UNIVERSITÉ DU QUÉBEC À MONTRÉAL

RÉSEAU DE NEURONES ARTIFICIELS ET APPLICATIONS À LA CLASSIFICATION D'IMAGES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES

PAR

SAMUEL BÉDARD-VENNE

MAI 2018

REMERCIEMENTS

Je tiens à remercier en premier lieu mon directeur de maîtrise, Simon Guillotte. Merci de m'avoir fait confiance, de m'avoir laissé choisir mon sujet de mémoire et pour le temps que tu m'as accordé. Ton aide m'a permis d'organiser ma pensée et mes idées tout en grandissant au niveau intellectuel. Ta façon d'approcher certains problèmes m'a poussé à dépasser mes limites et à me questionner davantage. Cette démarche a favorisé la créativité et une compréhension plus profonde des concepts dont il a été question lors de ce mémoire. Je suis convaincu que ce baggage de connaissances me sera utile tout au long de ma carrière professionnelle.

Je désire également remercier Sorana Froda et François Watier de m'avoir accueilli chaleureusement dans le programme de statistique. Vos précieux conseils ont permis de mieux orienter mon parcours. Merci également pour votre disponibilité et votre enseignement.

Je tiens aussi à remercier Fabrice Larribe, Karim Oualkacha et René Ferland. Dès mes premiers cours au département de mathématiques, vous avez su me transmettre votre passion pour la statistique qui se reflète au travers de votre enseignement.

Je remercie ma famille et mes amis pour tout le support que vous m'avez apporté tout au long de mon parcours académique. Merci pour vos nombreux encouragements. Sans vous rien de cet accomplissement n'aurait été possible.

TABLE DES MATIÈRES

LISTE DES FIGURES

RÉSUMÉ

Ce travail a pour sujet l'étude des réseaux de neurones, plus particulièrement la formulation mathématique du modèle. On y présente en premier lieu une introduction permettant de faire le lien entre la régression logistique multinomiale et le réseau de neurones. Le réseau de neurones MLP est ensuite défini et on s'intéresse à des questions d'inférence et d'estimation. On montre comment arriver à une forme récursive pour le calcul des dérivées de la fonction de perte par rapport aux paramètres du modèle. L'implémentation informatique est par la suite introduite afin de faire le lien entre la théorie et la façon dont on configure le réseau de neurones dans les modules couramment utilisés par la communauté scientifique. On montre également quelques propriétés statistiques pour l'algorithme de régularisation Dropout. Par la suite, le réseau de neurones à convolution est présenté tout en faisant un lien avec le modèle précédent. Le travail se conclut par un chapitre d'application, qui présente les résultats obtenus des différents modèles définis dans ce travail et aïnsi que certaines variations de ceux-ci.

MOTS-CLÉS : intelligence artificielle, réseaux de neurones artificiels, classification d'images

INTRODUCTION

En octobre 2015, le programme AlphaGo développé par la division Deepmind de Google a réussi à vaincre Fan Hui, le triple champion du Championnat européen de go (DeepMind, 2016). AlphaGo devient le premier programme informatique à vaincre un joueur professionnel au jeu de go. Cet exploit est répété à une deuxième reprise en mars 2016 lorsque le programme réussit à battre Lee Sedol, considéré comme le plus grand joueur de go de la dernière décennie. Originaire de Chine, le jeu de go date de plus 2500 ans. Sa complexité repose sur le nombre de configurations légales possibles de la table de jeu. Ce nombre est en fait plus grand que le nombre de particules dans l'univers. Le jeu de go est un Gogol (10^{100}) fois plus complexe que le jeu d'échec. Cet exploit, que plusieurs croyaient seulement possible avant plusieurs années a été réalisé en utilisant l'apprentissage automatique. Plus particulièrement, en utilisant une combinaison de réseau de neurones convolutifs et d'arbre de recherche «Search tree» (Silver *et al.*, 2016).

L'apprentissage automatique «machine learning» est la science et l'art qui étudie la façon de développer des algorithmes capables d'apprendre à partir de données. Dans le cadre de ce mémoire, on s'intéresse à des problèmes d'apprentissage supervisé, c'est-à-dire lorsque l'on observe certains exemples et les classes correspondantes. Le but d'un apprentissage supervisé est que le modèle puisse apprendre à partir d'observations pour ensuite être généralisable sur des données à l'extérieur de l'échantillon sur lequel on a entraîné les paramètres. L'intention devient que le modèle soit capable de prédire la sortie associée à certains exemples pour lesquels on ne connait pas la réponse.

Étant donné que les ordinateurs ont récemment gagné en puissance et que l'on traite désormais beaucoup plus de données, le réseau de neurones est souvent un modèle de prédilection dans le domaine de l'apprentissage automatique. Le modèle est présentement à la fine pointe due à sa capacité à résoudre des tâches complexes. Il est utilisé pour un grand nombre d'applications dont la reconnaissance d'images, le traitement automatique du langage naturel ainsi que pour les systèmes de recommandation.

On utilisera le modèle de réseau de neurones pour des fins d'applications à la classification d'images puisqu'il performe de façon remarquable pour la vision par ordinateur. Le MNIST «Mixed National Institute of Standards and Technology database», une grande base de données de chiffres écrits à la main fréquemment utilisée comme test de performance «Benchmark» pour entraîner divers systèmes de traitement d'images, sera employé pour tester le modèle et ses variantes. Les codes pour les différents modèles et prototypes développés sont disponibles en ligne sur mon Github à l'adresse `https://github.com/samuelBedard/` dans la section MNIST.

La plupart des publications dans le domaine portent davantage sur des applications informatiques empiriques. Dans le cadre de ce mémoire, il est question d'explorer les développements des plus récentes applications informatiques utilisées pour les réseaux de neurones dans le but de définir et formuler d'une façon mathématique le modèle et ses propriétés.

Pour ce faire, le chapitre 1 vise à faire le lien entre la régression logistique multinomiale et le réseau de neurones. Au chapitre 2, on définit le modèle pour un réseau de type MLP «Multilayer Perceptron». On passe à l'estimation des paramètres en définissant les techniques les plus couramment utilisées pour le calcul du gradient. On montre à l'aide d'un exemple comment on peut arriver à une forme

récursive pour le calcul des dérivées par rapport à chacun des paramètres. Le lien est ensuite fait pour déterminer comment on passe à l'implémentation informatique matricielle utilisée dans la plupart des librairies les plus utilisées à l'heure actuelle pour les réseaux de neurones. On verra également certaines techniques de régularisation dont Dropout. Le travail se termine par une introduction au réseau de neurones convolutifs. La performance des différents modèles et variantes sera comparée afin de déterminer le meilleur modèle.

CHAPITRE I

RÉGRESSION LOGISTIQUE MULTINOMIALE

Puisque ce mémoire porte en grande partie sur les réseaux de neurones, cette section vise à faire un lien entre la régression logistique multinomiale telle que l'on est habitué de la voir en statistique et le réseau de neurones. Une architecture pour un réseau de neurones artificielle MLP «multilayer perceptron» a une forme similaire à celle de la figure 1.1.

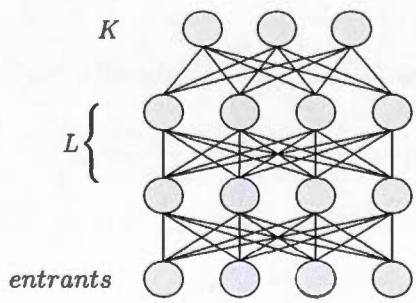

Figure 1.1: Architecture pour un réseau de neurones MLP avec 4 entrants $x \in \mathbb{R}^4$, $K = 3$ classes distinctes pour la sortie et deux couches cachées $L = 2$.

Un réseau de neurones MLP est toujours composé d'une couche d'entrée pour

accueillir les entrants d'un jeu de données, de $L \geq 0$ couche(s) intermédiaire(s) que l'on appelle couches cachées et d'une couche de sortie dont la dimension dépend du nombre de classes pour la variable dépendante.

Lorsqu'il est question de classification avec un réseau de neurones, la variable expliquée $y \in \mathbb{R}^K$ contient l'information pour la classe observée et $x \in \mathbb{R}^p$ est le vecteur d'entrants contenant les p variables explicatives. Une observation de la variable expliquée y prend la forme d'un vecteur où la $k^{\text{ème}}$ composante prend la valeur 1 si la classe correspondante est observée. C'est-à-dire que y a l'une des formes suivantes selon la classe associée :

$$\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \cdots \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}. \tag{1.1}$$

Ce problème est en fait modélisé à l'aide de la loi multinomiale. Un jeu de données contenant les N observations de la variable dépendante y est représenté au niveau probabiliste par les réalisations de N épreuves indépendantes d'une loi multinomiale. Les observations $y_1, \ldots, y_N$ sont indépendamment distribuées selon une loi multinomiale ; pour la $i^{\text{ème}}$ observation $y_i \sim Multi(n, p_{i1}, \ldots, p_{iK})$. Dans le cadre des applications utilisées pour ce mémoire, $n = 1$ puisque pour chaque observation, on observe une seule classe. On le voit particulièrement dans la réprésentation 1.1. On dénote par $y_i^{(k)}$, la $k^{\text{ème}}$ entrée du vecteur y_i, dont le support est $\{0, 1\}$. Pour la $i^{\text{ème}}$ observation, le vecteur $y_i = (y_i^{(1)}, \ldots, y_i^{(K)})$ suit une distribution de paramètres n et p où $p = (p_{i1}, \ldots, p_{iK})$. Notre distribution de probabilité a la forme suivante

$$f(y_i, n, p) = \binom{n}{y_i^{(1)}, \ldots, y_i^{(K)}} p_{i1}^{y_i^{(1)}} \cdots p_{iK}^{y_i^{(K)}} \tag{1.2}$$

où $\sum_{k=1}^{K} y_i^{(k)} = n$ et $\sum_{k=1}^{K} p_{ik} = 1$. Étant donné la contrainte sur nos paramètres, il est possible d'optimiser notre vraisemblance de deux façons. La première façon

consiste à estimer les paramètres de la vraisemblance dans un espace de dimension $K-1$ dans $\mathbb{R}^K$. En faisant de la sorte, on trouve une estimation pour $\hat{p}_1, \ldots, \hat{p}_K$. Par exemple, supposons qu'on a seulement trois paramètres p_1, p_2 et p_3, alors les 3 paramètres à estimer sont contraints à vivre dans un plan de dimension 2 à l'intérieur de $\mathbb{R}^3$.

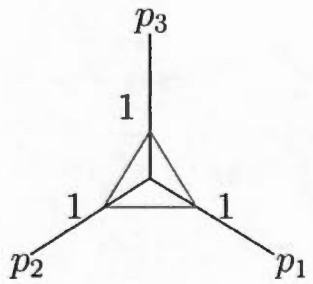

Figure 1.2: Les paramètres p_1, p_2 et p_3 sont restreints dans le plan délimité par les traits en bleu.

En apprentissage automatique, cette approche est utilisée lorsque l'on utilise la fonction softmax définie en 1.3. Pour le réseau de neurones, cette fonction est souvent utilisée pour faire le passage entre la dernière couche cachée et la sortie du réseau.

$$p_k = \frac{exp(z_k)}{\sum_{i=1}^{K} exp(z_i)}. \tag{1.3}$$

Lors de la modélisation d'une régression multinomiale, on désire relier la distribution $p_{i1}, \ldots, p_{iK}$ à une variable explicative x_i. La fonction softmax transforme les entrants de K prédicteurs linéaires en une probabilité de sorte que selon un entrant quelconque x_0, on associe une probabilité différente à chacune des classes. Pour chaque entrant x_{0j}, $j = 1, \ldots, N$, compris dans le jeu de données, la fonction softmax permet de produire une distribution de probabilité sommant à 1.

Puisque l'on a une contrainte sur les paramètres, il est également possible d'op-

timiser nos paramètres dans un espace de dimension $K-1$ dans $\mathbb{R}^{K-1}$, d'où la deuxième façon. L'équation 1.2 peut être réécrite de la manière suivante

$$\begin{aligned} f(y_i,n,p) &= exp\left[log\binom{n}{y_i^{(1)},\ldots,y_i^{(K)}} + \sum_{k=1}^{K-1} y_i^{(k)} log(p_{ik})\right] \\ &= h(y_i)exp\left(\sum_{k=1}^{K} y_i^{(k)} log(p_{ik})\right) \\ &= h(y_i)exp\left[\sum_{k=1}^{K-1} y_i^{(k)} log(p_{ik}) + (n - \sum_{k=1}^{K-1} y_i^{(k)}) log(p_{iK})\right] \\ &= h(y_i)exp\left[\sum_{k=1}^{K-1} log\left(\frac{p_{ik}}{p_{iK}}\right) + nlog(p_{iK})\right] \end{aligned}$$

En posant $\eta_i = \left(\frac{p_{ik}}{p_{iK}}\right)$, on trouve que $p_{iK} = (\sum_{k=1}^{K-1} exp(\eta_i) + 1)^{-1}$ et on obtient la vraisemblance paramètrisée en η. On a

$$\begin{aligned} f(y_i,n,\eta) &= h(y_i)exp\left[\sum_{k=1}^{K-1} y_i^{(k)}\eta_i - nlog(\sum_{k=1}^{K-1} exp(\eta_i) + 1)\right] \\ &= h(y_i)c(\eta)exp\left(\sum_{k=1}^{K-1} y_i^{(k)}\eta_i\right) \end{aligned}$$

Cette deuxième méthode d'optimisation est plus couramment utlisée lorsque l'on effectue une régression logistique multinomiale en statistique. On modélise $K-1$ régressions logistique binaires indépendantes par $K-1$ transformations logit où la $K^{\text{ème}}$ classe est choisie arbitrairement comme étant une classe pivot. Tel que l'on peut le voir dans la figure 1.3, la régression multinomiale est en faite une forme simpliste d'un réseau de neurones où l'on a $L=0$ couche cachée.

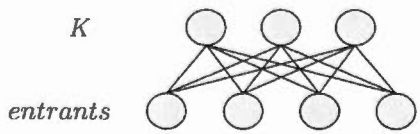

Figure 1.3: Architecture pour un réseau de neurones MLP avec 4 entrants et $K=3$ classes distinctes pour la sortie et aucune couche cachée.

CHAPITRE II

RÉSEAU DE NEURONES MLP

2.1 Le modèle

Le modèle pour un réseau de neurones (MLP) est défini récursivement de la façon suivante :

$$x_{l+1} = \begin{cases} g(\theta_l x_l) & \text{si } l = 0, 1, ..., L-1 \\ f(\theta_L x_L) & \text{si } l = L \end{cases}$$

où L est le nombre de couches cachées. On définit la matrice de poids partitionnée $\theta_l = (b_l|W_l)$ comme étant la matrice comprenant la matrice de poids $W_l \in \mathbb{R}^{n_{l+1} \times n_l}$ entre les neurones et le vecteur biais $b_l \in \mathbb{R}^{n_{l+1}}$. On prend les notations suivantes $n_0, n_1, \ldots, n_L, n_{L+1}$ pour signifier le nombre de neurones respectif dans chacune des $L+2$ couches du réseau allant de la couche d'entrée jusqu'à la couche de sortie. Ainsi, x_0 est vu comme un vecteur de variables explicatives pour le réseau, alors que x_{L+1} représente le vecteur ou scalaire de variable(s) expliquée(s) par celui-ci. Les vecteurs $x_1, \ldots, x_L$ contiennent les neurones pour chacune des couches cachées du réseau.

On choisit la fonction d'activation g afin de faire les connexions entre les différentes couches cachées d'un réseau de neurones. On choisira une fonction d'activation pouvant être distincte pour la sortie du réseau que l'on dénote par f . Cette dernière transforme les neurones de la dernière couche en sortie pour le réseau.

Par exemple, pour une tâche de classification binaire, on aura le choix d'utiliser f égale à la fonction sigmoïde ou softmax 1.3 selon que l'on utilise un ou deux neurones respectivement pour la couche de sortie. Dans le cas où l'on a une tâche de classification où le nombre de catégories est $K \geq 2$, la fonction softmax est souvent utilisée en pratique puisqu'elle permet d'associer une probabilité respective à chacune des différentes classes.

2.2 Inférence

On dénote l'ensemble des paramètres du modèle comme étant $\theta = \{\theta_0, \ldots, \theta_L\}$. Ainsi, on utilise la fonction

$$S_\theta(x_0) = f(\theta_L\, g(\theta_{L-1} g(\theta_{L-2} \circ \circ \circ \theta_1 g(\theta_0 x_0)))$$

et on cherche

$$\underset{\theta}{\operatorname{argmin}}\ \ell(S_\theta(x_0), y) + \lambda R(\theta)$$

où y est une observation de la variable expliquée et R le régularisateur pour le modèle. On se sert donc du jeu d'entraînement. Supposons que ce dernier contienne N paires d'observations d'un vecteur de variables explicatives et expliquées. Alors on a

$$(x_{0j}, y_j) \qquad j = 1, \ldots, N$$

et le problème revient à minimiser

$$\frac{1}{N}\left[\sum_{j=1}^{N} \ell(S_\theta(x_{0j}), y_j) + \lambda R(\theta)\right].$$

2.3 Estimation du modèle

2.3.1 Algorithme du gradient

L'approximation des paramètres pour un réseau de neurones se fait de la même façon que l'on aurait pu le faire pour un problème standard de minimisation en

statistique ou en apprentissage automatique. On minimise une fonction de perte par rapport à ses paramètres à laquelle s'ajoute un paramètre de régularisation. On cherche à minimiser une fonction sous forme de sommation

$$Q(\theta) = \frac{1}{N}\left[\sum_{j=1}^{N} \ell(S_\theta(x_{0j}), y_j) + \lambda R(\theta)\right] \tag{2.1}$$

où θ^* est le paramètre qui minimise 2.1 à estimer.

