

Apprentissage & Introduction à PyTorch



M. Lilian BUZER

1. L'apprentissage

La question légitime qui interpelle tout élève en début de ce cours est la suivante : comment une machine peut-elle apprendre ? Nous allons répondre à cette question dans ce chapitre et pour cela nous allons présenter les concepts essentiels vous permettant de comprendre comment un réseau de neurones peut apprendre.

1.1 Le réseau : une fonction de $\mathbb{R}^n \rightarrow \mathbb{R}$

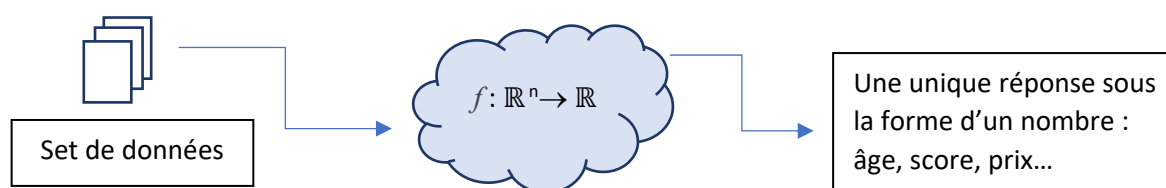
Avant d'apprendre, il faut déjà que le « programme » parle ! Ainsi, il faut que l'on construise un traitement qui à partir de données source puisse fournir une réponse : un âge, un score, une catégorie... Les données en entrée, monde informatique oblige, seront numériques :

- Une image se représente sous la forme d'une matrice de valeurs entre 0 et 255.
- Un son se modélise comme une suite de valeurs issues de l'enregistrement depuis un micro.
- L'état de Pacman se traduit par sa position x,y , le score de partie, la position des fantômes...

Ainsi en entrée, nous avons des valeurs numériques et en sortie une valeur numérique ; à ce niveau, vous devez reconnaître la définition d'une fonction mathématique f de $\mathbb{R}^n \rightarrow \mathbb{R}$ et nous choisissons naturellement de modéliser notre système d'apprentissage avec une fonction. Ceci a un impact immédiat sur le fonctionnement du système, car cela implique que toutes les entrées aient la même taille :

- Pour un set d'images, il faut donc retailler et redimensionner les images pour qu'elles aient toutes la même résolution et la même profondeur comme par exemple : 32x32x3
- Pour un son, il faut faire en sorte que tous les échantillons fassent la même durée avec la même vitesse d'échantillonnage.
- Pour des phrases, il faut que toutes les phrases contiennent le même nombre de mots quitte à ajouter des mots « vides » pour compléter la fin de phrase.

Voici donc notre premier système :

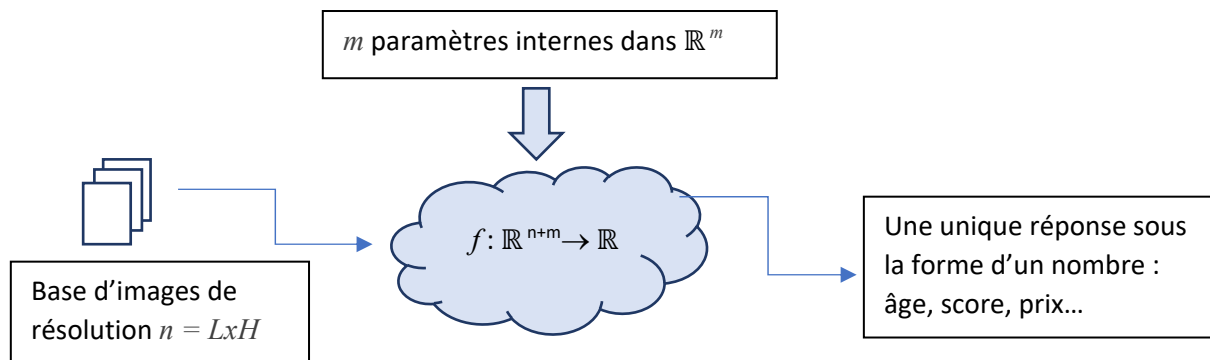


Nous avons construit un système qui prend des données en entrée et fournit une réponse en sortie. Cependant, vous l'avez remarqué, le système en question est incapable d'évoluer et d'apprendre.

1.2 Les paramètres d'apprentissage

Nous savons maintenant utiliser une fonction f pour fournir une réponse. Il faut faire en sorte que cette fonction puisse évoluer sinon aucun apprentissage ne sera possible. Pour cela, nous allons ajouter des paramètres supplémentaires, dit **paramètres internes** ou **paramètres d'apprentissage**. Ces paramètres d'apprentissage sont aussi appelés **poids du réseau** (**weights** en anglais). Ils sont f

utilisés comme des entrées supplémentaires de la fonction qui s'en sert, au même titre que les données de la base, pour calculer le résultat en sortie :



Ces paramètres vont permettre l'apprentissage : en effet, si les réponses fournies par la fonction f sont incorrectes, on peut faire évoluer ces paramètres internes pour tenter d'obtenir une autre réponse, éventuellement meilleure. Durant la phase d'apprentissage, les images en entrée sont des données « constantes » au sens où elles ne sont pas modifiées durant l'apprentissage. La fonction f est généralement mise en place au tout début de l'expérimentation. Elle sera conservée, telle quelle, durant toute la phase d'apprentissage. **Les seuls éléments pouvant évoluer durant la phase d'apprentissage sont les paramètres internes.**

Prenons un exemple, purement académique ! Nous allons considérer que chaque image possède un unique pixel de valeur x . Nous allons prendre pour fonction un polynôme du second degré pour construire la fonction $f(x,a,b,c) = ax^2+bx+c$. Ainsi les paramètres internes sont : a , b et c . Sans surprise, différentes valeurs pour les paramètres internes fournissent des réponses différentes pour une même valeur d'entrée $x = 2$:

- Pour $a = 1$, $b = 3$ et $c = 5$, la réponse est égale à : $f(x) = 2^2+3x2+5 = 15$
- Pour $a = 3$, $b = 1$ et $c = 3$, la réponse est égale à : $f(x) = 3.2^2+1x2+3 = 17$

Comment initialiser les paramètres internes ? Impossible de répondre à cette question. Ils seront donc initialisés aléatoirement sur une plage de valeurs connue.

1.3 L'erreur

Il nous manque encore une notion essentielle pour que l'apprentissage puisse exister : il faut que l'on soit capable d'évaluer la qualité de la réponse. Pour cela, nous mettons en place une approche utilisant des vérités terrain (des échantillons modèles). Ainsi pour chaque échantillon en entrée, on connaît la réponse exacte :

- Pour l'évaluation de l'âge, nous traiterons une base de photos de personnes et pour chaque photo nous aurons l'âge exact de la personne.
- Pour la classification : on dispose d'un set d'images et pour chacune d'entre elles, on va donner la catégorie associée : chien, chat, grenouille...
- Pour la description automatique d'images : pour chaque image, on dispose d'une phrase décrivant le contenu de l'image.

- Pour la transcription automatique (Speech to Text), nous disposons de plusieurs phrases enregistrées ainsi que leurs transcriptions.

