peuvent ajuster automatiquement les poids et les biais d'un réseau de neurones artificiels. Ce réglage se produit en réponse à des stimuli externes, sans intervention directe d'un programmeur. Ces algorithmes d'apprentissage nous permettent d'utiliser des neurones artificiels d'une manière radicalement différente des portes logiques classiques. Au lieu de disposer explicitement un circuit de et d'autres portes, nos réseaux de neurones peuvent simplement apprendre à résoudre des problèmes, parfois des problèmes où il serait extrêmement difficile de concevoir directement un circuit conventionnel.

Neurones sigmoïdes

Les algorithmes d'apprentissage semblent formidables. Mais comment concevoir de tels algorithmes pour un réseau de neurones ? Supposons que nous ayons un réseau de perceptrons que nous aimerions utiliser pour apprendre à résoudre un problème. Par exemple, les entrées du réseau peuvent être les données de pixels brutes d'une image numérisée et manuscrite d'un chiffre. Et nous aimerions que le réseau apprenne les poids et les biais afin que la sortie du réseau classe correctement le chiffre. Pour voir comment l'apprentissage pourrait fonctionner, supposons que nous apportions un petit changement dans un certain poids (ou biais) dans le réseau. Ce que nous aimerions, c'est que ce petit changement de poids n'entraîne qu'un petit changement correspondant dans la sortie du réseau. Comme nous le verrons dans un instant, cette propriété rendra l'apprentissage possible.

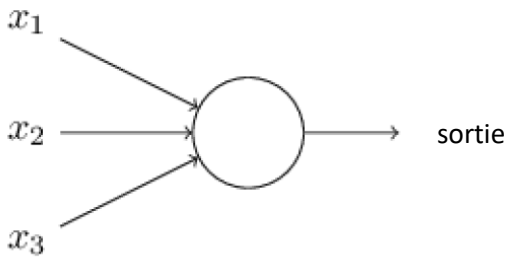
Schématiquement, voici ce que nous voulons (évidemment ce réseau est trop simple pour faire de la reconnaissance d'écriture manuscrite !) :



Un petit changement dans les poids ou les biais

entrainent un petit changement à la sortie

S'il était vrai qu'un petit changement dans un poids (ou un biais) ne provoque qu'un petit changement dans la sortie, alors nous pourrions utiliser ce fait pour modifier les poids et les biais afin que notre réseau se comporte davantage comme nous le souhaitons. Par exemple, supposons que le réseau classe par erreur une image comme un "8" alors qu'elle devrait être un "9". image comme un "9". Et puis nous répétions cela, en changeant les poids et les biais encore et encore pour produire de mieux en mieux. Le réseau apprendrait. Le problème est que ce n'est pas ce qui se passe lorsque notre réseau contient des perceptrons. En fait, un petit changement dans les poids ou le biais d'un seul perceptron dans le réseau peut parfois faire basculer complètement la sortie de ce perceptron, disons de 0 à 1. Ce basculement peut alors faire basculer le comportement du reste du réseau. changer complètement d'une manière très compliquée. Ainsi, même si votre "9" peut maintenant être classé correctement, le comportement du réseau sur toutes les autres images est susceptible d'avoir complètement changé d'une manière difficile à contrôler. Cela rend difficile de voir comment modifier progressivement les poids et les biais pour que le réseau se rapproche du comportement souhaité. Il existe peut-être un moyen astucieux de contourner ce problème. Mais il n'est pas immédiatement évident de savoir comment faire apprendre un réseau de perceptrons. Nous pouvons surmonter ce problème en introduisant un nouveau type de neurone artificiel appelé neurone sigmoïde. Les neurones sigmoïdes sont similaires aux perceptrons, mais modifiés de sorte que de petits changements dans leur poids et leur biais ne provoquent qu'un petit changement dans leur sortie. C'est le fait crucial qui permettra à un réseau de neurones sigmoïdes d'apprendre. Nous allons représenter les neurones sigmoïdes de la même manière que nous avons représenté les perceptrons :