Une façon naturelle d'approcher θ^* est d'utiliser l'algorithme du gradient (GD). Le gradient est utilisé afin de trouver la direction pour laquelle la fonction $Q(\theta)$ décroît le plus rapidement. Lorsque θ se situe à un certain point, disons θ', cette direction correspond à la direction négative du gradient de la fonction Q, c'est-à-dire $-\nabla Q(\theta')$. Pour un taux α assez petit, on a que si

$$\theta'' = \theta' - \alpha \nabla Q(\theta'),$$

alors $Q(\theta') \geq Q(\theta'')$. L'idée est donc de partir d'un point de départ pour θ_0 pour la fonction de perte afin de se diriger à petits pas vers un minimum en espérant que celui-ci soit un minimum global. Ainsi, on a une séquence de points $\theta_0, \theta_1, \theta_2, \ldots$ tel que pour

$$\theta_{n+1} = \theta_n - \alpha \nabla Q(\theta_n), n \geq 0$$

on a que

$$Q(\theta_0) \geq Q(\theta_1) \geq Q(\theta_2) \geq \ldots$$

En pseudocode, on a les étapes suivantes pour la minimisation :

1: On initialise les paramètres pour le réseau
2: Pour n itérations
3: $\quad \theta_{n+1} = \theta_n - \alpha \nabla Q(\theta_n)$

Il est connu que si la fonction objective est convexe, cet algorithme mène à une solution optimale.

2.3.2 Algorithme du gradient stochastique

Lorsque le nombre d'observations N est grand pour un certain jeu de données, évaluer le gradient de la somme définie en 2.1 peut devenir très lourd au niveau computationnel. C'est pourquoi, en pratique, l'algorithme du gradient stochastique (SGD) est davantage utilisé. Contrairement au GD, l'algorithme met à jour les paramètres pour chaque observation dans l'ensemble d'entraînement. Afin d'introduire l'algorithme, on prend ici $J(\theta)_j$ la fonction de perte évaluée pour l'observation j, c'est-à-dire

$$J(\theta)_j = \ell(S_\theta(x_{0j}), y_j) \qquad j = 1, \ldots, N.$$

Le SGD approxime la vraie valeur du gradient par sa valeur évaluée sur une observation. La convergence de l'algorithme repose en grande partie sur deux conditions. Si α_t est le pas de l'itération pour la $t^{\text{ème}}$ mise à jour des paramètres, on a besoin que $\alpha_t \to 0$, $\sum_t \alpha_t^2 < \infty$ et $\sum_t \alpha_t = \infty$. Par exemple, on aurait que $\alpha_t = 1/t$ serait un bon choix étant donné qu'il satisfait ces trois conditions. Essentiellement, lorsque α_t décroît selon un taux approprié et que ces conditions sont respectées, le SGD converge vers le minimum global si la fonction de perte est convexe ou vers un minimum local si celle-ci est pseudoconvexe. Ces résultats proviennent en fait du théorème de Robbins-Siegmund (Robbins, 1971). Un choix couramment utilisé pour la vitesse de convergence $\alpha_t = \frac{\alpha_0}{1+\delta t}$. On a ici que α_0 représente la valeur initiale pour le pas d'itération et δ est appelée constante de décroissance. On choisit habituellement $\delta < 10^{-3}$.

En pseudo-code, on a les étapes suivantes pour la minimisation :

1: On initialise les paramètres pour le réseau

2: Pour n itérations

3: Mélanger aléatoirement les observations dans l'ensemble d'entraînement

4: Pour chacune des observations dans l'ensemble d'entraînement définie à l'étape 3. ;

5: $\theta_{n+1} := \theta_n - \alpha_t(\nabla J(\theta_n)_j + \nabla \frac{\lambda}{2} R(\theta_n))$

On utilise le terme période «epochs» pour désigner n le nombre de fois où l'on passe sur l'ensemble d'entraînement. Lorsque N est grand, on peut apprendre de nos données en passant une seule fois sur le jeu de données. Par contre, plus N est petit, plus on devra passer de fois sur le jeu de données afin que l'algorithme du gradient puisse apprendre des données. Afin de déterminer le nombre optimal de périodes, on utilise souvent une procédure d'arrêt forcé «early stopping». L'idée derrière cette procédure est d'arrêter l'algorithme avant convergence afin d'éviter le surajustement du modèle sur l'ensemble d'entraînement. En pratique, on peut représenter sur un graphique l'erreur moyenne sur l'ensemble d'entraînement à laquelle on compare l'erreur moyenne sur l'ensemble de validation. Au départ, l'erreur moyenne va diminuer sur les deux ensembles jusqu'à un certain point où l'on aura que l'erreur sur l'ensemble d'entraînement va diminuer alors que celle sur l'ensemble validation va commencer à augmenter. On peut percevoir ce point comme étant une indication de surajustement pour le modèle sur l'ensemble d'entraînement.

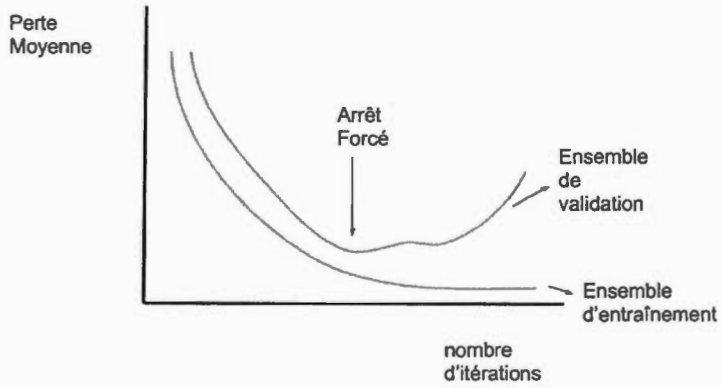

Figure 2.1: Exemple où l'on a surajustement sur l'ensemble de validation. On arrête d'itérer lorsque l'erreur moyenne sur l'ensemble de validation est à son minimum (i.e avant qu'elle commence à augmenter).

2.3.3 Algorithme du gradient stochastique «Mini-Batch»

Un compromis entre le GD et le SGD est d'utiliser le gradient stochastique «mini-batch». Au lieu d'approximer la valeur du gradient pour chaque observation dans le jeu de données, on peut utiliser à chaque fois un certain nombre d'observations choisi aléatoirement que l'on regroupe en «mini-batch» d'une certaine grandeur, disons b, pour calculer le gradient. On arrive ainsi à réduire la variance de nos paramètres lorsque ceux-ci sont réévalués. Ce qui peut mener à une convergence relativement plus stable des paramètres. L'autre avantage par rapport au SGD est que l'on peut accélérer le calcul du gradient en utilisant certaines librairies permettant l'optimisation matricielle. Avec une bonne vectorisation, le gradient stochastique «mini-batch» permet de traiter plusieurs observations en parallèle et par conséquent, il peut être plus efficace que le SGD.

En pseudo-code, on a les étapes suivantes pour la minimisation :

1: On initialise les paramètres pour le réseau

2: Pour n itérations

3: Mélanger aléatoirement les observations dans l'ensemble d'entraînement

4: Pour chacune des «mini-batch» c.-à-d. pour la séparation suivante de l'ensemble d'entraînement de l'étape 3 : $(i = 1, b+1, 2b+1, \ldots, ..., N-b+1)$;

5: $\theta_{n+1} := \theta_n - \alpha_t(\nabla \frac{1}{b} \sum_{j=i}^{i+(b-1)} J(\theta_n)_j + \nabla \frac{\lambda}{2b} R(\theta_n))$

2.4 Dérivabilité des fonctions et réseau MLP

Afin d'introduire les calculs pour les dérivées lors du calcul du gradient, on utilise certaines définitions en lien avec l'approche matricielle du calcul différentiel.

Definition 1. *Soit ϕ une fonction différentiable par rapport à un vecteur $x \in \mathbb{R}^n$ tel que $\phi : \mathbb{R}^n \to \mathbb{R}$, alors le vecteur $1 \times n$*

$$D\phi(x) = \frac{\partial \phi(x)}{\partial x^T}$$

est la dérivée de $\phi(x)$.

Definition 2. *Une forme plus générale est celle où l'on a une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$. Dans ce cas, la dérivée $Df(x)$ prend la forme d'une matrice $m \times n$ de la forme*

$$Df(x) = \begin{bmatrix} Df_1(x) \\ \vdots \\ Df_m(x) \end{bmatrix} = \frac{\partial f(x)}{\partial x^T}$$

On remarque à travers cette forme que notre dérivée de la définition 1 est simplement un cas particulier de la définition 2. On appelle cette matrice le Jacobien de f. Afin de parvenir au calcul des dérivées, on aura également besoin des deux définitions qui suivent.

Definition 3. *Si A est une matrice $m \times n$, alors sa vectorisation sera donnée par la forme suivante*

$$vec(A) = \begin{bmatrix} a_{11} & , \ldots , & a_{m1}, & a_{12} & , \ldots , & a_{m2} & , \ldots , & a_{1n} & , \ldots , & a_{mn} \end{bmatrix}^T \quad (2.2)$$

Definition 4. *Si A est une matrice $m \times n$ et B une matrice $p \times q$, alors le produit de kronecker $A \otimes B$ prend la forme suivante*

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix} \tag{2.3}$$

À partir de ces quatre définitions et du théorème de dérivation des fonctions composées, on est en mesure de calculer la dérivée par rapport à chacun des poids pour le réseau.

Afin de comprendre comment effectuer le calcul de chacun des poids pour le réseau, prenons un exemple simple. Considérons le modèle suivant avec seulement une couche cachée et sans vecteur de biais. C'est-à-dire

$$\theta_0 = W_0 \in \mathbb{R}^{n_1 \times n_0} \qquad S_\theta(x_0) = f(\theta_1 g(\theta_0 x_0)) \in \mathbb{R}^K \qquad \theta = \begin{bmatrix} vec(\theta_0) \\ vec(\theta_1) \end{bmatrix} \in \mathbb{R}^{n_0 n_1 + n_1 K}$$

$$\theta_1 = W_1 \in \mathbb{R}^{K \times n_1} \qquad g : \mathbb{R}^{n_1} \to \mathbb{R}^{n_1} \qquad f : \mathbb{R}^K \to \mathbb{R}^K$$

et avec une fonction de perte sans régularisateur

$$Q(\theta) = J(\theta) = \ell(S_\theta(x_0), y).$$

Afin de déterminer la dérivée de notre fonction de perte par rapport à θ, l'ensemble de nos paramètres, on utilise la vectorisation des deux matrices θ_0 et θ_1. Pour la transformation de nos paramètres et les dérivées de ceux-ci, on peut se référer au schéma suivant qui représente à travers une série de transformations la fonction qui transporte θ vers la fonction de perte. On veut d'abord déterminer $D\ell(S_\theta(x_0), y)$. En appliquant la règle de la chaîne une première fois, on obtient que

$$D\ell(S_\theta(x_0), y) = D\ell(S_\theta(x_0))_{1 \times K} DS_\theta(x_0)_{K \times (n_0 n_1 + n_1 K)}.$$

Puisque $\ell : \mathbb{R}^K \to \mathbb{R}$, alors $D\ell(S_\theta(x_0)) \in \mathbb{R}^{1 \times K}$. Les dimensions sont mises en index pour chacune des dérivées. À titre d'exemple, supposons que l'on prend la norme

$$\begin{array}{ccccccccc} \mathbb{R}^{n_0 n_1 + n_1 K} & \xrightarrow{j} & \mathbb{R}^{n_1 + n_1 K} & \xrightarrow{h^*} & \mathbb{R}^K & \xrightarrow{f} & \mathbb{R}^K & \xrightarrow{\ell} & \mathbb{R} \\ \begin{bmatrix} vec(\theta_0) \\ vec(\theta_1) \end{bmatrix} & \longmapsto & \begin{bmatrix} g(\theta_0 x_0) \\ vec(\theta_1) \end{bmatrix} & \longmapsto & \left[\theta_1 g(\theta_0 x_0)\right] & \longmapsto & \left[f(\theta_1 g(\theta_0 x_0))\right] & \longmapsto & \ell(S_\theta(x_0, y)) \end{array}$$

L_1 au carré pour la fonction de perte. Alors on aura la forme suivante

$$\ell(S_\theta(x_0), y) = \|S_\theta(x_0) - y\|^2 = \sum_{i=1}^{K} (y_i - S_\theta(x_0)_i)^2$$

et la dérivée est donnée par

$$\frac{\partial \ell}{\partial S_\theta(x_0)^T} = -2 \left[y_1 - S_\theta(x_0)_1, \ldots, y_K - S_\theta(x_0)_K\right].$$

Maintenant qu'on a déterminé comment trouver $D\ell(S_\theta(x_0))$, on s'intéresse à trouver une expression pour $DS_\theta(x_0)$. Si on dénote $h(\theta) = h^*(j(\theta)) = \theta_1 g(\theta_0 x_0)$, on a que h est une fonction qui transporte le paramètre θ dans $\mathbb{R}^K$ c'est-à-dire que $h : \mathbb{R}^{n_1 n_0 + n_1 K} \to \mathbb{R}^K$. En appliquant la règle de la chaîne une deuxième fois et puisque $f : \mathbb{R}^K \to \mathbb{R}^K$ on a que

$$DS_\theta(x_0) = Df(\theta_1 g(\theta_0 x_0))_{K \times K} D(h(\theta))_{K \times (n_1 n_0 + K n_1)}$$

On cherche maintenant à trouver $D(h(\theta))$. Mais $h(\theta)$ est encore une fois une fonction composée, car $h(\theta) = h^*(j(\theta))$ où

$$j(\theta) = \begin{bmatrix} g(\theta_0 x_0) \\ vec(\theta_1) \end{bmatrix} \qquad j : \mathbb{R}^{n_1 n_0 + K n_1} \to \mathbb{R}^{n_1 + K n_1}$$

et

$$h^*(j(\theta)) = \left[\theta_1 g(\theta_0 x_0)\right] \qquad h^* : \mathbb{R}^{n_1 + K n_1} \to \mathbb{R}^K.$$

Ainsi on a que

$$D(h(\theta))_{K\times(n_1n_0+Kn_1)} = Dh^*(j(\theta))_{K\times(n_1+Kn_1)}D(j(\theta))_{(n_1+Kn_1)\times(n_1n_0+Kn_1)}$$

et l'on obtient au final l'expression suivante avec les dimensions correspondantes à chacune des dérivées. Si on regarde les dimensions du schéma défini précédemment, on remarque que l'on se déplace exactement à l'inverse de celui-ci lorsqu'on effectue le calcul des dérivées.

$$\begin{aligned} D\ell(S_\theta(x_0), y) = D\ell(S_\theta(x_0))_{1\times K}Df(\theta_1 g(\theta_0 x_0))_{K\times K}Dh^*(j(\theta))_{K\times(n_1+Kn_1)} \\ \times D(j(\theta))_{(n_1+Kn_1)\times(n_1n_0+Kn_1)}. \end{aligned} \tag{2.4}$$

Tel qu'on peut le voir, puisque $D\ell(S_\theta(x_0), y) \in \mathbb{R}^{1\times(n_1n_0+Kn_1)}$, notre dérivée aura la forme d'un vecteur ligne correspondant à la même dimension que θ^T. Maintenant qu'on a une expression finale bien définie pour $D\ell(S_\theta(x_0), y)$, on aimerait avoir une expression nous permettant d'identifier chacune des dérivées. On cherche les dérivées de notre fonction de perte par rapport aux poids compris dans θ_0 et θ_1. Le but est de trouver une expression définie pour chacun des quatre blocs compris dans l'expression précédente.

Pour le premier bloc, on a vu précédemment dans l'exemple avec la perte quadratique que $D\ell(S_\theta(x_0))_{1\times K}$ dépend directement de la fonction de perte choisie. On peut donc passer directement à notre deuxième bloc $Df(\theta_1 g(\theta_0 x_0))_{K\times K}$. Rappelons que dans la partie sur la description du modèle on a posé $x_1, \ldots, x_L$ comme étant les vecteurs qui contiennent les neurones pour chacune des couches cachées du réseau. C'est-à-dire que $x_1 = g(\theta_0 x_0), x_2 = g(\theta_1 x_1), \ldots, x_{L+1} = g(\theta_L x_L)$. Pour la suite des choses, on posera également $a_1 = \theta_0 x_0, a_2 = \theta_1 x_1, \ldots, a_{L+1} = \theta_L x_L$ pour les transformations intermédiaires. Pour notre exemple, en suivant cette notation, on a $x_1 = g(\theta_0 x_0)$, $a_1 = \theta_0 x_0$ et $a_2 = \theta_1 x_1 = \theta_1 g(\theta_0 x_0)$.

En suivant cette notation, on cherche $Df(\theta_1 g(\theta_0 x_0)) = Df(\theta_1 x_1) = Df(a_2)$ et on a que $a_2 \in \mathbb{R}^K$ et $f(a_2) \in \mathbb{R}^K$. Par la définition 2, on trouve que

$$Df(a_2) = \begin{bmatrix} \frac{\partial f_1}{\partial a_2^1} & \cdots & \frac{\partial f_1}{\partial a_2^K} \\ \frac{\partial f_2}{\partial a_2^1} & \cdots & \frac{\partial f_2}{\partial a_2^K} \\ \vdots & \vdots & \ddots \\ \frac{\partial f_K}{\partial a_2^1} & \cdots & \frac{\partial f_K}{\partial a_2^K} \end{bmatrix} = \begin{bmatrix} f_1'(a_2^1) & 0 & \cdots & 0 \\ 0 & f_2'(a_2^2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f_K'(a_2^K) \end{bmatrix} = diag(f'(a_2)).$$

On obtient une matrice diagonale étant donné que f_1 dépend seulement de a_2^1, f_2 de a_2^2 et ainsi de suite jusqu'à f_K qui dépend seulement de a_2^K.