On parle ici d'**apprentissage supervisé**. L'apprentissage non supervisé est une thématique de recherche consistant à catégoriser, par exemple, des photos d'animaux sans connaissance à priori des espèces présentes sur les images.

Une fois que nous avons des échantillons et les réponses attendues, nous pouvons évaluer la qualité du réseau. Pour cela, nous mettons en place une **fonction d'erreur** indiquant si la réponse de la fonction f est éloignée de la réponse attendue. Par convention, la fonction d'erreur doit être positive et sa valeur doit augmenter plus la réponse s'éloigne de la valeur recherchée.

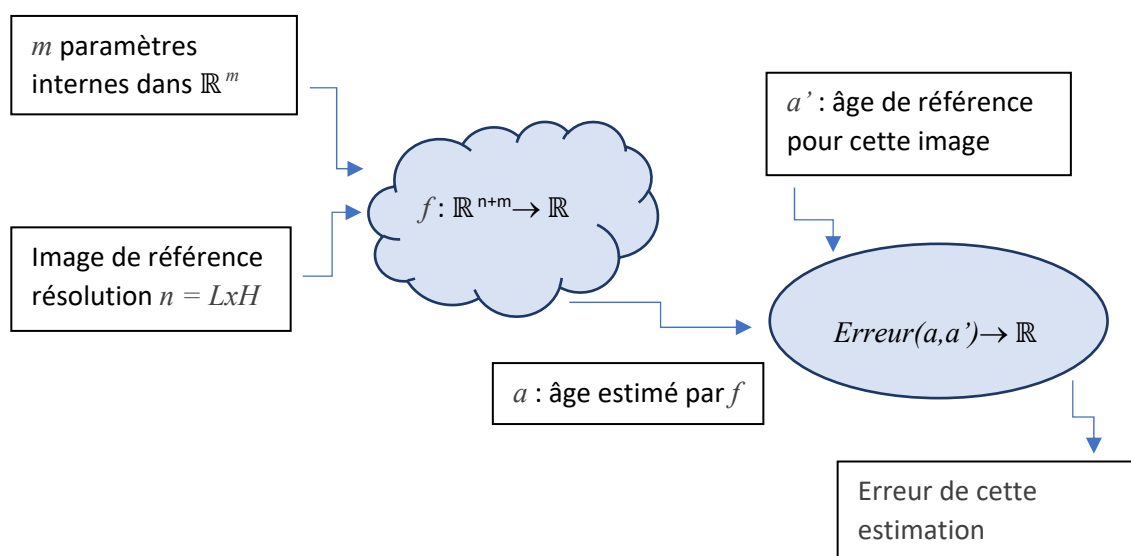
Prenons comme exemple le problème de la détection de l'âge d'une personne. Nous avons en entrée une image IMG et l'âge age_{ref} de la personne présente sur l'image. Nous calculons la réponse age_{estim} fournie par la fonction f correspondant à l'âge estimé. Nous pouvons proposer comme fonction d'erreur la fonction suivante :

$$Erreur(age_{ref}, age_{estim}) = abs(age_{ref} - age_{estim})$$

Voici différentes réponses possibles pour une même image où la personne a 37 ans :

- | | |
|--|---|
| • Age estimé : 37, Erreur = $ 37-37 = 0$ | correct, car on a trouvé la bonne réponse |
| • Age estimé : 39, Erreur = $ 37-39 = 2$ | faible erreur, 2 ans trop vieux |
| • Age estimé : 30, Erreur = $ 37-30 = 7$ | l'erreur augmente car l'écart augmente |
| • Age estimé : 87, Erreur = $ 37-87 = 40$ | 40 ans trop vieux, l'erreur est forte |

Dans cet exemple, la valeur de l'erreur est facile à comprendre car elle correspond à un nombre d'années. Mais rarement la valeur de l'erreur aura une unité, nous ne serons donc pas en mesure de savoir si 0.1 ou 150 correspond à une erreur importante ou non. Cependant, il faut garder à l'esprit que **si la valeur de l'erreur augmente cela suggère que la qualité de la réponse baisse** et réciproquement. Intégrons maintenant la fonction Erreur dans notre flux de calcul :



Durant la phase d'apprentissage, nous nous concentrons sur la valeur de l'erreur et moins sur la réponse estimée. Ainsi, la fonction f va momentanément être mise de côté pour étudier la nouvelle fonction err définie ainsi :

$$Err(I, w, ref_I) = Erreur(f(I, w), ref_I)$$

Où I correspond à une donnée en entrée (une image par exemple), w les paramètres d'apprentissage et ref_I la réponse exacte associée à la donnée en entrée I .

L'apprentissage supervisé, pour être efficace, doit disposer de plusieurs milliers de données en entrée. Plus la taille de l'échantillon de référence est importante, plus vous augmentez la qualité de l'apprentissage. On peut citer quelques bases annotées utilisées régulièrement dans la formation et la recherche :

- MNIST : base d'images correspondant aux chiffres manuscrits de 0 à 9 ; utilisée à l'origine pour construire les machines automatisées de tri postal ; 60 000 images, résolution de 28x28 en niveaux de gris : <http://yann.lecun.com/exdb/mnist/>
- CIFAR10 : base d'images pour la classification d'images. 60 000 images, résolution de 32x32 en couleur, 10 classes : avion, voiture, oiseau, chat, daim, chien, grenouille, cheval, navire et camion. <https://www.cs.toronto.edu/~kriz/cifar.html>
- CIFAR100 : 100 classes cette fois et 600 images par classe.
- ImageNet : base d'images qui sert de référence pour le problème de classification : 1000 classes, plus d'un million d'images en couleur de taille variable.
- COCO : base d'images pour la segmentation et la description d'images : 80 000 images accompagnées de leurs descriptions.

Pour estimer l'erreur totale sur l'ensemble de la base d'apprentissage, il nous suffit de sommer les erreurs induites par chaque entrée de la base :

$$ErrTot(f, w, DataSet) = \sum_{I, ref_I \in DataSet} Err(I, w, ref_I)$$

1.4 Comment apprendre ?

Si nous examinons la fonction $ErrTot(...)$, elle prend en entrée une fonction f , des paramètres internes w et une base d'apprentissage. Sur ces trois entrées, deux sont fixes : la base et la fonction f car la base ainsi que la fonction ne changent pas durant la phase d'apprentissage. Les seules entrées susceptibles d'évoluer sont les paramètres d'apprentissage w . Par conséquent, lorsque nous avons choisi une fonction f et la base à traiter, le problème de l'apprentissage se ramène à cette formulation :

$$\underset{w}{Min} \ ErrTot(w, f, DataSet)$$

Vous devez ici reconnaître un problème d'optimisation familier : la recherche du minimum d'une fonction. Si vous n'êtes pas familier avec ce domaine, voici une méthode d'optimisation naïve pour vous prouver que toute fonction peut être minimisée :

Voici le code associé :