Tout comme un perceptron, le neurone sigmoïde a des entrées, x_1, x_2, … mais au lieu d'être juste 0 ou 1, ces entrées peuvent également prendre n'importe quelle valeur entre 0 et 1. Ainsi, par exemple, $0{,}368$ … est une entrée valide pour un neurone sigmoïde. Tout comme un perceptron, le neurone

sigmoïde a des poids pour chaque entrée, w_1, w_2, ...et un biais global, b. Mais la sortie n'est pas 0 ou 1. Au lieu de cela, c'est $\sigma(w \cdot x + b)$, où σ est appelée la fonction sigmoïde $(*)$, et est définie par :

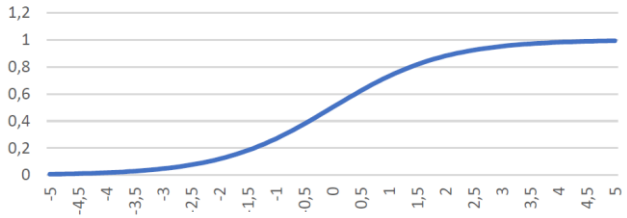
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Pour le dire un peu plus explicitement, la sortie d'un neurone sigmoïde avec des entrées x_1, x_2, ..., des poids w_1, w_2, ... et le biais b est :

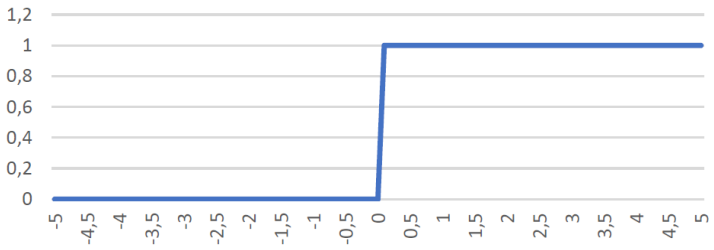
$$\frac{1}{1 + e^{(-\sum w_i x_i - b)}}$$

A première vue, les neurones sigmoïdes apparaissent très différents des perceptrons. La forme algébrique de la fonction sigmoïde peut sembler opaque et rébarbative si vous ne la connaissez pas déjà. En fait, il existe de nombreuses similitudes entre les perceptrons et les neurones sigmoïdes, et la forme algébrique de la fonction sigmoïde s'avère être plus un détail technique qu'un véritable obstacle à la compréhension.

Pour comprendre la similitude avec le modèle perceptron, supposons que $z \equiv w \cdot x + b$ est un grand nombre positif. Alors $e^{-z} \approx 0$ et donc $\sigma(z) \approx 1$. En d'autres termes, lorsque $z \equiv w \cdot x + b$ est grand et positif, la sortie du neurone sigmoïde est d'environ 1, tout comme elle l'aurait été pour un perceptron. Supposons par contre que $z \equiv w \cdot x + b$ soit très négatif. Alors $e^{-z} \longrightarrow +\infty$, et $\sigma(z) \approx 0$. Ainsi, lorsque $z \equiv w \cdot x + b$ est très négatif, le comportement d'un neurone sigmoïde se rapproche également étroitement d'un perceptron. Ce n'est que lorsque $w \cdot x + b$ est petit qu'il y a beaucoup d'écart par rapport au modèle perceptron. Qu'en est-il de la forme algébrique de ? Comment pouvons-nous comprendre cela ? En fait, la forme exacte de σ n'est pas si importante - ce qui compte vraiment, c'est la forme de la fonction lorsqu'elle est tracée :



Cette forme est une version lissée d'une Fonction sigmoïde



Fonction marche

Si σ avait en fait été une fonction échelonnée, alors le neurone sigmoïde serait un perceptron, puisque la sortie serait 1 ou 0 selon que $w \cdot x + b$ était positif ou négatif.