Après avoir déterminé la forme de la dérivée par rapport à notre deuxième bloc, on cherche la dérivée de notre troisième bloc, c'est-à-dire $Dh^*(j(\theta))_{K\times(n_1+Kn_1)}$. Comme la fonction $h^* : \mathbb{R}^{n_1+n1K} \to \mathbb{R}^K$ transporte

$$\begin{bmatrix} g(\theta_0 x_0) \\ vec(\theta_1) \end{bmatrix} \mapsto \left[\theta_1 g(\theta_0 x_0)\right],$$

lorsqu'on dérive $\theta_1 g(\theta_0 x_0) = \theta_1 x_1$, par rapport à chacun des éléments de x_1 et θ_1 un à la suite de l'autre, on obtient que

$$Dh^*(j(\theta)) = \left[\theta_1 \mid x_1^1 I_{K\times K} \mid x_1^2 I_{K\times K} \mid \ldots \mid x_1^{n_1} I_{K\times K}\right] = \left[\theta_1 \mid x_1^T \otimes I_{K\times K}\right].$$

Il ne reste plus qu'à trouver notre dérivée de notre quatrième bloc, c'est-à-dire $D(j(\theta))$. Comme la fonction $j : \mathbb{R}^{n_0 n_1+n1K} \to \mathbb{R}^{n_1+n_1 K}$ transporte

$$\theta = \begin{bmatrix} vec(\theta_0) \\ vec(\theta_1) \end{bmatrix} \mapsto \begin{bmatrix} g(\theta_0 x_0) \\ vec(\theta_1) \end{bmatrix}$$

si on dérive $g(\theta_0 x_0)$ et $vec(\theta_1)$, par rapport à chacun des éléments de θ_0 et θ_1 un à la suite de l'autre, on obtient que

$$D(j(\theta)) = \left[\begin{array}{c|c} Dg(\theta_0 x_0)_{n_1\times n_1}(x_0^T \otimes I_{n_1\times n_1})_{n_1\times n_0 n_1} & 0_{n_1\times n_1 K} \\ \hline 0_{n_1 K\times n_1 n_0} & I_{n_1 K\times n_1 K} \end{array}\right]$$

Par le même raisonnement qu'auparavant, lorsqu'on cherche à déterminer $Dg(\theta_0 x_0)$, on arrive encore à une matrice diagonale ;

$$Dg(a_1) = \begin{bmatrix} \frac{\partial g_1}{\partial a_1^1} & \cdots & \frac{\partial g_1}{\partial a_1^{n_1}} \\ \frac{\partial g_2}{\partial a_1^1} & \cdots & \frac{\partial g_2}{\partial a_1^{n_1}} \\ \vdots & \vdots & \ddots \\ \frac{\partial g_{n_1}}{\partial a_1^1} & \cdots & \frac{\partial g_{n_1}}{\partial a_1^{n_1}} \end{bmatrix} = \begin{bmatrix} g_1'(a_1^1) & 0 & \cdots & 0 \\ 0 & g_2'(a_1^2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g_{n_1}'(a_1^{n_1}) \end{bmatrix} = diag(g'(a_1)).$$

Étant donné qu'on a maintenant une expression bien définie pour chacun de nos blocs, il ne nous reste plus qu'à les regrouper ensemble pour obtenir une expression finale pour $D\ell(S_\theta(x_0), y)$. Puisque

$$D\ell(S_\theta(x_0)) = \Big[D\ell(S_\theta(x_0)_1), \ldots, D\ell(S_\theta(x_0)_K)\Big] \in \mathbb{R}^{1\times K},$$

On obtient que $D\ell(S_\theta(x_0))_{1\times K} diag(f'(a_2))_{K\times K} \in \mathbb{R}^{1\times K}$. Ce vecteur ligne aura la forme suivante

$$D\ell(S_\theta(x_0))diag(f'(a_2)) = \Big[D\ell(S_\theta(x_0)_1)f'(a_2^1), \ldots, D\ell(S_\theta(x_0)_K)f'(a_2^K)\Big] := \delta^{(2)}.$$

Une remarque sur la notation $\delta^{(2)}$. Cette quantité sera utilisée à plusieurs reprises par la suite. Comme nous le verrons en 2.6, cette notation nous permettra d'obtenir une équation récursive pour le calcul des dérivées pour chacun des θ_l, $l \in \{0, \ldots, L\}$. On utilise $\delta^{(l)}$ pour signifier la dérivée de la perte ℓ par rapport aux a_l associés aux neurones de la $l^{\text{ème}}$ couche. Ici, nous avons supposé $L = 1$ pour simplifier, et donc pour obtenir $\delta^{(2)}$ à partir de 2.6, il suffit de remplacer $L = 1$.

Notre expression finale 2.4 a la forme suivante :

$$\begin{aligned} D\ell(S_\theta(x_0), y) &= \delta^{(2)} \Big[\theta_1 \mid x_1^T \otimes I_{K\times K}\Big] \\ &\qquad \times \left[\begin{array}{c|c} Dg(\theta_0 x_0)_{n_1\times n_1}(x_0^T \otimes I_{n_1\times n_1})_{n_1\times n_0 n_1} & 0_{n_1\times n_1 K} \\ \hline 0_{n_1 K\times n_1 n_0} & I_{n_1 K\times n_1 K} \end{array}\right] \\ &= \delta^{(2)} \Big[\ \theta_1 Dg(\theta_0 x_0)_{n_1\times n_1}(x_0^T \otimes I_{n_1\times n_1})_{n_1\times n_0 n_1} \ \Big|\ x_1^T \otimes I_{K\times K})\ \Big] \\ &= \Big[\ \delta^{(2)}\theta_1 Dg(\theta_0 x_0)_{n_1\times n_1}(x_0^T \otimes I_{n_1\times n_1})_{n_1\times n_0 n_1} \ \Big|\ \delta^{(2)} x_1^T \otimes I_{K\times K})\ \Big]. \end{aligned}$$

On remarque à travers cette expression que le côté droit de l'expression correspond à la dérivée de notre fonction de perte par rapport à θ_1 alors que le côté gauche correspond à la dérivée par rapport à θ_0.

Pour le cas plus général, lorsque le nombre de couches cachées $L > 2$, il est facile de voir que le résultat se généralise. Notre expression finale pour notre vecteur de dérivées aura la forme d'un vecteur ligne partitionné où la première partition en partant de la gauche correspond à chacune des dérivées par rapport à θ_0, la deuxième partition en partant de la gauche correspond à chacune des dérivées par rapport à θ_1, et ainsi de suite jusqu'à la dernière partition qui correspond à chacune des dérivées par rapport à θ_L. On a la forme suivante pour la dérivée par rapport à θ_L

$$D\ell(S_\theta(x_0))diag(f'(a_{L+1}))(x_L^T \otimes I_{K\times K}).$$

Supposons qu'on souhaite maintenant déterminer la dérivée par rapport à θ_l, $l \in \{0, \ldots, L-1\}$, alors on a la forme suivante

$$D\ell(S_\theta(x_0))diag(f'(a_{L+1}))\theta_L Dg(\theta_{L-1}x_{L-1})\theta_{L-1}Dg(\theta_{L-2}x_{L-2})\cdots\theta_{l+1} \times Dg(\theta_l x_l)(x_l^T \otimes I_{n_{l+1}\times n_{l+1}}).f$$

Lorsqu'on passe à l'implémentation informatique, on veut réduire ces expressions afin d'obtenir une forme la plus compacte possible. On cherche entre autres à réduire chacune des matrices diagonales sous une forme vectorielle. De plus, on aimerait obtenir une équation récursive nous permettant d'obtenir la dérivée par rapport à chacune des matrices de poids θ_l, $l \in \{0, \ldots, L\}$, une à la suite de l'autre. La définition suivante va nous permettre de réduire nos deux expressions finales afin d'obtenir la forme compacte voulue.

Définition 5 : Si A et B sont deux matrices $m \times n$, alors le produit matriciel

d'Hadamar $A \odot B = (a_{ij}b_{ij})$ prend la forme suivante

$$A \odot B = \begin{bmatrix} a_{11}b_{11} & \dots & a_{1n}b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & \dots & a_{mn}b_{mn} \end{bmatrix} \tag{2.5}$$

La première chose que l'on va réduire est l'expression $D\ell(S_\theta(x_0))diag(f'(a_{L+1}))$, si on regarde cette expression, on remarque que l'on peut facilement l'a réécrire en utilisant le produit matriciel d'Hadamar sous la forme suivante

$$D\ell(S_\theta(x_0))diag(f'(a_{L+1})) = D\ell(S_\theta(x_0)) \odot (f'(a_{L+1}))^T.$$

Étant donné que l'on cherche à réduire chacune des formes où l'on a des matrices diagonales, on veut réduire $Dg(\theta_l x_l)$, $l \in \{0, \dots, L-1\}$ et $(x_l^T \otimes I_{n_{l+1} \times n_{l+1}})$, $l \in \{0, \dots, L\}$ à une forme vectorielle. Ainsi, on veut passer de $Dg(\theta_l x_l)$ à $g'(a_l)$ et de $(x_l^T \otimes I_{n_{l+1} \times n_{l+1}})$ à x_l. Pour ce faire, au lieu d'utiliser une forme complète pour le calcul des dérivées, on utilise une forme récursive. On pose

$$\delta^{(l)} = \begin{cases} \delta^{(l+1)}\theta_l \odot (g'(a_l))^T & \text{si } l = 1, ..., L \\ D\ell(S_\theta(x_0)) \odot (f'(a_{L+1}))^T & \text{si } l = L+1 \end{cases} \tag{2.6}$$

et on trouve chacune de nos dérivées sous une forme matricielle, en remarquant que l'on garde les mêmes propriétés que dans nos expressions complètes lorsque l'on utilise les résultats suivants ;

$$\begin{aligned} \tfrac{\partial J(\theta)}{\partial \theta_L} &= (x_L \delta^{(L+1)})^T \\ \tfrac{\partial J(\theta)}{\partial \theta_{L-1}} &= (x_{L-1} \delta^{(L)})^T \\ &\vdots \\ \tfrac{\partial J(\theta)}{\partial \theta_0} &= (x_0 \delta^{(1)})^T . \end{aligned}$$

De plus, ces dérivées sont de même dimension que les matrices $\theta_0, \dots \theta_L$.

2.5 Implémentation informatique

Dans la partie qui suit, on verra comment appliquer un réseau de neurones pour une tâche de classification ou de régression à partir d'opérations matricielles. Grâce à une série de multiplications matricielles, il sera possible de propager la matrice de variables expliquées X_0 directement dans le réseau afin d'obtenir les sorties associées à chacune des observations comprises dans X_0. Notre algorithme permet de calculer de façon matricielle la dérivée par rapport à chacun des poids compris à l'intérieur de $\theta = \{\theta_0, \ldots, \theta_L\}$. En utilisant une carte graphique pour les calculs des multiplications matricielles, on peut arriver à paralléliser un grand nombre de données et augmenter significativement la vitesse de calcul. Dans la plupart des librairies les plus utilisées pour les réseaux de neurones tels Tensorflow (Abadi *et al.*, 2015) développées par Google ou Theano (Theano Development Team, 2016) par l'Université de Montréal, on utilise l'implémentation informatique présentée dans cette section pour un réseau MLP.

2.5.1 Tâche de classification

Pour une tâche de classification, on a un vecteur de variables expliquées $y \in \mathbb{R}^K$ pour chaque observation où K représente le nombre de classes. Rappelons qu'une observation de la variable expliquée y prend la forme d'un vecteur où la $k^{\text{ème}}$ composante prend la valeur 1 si y correspond à la classe i. C'est-à-dire que y prend une forme telle que définie en 1.1. Pour l'implémentation informatique $Y \in \mathbb{R}^{N \times K}$ est défini comme étant une matrice dont chaque rangée correspond à un vecteur de variables expliquées $y^T \in \mathbb{R}^{1 \times K}$. En informatique, le terme associé à cette représentation est connu en anglais sous le nom "one-hot encoding". La matrice

Y prend la forme suivante :

$$Y = \begin{bmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_N^T \end{bmatrix}. \tag{2.7}$$

Tel que défini dans la partie inférence $S_\theta(x_0) \in \mathbb{R}^K$ représente la sortie obtenue par le réseau et correspond à notre prédiction. Lorsque la fonction softmax est utilisée pour faire le transfert entre les neurones de l'avant-dernière couche du réseau et la couche de sortie, chaque entrée $S_\theta(x_0)^{(k)}$ du vecteur $S_\theta(x_0)$ correspond à la probabilité estimée par le modèle que $y^{(k)} = 1$, c'est-à-dire que la $k^{\text{ème}}$ entrée du vecteur y appartienne à la classe k. Comme on a pu le voir dans la section 1, la fonction softmax permet d'écraser un vecteur de dimension K en un vecteur de probabibilité qui somme à 1. Ainsi, lorsque cette fonction est utilisée, on choisit un réseau avec K neurones pour la couche de sortie lorsque l'on a K différentes classes. Il est à noter que la fonction sigmoïde est souvent utilisée dans le cas où l'on a seulement deux classes et correspond à la deuxième façon de paramétriser la vraisemblance (avec η) dans la section 1. Si c'est la fonction sigmoïde qui est utilisée, on doit utiliser un seul neurone pour la couche de sortie. Comme lorsque l'on utilise la régression logistique, la sortie du réseau nous donne la probabilité que $y = 1$.

Rappelons que dans la section 2.1, on a utilisé les notations suivantes

$$n_0, n_1, \ldots, n_L, n_{L+1}$$

pour signifier le nombres de neurones respectif dans chacune des $L + 2$ couches du réseau allant de la couche d'entrée (la couche 0) jusqu'à la couche de sortie (la couche $L+1$). De plus, $\theta_l = (b_l | W_l)$, $l = 0, ..., L$ représente les matrices à l'intérieur desquelles chacun des poids $w_{ij}^{(l)}$ et $b_i^{(l)}$ est à estimer. Avec cette notation, on peut

maintenant écrire de façon explicite la fonction de perte entropie croisée «cross entropy» à laquelle s'ajoute le régularisateur pour la dernière couche du réseau. Dans ce cas, on a utilisé la norme L_2. On a

$$Q(\theta) = \underbrace{-\frac{1}{N}\sum_{j=1}^{N}\sum_{k=1}^{K}\mathbb{1}\left\{ y_j^{(k)} = k\right\} log(S_\theta(x_{0j})_k)}_{J(\theta)} + \underbrace{\frac{\lambda}{2N}\sum_{l=0}^{L}\sum_{i=1}^{n_l}\sum_{j=1}^{n_{l+1}}(w_{ij}^{(l)})^2}_{R(\theta)}$$

où $S_\theta(x_{0j})_k$ représente la $k^{\text{ème}}$ entrée du vecteur de sortie $S_\theta(x_{0j})$ pour la $j^{\text{ème}}$ observation dans le jeu d'entraînement. On dénote $y_j^{(k)}$ est la $k^{\text{ème}}$ entrée de la variable expliquée pour la $j^{\text{ème}}$ observation. La partie $J(\theta)$ vient du fait que maximiser la vraisemblance tel que définie dans la partie 1 revient en fait à minimiser la déviance qui correspondant à $-log$ de notre vraisemblance. Comme dans un modèle de régression logistique avec régularisation Tychonoff «ridge regression», on n'inclut pas le biais b_l pour le régularisateur $R(\theta)$.

Lorsque l'on implémente l'algorithme, on cherche à minimiser la fonction de perte $Q(\theta)$. L'objectif est de parvenir à calculer la fonction $Q(\theta)$ et par la suite chacune des dérivées partielles $\frac{\partial Q(\theta)}{\partial \theta_{ij}^{(l)}}$ où $\theta_{ij}^{(l)}$ représente l'entrée (i,j) de la matrice θ_l . Cette étape permettra de calculer le gradient $\nabla Q(\theta)$. Afin de simplifier le calcul du gradient, $Q(\theta)$ est séparé en deux parties, soit la fonction de perte à laquelle s'ajoute le régularisateur. On aura qu'à additionner les dérivées par rapport à chacune des deux parties afin d'obtenir le gradient.

$$Q(\theta) = J(\theta) + R(\theta)$$

Après avoir initialisé les poids $\theta = \{\theta_0, \ldots, \theta_L\}$, chaque observation x_{0i} dans l'ensemble d'entraînement est propagée à travers le réseau afin d'obtenir un vecteur de prédiction pour la sortie $S_\theta(x_{0i}) \in \mathbb{R}^K$. On prend X_0 la matrice correspondant à notre jeu de données où chacune des rangées représente un vecteur ligne x_{0i}^T, $i = 1, \ldots, N$. La matrice X_0 est transformée à travers $X_1, \ldots, X_{L+1}$ par une

fonction d'activation lors du passage vers les différentes couches supérieures du réseau. Une note sur la notation. On utilise dans cette section la même notation que dans la partie 2.4 mais avec les lettres majuscules puisque les transformations sont maintenant représentées dans le réseau sous forme de matrice. On fait la série d'opérations suivante pour obtenir les sorties $S_\theta(x_{0i})$ associées à chacune de nos observations x_{0i} sous forme matricielle. On désigne souvent cette série d'opérations comme étant l'étape de propagation.

$$
\begin{aligned}
A_1 &= X_0 W_0^T + b_0^T \\
X_1 &= g(A_1) \\
A_2 &= X_1 W_1^T + b_1^T \\
X_2 &= g(A_2) \\
&\vdots \\
A_{L+1} &= X_L W_L^T + b_L^T \\
X_{L+1} &= S_\theta(X_0) = f(A_{L+1})
\end{aligned}
$$

Pour une tâche de classification, f est habituellement choisie comme étant la fonction softmax et g est une fonction d'activation entre les neurones à déterminer. Lorsque l'on additione le vecteur b_l^T, on ne fait pas ici une opération matricielle, chacune des entrées du vecteur ligne b_l^T est additionnée à chacune des entrées des rangées de $X_l W_l^T$ une à la suite de l'autre afin de donner les biais respectifs.