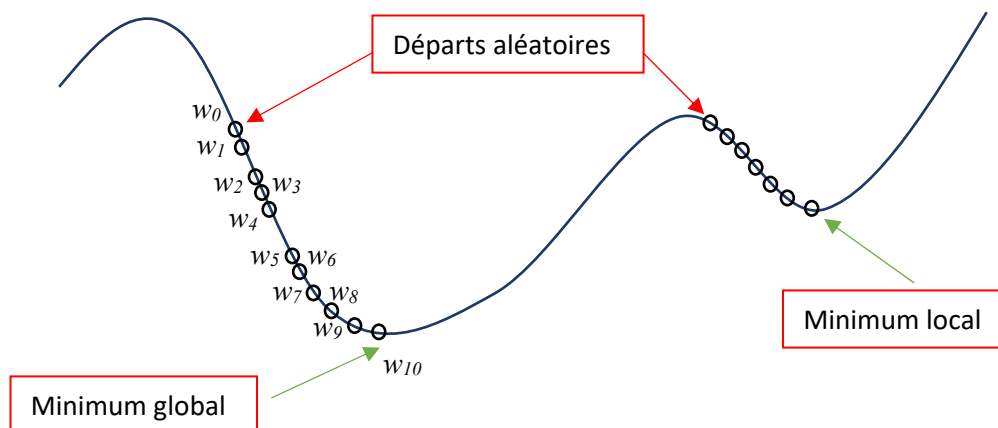
```

Function OptimNaif(ErrTot,w,f,Base) :
    Erreur = ErrTot(w,f,Base)
    Tant qu'il me reste du temps :
        # on génère des poids w' dans le voisinage de w
        pour i allant de 0 à len(w) :
            w'[i] = w[i] + pas * rand(-1,1)
        Erreur' = ErrTot(w',f,Base)
        # si les poids génèrent une erreur moindre, on les garde
        Si Erreur' < Erreur :
            w' → w
            Erreur' → Erreur

```

Ainsi, successivement, nous testons de nouveaux paramètres dans le voisinage des valeurs des paramètres actuels. Si les réponses sont de meilleure qualité, nous conservons ces nouveaux paramètres sinon nous les rejetons. Itérativement, nous diminuons l'erreur. L'apprentissage fonctionne !

Néanmoins, ni cette méthode, ni aucune autre méthode d'optimisation peut nous garantir de converger vers l'erreur optimale. En effet, il existe le problème des minimas locaux qui piègeront tout algorithme de minimisation. Voici un exemple :



Dans la partie gauche du graphique, les paramètres internes générés aléatoirement positionnent $ErrTot(w_0)$ dans une pente qui descend vers le minimum global. En utilisant notre algorithme naïf, nous descendons le long de cette pente chaque fois que de meilleurs paramètres internes sont trouvés. Au final, nous convergions vers le minimum.

En commençant à droite, nous minimisons, de la même manière, l'erreur totale à chaque changement de paramètres internes. Cependant, en suivant cette pente, nous atteignons cette fois un minimum local. Il n'y aura pas de moyen évident pour sortir de ce piège. Bien souvent, il faudra relancer le problème à partir d'une autre position.

1.5 Les écueils de l'apprentissage

Bien que les réseaux de neurones aient permis de grandes avancées en quelques années, il faut rester humble, car les réseaux d'aujourd'hui sont les résultats des travaux de plusieurs milliers de chercheurs ayant effectués des milliers de tentatives avant d'arriver à une solution fonctionnelle. On l'oublie trop souvent. Voici les différents écueils que vous pouvez rencontrer lors de la mise en place d'une méthode d'apprentissage :

La non convergence : une phase d'apprentissage peut ne pas converger ! En effet, si vous prenez un réseau connu, avec sa base d'entraînement et la méthode de gradient utilisée pour son apprentissage et que vous relancez l'apprentissage de zéro, vous n'allez pas forcément obtenir la convergence. Cela semble étrange, mais il faut être conscient que les phases d'apprentissage dépendent énormément des valeurs d'initialisation des paramètres internes et que ces dernières sont générées aléatoirement au lancement. Cela pose problème car certains réseaux sont très difficiles à faire converger et il semble impossible de pouvoir retrouver des paramètres internes qui fonctionnent. Des entités de recherche comme Google ou Microsoft ont pu lancer des expériences en série sur plusieurs jours et sur plusieurs GPU en parallèle. Ainsi, sur 1000 ou 10 000 essais, seuls 1 ou 2 ont pu s'avérer satisfaisants. Avec les moyens que l'on dispose de notre côté, il est difficile de réentraîner de tel réseau from scratch.

Un mauvais score : si votre réseau converge (l'erreur a diminué au fur et à mesure de l'apprentissage) cela ne signifie pas qu'il a atteint un taux de prédiction intéressant. Ne soyez pas impressionnés par un pourcentage, un réseau qui prédit le sexe d'une personne avec un taux de réussite de 60% ne fait en fait que 10% de mieux qu'un tirage à pile ou face qui fournit, de base, une performance de 50 %.

Le surapprentissage : vous avez entraîné un réseau qui obtient un taux de bonnes prédictions de 99%, c'est un miracle ! Non pas forcément. Cette situation peut être synonyme d'un problème qui concerne tout processus d'apprentissage. En effet, si trop de neurones sont présents dans le réseau, il peut apparaître une sorte de phénomène d'apprentissage par cœur. Ainsi, le réseau arrive à identifier chaque image et à retenir la réponse associée. Il n'y a donc plus d'apprentissage au sens où le système n'a pas recherché des critères utiles pour fournir une réponse de qualité, mais il semble avoir appris par cœur la réponse associée à chaque input. Pour prévenir ce phénomène, on sépare la base d'apprentissage en deux : on garde 80% des données pour la phase d'apprentissage et 20% des données pour la phase de validation. Ainsi, on ne présente jamais le groupe de validation durant l'apprentissage et les performances du réseau sont évaluées uniquement sur ses performances par rapport à des données « nouvelles ». Cette approche est très rigoureuse et permet de détecter le surapprentissage.

L'évidence : vous avez entre les mains un système connu pour pouvoir apprendre et si vous le mettez dans des situations évidentes, il ne va pas s'embêter il va aller à la conclusion la plus rapide. Ce phénomène pourra parfois vous dérouter. Ainsi, si vous lancez un problème de classification entre des grenouilles, des dauphins et des lions, en quelques secondes, quelle que soit la taille de la base d'images, vous allez arriver à un score de 100% de prédictions justes. Une nouvelle configuration de réseau très intelligente ? Un algorithme de descente ultra-performant ? Non rien de cela... En faisant

d'autres tests, vous finirez par vous apercevoir qu'avec un seul neurone, le score restera à 100%. Pourquoi ? Parce que pour le réseau, la réponse est évidente. En moyennant les couleurs de l'image, il obtient un ton moyen qui représente sa catégorie : un ton plutôt vert est associé à une grenouille, plutôt jaune au lion et plutôt bleu au dauphin.