En utilisant la fonction σ réelle, nous obtenons, comme déjà sous-entendu ci-dessus, un perceptron lissé. En effet, c'est **la régularité de la fonction** qui est **le fait crucial**, pas sa forme détaillée. La régularité de σ signifie que de petits changements Δw_i dans les poids et dans le biais produiront un

petit changement dans la sortie du neurone. En fait, le calcul nous dit que $\Delta sortie$ est bien approché par

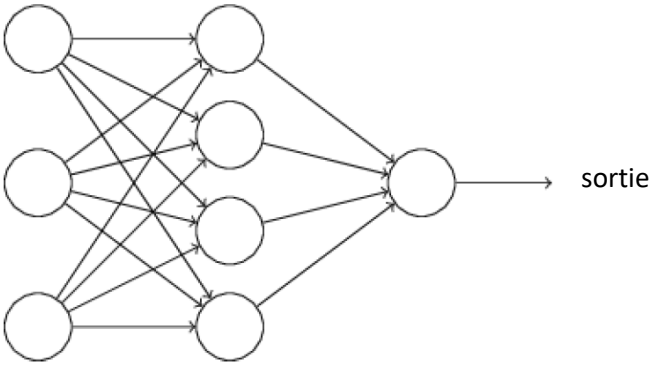
$$\Delta sortie = \sum \frac{\partial sortie}{\partial w_i} \Delta w_i + \frac{\partial sortie}{\partial b} \Delta b$$

où la somme est sur tous les poids, , et $\frac{\partial sortie}{\partial w_i}$ et $\frac{\partial sortie}{\partial b}$désignent les dérivées partielles de la sortie par rapport à w_i et b, respectivement. Si c'est la forme de σ qui compte vraiment, et non sa forme exacte, alors pourquoi utiliser la forme particulière utilisée pour σ dans l'équation? En fait, nous considérerons occasionnellement les neurones dont la sortie est $f(w \cdot x + b)$ pour une autre fonction d'activation $f(\cdot)$. La principale chose qui change lorsque nous utilisons une fonction d'activation différente est que les valeurs particulières des dérivées partielles dans l'équation ci-dessus changent. Il s'avère que lorsque nous calculons ces dérivées partielles plus tard, l'utilisation de σ simplifiera l'algèbre, simplement parce que les exponentielles ont de belles propriétés lorsqu'elles sont différenciées.

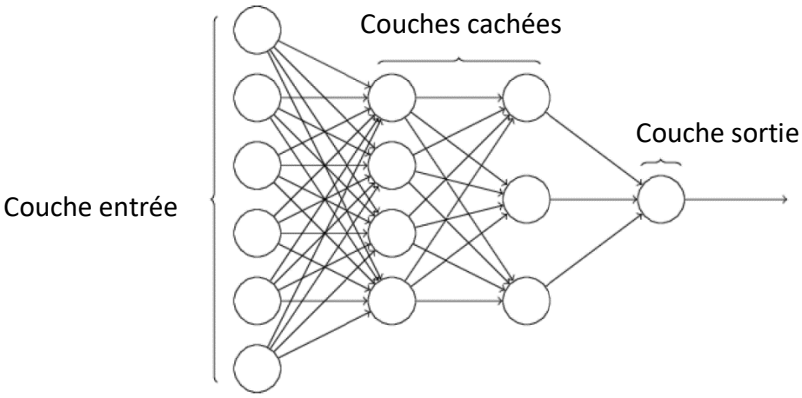
Comment doit-on interpréter la sortie d'un neurone sigmoïde ? De toute évidence, une grande différence entre les perceptrons et les neurones sigmoïdes est que les neurones sigmoïdes ne produisent pas seulement 0 ou 1. Ils peuvent avoir comme sortie n'importe quel nombre réel entre 0 et 1, donc des valeurs telles que $0{,}173\ldots$ et $0{,}689\ldots$ sont des sorties légitimes. Cela peut être utile, par exemple, si nous voulons utiliser la valeur de sortie pour représenter l'intensité moyenne des pixels dans une image entrée dans un réseau de neurones. Mais parfois, cela peut être une nuisance. Supposons que nous voulions que la sortie du réseau indique soit "l'image d'entrée est un 9" ou "l'image d'entrée n'est pas un 9". Évidemment, il serait plus facile de le faire si la sortie était un 0 ou un 1, comme dans un perceptron. Mais en pratique, nous pouvons mettre en place une convention pour gérer cela, par exemple, en décidant d'interpréter toute sortie d'au moins $0{,}5$ comme indiquant un "9", et toute sortie inférieure à $0{,}5$ comme indiquant "pas un 9". J'indiquerai toujours explicitement quand nous utilisons une telle convention, donc cela ne devrait pas causer de confusion.