Après avoir propagé chacune des observations dans le réseau, on est maintenant en mesure d'évaluer la fonction de perte car on connaît maintenant $S_\theta(X_0)$. On s'intéresse à calculer le gradient $\nabla Q(\theta)$. Rappelons que l'on a défini plus tôt $J(\theta)$ comme étant la première partie de la fonction de perte sans le régularisateur. Si on pose

$$
\delta^{(l)} = \begin{cases} \delta^{(l+1)} W_l \odot g'(A_l) & \text{si } l = 1, ..., L \\ X_{L+1} - Y & \text{si } l = L+1 \end{cases}
$$

où $\odot$ représente le produit élément par élément, on a presque qu'exactement la forme finale trouvée en 2.6 dans la partie 2.4 mais sous une forme matricielle. Avec l'aide de cette nouvelle définition et le vecteur ligne $v = [1, \ldots, 1]$ tel que $v \in \mathbb{R}^{1\times N}$, on arrive à une forme claire pour chacune de nos dérivées sous une forme matricielle. On a que

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial W_L} &= X_L^T \delta^{(L+1)}, & \frac{\partial J(\theta)}{\partial b_L} &= v\delta^{(L+1)} \\
\frac{\partial J(\theta)}{\partial W_{L-1}} &= X_{L-1}^T \delta^{(L)}, & \frac{\partial J(\theta)}{\partial b_{L-1}} &= v\delta^{(L)} \\
&\vdots & & \\
\frac{\partial J(\theta)}{\partial W_0} &= X_0^T \delta^{(1)}, & \frac{\partial J(\theta)}{\partial b_0} &= v\delta^{(1)}
\end{aligned}$$

Cette forme matricielle est encore une fois très similaire à celle trouvée dans la partie 2.4. Deux petites distinctions sont à faire ici. La première est en lien avec le vecteur de biais b_l. Étant donné que celui-ci n'est pas multiplié par sa matrice correspondante X_l, on obtient que notre dernière dérivée est égale à 1, d'où la multiplication avec le vecteur v. La deuxième différence à faire est au niveau de $\delta^{(L+1)}$. On a vu dans la section 2.4 que $\delta^{(L+1)} = D\ell(S_\theta(x_0)) \odot (f'(a_{L+1}))^T$. On a également vu que changer la fonction de perte pour le réseau influence seulement notre dérivée par rapport à θ_L à travers $\delta^{(L+1)}$. On aimerait donc vérifier que $\delta^{(L+1)} = X_{L+1} - Y$ lorsque l'on utilise la fonction softmax pour la sortie du réseau. Si on reprend notre petit exemple de la partie 2.4, on avait que pour la sortie du réseau $S_\theta(x_0) = f(\theta_1 g(\theta_0 x_0)) \in \mathbb{R}^K$. Rappelons, qu'on a choisi pour J la fonction de perte entropie croisée

$$J(\theta) = -\frac{1}{N}\sum_{j=1}^{N}\sum_{k=1}^{K} \mathbb{1}\left\{ y_j^{(k)} = k \right\} log(S_\theta(x_{0j})_k).$$

Notre première dérivée $D\ell(S_\theta(x_0))$ pour l'observation i est donnée par

$$\frac{\partial \ell}{\partial S_\theta(x_0)^T} = -\frac{1}{N}\left[\mathbb{1}\left\{ y_i^{(1)} = k \right\} \frac{1}{S_\theta(x_{01})_1}, \ldots, \mathbb{1}\left\{ y_i^{(K)} = K \right\} \frac{1}{S_\theta(x_{01})_K}\right] \in \mathbb{R}^{1\times K}.$$

Notre deuxième dérivée $Df(\theta_1 g(\theta_0 x_0))_{K\times K} = Df(a_2) = diag(f'(a_2))$. On va maintenant regarder la $i^{\text{ème}}$ entrée sur la diagonale de la matrice $Df(\theta_1 g(\theta_0 x_0))_{K\times K}$. Puisque l'on a choisit pour f la fonction de softmax, on a que

$$f(a_2^i) = \frac{exp(a_2^i)}{\sum_{j=1}^K exp(a_2^j)} = softmax(a_2^i) \qquad i \in \{1, \ldots, K\}$$

et par la régle de dérivation du quotient on a que

$$\begin{aligned} f'(a_2^i) &= \frac{exp(a_2^i)}{\sum_{j=1}^K exp(a_2^j)} - \frac{exp(a_2^i)exp(a_2^i)}{(\sum_{j=1}^K exp(a_2^j))^2} \\ &= softmax(a_2^i) - softmax(a_2^i)^2 \\ &= S_\theta(x_0)_i - S_\theta(x_0)_i^2 \end{aligned}$$

La multiplication $\delta^{(L+1)} = D\ell(S_\theta(x_0)) \odot (f'(a_{L+1}))^T$ nous donne ainsi un vecteur ligne de dimension $1 \times K$ dont la $i^{\text{ème}}$ composante est donnée par

$$-\mathbb{1}\left\{ y^{(i)} = k\right\} \frac{1}{S_\theta(x_0)_i}(S_\theta(x_0)_i - S_\theta(x_0)_i^2) = \mathbb{1}\left\{ y^{(i)} = k\right\}(S_\theta(x_0)_i - 1)$$

Puisque la matrice $\delta^{(L+1)} \in \mathbb{R}^{N\times K}$ est composée de chacun de ces vecteurs lignes, on voit que maintenant directement pourquoi $\delta^{(L+1)} = X_{L+1} - Y$ lorsque l'on choisit pour f la fonction de softmax.

En résumé, en calculant $\delta^{(l+1)}$ de façon récursive de la dernière couche vers la première, on obtient une à la suite de l'autre nos matrices $\frac{\partial J(\theta)}{\partial W_L}, \frac{\partial J(\theta)}{\partial W_{L-1}}, \ldots, \frac{\partial J(\theta)}{\partial W_0}$ et vecteurs $\frac{\partial J(\theta)}{\partial b_L}$, $\frac{\partial J(\theta)}{\partial b_{L-1}}, \ldots, \frac{\partial J(\theta)}{\partial b_0}$. Pour obtenir les dérivées par rapport aux vecteurs de biais, c'est-à-dire $\frac{\partial J(\theta)}{\partial b_L}$, $\frac{\partial J(\theta)}{\partial b_{L-1}}, \ldots, \frac{\partial J(\theta)}{\partial b_0}$, on somme les colonnes des matrices de $\delta^{(L+1)}, \delta^{(L)}, \ldots, \delta^{(1)}$ en multipliant ces matrices par le vecteur v.

Les entrées des matrices $\frac{\partial R(\theta)}{\partial W_L}, \frac{\partial R(\theta)}{\partial W_{L-1}}, ..., \frac{\partial R(\theta)}{\partial W_0}$ sont facilement obtenues en remarquant que la dérivée

$$\frac{\partial R(\theta)}{w_{ij}^{(l)}} = \frac{\lambda}{N} w_{ij}^{(l)}.$$

Ainsi, on a que

$$\frac{\partial R(\theta)}{\partial W_l} = \frac{\lambda}{N} W_l$$

et on trouve chacune de nos dérivées en additionnant

$$\frac{\partial Q(\theta)}{\partial W_l} = \frac{\partial J(\theta)}{\partial W_l} + \frac{\partial R(\theta)}{\partial W_l} \qquad l = 0, ..., L$$

Étant donné que $R(\theta)$ ne dépend pas des vecteurs biais, on a que

$$\frac{\partial Q(\theta)}{\partial b_l} = \frac{\partial J(\theta)}{\partial b_l} \qquad l = 0, ..., L$$

On a maintenant un algorithme récursif permettant de calculer les dérivées de façon matricielle pour chacune des matrices de poids et vecteurs de biais.

2.5.2 Tâche de régression

Pour une tâche de régression, les résultats sont facilement dérivables puisque l'on a un seul neurone dans la couche de sortie ($K = 1$). En pratique, on utilise souvent la perte quadratique à laquelle s'ajoute le régularisateur. Si on utilise encore une fois la norme L_2 pour le régularisateur, l'expression explicite pour la fonction de perte est la suivante :

$$Q(\theta) = \underbrace{\frac{1}{2N}\sum_{j=1}^{N}(y_i - S_\theta(x_{0i}))^2}_{J(\theta)} + \underbrace{\frac{\lambda}{2N}\sum_{l=0}^{L}\sum_{i=1}^{n_l}\sum_{j=1}^{n_{l+1}}(w_{ij}^{(l)})^2}_{R(\theta)}.$$

La perte quadratique est définie comme $\ell(y, S_\theta(x_0)) = \frac{1}{2}(y - S_\theta(x_0))^2$. Comme la perte est une fonction de $\ell : \mathbb{R} \to \mathbb{R}$, on a que $D\ell(S_\theta(x_0)) \in \mathbb{R}$. Si on regarde le cas d'un seul exemple, la première dérivée $D\ell(S_\theta(x_0))$ pour une certaine observation est donnée par

$$\frac{\partial \ell}{\partial S_\theta(x_0)} = -(y - S_\theta(x_0)) \in \mathbb{R}.$$

Puisque la sortie du réseau $S_\theta(x_0) = f(\theta_L g(\theta_{L-1} x_{L-1})) \in \mathbb{R}$, on a également que notre deuxième dérivée $Df(\theta_L g(\theta_{L-1} x_{L-1})) \in \mathbb{R}$. Habituellement, pour une tâche de régression on utilise soit f comme étant la fonction identé, soit comme étant la fonction de sigmoïde lorsque $y \in [0, 1]$. Si f est la fonction identitée, il est facile de

voir que le résultat pour chacune des dérivées est le même que pour une tâche de classification car $Df(\theta_1 g(\theta_0 x_0)) = 1$. Cependant, si on utilise la fonction sigmoïde, c'est-à-dire si $f(a_{L+1}) = \frac{1}{1+exp(-a_{L+1})}$, on a que $Df(a_{L+1}) = f(a_{L+1})(1 - f(a_{L+1}))$. Par conséquent, c'est seulement le résultat pour $\delta^{(L+1)}$ qui change, tout le reste de l'algorithme demeure inchangé. Ainsi on a

$$\delta^{(l)} = \begin{cases} \delta^{(l+1)} W_l \odot g'(A_l) & \text{si } l = 1, ..., L \\ -(Y - S_\theta(X_0)) \odot f(A_{L+1}) \odot (1 - f(A_{L+1})) & \text{si } l = L+1 \end{cases} \tag{2.8}$$

Pour une tâche de régression le vecteur colonne $Y \in \mathbb{R}^N$représente la variable expliquée. $S_\theta(X_0) \in \mathbb{R}^N$ est également un vecteur colonne et correspondant à notre prédiction. Dans l'équation 2.8, 1 représente un vecteur de 1 appartenant à $\mathbb{R}^N$.

2.5.3 Régularisation Dropout

Développée en partie par Geoffrey Hinton (Hinton *et al.*, 2014), Dropout est une nouvelle méthode de régularisation qui a permis d'améliorer considérablement les résultats et la performance des réseaux de neurones. Les trois prochains paragraphes porteront sur les grandes idées motivant cette technique de régularisation.

Les réseaux de neurones sont particulièrement sujets à deux types de problèmes. Premièrement, lorsque l'architecture d'un réseau devient complexe comme c'est le cas avec les réseaux de type profond «Deep», c'est-à-dire lorsque l'on a plusieurs couches cachées, le nombre de paramètres peut devenir grand, faisant en sorte qu'il peut être long pour l'ordinateur d'estimer les paramètres d'un modèle. On aimerait donc ne pas avoir à réestimer un modèle plusieurs fois.

Lorsque l'on estime un modèle sur un ensemble de données, un danger potentiel est de surestimer les paramètres d'un modèle. Plus particulièrement lorsqu'on a peu de données, un réseau de neurones pourrait apprendre à travers ses couches cachées

certaines relations complexes entre les entrées et les sorties du réseau pouvant être dues seulement à un bruit lié à l'échantillonnage des données. Ce problème est connu sous le nom de surajustement. Plusieurs techniques ont été développées afin de réduire le surajustement et d'améliorer les prévisions sur un jeu de données à l'extérieur de l'ensemble d'entraînement «out-sample». En particulier, l'ajout d'un régularisateur au modèle ou la technique d'arrêt forcé telle que décrite dans la partie 2.3.2.

Les meilleurs modèles, c'est-à-dire ceux qui ont permis de gagner les grandes compétitions en apprentissage automatique utilisent pour la plupart du temps une méthode qu'on appelle une moyenne de différents modèles «model averaging». La méthode consiste à utiliser une moyenne arithmétique ou géométrique de plusieurs différents modèles afin de faire une prévision. L'idée derrière Dropout est de laisser tomber aléatoirement certaines unités (neurones) à travers le réseau afin d'éviter une trop grande coadaptation entre ceux-ci en utilisant une moyenne d'un nombre exponentiel de modèles possibles. On utilise cette moyenne pour faire les prédictions. La coadaptation entre les unités intervient lorsque certains neurones comptent en trop grande partie sur la sortie de certains autres. En laissant tomber aléatoirement certains neurones, on force les neurones à compter sur la population de neurones qui affectent directement leur comportement. Cette procédure permet d'éviter le surajustement. Bien que l'idée paraisse complexe à première vue, l'implémentation est relativement simple.

On va maintenant présenter le modèle. Rappelons que dans un réseau où l'on a L couches cachées, lorsque l'on propage l'entrant x_0 de façon vectorielle à travers le réseau, on a les étapes suivantes

$$a_1 = \theta_0 x_0$$
$$x_1 = g(a_1)$$

$$
\begin{aligned}
a_2 &= \theta_1 x_1 \\
x_2 &= g(a_2) \\
&\vdots \\
a_{L+1} &= \theta_L x_L \\
x_{L+1} &= S_\theta(x_0) = f(a_{L+1})
\end{aligned}
$$

Dropout agit en retenant chaque unité de façon indépendante avec une probabilité p. On définit $r_0, \ldots, r_L$ de même dimension que $x_0, \ldots, x_L$, c'est-à-dire de dimension $n_0, \ldots, n_L$, comme étant des vecteurs où chaque entrée $r_l^{(i)}, l \in \{0, \ldots, L\}$ suit une loi de Bernoulli de paramètre p_l ; $r_l^{(i)} \sim Bernoulli(p_l)$. Dans la littérature, on appelle souvent ces vecteurs les masques «masks» qui multiplient les vecteurs de sorties de chacune des couches du réseau. Nos étapes lorsque l'on propage l'entrant x_0 à travers le réseau deviennent

$$
\begin{aligned}
a_1 &= \theta_0 \tilde{x}_0 = \theta_0 (r_0 \odot x_0) \\
x_1 &= g(a_1) \\
a_2 &= \theta_1 \tilde{x}_1 = \theta_1 (r_1 \odot x_1) \\
x_2 &= g(a_2) \\
&\vdots \\
a_{L+1} &= \theta_L \tilde{x}_L = \theta_L (r_L \odot x_L) \\
x_{L+1} &= S_\theta(x_0) = f(a_{L+1})
\end{aligned}
$$

où $\odot$, à titre de rappel, représente le produit d'Hadamar défini en 2.5. Plusieurs tests tendent à montrer que pour un grand nombre d'applications, la probabilité optimale semble être 0.5 pour les couches cachées et 0.2 pour la couche d'entrée, c'est-à-dire, $p_l = 0.2$ si $l = 0$ et $p_l = 0.5$ si $l \in \{1, \ldots, L\}$ (Hinton *et al.*, 2014). On dit que l'on échantillonne de différents modèles étant donné que pour chaque observation $x_{0j} \quad j = 1, \ldots, N$, comprise dans le jeu de données, on peut avoir

une architecture différente. À chaque fois qu'on propage une observation dans le réseau, on échantillonne une différente série de masques $r_0, \ldots, r_L$.

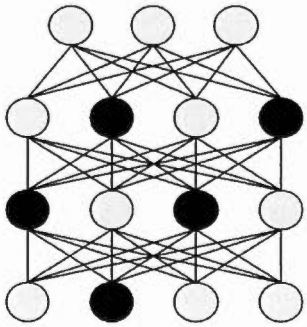

Figure 2.2: Exemple d'architecture où l'on applique la régularisation Dropout sur la couche d'entrée et les couchée cachées pour un réseau de neurones MLP.