Des ensembles déséquilibrés : si vous cherchez à construire un réseau qui distingue les chats et les chiens et que vous fournissez 1000 photos de chats et 10 photos de chiens, le réseau va afficher un très bon score de 99%. En effet, en optimisant la fonction d'erreur, le réseau va vite détecter qu'en répondant toujours « chats » à la question sans regarder l'image en entrée, il a 99% de chance de voir juste car vous avez une majorité de photos de chats dans votre base. C'est idiot comme réaction, mais un réseau en phase d'apprentissage fait ce qu'il a à faire, et s'il trouve une brèche : une réponse facile qui marche à tous les coups, il choisira probablement cette option. Il en est de même des photos de chauve-souris prises de nuit (fond noir) face à des photos d'oiseaux prises de jour (fond bleu clair).

Les phénomènes rares : si vous cherchez à détecter des signes annonciateurs de tremblements de terre dans des relevés sismiques, la tâche peut s'avérer très difficile. En effet, ce sont des phénomènes rares et sur l'ensemble de vos données, par exemple 1000 jours d'enregistrement, il n'y aura qu'un seul tremblement de terre présent. Entraîner un réseau sur des cas aussi rares pose problème car les données sont déséquilibrées en faveur des moments sans évènement.

Remarque : lorsque vous manquez d'échantillons pour votre apprentissage, une technique consiste à utiliser du data augmentation pour simuler des données supplémentaires. On peut par exemple citer pour l'image : une symétrie verticale, une légère translation / rotation, l'ajout de bruit ou de tâches dans l'image d'origine. Ces techniques ont aussi pour objectif de rendre le réseau plus robuste aux perturbations.

1.6 Exercices

1.6.1 Exercice 1

L'apprentissage comportent deux phases distinctes :

- La phase d'apprentissage consistant, à partir d'une base de vérités terrains fixées, à faire évoluer les paramètres internes afin de minimiser l'erreur totale.
- Une phase de production, où les paramètres internes sont maintenant figés : le système se comporte comme un expert auquel on présente de nouvelles informations pour connaître sa réponse.

Que doit-on exporter comme données une fois que le système a fini la phase d'apprentissage ? Ces données doivent permettre au système de fonctionner pendant la phase de production. Entourez les bonnes réponses :

Les images contenues dans la base

La fonction/réseau f

La fonction d'erreur

Les réponses contenues dans la base

La méthode d'optimisation

Les paramètres internes

Remarque : vous pouvez à tout moment reprendre une phase d'apprentissage, même plusieurs jours après l'arrêt de la phase initiale. Pour cela, il vous suffit de relancer l'algorithme d'optimisation avec la fonction et les derniers paramètres internes choisis pour essayer d'obtenir de nouveaux meilleurs paramètres. On peut par ailleurs changer ou étendre la base de référence pour fournir plus d'échantillons.

1.6.2 Exercice 2

Attention, cet exercice est loin d'être facile et doit nécessiter plusieurs minutes de concentration pour fournir une réponse correcte.

Nous cherchons à apprendre le test booléen $T(x,y)$ suivant : 1 si $x \neq y$ et 0 si $x = y$

Pour cela, nous allons mettre en place nos vérités terrains, simple échantillonnage de cette fonction :

Vérités terrains :

x	y	Résultat
0	5	1
4.1	4	1
10	6	1
3	3	0

Construisez une fonction f servant de fonction d'apprentissage :

- Cette fonction doit avoir un seul paramètre interne a permettant l'apprentissage.
- Utilisant les fonctions mathématiques suivantes :
 - Exp, Log
 - Max, Min, Abs, Mean
 - +, /, -, x

Donnez une fonction d'erreur pouvant servir au problème d'apprentissage.

Intuisez une valeur a associé à une erreur totale nulle.

La fonction construite est-elle être identique à la fonction T ?

Comment améliorer le set de vérités terrains pour améliorer la similarité entre f et T ?

Appelez le responsable de salle pour valider vos réponses une fois terminé.

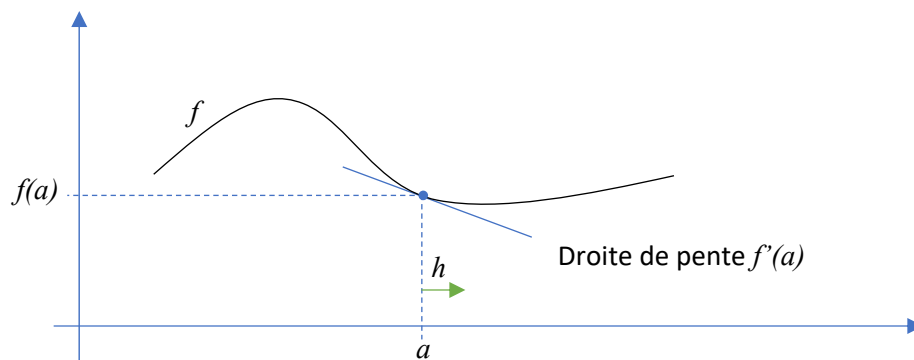
2 La méthode du gradient

2.1 Pour une fonction de $\mathbb{R} \rightarrow \mathbb{R}$

Nous cherchons à minimiser la fonction donnant l'erreur totale. Supposons qu'il s'agisse d'une fonction de \mathbb{R} dans \mathbb{R} . Dans ce cas particulier, le gradient correspond à la dérivée. Nous rappelons ici la formule d'approximation d'une fonction :

$$f(a+h) = f(a) + h \cdot f'(a) + \varepsilon(h) \quad \text{avec } \varepsilon(.) \text{ tend vers 0 lorsque } h \text{ tend vers 0}$$

Ce qui signifie qu'autour de la valeur a , pour une valeur de h « petite », la fonction f se comporte comme une droite de pente $f'(a)$:



On en déduit ainsi facilement la direction dans laquelle la recherche du minimum doit avoir lieu :

- Si $f'(a)$ est positive, alors la fonction croît, il faut donc chercher le minimum à gauche de a
- Si $f'(a)$ est négative, la fonction décroît, il faut donc chercher le minimum à droite de a

En conclusion, il faut se déplacer dans la **direction opposée** au signe de $f'(a)$. Ainsi, nous pouvons mettre en place l'algorithme de descente du gradient suivant :

```

Fonction GradientDescent(f,a) :
    pas = 0.01
    tant que du temps est disponible :
        a = a - pas * f'(a) # le signe - → direction opposée
  
```

Remarque : la méthode de descente du gradient ne garantit pas que la valeur de la fonction diminue à l'itération suivante. Par exemple, lorsque nous sommes proches d'un optimum, on peut « sauter » par-dessus pour atteindre finalement une valeur plus importante. En pratique, lorsque les valeurs atteintes à chaque itération se mettent à osciller, on suppose qu'un optimum est proche et que le pas du gradient est trop important pour continuer la convergence.