L'architecture des réseaux de neurones

Dans la section suivante, je présenterai un réseau de neurones qui peut faire un très bon travail de classification des chiffres manuscrits. En prévision de cela, il est utile d'expliquer une certaine terminologie qui nous permet de nommer différentes parties d'un réseau. Supposons que nous ayons le réseau :



Comme mentionné précédemment, la couche la plus à gauche de ce réseau est appelée couche d'entrée et les neurones de la couche sont appelés neurones d'entrée. La couche la plus à droite ou de sortie contient les neurones de sortie, ou, comme dans ce cas, un seul neurone de sortie. La couche intermédiaire est appelée couche cachée, car les neurones de cette couche ne sont ni des entrées ni des sorties. Par exemple, le réseau à quatre couches suivant a deux couches cachées :

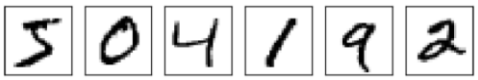


La conception des couches d'entrée et de sortie dans un réseau est souvent simple. Par exemple, supposons que nous essayions de déterminer si une image manuscrite représente un « 9 » ou non. Une façon naturelle de concevoir le réseau consiste à coder les intensités des pixels de l'image dans les neurones d'entrée. Si l'image est une image en niveaux de gris de 64 x 64, alors nous aurions 4096 = 64 × 64 neurones d'entrée, avec les intensités mises à l'échelle de manière appropriée entre 0 et 1. La couche de sortie contiendra un seul neurone, avec des valeurs de sortie inférieures à 0,5 indiquant "l'image d'entrée n'est pas un 9", et les valeurs supérieures à 0,5 indiquant "l'image d'entrée est un 9". Alors que la conception des couches d'entrée et de sortie d'un réseau de neurones est souvent simple, la conception des couches cachées peut être tout un art. En particulier, il n'est pas possible de résumer le processus de conception des couches cachées avec quelques règles simples. Au lieu de cela, les chercheurs en réseaux de neurones ont développé de nombreuses heuristiques de conception pour les couches cachées, qui aident les gens à obtenir le comportement qu'ils souhaitent de leurs réseaux. Par exemple, de telles heuristiques peuvent être utilisées pour aider à déterminer comment échanger le nombre de couches cachées par rapport au temps nécessaire pour former le réseau. Nous rencontrerons plusieurs de ces heuristiques de conception plus loin dans ce livre. Jusqu'à présent, nous avons discuté des réseaux de neurones où la sortie d'une couche est utilisée comme entrée de la couche suivante. De tels réseaux sont appelés réseaux de neurones feedforward. Cela signifie qu'il n'y a pas de boucles dans le réseau - les informations sont toujours transmises en avant, jamais en retour. Si nous avions des boucles, nous nous retrouverions dans des situations où l'entrée de la fonction dépendrait de la sortie. Ce serait difficile à comprendre, et nous n'autorisons donc pas de telles boucles. Cependant, il existe d'autres modèles de réseaux de neurones artificiels dans lesquels des boucles de rétroaction sont possibles. Ces modèles sont appelés réseaux de neurones récurrents. L'idée dans ces modèles est d'avoir des neurones qui s'activent pendant une durée limitée, avant de devenir inactifs. Ce déclenchement peut stimuler d'autres neurones, qui peuvent se déclencher un peu plus tard, également pour une durée limitée. Cela provoque l'activation d'encore plus de neurones et, avec le temps, nous obtenons une cascade de neurones. Les boucles ne posent pas de problèmes dans un tel modèle, car la sortie d'un neurone n'affecte son entrée qu'à un moment ultérieur, pas instantanément. Les réseaux neuronaux récurrents ont eu moins d'influence que les réseaux à action directe, en partie parce que les algorithmes d'apprentissage pour les réseaux récurrents sont (au moins à ce jour) moins puissants. Mais les réseaux récurrents restent extrêmement intéressants. Ils sont beaucoup plus proches dans l'esprit du fonctionnement de notre cerveau que les réseaux feedforward. Et il est possible que les réseaux récurrents puissent résoudre des problèmes importants qui ne peuvent être résolus qu'avec de grandes difficultés par des réseaux à action directe. Cependant, pour limiter notre champ d'application, dans ce livre, nous allons nous concentrer sur les réseaux feedforward les plus largement utilisés.