Si notre réseau comprend n unités (on n'inclut pas ici la couche de sortie) et que $p \in]0, 1[$, on échantillonne à travers 2^n architectures différentes puisque l'on a deux possibilités pour chaque unité comprise dans le réseau, soit l'unité est présente soit elle ne l'est pas. On peut également voir les choses selon une approche plus combinatoire. Supposons que notre réseau comprend n unités et que l'on a à choisir k neurones parmi ceux-ci, $k \in \{0, \ldots, n\}$ alors on a $\binom{n}{k}$ possibilités. On arrive au même résultat car

$$2^n = \sum_{k=0}^{n} \binom{n}{k}.$$

Pour le calcul du gradient, rien ne change réellement avec Dropout. Notre fonction de perte est représentée sous forme d'une sommation, notre gradient est en fait une sommation de dérivées par rapport à l'ensemble de paramètres compris dans le réseau. Si on regarde une observation, disons x_0 à l'intérieur d'un jeu de données, Dropout attribuera une architecture quelconque à cet exemple parmi les 2^n possibilités. Ainsi, la série de transformations qui transforme l'entrant x_0 en

sa sortie $S_\theta(x_0)$ sera unique à l'architecture échantillonnée. On peut réécrire les transformations de sorte que l'on ait une structure très similaire à notre modèle antérieur excepté que l'on a maintenant une fonction d'activation ϕ_l, $l \in \{1, \ldots, L\}$, distincte et dépendante du masque r_l échantillonné. C'est-à-dire ;

$$
\begin{aligned}
a_1 &= \theta_0 \tilde{x}_0 = (r_0 \odot x_0) \\
\tilde{x}_1 &= \phi_1(a_1) = (r_1 \odot x_1) = (r_1 \odot g(a_1)) \\
a_2 &= \theta_1 \tilde{x}_1 \\
\tilde{x}_2 &= \phi_2(a_2) = (r_2 \odot x_2) = (r_2 \odot g(a_2)) \\
&\vdots \\
\tilde{x}_L &= \phi_L(a_L) = (r_L \odot x_L) = (r_L \odot g(a_L)) \\
a_{L+1} &= \theta_L \tilde{x}_L \\
x_{L+1} &= S_\theta(x_0) = f(a_{L+1})
\end{aligned}
$$

De cette façon, on voit directement $\tilde{x}_0$ comme un vecteur d'entrant et $\tilde{x}_1, \ldots, \tilde{x}_L$ comme les vecteurs transformés par nos fonctions d'activation distinctes. Puisqu'on a la même forme qu'auparavant, on a le même résultat pour nos dérivées par rapport à chacun de nos paramètres excepté que l'on multiplie par $\tilde{x}_l$, $l \in \{0, \ldots, L\}$. C'est-à-dire

$$
\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_L} &= (\tilde{x}_L \delta^{(L+1)})^T \\
\frac{\partial J(\theta)}{\partial \theta_{L-1}} &= (\tilde{x}_{L-1} \delta^{(L)})^T \\
&\vdots \\
\frac{\partial J(\theta)}{\partial \theta_0} &= (\tilde{x}_0 \delta^{(1)})^T
\end{aligned}
$$

et notre forme récursive devient

$$
\delta^{(l)} = \begin{cases} \delta^{(l+1)} \theta_l \odot (\phi_l'(a_l))^T = \delta^{(l+1)} \theta_l \odot (r_l \odot g'(a_l))^T & \text{si } l = 1, ..., L \\ D\ell(S_\theta(x_0)) \odot (f'(a_{L+1}))^T & \text{si } l = L+1 \end{cases}
$$

Maintenant qu'on connait comment propager un entrant dans le réseau et entraîner notre modèle avec Dropout, on aimerait savoir comment on fera pour prédire la sortie du modèle au temps test «test time». On définit le temps test comme l'étape où l'on évalue nos résultats. Au temps test, on peut par exemple évaluer la fonction de perte sur l'ensemble d'entraînement (et/ou) sur un ensemble validation. Cette étape est souvent faite après une itération de gradient ou lorsqu'on veut évaluer la performance d'un modèle. Lorsqu'on évalue la performance, on prend souvent en considération le taux de bonne classification sur un ensemble de données validation ou un ensemble de données test ainsi que la perte.

Le problème retrouvé au temps test est qu'il est impossible de faire une prédiction en utilisant une moyenne des prédictions de tous les modèles échantillonnés lorsque l'on a entraîné notre modèle. Le nombre d'architectures étant beaucoup trop grand pour tout les considérer. Puisque l'on échantillonne d'un très grand nombre d'architectures lors de l'estimation des paramètres, on utilise l'espérance de notre modèle qui comprend maintenant une partie aléatoire pour approximer la moyenne des prédictions des différents modèles échantillonnés.

On peut calculer notre espérance de la sortie de notre modèle de deux façons, la première est l'approche probabiliste et la deuxième est l'approche combinatoire.

Débutons par un exemple simple afin d'avoir une certaine intuition de comment on arrive au calcul de la moyenne avec l'approche probabiliste. Supposons qu'on a une structure de réseau représentée par l'architecture suivante :

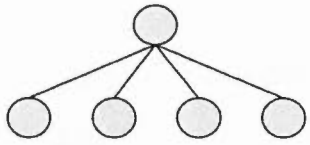

Figure 2.3: Exemple d'architecture simple (régression linéaire) avec un output $x_1 \in \mathbb{R}$

Si on prend le vecteur r_0 de dimension n_0, le vecteur qui multiplie notre couche d'entrée pour le réseau, on a que la $i^{\text{ème}}$ composante du vecteur r_0, c'est-à-dire $r_0^{(i)} \sim Bernoulli(p_0)$. On cherche d'abord l'architecture la plus simple possible. On donc la fonction identité, i.e $f(x) = x$ pour le transfert des neurones de la couche d'entrée vers la couche de sortie. Dans ce cas, notre réseau correspond exactement à une régression linéaire. Avec n_0 unités pour la couche d'entrant, si on garde la même notation que dans la présentation du modèle Dropout, on a que a_1 est la sortie associée à la transformation linéaire de x_0 par la matrice de poids $\theta_0 \in \mathbb{R}^{1 \times n_0}$, c'est-à-dire la transformation $\theta_0 x_0$. Pour la suite des choses, on définit $a_0 = x_0$. Puisque notre fonction de transfert est l'identité, on a que $x_1 = a_1$. La sortie de notre réseau est donc donnée par

$$a_1 = \sum_{j=1}^{n_0} \theta_{1j}^0 \tilde{x}_0^{(j)} = \sum_{j=1}^{n_0} \theta_{1j}^0 r_0^{(j)} x_0^{(j)}.$$

Il est facile de calculer notre espérance car on a un seul output $a_1 \in \mathbb{R}$. L'espérance est

$$E[a_1] = \sum_{j=1}^{n_0} E[\theta_{1j}^0 r_0^{(j)} x_0^{(j)}] = \sum_{j=1}^{n_0} \theta_{1j}^0 p_0 E[x_0^{(j)}] = \sum_{j=1}^{n_0} \theta_{1j}^0 p_0 x_0^{(j)}.$$

Supposons qu'on passe maintenant un réseau avec K classes distinctes et aucune couche cachée sans toutefois appliquer une fonction de transfert, on a maintenant $a_1 \in \mathbb{R}^K$ et on a la matrice $\theta_0 \in \mathbb{R}^{K \times n_0}$ pour faire le transfert entre notre couche d'entrée et de sortie. L'architecture devient similaire à celle représentée dans la

figure 1.3. On cherche maintenant à calculer l'espérance du vecteur a_1. On s'intéresse donc à l'espérance de chacune des entrées du vecteur a_1. On trouve que l'espérance pour la $i^{\text{ème}}$ entrée du vecteur a_1 est donnée par

$$E[a_1^{(i)}] = \sum_{j=1}^{n_0} \theta_{ij}^{(0)} p_0 E[x_0^{(j)}] = \sum_{j=1}^{n_0} \theta_{ij}^{(0)} p_0 x_0^{(j)}.$$

On va maintenant voir le cas plus général où l'on a un nombre quelconque de couches cachées, c'est-à-dire lorsque $L \geq 1$. On obtient que l'espérance en partant de l'unité i dans la couche l, $l \in \{1, \ldots, L+1\}$ est donnée par la forme récursive suivante

$$E[a_l^{(i)}] = \sum_{j=1}^{n_{l-1}} \theta_{ij}^{(l-1)} p_{l-1} E[a_{l-1}^{(j)}]$$

où $E[a_0^{(j)}] = E[x_0^{(j)}] = x_0^{(j)}$. On conclut que si notre fonction d'activation entre chacune des couches pour le réseau est l'identité, on peut trouver facilement l'espérance de la sortie du réseau en multipliant chacune des matrices de poids θ_l, $l \in \{0, \ldots, L\}$, par la probabilité à laquelle on retient nos entrants entre nos différentes couches, c'est-à-dire par p_l, $l \in \{0, \ldots, L\}$. La figure 2.4, similaire et tirée de la figure 2 de l'article (Hinton *et al.*, 2014), montre bien comment on applique Dropout à un réseau lorsque l'on entraîne notre modèle et lorsque l'on évalue nos résultats au temps test.

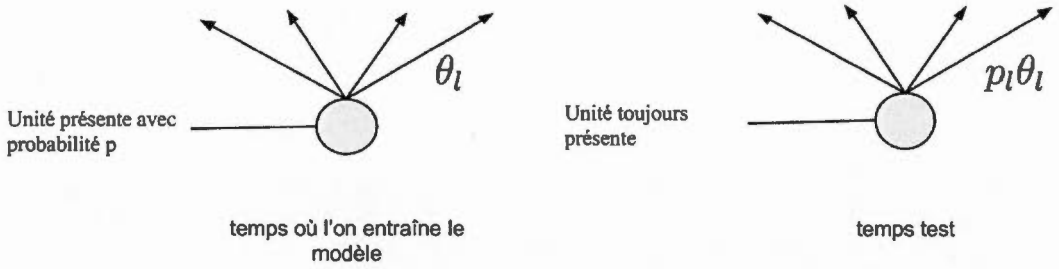

Figure 2.4: À gauche, on a une unité présente avec une probabilité p_l connectée aux unités de la couche supérieure par un vecteur de poids θ_l. À droite, on a qu'au temps test l'unité est toujours présente, mais on multiplie les poids connectés à la couche supérieure par p_l. L'espérance de la sortie du réseau au temps test est la même que l'espérance de la sortie lorsque l'on entraîne le modèle.

En pratique lorsque l'on implémente Dropout au temps test, on multiplie chacun des poids sortants d'une unité par la probabilité que l'unité soit retenue. On vient de voir que si notre fonction d'activation est l'identité cette pratique revient à faire l'espérance.

Toujours en partant avec notre exemple simple de régression linéaire, on va maintenant passer à l'approche combinatoire. Rappelons que si $p \in]0,1[$, on a la possibité d'échantillonner d'une structure avec k unités parmi n unités, $k \in \{0,\ldots,n\}$. Ainsi, on cherche l'espérance pour la sortie de notre réseau, c'est-à-dire $E[a_1]$. Si $B = \{B_1,\ldots,B_{2^n}\}$ est l'ensemble de tous les possibilités d'architectures et que x_0 est un entrant fixé, alors l'espérance de la sortie de notre réseau est donnée par

$$E[a_1] = \sum_{i \in B} P(B_i) a_1(B_i, x_0).$$

Dépendamment des k unités choisies lors de l'échantillonnage, on a une sortie différente pour le réseau. On va donc énumérer pour chacune des $\binom{n}{k}$ possibilités, les sorties correspondantes, c'est-à-dire

$$\binom{n}{0} \to 0 \quad \binom{n}{1} \to \sum_{j=1}^{n_0}(\theta_{1j}^0 x_0^{(j)}) \quad \binom{n}{2} \to$$
$$\sum_{1\leq i<j\leq n_0}(\theta_{1i}^0 x_0^{(i)} + \theta_{1j}^0 x_0^{(j)}) \quad ,\ldots, \quad \binom{n}{n} \to (\sum_{j=1}^{n_0} \theta_{1j}^0 x_0^{(j)})$$

Pour cet exemple, on va montrer que si $p_0 = 0.5$, calculer l'espérance revient également à multiplier chacun des poids sortants d'une unité par la probabilité que l'unité soit retenue. Bien que calculer l'espérance avec l'approche combinatoire revient au même qu'avec l'approche probabiliste, il est beaucoup plus facile d'utiliser l'approche probabiliste. Cependant, l'approche combinatoire est importante pour la suite des choses. La raison pour laquelle on utilise $p_0 = 0.5$ est que cette probabilité rend chacune des architectures possibles d'échantillonner équiprobables. En faisant de la sorte, on a que $P(B_i) = \frac{1}{2^n}$ et l'on peut factoriser le terme dans l'espérance. Cette propriété nous permettra de calculer facilement l'espérance.

$$E[a_1] = \frac{1}{2^n}\sum_{i\in B} a_1(B_i, x_0).$$

Afin de calculer $\sum_{i\in B} a_1(B_i, x_0)$, on va dénombrer le nombre de fois que chacun des termes $\theta_{1i}^0 x_0^{(i)}$ apparait dans chacune de nos sommations en incluant le 0, on trouve que pour chacune des sommes correspondantes, on dénombre

$$0 \to 0, \quad \sum_{j=1}^{n_0}(\theta_{1j}^0 x_0^{(j)}) \to \binom{n-1}{0}, \quad \sum_{1\leq i<j\leq n_0}(\theta_{1i}^0 x_0^{(i)} + \theta_{1j}^0 x_0^{(j)}) \to$$
$$\binom{n-1}{1} \quad ,\ldots, \quad (\sum_{j=1}^{n_0} \theta_{1j}^0 x_0^{(j)}) \to \binom{n-1}{n-1}$$

Puisque

$$2^{n-1} = \sum_{k=0}^{n-1}\binom{n-1}{k}$$

On trouve que

$$E[a_1] = \frac{2^{n-1}}{2^n}\sum_{j=1}^{n_0}(\theta_{1j}^0 x_0^{(j)})$$
$$= \frac{1}{2}\sum_{j=1}^{n_0}(\theta_{1j}^0 x_0^{(j)})$$

et l'on remarque encore une fois que l'on multiplie chacun des poids sortants d'une unité par la probabilité $p_0 = \frac{1}{2}$ que l'unité soit retenue.

Dans le cas où la fonction d'activation entre les couches de notre réseau est la fonction logistique nous verrons que cette pratique revient à faire une moyenne géométrique normalisée pondérée que l'on notera $NWGM$ pour «normalized weighted geometric mean» afin de suivre la même notation que les articles (Baldi et Sadowski, 2013) et (Baldi et Sadowski, 2014), ayant étudié théoriquement Dropout. On utilise en pratique la $NWGM$ afin de faire une approximation notre espérance.

On va maintenant pas à pas complexifier notre architecture afin d'arriver à une structure avec plusieurs couches cachées où l'on a une fonction de transfert entre les différentes couches. On passe maintenant à une architecture similaire à celle de la figure 1.3, i.e de type régression multinomiale, mais cette fois avec fonction d'activation logistique entre la couche d'entrée et la couche de sortie. La fonction d'activation logistique a la forme suivante

$$\sigma(x) = \frac{1}{1 + ce^{-\lambda x}}. \tag{2.9}$$

Soit $B = \{B_1, \ldots, B_{2^n}\}$, l'ensemble de toutes les possibilités d'architectures et x_0 un entrant fixé, la moyenne géométrique pondérée des sorties du réseau est donnée par

$$G = \prod_{i \in B} \sigma(a_1(B_i, x_0))^{P(B_i)}$$

où $P(B_i)$ représente la distribution de probabilités associée à chacune des différentes architectures. Si par exemple, chacune de nos de 2^n architectures sont équiprobables, on a que notre moyenne géométrique est donnée par

$$G = \prod_{i \in B} \sigma(a_1(B_i, x_0))^{\frac{1}{2^n}}.$$

Puisque l'on modélise une structure avec fonction d'activation logistique, la sortie de notre unité modélise la contribution de l'unité et celle de son complémentaire,

nous donnant un nombre compris dans l'intervalle $[0, 1]$. La moyenne géométrique pondérée de la sortie complémentaire est donnée par

$$G^c = \prod_{i \in B} [1 - \sigma(a_1(B_i, x_0))]^{P(B_i)}.$$

On va maintenant utiliser l'approximation de notre moyenne, la $NWGM$ définie comme étant

$$NWGM = \frac{G}{G + G^c}$$

Ainsi, on trouve que

$$NWGM = \frac{\prod_{i \in B} \sigma(a_1(B_i, x_0))^{P(B_i)}}{\prod_{i \in B} \sigma(a_1(B_i, x_0))^{P(B_i)} + \prod_{i \in B} [1 - \sigma(a_1(B_i, x_0))]^{P(B_i)}} \tag{2.10}$$

$$= \frac{1}{1 + \prod_{i \in B} \left[\frac{1 - \sigma(a_1(B_i, x_0))}{\sigma(a_1(B_i, x_0))}\right]^{P(B_i)}} \tag{2.11}$$

Pour la fonction logistique, on a l'identité suivante

$$\frac{1 - \sigma(x)}{\sigma(x)} = ce^{-\lambda x}$$

Si on utilise cette identité dans l'équation 2.10, on obtient que

$$\begin{aligned} NWGM &= \frac{1}{1 + \prod_{i \in B} \left[ce^{-\lambda a_1(B_i, x_0)}\right]^{P(B_i)}} \\ &= \frac{1}{1 + ce^{-\lambda \sum_{i \in B} a_1(B_i, x_0) P(B_i)}} \\ &= \frac{1}{1 + ce^{-\lambda E[a_1]}} \end{aligned}$$

et l'on remarque que l'on peut calculer notre $NWGM$ facilement puisque l'on a que notre $NWGM = \sigma(E[a_1])$. Calculer la $NWGM$ dans le cas où la fonction d'activation est la fonction logistique revient donc également à multiplier chacun des poids sortants d'une unité par la probabilité p_0 que l'unité soit retenue.