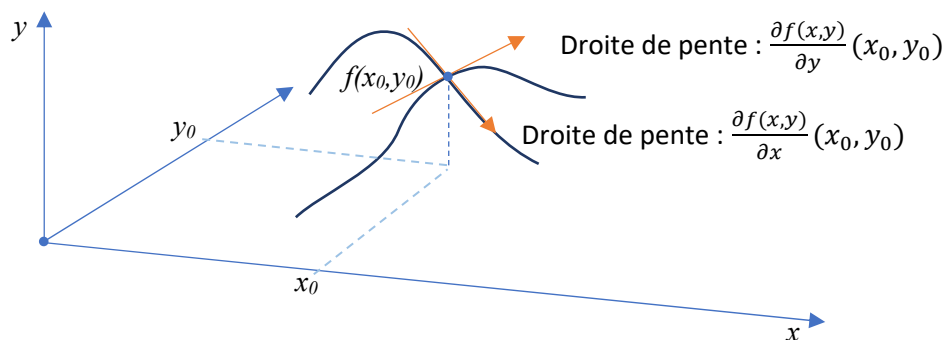
2.2 Pour une fonction de $\mathbb{R}^n \rightarrow \mathbb{R}$

Pour minimiser une telle fonction, nous utilisons les dérivées partielles qui étendent la notion de dérivée d'une fonction de $\mathbb{R} \rightarrow \mathbb{R}$ à une fonction de $\mathbb{R}^n \rightarrow \mathbb{R}$.

Prenons l'exemple d'une fonction à deux variables $f(x,y) \rightarrow \mathbb{R}$. En ajoutant un paramètre de perturbation h très petit, nous obtenons des formules similaires au cas 1D :

$$\begin{aligned} f(x+h,y) &= f(x,y) + h \cdot \partial f(x,y) / \partial x + \varepsilon(h) \\ \text{et} \\ f(x,y+h) &= f(x,y) + h \cdot \partial f(x,y) / \partial y + \varepsilon(h) \end{aligned}$$

Comment s'interprètent les deux dérivées partielles ? Si nous faisons uniquement évoluer la valeur de x , nous obtenons comme dans le cas 1D une droite tangente au point courant $f(x_0, y_0)$. De la même manière, si nous faisons évoluer y , nous obtenons une autre droite tangente en ce point. En combinant ces « deux directions tangentes », nous obtenons un plan tangent qui approche le comportement de la fonction autour du point $((x_0, y_0), f(x_0, y_0))$:



Pour le gradient, nous utilisons la notation : ∇f qui correspond au vecteur des dérivées partielles de la fonction f :

$$\nabla f = \begin{pmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{pmatrix}$$

Ainsi, nous avons :

$$f(x+h_0, y+h_1) \sim f(x,y) + (h_0, h_1) \cdot \nabla f(x,y)$$

Dans cette écriture, le symbole \cdot entre (h_0, h_1) et $\nabla f(x,y)$ désigne le produit scalaire entre 2 vecteurs.

Pour minimiser la valeur de f , la logique est similaire au cas 1D : il faut se déplacer dans le sens opposé au gradient. Ainsi, nous pouvons mettre en place l'algorithme de descente du gradient pour une fonction de $\mathbb{R}^n \rightarrow \mathbb{R}$. A noter que ci-dessous x désigne un vecteur de \mathbb{R}^n :

```
Def GradientDescent(f,x) :  
    pas = 0.01  
    tant que du temps est disponible :  
        x = x - pas *  $\nabla f(x)$     # le signe -  $\rightarrow$  direction opposée
```

Le gradient est un vecteur de taille égale aux nombres de paramètres de la fonction. Par exemple, pour une fonction de $f: \mathbb{R}^n \rightarrow \mathbb{R}$, le gradient est un vecteur avec n paramètres correspondant aux n dérivées partielles de f . Ainsi, on peut effectuer le calcul $x = x - pas * \nabla f$ car x et ∇f sont deux vecteurs de même dimension. A noter que le paramètre pas est un réel et que le signe $*$ désigne la multiplication entre un réel et un vecteur.

2.3 Calcul du gradient automatisé

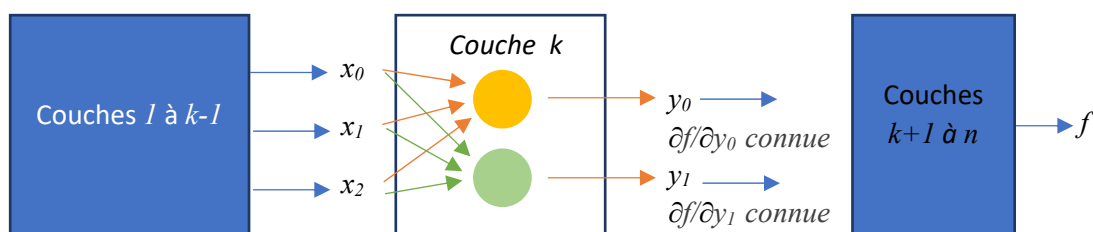
Les bibliothèques de réseaux de neurones (Tensorflow, Pytorch) comme la bibliothèque Numpy (avec Autograd) permettent de calculer le gradient d'une fonction de manière automatique. Pour cela, on utilise un système en trois phases :

- Dans la première phase appelée **Forward**, on calcule la valeur de la fonction. Ici les calculs sont effectués de manière progressive comme on les ferait à la main. Parallèlement, la bibliothèque construit un graph de calcul qui représente le flot d'opérations effectuées pour arriver au résultat final. Dans ce graph sont stockés les résultats intermédiaires de chaque calcul.
- Dans la deuxième phase appelée **Backward** ou aussi **rétropropagation**, la bibliothèque va parcourir le graph de calcul en sens inverse. Cette phase permet de construire toutes les dérivées partielles par rapport aux paramètres internes. Cette phase utilise les formules classiques de dérivation.
- Une troisième phase consiste à remettre à zéro les informations stockées pour le calcul du gradient lors de la rétropropagation. A ce niveau, les résultats intermédiaires stockés dans le graph doivent être mis à zéro pour éviter qu'il soit pris en compte lors de la prochaine itération.

2.4 Le calcul du gradient par rétropropagation

Les réseaux de neurones avec lesquels nous allons travailler sont décrits sous forme de couches (layer). Chaque couche prend les informations en entrée, effectue des calculs sur ces données et fournit des résultats pour la couche suivante. Le réseau complet constitue la fonction f et nous cherchons à déterminer les dérivées partielles de f par rapport à chaque paramètre interne. Une fois les dérivées partielles déterminés, nous connaissons le gradient nécessaire à l'algorithme de descente.

Durant la passe Backward, nous allons partir de la couche terminale (à droite) et reculer à chaque itération pour déterminer les dérivées partielles de la couche précédente. Ainsi, à l'itération $n-k$, nous pouvons modéliser notre système comme ceci :



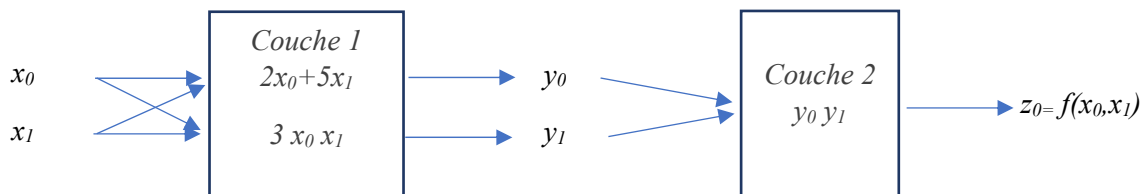
Les itérations précédentes ont permis de déterminer les valeurs de $\frac{\partial f}{\partial y_0}$ et $\frac{\partial f}{\partial y_1}$ et maintenant nous cherchons à déterminer les valeurs de $\frac{\partial f}{\partial x_0}$, $\frac{\partial f}{\partial x_1}$ et $\frac{\partial f}{\partial x_2}$. Pour cela, nous utilisons le théorème général des dérivations :

$$\frac{\partial f}{\partial x_i} = \sum_{j=0}^m \frac{\partial f}{\partial y_j} \times \frac{\partial y_j}{\partial x_i}$$