Un réseau simple pour classer les chiffres manuscrits

Après avoir défini les réseaux de neurones, revenons à la reconnaissance de l'écriture manuscrite. Nous pouvons diviser le problème de la reconnaissance des chiffres manuscrits en deux sous-problèmes. Premièrement, nous aimerions un moyen de diviser une image contenant de nombreux chiffres en une séquence d'images distinctes, chacune contenant un seul chiffre. Par exemple, nous aimerions diviser l'image

en six images distinctes,

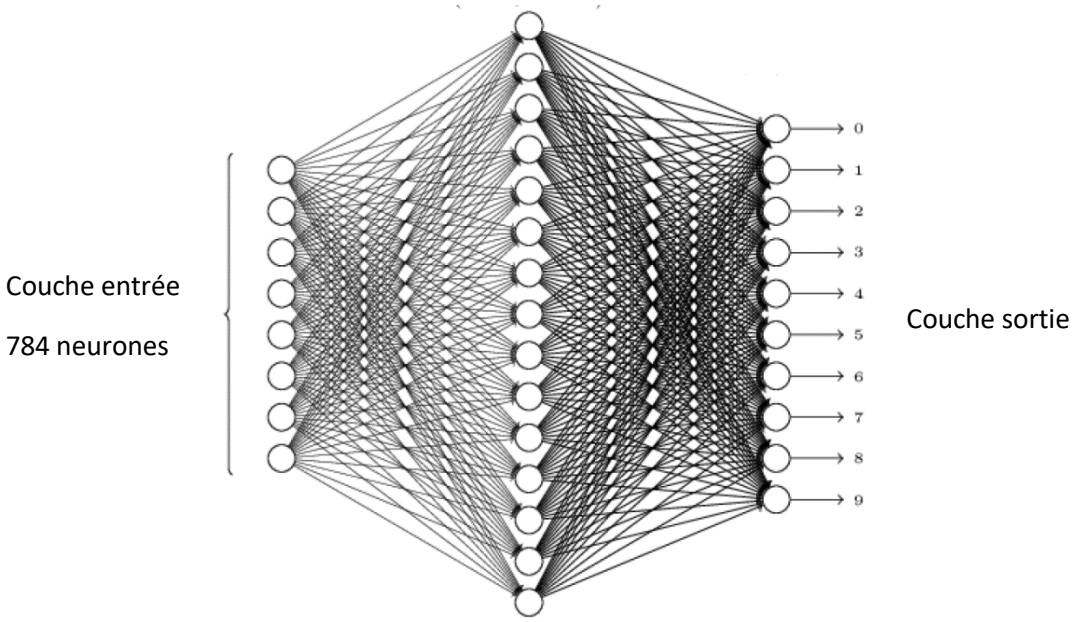


Nous, les humains, résolvons facilement ce problème de segmentation, mais il est difficile pour un programme informatique de décomposer correctement l'image. Une fois que l'image a été segmentée, le programme doit alors classer chaque chiffre individuel. Ainsi, par exemple, nous aimerions que notre programme reconnaisse que le premier chiffre ci-dessus,

Est un 5.