On a vu que dans la partie 1 que l'on utilise la fonction softmax afin de normaliser la sortie associée à K unités exponentielles de sorte que l'on ait la distribution

de probabilités correspondantes. Lorsque l'on multiplie chacun des poids sortants d'une unité par la probabilité qu'elle soit retenue et que la fonction softmax est utilisée, on effectue la moyenne géométrique normalisée pondérée. On aimerait donc vérifier que la fonction softmax satisfait les mêmes propriétés que la fonction logistique dans le cas où l'on a une architecture simple, c'est-à-dire que

$$NWGM = softmax(E[a_1]).$$

Avec $K > 1$ classes, $a_1 \in \mathbb{R}^K$. Pour une architecture B_i fixée et un entrant x_0 fixé, on a que pour pour la classe k, notre fonction softmax est donnée par

$$softmax(a_1^{(k)}(B_i, x_0)) = \frac{exp(\lambda a_1^{(k)}(B_i, x_0))}{\sum_{j=1}^{K} exp(\lambda a_1^{(j)}(B_i, x_0))}. \tag{2.12}$$

Si on effectue encore la moyenne géométrique pour la classe k , on obtient que celle-ci est donnée par

$$G = \prod_{i \in B} softmax(a_1^{(k)}(B_i, x_0))^{P(B_i)}.$$

Pour la sortie complémentaire on a que

$$G^c = \sum_{j=1, j \neq k}^{K} \prod_{i \in B} softmax(a_1^{(j)}(B_i, x_0))^{P(B_i)}.$$

Ainsi notre $NWGM$ pour l'unité k est donnée par

$$\begin{aligned}
NWGM &= \frac{G}{G+G^c} \\
&= \frac{\prod_{i\in B} softmax(a_1^{(k)}(B_i,x_0))^{P(B_i)}}{\sum_{l=1}^{K}\prod_{i\in B} softmax(a_1^{(l)}(B_i,x_0))^{P(B_i)}} \\
&= \frac{\prod_{i\in B}\left[\frac{exp(\lambda a_1^{(k)}(B_i,x_0))}{\sum_{j=1}^{K} exp(\lambda a_1^{(j)}(B_i,x_0))}\right]^{P(B_i)}}{\sum_{l=1}^{K}\prod_{i\in B}\left[\frac{exp(\lambda a_1^{(l)}(B_i,x_0))}{\sum_{j=1}^{K} exp(\lambda a_1^{(j)}(B_i,x_0))}\right]^{P(B_i)}} \\
&= \frac{1}{1+\sum_{l=1,l\neq k}^{K}\prod_{i\in B}\left[\frac{exp\left(\lambda a_1^{(l)}(B_i,x_0)\right)}{exp\left(\lambda a_1^{(k)}(B_i,x_0)\right)}\right]^{P(B_i)}} \\
&= \frac{1}{1+\sum_{l=1,l\neq k}^{K}\left[\frac{exp\left(\lambda\sum_{i\in B} a_1^{(l)}(B_i,x_0)P(B_i)\right)}{exp\left(\lambda\sum_{i\in B} a_1^{(k)}(B_i,x_0)P(B_i)\right)}\right]} \\
&= \frac{exp\left(\lambda\sum_{i\in B} a_1^{(k)}(B_i,x_0)P(B_i)\right)}{\sum_{l=1}^{K} exp\left(\lambda\sum_{i\in B} a_1^{(l)}(B_i,x_0)P(B_i)\right)} \\
&= \frac{exp\left(\lambda E[a_1^{(k)}]\right)}{\sum_{l=1}^{K} exp\left(\lambda E[a_1^{(l)}]\right)}.
\end{aligned}$$

On voit qu'on a bien la propriété voulue, car $NWGM = softmax(E[a_1])$. Si notre fonction d'activation est la fonction softmax, on voit qu'on a encore qu'à multiplier chacun des poids sortants d'une unité par la probabilité p_0 que l'unité soit retenue pour obtenir l'approximation de notre espérance. Nous avons vu que la fonction softmax s'exprime sous la forme 2.12 que l'on peut également exprimer de la façon suivante :

$$softmax(a_1^{(k)}(B_i,x_0)) = \frac{1}{1+exp(-\lambda a_1^{(k)}(B_i,x_0))\sum_{j=1,l\neq k} exp(\lambda a_1^{(j)}(B_i,x_0))}.$$

Si on prend

$$c = \sum_{j=1,l\neq k} exp(\lambda a_1^{(j)}(B_i,x_0))$$

on voit que l'on se ramène à la forme exprimée en 2.9. La fonction softmax est donc un cas particulier de notre fonction logistique. Pierre Baldi et Peter Sadowski

(Baldi et Sadowski, 2014), ont démontré que la classe de fonction qui satisfait la propriété suivante

$$\frac{G}{G+G^c}f(x) = f(E[x]) \tag{2.13}$$

comprend la fonction constante $f(x) = a$ si $0 \leq a \leq 1$ et tous les fonctions logistiques tel qu'exprimé par la forme 2.9. Si on a une architecture de type profonde avec plusieurs couches cachées et que l'on cherche l'espérance du vecteur de sorties de notre réseau, il est facile de voir que l'on peut appliquer récursivement les formules trouvées pour calculer notre espérance. Bien que la $NWGM$ soit utile pour approximer notre espérance dans le cas où l'on a une fonction logistique ou softmax, la propriété que l'on recherche pour une fonction d'activation est la suivante. On veut que

$$E[f(x)] \approx f(E[x]). \tag{2.14}$$

La fonction d'activation ReLu «rectified linear unit», étant l'une des fonctions d'activation la plus couramment utilisée à l'heure actuelle, en partie puisqu'elle permet d'adresser le problème de fuite du gradient «vanishing gradient problem» (Pascanu *et al.*, 2012) par conséquent d'arriver à une convergence plus rapide des paramètres, satisfait cette propriété.

CHAPITRE III

RÉSEAU DE NEURONES À CONVOLUTION

3.1 Types d'image

Dans le cadre de ce mémoire, le réseau de neurones à convolution sera utilisé pour des applications de classification d'images. On va donc d'abord définir trois types d'images. C'est-à-dire, l'image booléenne, l'image à intensité de gris et l'image RVB. Les deux derniers formats d'images que l'on définit dans cette section sont couramment utilisés en classification d'images par ordinateur.

3.1.1 Images booléennes «boolean images»

L'image booléenne est la version la plus simple d'un format qu'une image puisse prendre pour l'ordinateur. Chacun des pixels de la représentation matricielle de l'image est compris dans l'ensemble $\{0, 1\}$. Le 0 représente la couleur d'intensité minimale ; le noir, et le 1 représente la couleur d'intensité maximale ; le blanc. On voit sur la figure 3.1 une image booléenne et sa représentation matricielle.

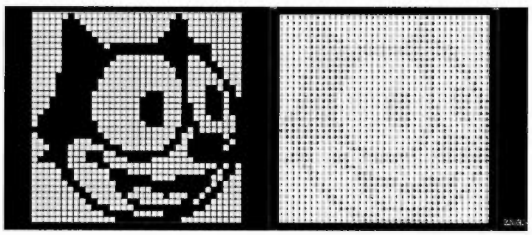

Figure 3.1: Exemple d'une image booléenne de Félix le chat exprimée sous la forme d'une matrice binaire 35×35.

3.1.2 Images à intensités de gris «grayscale images»

Les images à intensités de gris sont aussi représentées de façon matricielle. Chacune des entrées de la représentation matricielle de l'image correspond à un nombre entier compris dans l'ensemble $\{0, \ldots, 255\}$. L'intensité maximale 255 est le blanc et l'intensité minimale 0 est le noir. Les nombres compris entre 0 et 255 sont des intensités intermédiaires de gris allant de l'intensité minimale vers l'intensité maximale.

Figure 3.2: Représentation du canal bleu d'une image de piments de dimension $512 \times 512 \times 3$ en intensité de gris

3.1.3 Images RVB «RGB images»

Une image RVB, rouge vert bleu, peut être représentée sous la forme de trois matrices de même dimension, mais d'intensités distinctes du même type de format

que pour les intensités de gris 3.1.2. Chacune des trois matrices représente une intensité respective distincte de rouge vert et bleu. On a un total de 256^3 couleurs différentes possible pour un pixel compris dans l'image selon l'intensité choisie pour chacune des trois couleurs puisque l'on a 256 choix pour chacune des intensités. L'image suivante représente la même image de piments dans un système RVB.

Figure 3.3: Représentation de l'image piments de dimension $512 \times 512 \times 3$ dans sa représentation RVB

L'image qui suit représente la décomposition de notre volume de sorte que l'on puisse voir sa représentation dans chacune des couleurs.

Figure 3.4: Représentation de l'image piments de dimension $512 \times 512 \times 3$ pour sa représentation dans chacun des canaux respectifs, rouge, vert et bleu

3.2 Introduction au modèle

Lorsque l'on implémente un réseau de neurones de MLP sur le MNIST dans la section 4, les images comprises dans notre jeu de données sont décomposées sous une forme vectorielle faisant en sorte que le modèle ne puisse exploiter les caractéristiques propres à certaines régions comprises dans l'image. Le réseau de neurones à convolution peut tirer profit d'une topologie de type grille pour l'entrant comme c'est le cas pour la représentation matricielle d'une image (Goodfellow *et al.*, 2016). Le terme convolution vient du fait que le réseau utilise la convolution au sens mathématique lorsque l'on propage les entrants à travers celui-ci. Nous verrons dans ce chapitre quel type de convolution est utilisé en pratique.

3.2.1 Distinction avec le réseau MLP

Pour un réseau de neurones MLP, on a pu voir que notre réseau transforme chacun des entrants compris dans notre jeu de données exprimés sous forme vectorielle à travers les neurones des couches cachées du réseau. Les différentes transformations faites dans le réseau s'expriment également sous forme vectorielle. Dans le cas d'un réseau convolutif, les couches en partant de la couche d'entrée jusqu'à la couche de sortie sont représentées sous forme de volumes de largeur et hauteur généralement égales. Pour un réseau MLP, les neurones d'une couche supérieure à la couche d'entrants sont entièrement connectés «Fully-Connected» aux neurones de sa couche antérieure, c'est-à-dire que l'on a des connexions ou des poids distincts avec chacun des entrants compris dans une couche précédente. On verra que ce n'est pas toujours le cas pour un réseau de neurones à convolution. Du fait que les neurones d'une couche ne sont pas toujours entièrement connectés avec une couche antérieure, on aura ce qu'on appelle des poids partagés «shared weights». On verra qu'à travers une épaisseur d'une couche de convolution, chaque neurone partage les mêmes poids.

3.3 Architecture du modèle

On utilise habituellement trois types de couches pour former un réseau convolutif. Celui-ci est composé de couche(s) de convolution (CONV), de couche(s) de pooling (POOL) et de couche(s) entièrement connectées (FC). La ou les couches entièrement connectées sont souvent utilisées pour la sortie du réseau. Habituellement, une couche de convolution est suivie d'une fonction d'activation puis d'une couche de pooling, cette séquence peut être répétée plusieurs fois jusqu'à la couche entièrement connectée pour former un réseau convolutif que l'on dénote souvent sous la notation CONVNET. Notre modèle aura donc une architecture de type ENTRANT-CONV-ACTIVATION-POOL-FC où les trois étapes intermédiaires CONV-ACTIVATION-POOL peuvent se répéter à plusieurs reprises avant que l'on arrive à une couche entièrement connectée FC. Il est commun d'utiliser plus qu'une couche entièrement connectée FC avant la sortie du réseau. L'image suivante tirée de (Stanford, 2015) montre un type d'architecture permettant de classifier 5 différents types d'images.

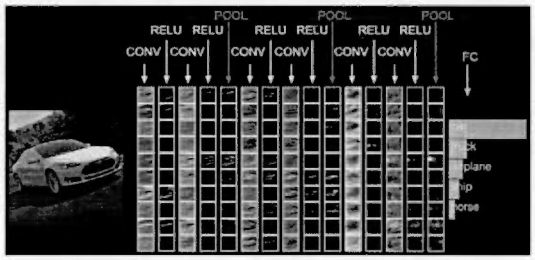

Figure 3.5: Exemple d'architecture CONVNET avec fonction d'activation ReLu. L'image est transformée plusieurs fois à travers les couches du réseau convolutif. On cherche à classifier différentes images (Voitures,Camion, Avion, Bateau, Cheval)

On va maintenant définir chacune des étapes à travers lesquelles l'entrant d'un

réseau convolutif passe avant que le réseau lui accorde un score pour chacune des catégories utilisées pour la classification.

L'entrant : puisque l'on utilise le réseau convolutif pour la classification d'images, l'entrant sera représenté sous la forme d'un volume. Pour le traitement à l'intérieur du réseau, l'image est habituellement représentée sous la forme d'un volume dont la largeur est égale à la hauteur. La profondeur du volume représente le nombre de canaux «channels». Pour une image en couleur, on a trois canaux qui représentent les intensités de rouge, de vert et de bleu. Dans le cas d'une image à intensité de gris, on a seulement un canal qui représente l'intensité de gris dans l'image. On utilisera également le mot entrant au sens large pour désigner l'entrant d'une certaine couche, peu importe où on se situe dans le réseau.

La couche de convolution : pour passer d'une couche d'entrants ou d'une couche de pooling à une couche de convolution, on applique une convolution au sens mathématique. D'où le nom de réseau de neurones à convolution. La convolution peut transformer la hauteur, la largeur et la profondeur du volume d'entrant. On définira la convolution plus en détail dans la section 3.4.

La fonction d'activation : la fonction d'activation est appliquée après la convolution et laisse la taille volume inchangé. Par exemple, si on a une fonction d'activation ReLu, on applique la fonction $f(x) = max(0, x)$ à chacun des éléments compris dans le volume associé à la couche.

La couche de pooling : la couche de pooling réduit la dimension (i.e la hauteur et la largeur) du volume d'entrants sans toutefois affecter sa profondeur. Les opérations les plus courantes sont le max-pooling 3.3 et le moyenne-pooling 3.4

«average pooling».

La couche entièrement connectée : les neurones d'une couche entièrement connectée sont connectés avec chacun des neurones du volume correspondant à son entrant, c'est-à-dire avec chacun des neurones de la couche précédente. Si on décompose notre volume d'entrants sous forme vectorielle, on a exactement un réseau de neurones MLP.

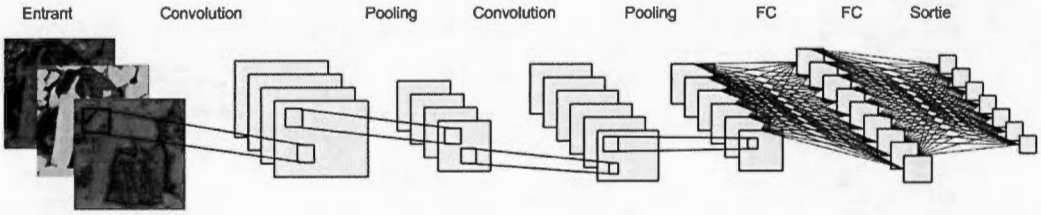

Figure 3.6: Exemple de réseau convolutif deux étapes CONV-POOL suivies par deux couches FC (entièrement connectées).

3.4 La convolution

Si f et g sont toutes deux fonctions continues d'une variable, on définit la convolution

$$\begin{aligned} s(t) = (f * g)(t) &= \int_{-\infty}^{\infty} f(\tau) g(t-\tau) d\tau \\ &= \int_{-\infty}^{\infty} f(t-\tau) g(\tau) d\tau = (g * f)(t). \end{aligned}$$

On décrit souvent la convolution comme une moyenne pondérée de f où g représente la pondération associée à f. La fonction f est appelée la fonction d'entrant «input function». Ce qu'on appelle ici la pondération correspond au noyau «kernel» que l'on définira par la suite. On se sert généralement de la version discrète

de la convolution définie sous forme de sommation lorsque l'on traite de fonctions à une variable, car on utilise l'informatique dans plupart des applications en apprentissage automatique. La forme discrète est la suivante

$$\begin{aligned} s(t) = (f * g)(t) &= \sum_{\tau=-\infty}^{\infty} f(\tau) g(t-\tau) \\ &= \sum_{\tau=-\infty}^{\infty} f(t-\tau) g(\tau) = (g * f)(t). \end{aligned}$$

Plus particulièrement, lorsqu'il est question de classification d'images, l'entrant correspond à un volume en deux ou trois dimensions. Dans un réseau convolutif, si I est un entrant et si K est un noyau, alors la convolution est définie comme

$$\begin{aligned} S(i,j) = (I * K)(i,j) &= \sum_m \sum_n I(i+m, j+n) K(r-m, r-n) \\ &= \sum_m \sum_n I((i+r-1)-m, (j+r-1)-n) K(m+1, n+1) \end{aligned} \tag{3.1}$$

où $m, n \in \{0, \ldots, r-1\}$. Ici, r correspond à la dimension du noyau. Dans ce cas, on a pris la convolution en deux dimensions pour simplifier les choses. Pour une représentation visuelle de cette sommation, les figures 3.8 et 3.12 montrent toutes deux une convolution en deux dimensions alors que la figure 3.7 représente les déplacements du noyau pour une convolution en trois dimensions. Puisque la base du noyau est toujours carrée, le terme dimension sera utilisé pour désigner sa hauteur ou sa largeur. À travers la première égalité de la formule 3.1, on a une rotation de 180 degrés du noyau alors que dans la deuxième partie, c'est la partie de l'entrant affectée par le noyau qui effectue cette même rotation. L'opération est la même, d'où l'égalité. Lorsqu'un réseau convolutif est appliqué, la rotation n'est pas nécessaire. C'est pourquoi on utilise en pratique plus souvent la corrélation croisée (Goodfellow *et al.*, 2016). L'opération est très similaire, mais on n'effectue

pas de rotation pour le noyau. C'est-à-dire :

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m+1, n+1). \tag{3.2}$$

Il est à noter que la profondeur du noyau sera toujours la même que la profondeur de l'entrant, peu importe la couche à laquelle on se retrouve dans le réseau convolutif. Le noyau K est l'équivalent de nos poids lorsqu'on l'on avait un réseau MLP. Dépendamment de la profondeur de l'entrant, les noyaux sont également représentés sous forme de volumes. On déterminera un nombre de filtre(s). Ce qu'on appelle le nombre de filtre(s) est en fait le nombre de noyaux par entrant et détermine la profondeur de la sortie associée à la convolution.