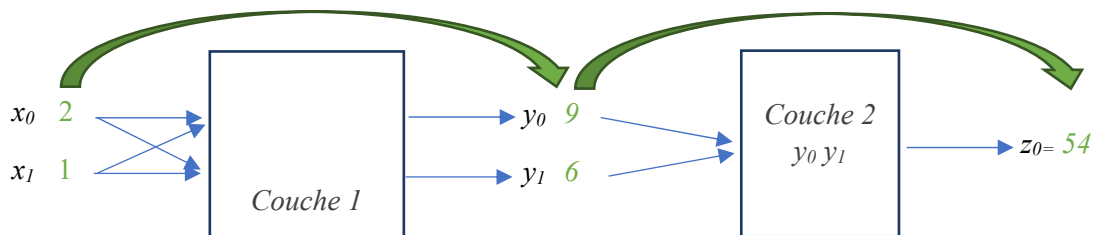
Les valeurs de $\partial f / \partial y_j$ ont été déterminées à l'itération précédente. Les termes $\partial y_j / \partial x_i$ correspondent aux dérivées partielles des fonctions utilisées dans la couche courante k . Ces fonctions sont connues et par exemple, on peut avoir $y_3 = \ln(x_1)$. On peut donc exprimer leurs dérivées en utilisant les formules de dérivation : $\partial y_3 / \partial x_1 = \ln'(x_1) = 1/x_1$. Comme les valeurs des x_i sont connues depuis la passe Forward, il peut évaluer l'expression : $1/x_1$ pour connaître la valeur de $\partial y_3 / \partial x_1$. Il reste donc à évaluer la formule du théorème général de dérivation pour trouver la valeur de $\partial f / \partial x_i$.

Exemple : $f(x_0, x_1) = (2x_0 + 5x_1) \times (3x_0 x_1)$

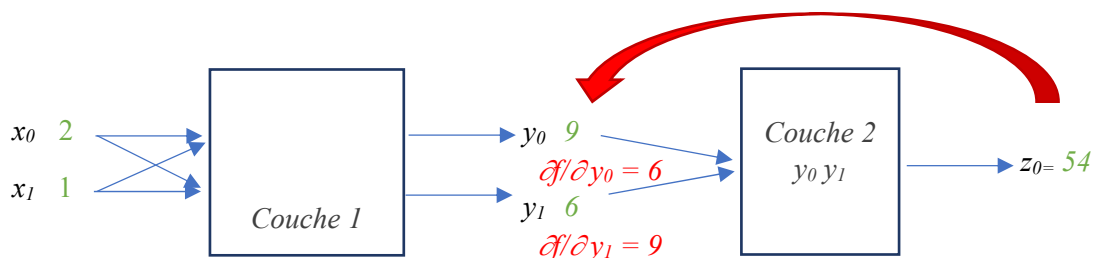
Voici le graph de calcul que nous pouvons associer à cette fonction :



Prenons $x_0=2$ et $x_1=1$. Nous allons tout d'abord effectuer la passe Forward :



La passe Forward est maintenant terminée. Nous commençons la rétropropagation en démarrant par la droite. La couche 2 effectue le calcul suivant $z_0(y_0, y_1) = y_0 \times y_1$. Les formules classiques de dérivation nous donnent $\partial z_0 / \partial y_0 = \partial(y_0 y_1) / \partial y_0 = y_1$. Or la valeur y_1 est connue car elle a été déterminée durant la passe Forward : $y_1=6$. Nous connaissons donc la valeur de $\partial f / \partial y_0$. Le raisonnement est identique pour y_1 : $\partial f / \partial y_1 = \partial z_0 / \partial y_1 = \partial(y_0 y_1) / \partial y_1 = y_0=9$.



Nous allons maintenant passer à l'étape suivante où nous allons rétropropager les valeurs des dérivées $\partial f / \partial y_0$ et $\partial f / \partial y_1$ pour trouver $\partial f / \partial x_0$ et $\partial f / \partial x_1$. Dans cette configuration plus complexe où $\partial f / \partial x_0$ dépend de $\partial f / \partial y_0$ et $\partial f / \partial y_1$, nous devons utiliser le théorème général :

$$\frac{\partial f}{\partial x_0} = \frac{\partial f}{\partial y_0} \times \frac{\partial y_0}{\partial x_0} + \frac{\partial f}{\partial y_1} \times \frac{\partial y_1}{\partial x_0}$$

Il reste à calculer :

- $\partial y_0 / \partial x_0 = \partial (2x_0 + 5x_1) / \partial x_0 = 2$
- $\partial y_1 / \partial x_0 = \partial (3x_0 x_1) / \partial x_0 = 3x_1 = 3 \times 1 = 3$

Ainsi : $\partial f / \partial x_0 = 6 \times 2 + 9 \times 3 = 39$

Nous calculons $\partial f / \partial x_1$ de la même manière :

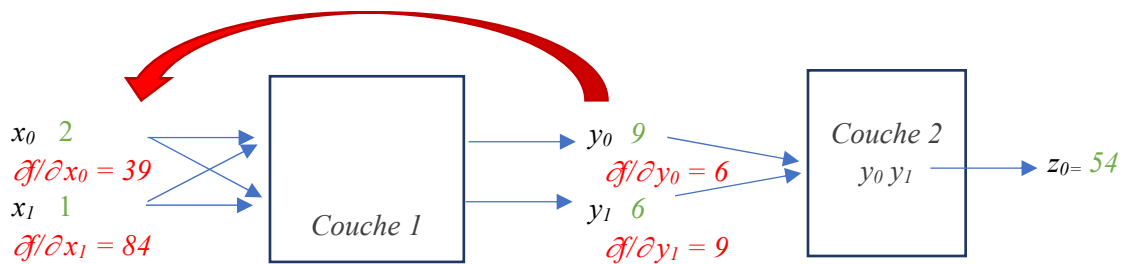
$$\frac{\partial f}{\partial x_1} = \frac{\partial f}{\partial y_0} \times \frac{\partial y_0}{\partial x_1} + \frac{\partial f}{\partial y_1} \times \frac{\partial y_1}{\partial x_1}$$

Il reste à calculer :

- $\partial y_0 / \partial x_1 = \partial (2x_0 + 5x_1) / \partial x_1 = 5$
- $\partial y_1 / \partial x_1 = \partial (3x_0 x_1) / \partial x_1 = 3x_0 = 6$

Ainsi : $\partial f / \partial x_1 = 6 \times 5 + 9 \times 6 = 84$

Nous obtenons finalement :



Quel est l'intérêt de cette approche ? Elle est relativement simple à mettre en place, informatiquement parlant. La méthode de la rétropropagation permet de faire les calculs de dérivation localement dans chaque bloc, ce qui est beaucoup plus simple que de dériver une formule complexe avec des centaines de variables.