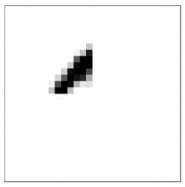
Nous allons nous concentrer sur l'écriture d'un programme pour résoudre le deuxième problème, c'est-à-dire la classification des chiffres individuels. Nous faisons cela parce qu'il s'avère que le problème de segmentation n'est pas si difficile à résoudre, une fois que vous avez un bon moyen de classer les chiffres individuels. Il existe de nombreuses approches pour résoudre le problème de segmentation. Une approche consiste à tester de nombreuses manières différentes de segmenter l'image, en utilisant le classificateur à chiffres individuels pour noter chaque segmentation d'essai. Une segmentation d'essai obtient un score élevé si le classificateur à chiffres individuel est sûr de sa classification dans tous les segments, et un score faible si le classificateur a beaucoup de problèmes dans un ou plusieurs segments. L'idée est que si le classificateur a des problèmes quelque part, alors il a probablement des problèmes parce que la segmentation a été mal choisie. Cette idée et d'autres variantes peuvent être utilisées pour résoudre assez bien le problème de segmentation. Ainsi, au lieu de nous soucier de la segmentation, nous allons nous concentrer sur le développement d'un réseau de neurones qui peut résoudre le problème le plus intéressant et le plus difficile, à savoir la reconnaissance des chiffres manuscrits individuels. Pour reconnaître les chiffres individuels, nous utiliserons un réseau de neurones à trois couches :

Couche cachée 15 neurones

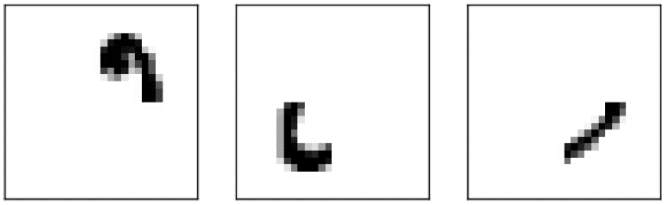


La couche d'entrée du réseau contient des neurones codant les valeurs des pixels d'entrée. Comme indiqué dans la section suivante, nos données d'entraînement pour le réseau seront constituées de nombreuses images de 28 x 28 pixels de chiffres manuscrits numérisés, et la couche d'entrée contient donc 784 = 28 x 28 neurones. Pour plus de simplicité, j'ai omis la plupart des 784 neurones d'entrée dans le diagramme ci-dessus. Les pixels d'entrée sont en niveaux de gris, avec une valeur de 0,0 représentant le blanc, une valeur de 1,0 représentant le noir, et entre les valeurs représentant des nuances de gris qui s'assombrissent progressivement. La deuxième couche du réseau est une couche cachée. Nous désignons le nombre de neurones dans cette couche cachée par nn, et nous expérimenterons différentes valeurs pour nn. L'exemple montré illustre une petite couche cachée, contenant seulement n=15 neurones. La couche de sortie du réseau contient 10 neurones. Si le premier neurone se déclenche, c'est-à-dire a une sortie 1, cela indiquera que le réseau pense que le chiffre est un 0. Si le deuxième neurone se déclenche, cela indiquera que le réseau pense que le chiffre est un 1. Et ainsi de suite . Un peu plus précisément, nous numérotons les neurones de sortie de 0 à 9 et déterminons quel neurone a la valeur d'activation la plus élevée. Si ce neurone est, disons, le neurone numéro 6, alors notre réseau devinera que le chiffre d'entrée était un 6. Et ainsi de suite pour les autres neurones de sortie. Vous pourriez vous demander pourquoi nous utilisons 10 neurones de sortie. Après tout, le but du réseau est de nous dire quel chiffre (0,1,2,…,9) correspond à l'image d'entrée. Une manière apparemment naturelle de procéder consiste à n'utiliser que 4 neurones de sortie, en traitant chaque neurone comme prenant une valeur binaire, selon que la sortie du neurone est plus proche de 0 ou de 1. Quatre neurones suffisent pour coder la réponse, puisque 24 =16 est plus que les 10 valeurs possibles pour le chiffre d'entrée. Pourquoi notre réseau devrait-il plutôt utiliser 10 neurones ? N'est-ce pas inefficace ? La justification ultime est empirique : on peut essayer les deux conceptions de réseau, et il s'avère que, pour ce problème particulier, le réseau à 10 neurones de sortie apprend à mieux reconnaître les chiffres que le réseau à 4 neurones de sortie. Mais cela nous laisse nous demander pourquoi l'utilisation de 10 neurones de sortie fonctionne mieux. Existe-t-il une heuristique qui nous indiquerait à l'avance que nous devrions utiliser l'encodage à 10 sorties au lieu de l'encodage à 4 sorties ? Pour comprendre pourquoi nous faisons cela, il est utile de réfléchir à ce que fait le réseau de neurones à partir des premiers principes. Considérons d'abord le cas où nous utilisons 10 neurones de sortie. Concentrons-nous sur le premier neurone de sortie, celui qui essaie de décider si le chiffre est ou non un 0. Il le fait en pesant les preuves de la couche cachée de neurones.