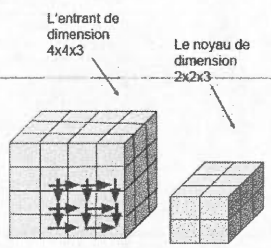

Figure 3.7: Exemple d'un volume pour l'entrant de dimension $4 \times 4 \times 3$ et un seul noyau de dimension $2 \times 2 \times 3$. Les flèches rouges représentent chacun des déplacements du coin inférieur gauche du noyau à travers l'entrant lorsqu'on a un pas de 1 pour les déplacements verticaux et horizontaux.

Il est nécessaire d'introduire quelques concepts afin de comprendre comment la convolution est utilisée dans le cadre d'un réseau convolutif. Les trois hyperparamètres suivants détermineront la dimension de la sortie associée à notre convolution.

Le pas : lorsque l'on effectue une convolution, on choisit ce qu'on appelle le pas «stride», i.e le pas auquel on déplace le noyau à travers l'entrant. Le pas horizontal représente le pas auquel on déplace horizontalement le noyau à travers l'image alors que le pas vertical représente le pas auquel on déplace verticalement le noyau à travers l'image. Afin de simplifier, pour la suite des choses, on utilisera un pas horizontal égal à notre pas vertical afin de simplifier le modèle.

La profondeur : le nombre de filtres est en fait le nombre de noyaux (matrices de poids) choisis par couche de convolution. Le nombre de filtres déterminera la profondeur de la sortie associée à la convolution.

La marge à zéro : il est commun d'utiliser une certaine épaisseur de marge de zéros autour de l'entrant afin de contrôler la hauteur et la largeur du volume de sortie.

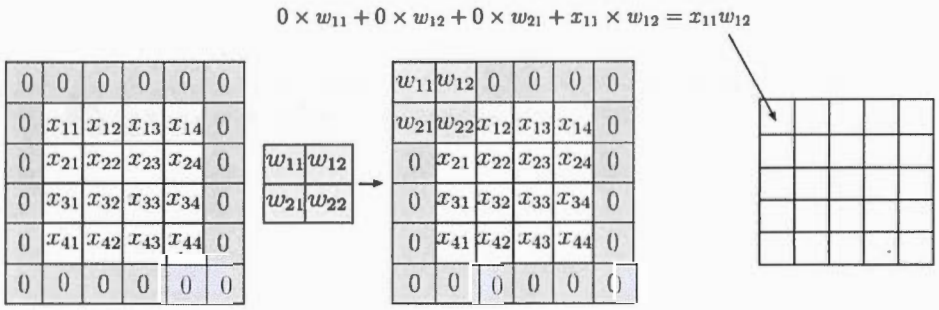

Figure 3.8: Exemple d'entrant $4 \times 4 \times 1$ avec couche de marge à zéro de taille 1 auquel on applique un noyau $2 \times 2 \times 1$. La sortie de notre convolution est de dimension $5 \times 5 \times 1$. On a utilisé un pas de 1 pour les déplacements du noyau. Dans ce cas, on a $d = 4$, $r = 2$, $p = 1$ et $s = 1$.

Il est connu que la dimension du volume de sortie est donnée par la formule

suivante $(d-r+2p)/s+1$ où d est dimension associée à l'entrant, r la dimension du noyau, p l'épaisseur de la marge à zéro et s l'amplitude du pas utilisée (Stanford, 2015).

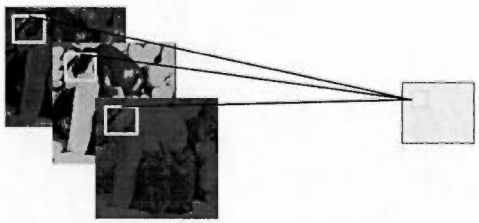

Figure 3.9: Convolution appliquée sur l'image piments en format RVB. On utilise un seul filtre de dimension r. Les trois carrés de gauche représente l'image RVB et le carré de droite représente la sortie qui sera de dimension $(d-r+2p)/s+1$.

3.5 L'opération pooling

L'idée derrière l'opération pooling est de retirer une certaine information dans un même voisinage pour l'entrant. On effectue l'opération pooling sur chacune des épaisseurs liées à la profondeur de l'entrant. Afin de garder la même notation, on prendra X_l pour représenter le volume associé à la sortie après la $l^{\text{ème}}$ séquence d'opérations CONV-ACTIVATION et Y_l pour la sortie après l'opération pooling appliquée à la matrice X_l. On prend par défaut X_0 pour désigner notre entrant, c'est-à-dire l'image. On rajoute l'indice k pour désigner la profondeur à laquelle on se trouve dans le volume, i.e $X_l^{(k)}$. Si $x_{l,i,j}^{(k)}$ est l'élément (i,j) dans la matrice $X_l^{(k)}$, et que l'on choisit le max-pooling, alors la sortie associée à l'opération pooling est donnée par

$$y_{l,i,j}^{(k)} = max_{pq}(x_{l,i+p,j+q}^{(k)}, 0) \tag{3.3}$$

où $p, q \in \{0, \ldots, r-1\}$. Dans ce cas r représente la dimension du voisinage choisi. On utilise des voisinages sans chevauchement pour le pooling. Si on avait plutôt choisi l'opération moyenne pooling pour le réseau, la sortie associée à l'opération pooling aurait été donnée par

$$y_{l,i,j}^{(k)} = \frac{1}{r^2} \sum_{pq} x_{l,i+p,j+q}^{(k)}. \tag{3.4}$$

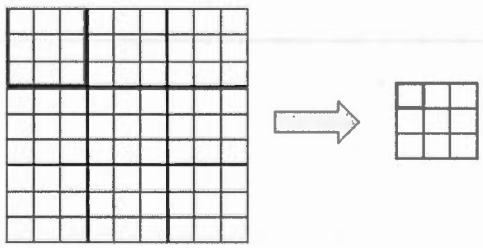

Figure 3.10: Dans cet exemple, on effectue l'opération pooling sur une épaisseur, disons k' d'un volume d'entrants $X_l^{(k)}$. Les voisinages sont délimités par les traits de couleur noire. Les carrés bleus représentent un voisinage spécifique sur lequel on effectue le pooling et la sortie associée. Les voisinages sont choisis sans chevauchement. Dans ce cas $r = 3$.

La figure 3.11 montre comment l'opération pooling affecte la dimension du volume lié à son entrant X_l.

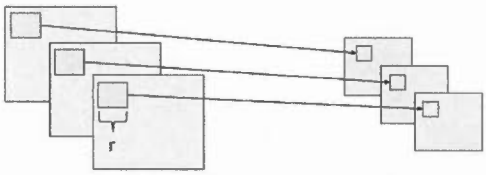

Figure 3.11: Application de l'opération pooling sur un entrant de profondeur 3. À gauche les carrés bleus, rouges et verts représentent respectivement ce qu'on a dénoterait les matrices $X_l^{(1)}, X_l^{(2)}$ et $X_l^{(3)}$. Les carrés de gauche sont les sorties correspondantes $Y_l^{(1)}, Y_l^{(2)}$ et $Y_l^{(3)}$.

3.6 Propagation dans un réseau convolutif

Pour la propagation dans le réseau, on réutilise exactement la même notation qu'auparavant excepté qu'on utilisera des lettres majuscules pour représenter nos couches sous forme de volume. Ainsi, les étapes pour la propagation d'un entrant sont :

$$
\begin{aligned}
A_1 &= X_0 * \theta_0 \\
X_1 &= g(A_1) \\
Y_1 &= POOL(g(A_1)) \\
A_2 &= Y_1 * \theta_1 \\
X_2 &= g(A_2) \\
Y_2 &= POOL(g(A_2)) \\
&\vdots \\
A_{L+1} &= Y_L \theta_L \\
X_{L+1} &= S_\theta(X_0) = f(A_{L+1})
\end{aligned}
$$

On n'effectue pas de convolution entre Y_L et θ_L car la dernière couche est habi-

tuellement entièrement connectée.

3.7 Dérivabilité des fonctions et réseau convolutif

Dans cette section, l'objectif est de réutiliser le travail fait dans la section 2.4 afin de déterminer une forme récursive pour le réseau convolutif. Dans le cas où l'on a une couche entièrement connectée, nos formules récursives pour le calcul des dérivées développées dans la partie 2.4 tiennent toujours puisque si on vectorise une couche entièrement connectée, on a exactement un modèle MLP. Pour une couche de pooling, la dérivée par rapport à l'entrant est également facilement calculable. La tâche la plus complexe revient donc de déterminer comment on calcule nos dérivées lorsque l'on a une couche de convolution. Afin de montrer comment on effectue le calcul, on va encore une fois utiliser un exemple simple très similaire à celui de la section 2.4. Puisque les couches d'un réseau convolutif sont exprimées sous forme de volume, on utilisera la vectorisation de chacune de nos transformations afin de parvenir à calculer les dérivées.

Pour la suite des choses, on suppose des entrants de profondeur 1 (i.e en deux dimensions) et un seul filtre (i.e un seul noyau) pour les convolutions. La théorie qui suit est facilement généralisable si on augmente le nombre de filtres et la profondeur des entrants. On réutilise la notation n_0 pour représenter la dimension de X_0 et on dénote r_0 pour représenter la dimension du noyau associé. On fixe également la marge à zéro $p_0 = 0$ et le pas $s_0 = 1$ pour simplifier le problème. Considérons un volume correspondant à l'entrant $X_0 \in \mathbb{R}^{n_0 \times n_0 \times 1}$, alors sa sortie correspondante après convolution sera $A_1 \in \mathbb{R}^{n_1 \times n_1 \times 1}$ où $n_1 = (n_0 - r_0 + 2p_0)/s_0 + 1$. Le noyau sera $\theta_0 \in \mathbb{R}^{r_0 \times r_0 \times 1}$. Pour arriver au calcul des dérivées, on devra considérer la vectorisation de X_0 et A_1, c'est-à-dire $vec(X_0)$ et $vec(A_1)$. La figure 3.12 permet comprendre comment on vectorise un réseau convolutif et de faire le lien avec un réseau MLP.

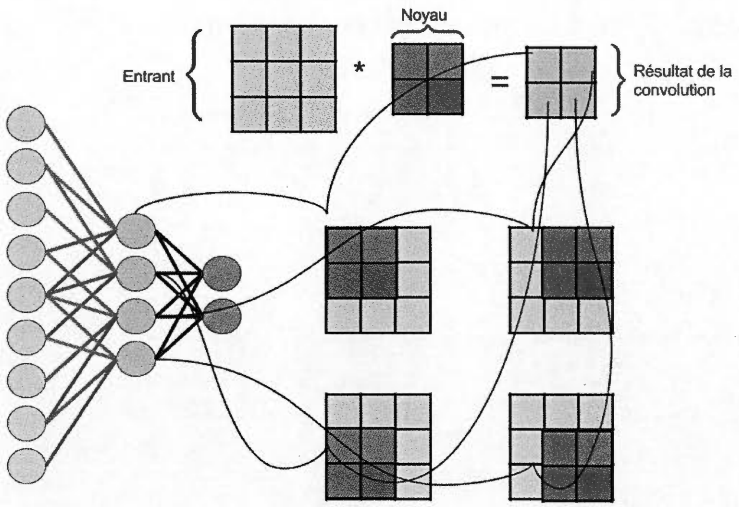

Figure 3.12: La première couche (bleu pâle) dans le réseau représente la vectorisation $vec(X_0)$. La deuxième couche représente le résultat de la convolution vectorisée $vec(A_1)$. La dernière couche (orange) représente une couche entièrement connectée. Les 4 figures de droite représentent les déplacements du noyau (représenté en vert foncé, bleu foncé, rouge et rose) à travers l'entrant X_l. Les lignes relient le résultat de la convolution à travers la vectorisation $vec(A_1)$ et le résultat associé à $X_l * \theta_l$, i.e le carré vert pâle.

Maintenant que l'on s'est familiarisé aux principales opérations effectuées dans un réseau convolutif, on va essayer de déterminer une forme récursive permettant de calculer les dérivées dans le cas où on aurait un modèle avec une série de convolutions. Pour l'exemple qui suit, on prend un modèle très similaire à celui utilisé dans la partie 2.4. Même si cet exemple n'est pas représentatif d'un vrai réseau convolutif, étant donné que l'on va omettre la couche de pooling et la couche entièrement connectée, il permettra tout de même d'arriver à déterminer une forme récursive pour les dérivées dans le cas où on l'on a une série de convolutions.

Notre modèle simple est paramétrisé de la façon suivante

$$\theta_0 = W_0 \in \mathbb{R}^{r_0 \times r_0} \qquad S_\theta(x_0) = X_2 = f(\theta_1 * g(\theta_0 * X_0)) \in \mathbb{R}^{n_2 \times n_2 \times 1}$$

$$\theta = \begin{bmatrix} vec(\theta_0) \\ vec(\theta_1) \end{bmatrix} \in \mathbb{R}^{r_0 r_0 + r_1 r_1} \qquad X_0 \in \mathbb{R}^{n_0 \times n_0 \times 1} \qquad X_1 \in \mathbb{R}^{n_1 \times n_1 \times 1}$$

$$\theta_1 = W_1 \in \mathbb{R}^{r_1 \times r_1} \qquad g : \mathbb{R}^{n_1 \times n_1 \times 1} \to \mathbb{R}^{n_1 \times n_1 \times 1} \qquad f : \mathbb{R}^{n_1 \times n_1 \times 1} \to \mathbb{R}^{n_2 \times n_2 \times 1}$$

$$A_1 = \theta_0 * X_0 \qquad X_1 = g(A_1) \qquad A_2 = \theta_1 * X_1 \qquad X_2 = f(A_2)$$

avec encore une fois une fonction de perte sans régularisateur

$$Q(\theta) = J(\theta) = \ell(S_\theta(x_0), y). \tag{3.5}$$

Comme on est habitué d'avoir une sortie pour le réseau $S_\theta(x_0) \in \mathbb{R}^K$, il peut paraître étrange que la sortie de notre réseau $S_\theta(x_0)$ soit représentée par un volume, (i.e $S_\theta(x_0) \in \mathbb{R}^{n_2 \times n_2 \times 1}$). Cela ne changera rien pour les calculs étant donné qu'on cherche seulement à trouver une forme récursive pour nos dérivés et que l'on vectorisera l'ensemble de nos paramètres et de nos transformations. On veut tout simplement ne pas avoir à ajouter une couche entièrement connectée afin de se simplifier la tâche et ne pas avoir à rajouter un troisième paramètre (i.e un θ_2). Pour la transformation des paramètres et les dérivées de ceux-ci, on utilisera encore une fois un schéma similaire à celui de la section 2.4 qui représente à travers une série de transformations un réseau vectorisé et la fonction qui transporte θ vers la fonction de perte.

$$\begin{array}{ccccccc} \mathbb{R}^{r_0 r_0 + r_1 r_1} & \xrightarrow{j} & \mathbb{R}^{n_1 n_1 + r_1 r_1} & \xrightarrow{h^*} & \mathbb{R}^{n_2 n_2} & \xrightarrow{f} & \\ \begin{bmatrix} vec(\theta_0) \\ vec(\theta_1) \end{bmatrix} & \longmapsto & \begin{bmatrix} g(vec(\theta_0 * x_0)) \\ vec(\theta_1) \end{bmatrix} & \longmapsto & \left[vec(\theta_1 * g(\theta_0 * x_0)) \right] & \longmapsto & \end{array}$$

$$\begin{array}{ccc} \mathbb{R}^{n_2 n_2} & \xrightarrow{\ell} & \mathbb{R} \\ \left[f(vec(\theta_1 * g(\theta_0 * x_0))) \right] & \longmapsto & \ell(S_\theta(x_0, y)) \end{array}$$

Maintenant qu'on a modélisé la transformation de nos paramètres, on peut réutiliser les résultats de la section 2.4. On va encore réécrire chacune de nos dérivées par bloc et leurs dimensions en indice.

$$\begin{aligned} D\ell(S_\theta(x_0), y) = D\ell(S_\theta(x_0))_{1\times n_2 n_2} Df(vec(\theta_1 * g(\theta_0 * x_0)))_{n_2 n_2 \times n_2 n_2} \\ \times Dh^*(j(\theta))_{n_2 n_2 \times (n_1 n_1 + r_1 r_1)} D(j(\theta))_{(n_1 n_1 + r_1 r_1)\times(r_0 r_0 + r_1 r_1)} \end{aligned} \tag{3.6}$$

Une remarque sur la notation. On utilise $h*$ afin d'avoir la même notation qu'au chapitre 2.4 et de désigner la deuxième transformation du schéma qu'on vient de présenter. Le signe $*$ sur le h n'a aucun lien avec le signe $*$ utilisé pour désigner la convolution.