D'ailleurs, nos résultats sont-ils justes ? Pour vérifier, plutôt que de dériver $f(x_0, x_1)$ nous allons plutôt utiliser la formule d'approximation pour vérifier notre calcul :

$$\partial f / \partial x_0 = [f(x_0 + \varepsilon, x_1) - f(x_0, x_1)] / \varepsilon$$

$$\partial f / \partial x_1 = [f(x_0, x_1 + \varepsilon) - f(x_0, x_1)] / \varepsilon \quad \text{avec } \varepsilon = 0.001$$

2.5 Exercices

2.5.1 Exercice 1 : descente du gradient pour une fonction de $\mathbb{R} \rightarrow \mathbb{R}$

Supposons que l'erreur totale s'écrive sous la forme : $Err(x,a) = a^2 + a.x + 1$

- x : donnée en entrée (constante)
- a : paramètre d'apprentissage.

Valeur à l'itération courante :

$$x = 3 \text{ et } a_0 = 5$$

Combien vaut $Err(a_0)$?

$$Err(a_0) = 5^2 + 3.5 + 1 = 41$$

Exprimez $Err'(a)$?

$$Err'(a) = 2a + x$$

Combien vaut $Err'(a_0)$?

$$Err'(5) = 2 \times 5 + 3 = 13$$

Devons-nous prendre pour prochaine valeur de a une valeur inférieure ou supérieure à la valeur courante ?

La dérivée est positive, on doit chercher à gauche de a

Prenons une valeur pour le *pas* de $1/13$, donnez la nouvelle valeur de a : a_1 ?

$$a_1 = a_0 - pas = 5 - 1 = 4$$

Calculez $Err(a_1)$?

$$Err(a_1) = 4^2 + 3.4 + 1 = 29$$

Que pouvons-nous conclure ?

Cette itération a permis de diminuer l'erreur totale

2.5.2 Exercice 2 : descente du gradient pour une fonction de $\mathbb{R}^2 \rightarrow \mathbb{R}$

Supposons que l'erreur totale s'écrive sous la forme : $Err(x, a, b) = a^2 + b.x + 1$

- x : donnée en entrée (constante)
- a, b : paramètres d'apprentissage.

Valeur à l'itération courante :

$$x = 6, a_0 = 3 \text{ et } b_0 = 2$$

Combien vaut $Err(a_0, b_0)$?

$$Err(a_0, b_0) = 3^2 + 2 \times 6 + 1 = 21$$

Exprimez $\nabla Err(a, b)$?

$$\partial Err / \partial a = 2a$$

$$\partial Err / \partial b = x$$

$$\nabla Err(a, b) = (2a, x)$$

Combien vaut $\nabla Err(a_0, b_0)$?

$$\nabla Err(a_0, b_0) = (2 \times 3, 6) = (6, 6)$$

Prenons une valeur de $pas = 1/6$,
donnez les nouvelles valeurs a_1, b_1 ?

$$\begin{aligned} (a_1, b_1) &= (a_0, b_0) - pas. \nabla Err(a_0, b_0) \\ &= (3, 2) - (6, 6)/6 \\ &= (2, 1) \end{aligned}$$

Calculez $Err(a_1, b_1)$?

$$Err(a_1, b_1) = 2^2 + 1 \times 6 + 1 = 11$$

Que pouvons-nous conclure ?

Cette itération a diminué l'erreur totale

Si nous avions un pas de 4, quels
auraient été les valeurs de a_1, b_1 ?

$$\begin{aligned} (a_1, b_1) &= (a_0, b_0) - pas. \nabla Err(a_0, b_0) \\ &= (3, 2) - 4 \times (6, 6) \\ &= (-19, -20) \end{aligned}$$

Calculez $Err(a_1, b_1)$?

$$Err(a_1, b_1) = (-19)^2 - 20 \times 6 + 1 = 242$$

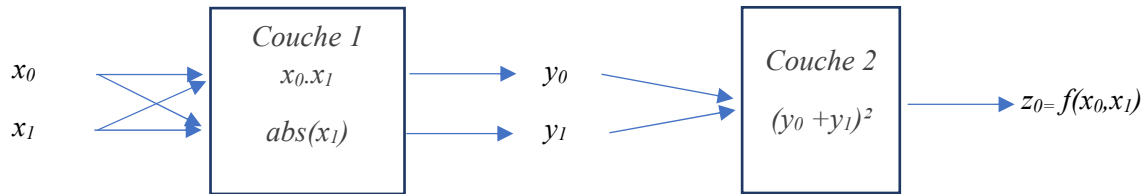
Que pouvons-nous conclure ?

La valeur de pas est trop importante
En conséquence, cette itération a fait
augmenter l'erreur totale

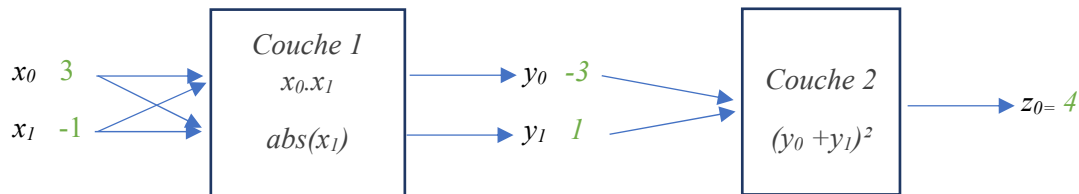
2.5.3 Exercice 3 : rétropropagation

Calculez le gradient de $f(x_0, x_1) = [(x_0 \cdot x_1) + \text{abs}(x_1)]^2$ en $x_0=3$ et $x_1=-1$

Graph de calcul associé à cette fonction :

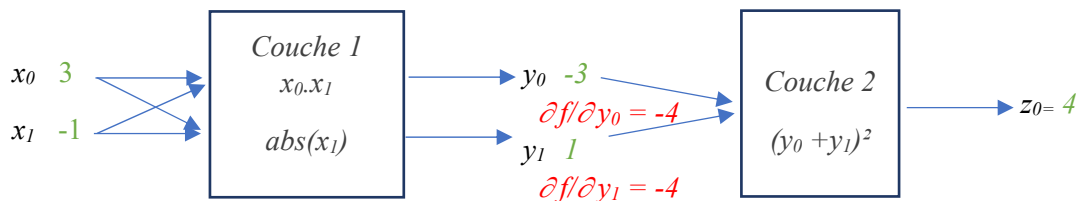


Passé Forward :



Rétropropagation sur le bloc de droite :

$$\partial z_0 / \partial y_0 = \partial((y_0 + y_1)^2) / \partial y_0 = 2(y_0 + y_1) = -4 \quad \text{et} \quad \partial z_0 / \partial y_1 = \partial((y_0 + y_1)^2) / \partial y_1 = 2(y_0 + y_1) = -4.$$