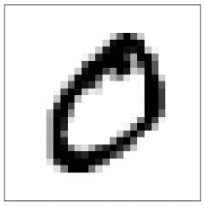
Que font ces neurones cachés ? Eh bien, supposons simplement pour les besoins de l'argument que le premier neurone de la couche cachée détecte si une image comme celle-ci est présente ou non :



Il peut le faire en pondérant fortement les pixels d'entrée qui se chevauchent avec l'image, et en ne pondérant que légèrement les autres entrées. De la même manière, supposons pour les besoins de l'argumentation que les deuxième, troisième et quatrième neurones de la couche cachée détectent si les images suivantes sont présentes ou non :



Comme vous l'avez peut-être deviné, ces quatre images forment ensemble l'image 0 que nous avons vue dans la ligne de chiffres indiquée précédemment :

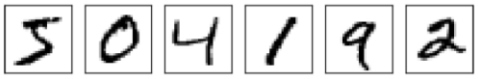


Donc, si ces quatre neurones cachés se déclenchent, nous pouvons conclure que le chiffre est un 0. Bien sûr, ce n'est pas la seule sorte de preuve que nous pouvons utiliser pour conclure que l'image était un 0 - nous pourrions légitimement obtenir un 0 dans de nombreuses autres manières (par exemple, à travers des traductions des images ci-dessus, ou de légères distorsions). Mais il semble sûr de dire qu'au moins dans ce cas, nous conclurions que l'entrée était un 0. En supposant que le réseau de neurones fonctionne de cette manière, nous pouvons donner une explication plausible pour laquelle il est préférable d'avoir 10 sorties du réseau, plutôt que 4. Si nous avions 4 sorties, alors le premier neurone de sortie essaierait de décider quel est le bit le plus significatif du chiffre. Et il n'y a pas de moyen facile de relier ce bit le plus significatif à des formes simples comme celles illustrées ci-dessus. Il est difficile d'imaginer qu'il existe une bonne raison historique pour laquelle les formes des composants du chiffre seront étroitement liées (disons) au bit le plus significatif de la sortie. Maintenant, avec tout ce qui a été dit, tout cela n'est qu'une heuristique. Rien ne dit que le réseau neuronal à trois couches doit fonctionner comme je l'ai décrit, les neurones cachés détectant des formes de composants simples. Peut-être qu'un algorithme d'apprentissage intelligent trouvera une affectation de poids qui nous permettra d'utiliser seulement 4 neurones de sortie. Mais en tant qu'heuristique, la façon de penser que j'ai décrite fonctionne plutôt bien et peut vous faire gagner beaucoup de temps dans la conception de bonnes architectures de réseaux neuronaux.