Au chapitre 2.4, on a posé $\delta^{(l)}$ comme étant égale aux dérivées de notre fonction de perte par rapport à nos a_l, $l \in \{1, \ldots, L+1\}$. Cette fois,puisque chacune de nos transformations sont représentées sous forme de volume dans un réseau convolutif, on pose $\delta^{(l)}$ comme étant les dérivées de la fonction de perte par rapport à A_l mais dans un espace non vectorisé. C'est-à-dire que $\delta^{(l)}$ a la forme d'un volume de même dimension A_l et sa représentation est la suivante :

$$\delta_l := \begin{bmatrix} \frac{\partial \ell}{\partial a^l_{11}} & \cdots & \frac{\partial \ell}{\partial a^l_{1n_l}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial a^l_{n_l 1}} & \cdots & \frac{\partial \ell}{\partial a^l_{n_l n_l}} \end{bmatrix}.$$

L'ensemble du réseau a été vectorisé pour effectuer le calcul des dérivées. Le calcul des deux premiers blocs donne la transposée de la vectorisation de $\delta^{(2)}$ suivante :

$$D\ell(S_\theta(x_0))Df(vec(\theta_1 * g(\theta_0 * x_0))) = \frac{\partial \ell}{\partial vec(A_2)^T} = vec(\delta^{(2)})^T. \tag{3.7}$$

Il n'est pas nécessaire de calculer explicitement cette expression parce que l'on veut seulement arriver à une équation pour nos dérivées. La dérivée pour le troisième

bloc est particulièrement importante dans l'équation 3.6. Lorsqu'on multiplie nos deux premiers blocs par notre troisième bloc, c'est-à-dire $vec(\delta^{(2)})^T$ par $Dh^*(j(\theta))$, on obtient la matrice partionnée suivante. Les dimensions des deux partitions sont données en indice.

$$vec(\delta^{(2)})^T Dh^*(j(\theta)) = \left[\begin{array}{c|c} vec(rot_{180}(\theta_1) \overset{\text{full}}{*} \delta^{(2)})^T_{1 \times n_1 n_1} & vec(\delta^{(2)} * X_1)^T_{1 \times r_1 r_1} \end{array} \right] \quad (3.8)$$

On voit apparaître deux nouvelles opérations dans l'équation 3.8. L'opération rot_{180} signifie qu'on fait une rotation de 180 degrés du noyau. Par exemple, si le noyau θ est defini par l'équation de droite, alors sa rotation de 180 degrés est donnée par l'équation de droite.

$$\theta = \begin{bmatrix} \theta_{11} & \theta_{12} \\ \theta_{21} & \theta_{22} \end{bmatrix} \qquad rot_{180}(\theta) = \begin{bmatrix} \theta_{22} & \theta_{21} \\ \theta_{12} & \theta_{11} \end{bmatrix}.$$

L'opération $\overset{\text{full}}{*}$ représente la convolution pleine (full convolution). Cette opération commutative est implémentée dans la plupart des logiciels ou librairies qui traitent les opérations matricielles. On peut facilement la trouver entre autres dans les logiciels comme Matlab ou la librairie Numpy de Python. La formule qui suit permet de déterminer le résultat de l'entrée (l, k) associée à une convolution pleine avec rotation du noyau comme celle qui est illustrée dans la partition de gauche de la matrice comprise dans l'équation 3.8.

Soit v le nombre de rangées du noyau K et h son nombre de colonnes. Si m est n sont le nombre de rangées et de colonnes respectif pour l'entrant I, alors le résultat pour la convolution pleine avec rotation du noyau et un pas de 1 est donné par

$$R(l,k) = \sum_{i=max(l-m+1,1)}^{min(v,l)} \sum_{j=max(k-n+1,1)}^{min(h,k)} I(l-i+1, k-j+1)K(i,j).$$

Le lecteur peut également se référer à la figure 3.13 afin d'avoir une représentation plus visuelle des déplacements du noyau à travers l'entrant lors d'une convolution pleine en deux dimensions.

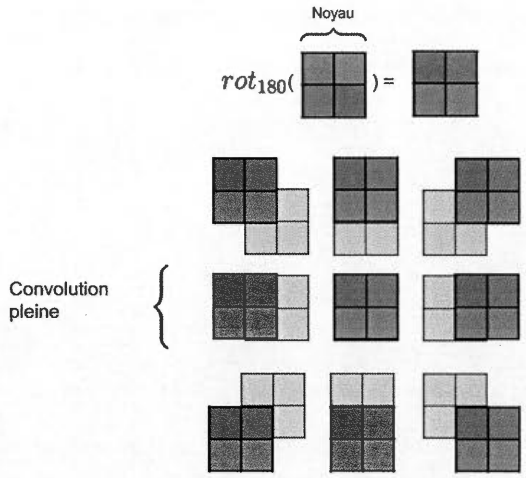

Figure 3.13: Exemple simple des déplacements du noyau sur un entrant de dimension 2×2 (en bleu pâle) lors d'une convolution pleine avec un pas de 1.

Il nous reste maintenant plus qu'à calculer notre quatrième bloc $Dj(\theta)$. On trouve que

$$D(j(\theta)) = \left[\begin{array}{c|c} Dg(vec(A_1))_{n_1n_1 \times n_1n_1}(\frac{\partial vec(A_1)}{\partial vec(\theta_0)})_{n_1n_1 \times r_0r_0} & 0_{n_1n_1 \times r_1r_1} \\ \hline 0_{r_1r_1 \times r_0r_0} & I_{r_1r_1 \times r_1r_1} \end{array}\right].$$

En multipliant $vec(\delta^{(2)})^T Dh^*(j(\theta))$ par $D(j(\theta))$, on obtient comme résultat final la matrice partionnée suivante.

$$\begin{aligned} D\ell(S_\theta(x_0), y) &= \left[\begin{array}{c|c} vec(rot180(\theta_1) \overset{\text{full}}{*} \delta^{(2)})^T Dg(vec(A_1))(\frac{\partial vec(A_1)}{\partial vec(\theta_0)}) & vec(\delta^{(2)} * X_1)^T \end{array}\right] \\ &= \left[\begin{array}{c|c} vec(rot180(\theta_1) \overset{\text{full}}{*} \delta^{(2)} \odot g'(A_1) * X_0)^T & vec(\delta^{(2)} * X_1)^T \end{array}\right] \\ &= \left[\begin{array}{c|c} vec(\delta^{(1)} * X_0)^T & vec(\delta^{(2)} * X_1)^T \end{array}\right]. \end{aligned}$$

D'une façon similaire à celle faite dans la section 2.4, il est facile de voir que notre résultat se généralise lorsque l'on ajoute plusieurs couches cachées. On trouve que

$$\delta^{(l)} = rot180(\theta_l) \overset{\text{full}}{*} \delta^{(l+1)} \odot g'(A_l) \tag{3.9}$$

si $l \in \{1, \ldots, L\}$ et si il a eu convolution entre les couches l et $l+1$. Si il y a également eu convolution entre la couche l et $l-1$, on voit que la convolution réapparaît lorsqu'on arrive à la dérivée par rapport au poid θ_l dont il est question, i.e :

$$\frac{\partial J(\theta)}{\partial \theta_l} = X_l * \delta^{(l+1)}.$$

Dépendamment de l'architecture du réseau convolutif, les δ_l sont définis autrement selon que l'on ait une convolution ou des neurones entièrement connectés entre deux couches. Tel que mentionné précédemment, si on utilise une couche entièrement connectée nos résultats de la section 2.4 sont toujours valides.

CHAPITRE IV

RÉSULTATS SUR LE MNIST

4.1 Description jeu de données

Le MNIST «Mixed National Institute of Standards and Technology database» est une base de données de chiffres écrits à la main. Dans le cadre de ce mémoire, la base de données a été téléchargée du site de Yan LeCun (LeCun, 2016) et comprend un ensemble d'entraînement de 60000 observations et un ensemble test de 10000 observations. On utilisera les dernières 5000 observations de l'ensemble d'entraînement comme ensemble de validation. L'ensemble d'entraînement sera utilisé pour développer et estimer les paramètres des modèles et leurs variantes comprises dans chacune des grandes catégories dont il a été question lors de ce mémoire. C'est-à-dire, la régression logistique multinomiale, le réseau de neurones MLP et le réseau de neurones à convolution. On a recours à l'ensemble de validation afin d'éviter le problème de surajustement. Sur cet ensemble, on détermine et l'on estime les hyperparamètres optimaux pour les différents modèles selon la performance résultante. Cet ensemble va nous permettre de sélectionner les meilleurs modèles selon la catégorie. À titre d'exemple, pour un réseau de neurones, un hyperparamètre peut entre autres être le nombre de couches cachées, le nombre de neurones dans chacune des couches cachées ou la force de régularisation que l'on a représenté par λ tout au long du chapitre 2. C'est sur l'ensemble test que l'on évaluera la performance officielle du modèle optimal selon la catégorie utilisée.

Ainsi notre séparation pour le MNIST est représentée dans la figure 4.1.

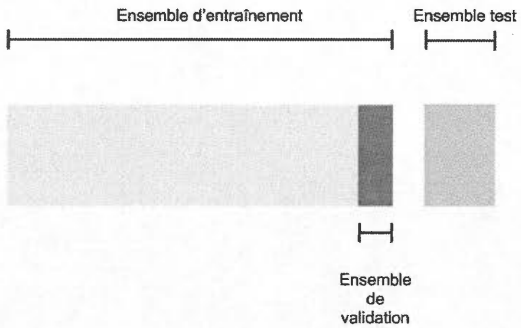

Figure 4.1: partition du jeu de données

Chacune des images comprises dans le MNIST est de dimensions 28 pixels par 28 pixels et représente un chiffre écrit à la main de 0 à 9. Les observations du jeu de donnée sont des images à intensité de gris tel qu'on les a définis dans la section 3.1.2. Pour chaque image, on a la classe correspondante à un chiffre pour la variable expliquée. L'objectif est de classifier correctement les chiffres sur l'ensemble test lorsque l'on ne connait pas la réponse. La figure 4.2 montre quatre exemples d'images comprises à l'intérieur du MNIST.

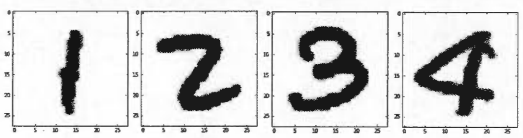

Figure 4.2: 4 chiffres écrit à la main provenant du MNIST

4.2 Résultats et expérimentations

Les résultats de la table 4.1 permettent de comparer les performances des modèles présentés dans chacune des catégories et certaines variantes de ceux-ci. On teste différentes architectures de réseau afin de déterminer la meilleure méthode de classification d'images. Avec la régression logistique, on obtient un taux d'er-

Méthode	Type d'Unité	Architecture	Erreur (%)
Régression Multinomiale	NA	NA	7.4
MLP	logistique	2 couches, 800 unités	3.42
MLP + Dropout	ReLu	2 couches, 1500 unités	2.31
MLP + Dropout + $L2$	ReLu	2 couches, 1500 unités	2.04
Convolutionnel	ReLu	2 CONV-POOL, 2 FC	1.25
Convolutionnel + Dropout	ReLu	2 CONV-POOL, 2 FC	0.69

Tableau 4.1: Comparaison de différentes modèles. Le pourcentage d'erreur représente le taux de mauvaise classification sur l'ensemble test.

reur de 7.4. On arrive à diminuer considérablement l'erreur lorsqu'on passe au réseau de neurones. Le premier réseau MLP avec fonction d'activation logistique a permis d'obtenir de relativement bons résultats avec une architecture moins dense tout en arrivant à une convergence rapide de la fonction de perte et du taux de classification sur l'ensemble test. La méthode Dropout et la fonction d'activation ReLu ont donné de meilleurs résultats en augmentant la taille du réseau à une architecture de deux couches cachées comprenant 1500 neurones chacune et en diminuant le taux d'apprentissage «learning rate» que l'on a dénoté par la notation α au chapitre 2. Pour les deux réseaux MLP avec Dropout, on a appliqué une probabilité de rétention des unités de $p = 0.5$ sur chacune des couches

cachées. Ajouter une probabilité de rétention de $p = 0.8$ sur les neurones de la couche d'entrants tel que suggéré par l'article (Hinton *et al.*, 2014) n'a pas permis d'améliorer les résultats. La norme L_2 pour le régularisateur combiné avec la méthode Dropout a permis de diminuer de quelque peu l'erreur à 2.04. C'est lorsqu'on est passé à un réseau de neurones convolutifs qu'on a vu apparaître les meilleurs résultats. Tel que l'on peut le voir dans le tableau 4.1, l'architecture pour les deux réseaux de neurones convolutifs utilisés sur le MNIST est composée deux séries convolution-pooling suivi de deux couches entièrement connectées. L'architecture utilisée pour les deux réseaux convolutifs est semblable à celle présentée dans la documentation disponible en ligne de la librairie Tensorflow (Abadi *et al.*, 2015). Celle-ci est composée de 50 filtres de dimension 5 pour la première convolution, et de 100 filtres de dimension 5 pour la deuxième et d'un voisinage de taille 2×2 pour les deux couches de pooling. On a utilisé un pas vertical et horizontal de 1 pour les déplacements des noyaux. Deux épaisseurs de marge à zéro ont été appliquées verticalement et horizontalement sur l'image et la sortie associées à la première couche de pooling afin de garder la même dimension pour le volume de sortie après la première et deuxième convolution. La première couche entièrement connectée est composée de 1500 neurones et la deuxième de 10 neurones de sortie pour la classification des 10 chiffres. Pour tous les réseaux de neurones exceptés celui avec fonction d'activation logistique, une variante de l'algorithme du gradient stochastique «Mini-Batch», sois la méthode d'estimation du moment adaptatif Adam (Kingma et Ba, 2014), a été utilisé et a permis d'obtenir de meilleurs résultats. Les modèles inclus dans la table 4.1 ont été développés à l'aide de la librairie Tensorflow. Ils sont disponibles sur mon Github à l'adresse `https://github.com/samuelBedard/MNIST/FinalProgram`. Les modèles ont déjà été entraînés dans les sections «saveMyNet». Il est donc possible de restaurer les modèles pour tester l'exactitude des résultats de la table 4.1.

CONCLUSION

Le but principal de ce mémoire est l'étude des algorithmes de pointe dans le domaine des réseaux de neurones. Étant donné la nouveauté du sujet et due à la faible quantité de références mathématiques et statistiques, il a été question de dériver la plupart des résultats et formules présentés dans le cadre de ce mémoire. Nous avons exposé le sujet de manière formelle en incluant les détails de plusieurs calculs. Ceux-ci sont omis dans les références que nous avons trouvées sur le sujet. Plus particulièrement, cela a été le cas lorsqu'il a été question formuler le modèle, définir une forme récursive pour le calcul du gradient et faire le lien avec l'implémentation informatique utilisée dans les logiciels modernes. Cette démarche a requis une quantité non négligeable de temps et de recherche. Elle a toutefois permis d'acquérir une solide base et compréhension du modèle qui s'est avérée transférable à des problèmes plus complexes tels qu'il a été le cas lorsqu'on est passé au réseau de neurones convolutifs. Il a également été intéressant d'explorer les récentes méthodes empiriques utilisées afin d'ajuster correctement un modèle sur des ensembles de données. Je considère que le bagage de connaissances présentées dans cette thèse constitue un outil intérressant pour quiconque voulant aquérir une solide base sur le sujet qui pourrait être applicable à des recherches complémentaires plus poussées ou directement dans un milieu professionnel.

BIBLIOGRAPHIE

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. et Zheng, X. (2015). TensorFlow : Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. Récupéré de `http://tensorflow.org/`

Baldi, P. et Sadowski, P. (2014). The dropout learning algorithm. *Artif. Intell.*, *210*, 78–122. `http://dx.doi.org/10.1016/j.artint.2014.02.004`. Récupéré de `http://dx.doi.org/10.1016/j.artint.2014.02.004`

Baldi, P. et Sadowski, P. J. (2013). Understanding dropout. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, et K. Q. Weinberger (dir.), *Advances in Neural Information Processing Systems 26* 2814–2822. Curran Associates, Inc.

DeepMind (2016). Alphago. Récupéré de `https://deepmind.com/research/alphago/`

Goodfellow, I., Bengio, Y. et Courville, A. (2016). Deep learning. Book in preparation for MIT Press

Hinton, G., Srivastava, N., Krizhevsky, A., Sutskever, I. et Salakhutdinov, R. (January 2014). Dropout : a simple way to prevent neural networks from overfitting. 1929–1958.

Kingma, D. P. et Ba, J. (2014). Adam : A method for stochastic optimization. *CoRR*, *abs/1412.6980*. Récupéré de `http://arxiv.org/abs/1412.6980`

LeCun, Y. (2016). The mnist database of handwritten digits. Récupéré de `http://yann.lecun.com/exdb/mnist/`

Pascanu, R., Mikolov, T. et Bengio, Y. (2012). Understanding the exploding gradient problem. *CoRR*, *abs/1211.5063*. Récupéré de `http://arxiv.org/abs/1211.5063`

Robbins, Herbert ; Siegmund, D. O. (1971). *A convergence theorem for non negative almost supermartingales and some applications.* Rustagi, Jagdish S. Optimizing Methods in Statistics. Academic Press.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. et Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*, 484–503. Récupéré de `http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html`

Stanford (2015). Stanford university cs231n. Récupéré de `http://cs231n.github.io/`

Theano Development Team (2016). Theano : A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, *abs/1605.02688*. Récupéré de `http://arxiv.org/abs/1605.02688`