Apprentissage avec les algorithmes de descente

Maintenant que nous avons un design pour notre réseau de neurones, comment peut-il apprendre à reconnaître les chiffres ? La première chose dont nous aurons besoin est un ensemble de données à

partir duquel apprendre - un ensemble de données d'apprentissage. Nous utiliserons l'ensemble de données MNIST, qui contient des dizaines de milliers d'images numérisées de chiffres manuscrits, ainsi que leurs classifications correctes. Le nom du MNIST vient du fait qu'il s'agit d'un sous-ensemble modifié de deux ensembles de données collectées par le NIST, l'Institut national des normes et de la technologie des États-Unis. Voici quelques images du MNIST :



Comme vous pouvez le voir, ces chiffres sont, en fait, les mêmes que ceux présentés au début de ce chapitre comme un défi à reconnaître. Bien entendu, lors du test de notre réseau, nous lui demanderons de reconnaître les images qui ne sont pas dans l'ensemble d'apprentissage ! Les données MNIST se présentent en deux parties. La première partie contient $60\,000$ images à utiliser comme données d'apprentissage. Ces images sont des échantillons d'écriture scannés de 250 personnes, dont la moitié étaient des employés du US Census Bureau et la moitié des lycéens. Les images sont en niveaux de gris et mesurent 28×28 pixels. La deuxième partie de l'ensemble de données MNIST comprend $10\,000$ images à utiliser comme données de test. Encore une fois, ce sont des images en niveaux de gris de 28 par 28. Nous utiliserons les données du test pour évaluer dans quelle mesure notre réseau de neurones a appris à reconnaître les chiffres. Pour en faire un bon test de performance, les données de test ont été extraites d'un ensemble différent de 250 personnes que les données de formation d'origine (bien qu'il s'agisse toujours d'un groupe divisé entre les employés du Census Bureau et les étudiants du secondaire). Cela nous donne l'assurance que notre système peut reconnaître les chiffres des personnes dont il n'a pas vu l'écriture pendant la formation. Nous utiliserons la notation xx pour désigner une entrée d'apprentissage. Il sera pratique de considérer chaque entrée d'apprentissage xx comme un vecteur de 28 × 28 = 784 dimensions. Chaque entrée du vecteur représente la valeur de gris pour un seul pixel de l'image. Nous noterons la sortie souhaitée correspondante par y=$y(x)$, où y est un vecteur à 10 dimensions. Par exemple, si une image d'entraînement particulière, x, représente un 6, alors $y(x) = (0{,}0{,}0{,}0{,}0{,}0{,}1{,}0{,}0{,}0)^T$ est la sortie souhaitée du réseau .

Ce que nous aimerions, c'est un algorithme qui nous permette de trouver des poids et des biais afin que la sortie du réseau se rapproche de $y(x)$ pour toutes les entrées d'apprentissage x. Pour quantifier dans quelle mesure nous atteignons cet objectif, nous définissons une fonction de coût :

$$C(w,b) = \frac{1}{2n}\sum_x \|y(x) - a\|^2$$

Ici, w désigne la collection de tous les poids du réseau, tous les biais, n est le nombre total d'entrées d'apprentissage, a est le vecteur des sorties du réseau lorsque x est entré, et la somme est sur toutes les entrées d'apprentissage. Bien sûr, la sortie aa dépend de x, w et b. Nous appellerons C la fonction coût; elle est aussi parfois connue sous le nom d'erreur quadratique moyenne ou simplement MSE. En examinant la forme de la fonction de coût quadratique, nous voyons que $C(w,b)$ est non négatif, puisque chaque terme de la somme est non négatif. De plus, le coût $C(w,b)$ devient petit, c'est-à-dire $C(w,b) \approx 0$, précisément lorsque $y(x)$ est approximativement égal à la sortie, a, pour toutes les entrées d'apprentissage, x. Notre algorithme d'entraînement a donc fait du bon travail s'il peut trouver des poids et des biais de sorte que $C(w,b) \approx 0$. En revanche, ça ne marche pas si bien quand $C(w,b)$ est grand, cela voudrait dire que $y(x)$ n'est pas proches de la sortie aa pour un grand nombre d'entrées. Le but de notre algorithme d'apprentissage sera donc de minimiser le coût $C(w,b)$ en fonction des poids et des biais. En d'autres termes, nous voulons trouver un ensemble de poids et de

biais qui rendent la fonction coût aussi faible que possible. Nous pouvons le faire en utilisant l'algorithme du gradient.

Bibliographie :

M.A. Nielson: *Neural Networks and Deep Learning*, 2015