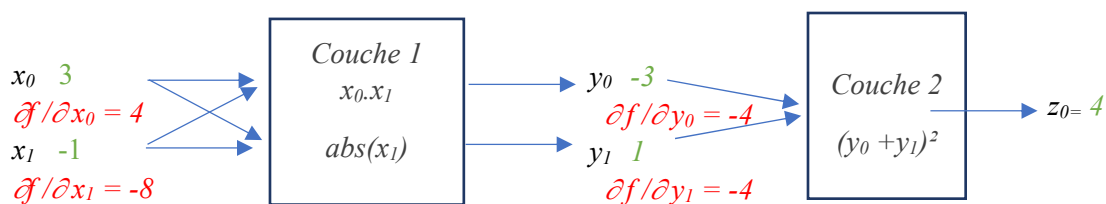
Rétropropagation sur le bloc de gauche :

- $\partial y_0 / \partial x_0 = \partial(x_0 \cdot x_1) / \partial x_0 = x_1 = -1$
- $\partial y_1 / \partial x_0 = \partial \text{abs}(x_1) / \partial x_0 = 0$

Ainsi : $\partial f / \partial x_0 = \partial f / \partial y_0 \times \partial y_0 / \partial x_0 = -4 \times -1 = 4$

- $\partial y_0 / \partial x_1 = \partial(x_0 \cdot x_1) / \partial x_1 = x_0 = 3$
- $\partial y_1 / \partial x_1 = \partial \text{abs}(x_1) / \partial x_1 = -x_1 / |x_1| = -1$ car $x_1 < 0$

Ainsi : $\partial f / \partial x_1 = \partial f / \partial y_0 \times \partial y_0 / \partial x_1 + \partial f / \partial y_1 \times \partial y_1 / \partial x_1 = -4 \times 3 + -4 \times -1 = -8$



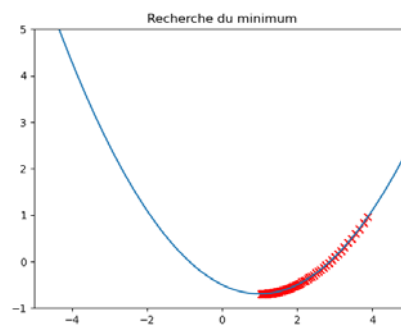
2.5.4 Exercice 4 : descente du gradient

Nous vous proposons de programmer l'algorithme du gradient pour trois fonctions différentes :

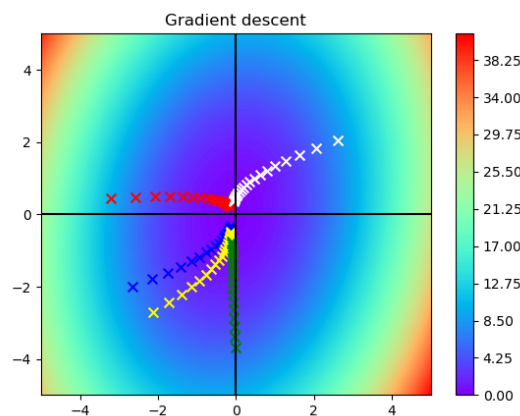
- Fichier « gradient descent.py » avec une fonction $\mathbb{R} \rightarrow \mathbb{R}$
- Fichier « gradient descent 2D.py » avec une fonction de $\mathbb{R}^2 \rightarrow \mathbb{R}$
- Fichier « gradient descent 2Dmax.py » avec une fonction $\mathbb{R}^2 \rightarrow \mathbb{R}$ utilisant un max

Il est conseillé de calculer littéralement l'expression de la dérivée et d'utiliser cette nouvelle fonction pour calculer le gradient. Voici les résultats graphiques obtenus pour chaque exercice :

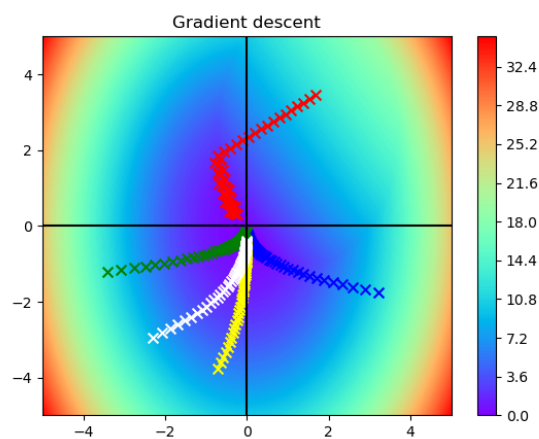
- Minimisation d'une fonction de $\mathbb{R} \rightarrow \mathbb{R}$



- Minimisation d'une fonction de $\mathbb{R}^2 \rightarrow \mathbb{R}$ - représentation par niveaux de couleurs



- Minimisation d'une fonction contenant un max - représentation par niveaux de couleurs



3 Découverte de PyTorch

3.1 Installer PyTorch

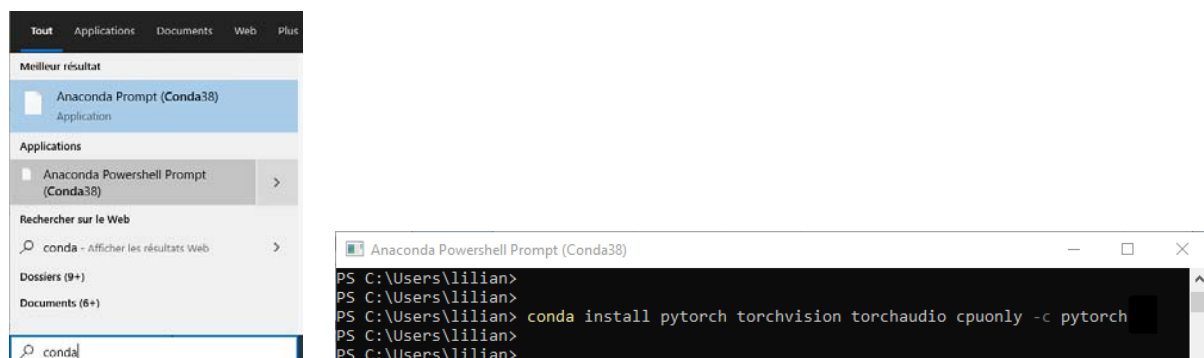
La librairie PyTorch, maintenue par Facebook, permet de mettre en place de puissants réseaux de neurones. De plus, cette librairie offre une grande simplicité d'utilisation et dispose d'une documentation complète en ligne. Elle peut d'exploiter les performances des CPU ou des GPU. Pour son installation, une page web interactive a été mise en place pour expliquer la marche à suivre suivant votre système et votre configuration. Rendez-vous donc sur www.PyTorch.org. Descendez dans la page pour arriver dans la zone contenant le tableau avec les cases orange. Voici les différents choix qui s'offre à vous :

- Par défaut vous installerez avec la version Stable et non le Night Build.
- Choisissez votre OS.
- Si vous travaillez avec Anaconda, vous devez cliquer sur la case CONDA. Si vous travaillez avec une version de Python classique, cliquez sur Pip.
- Sélectionnez Python – **IL VOUS FAUT UNE VERSION 64 bits de Python**
- Nous vous conseillons d'installer la version CPU surtout si vous êtes débutant.

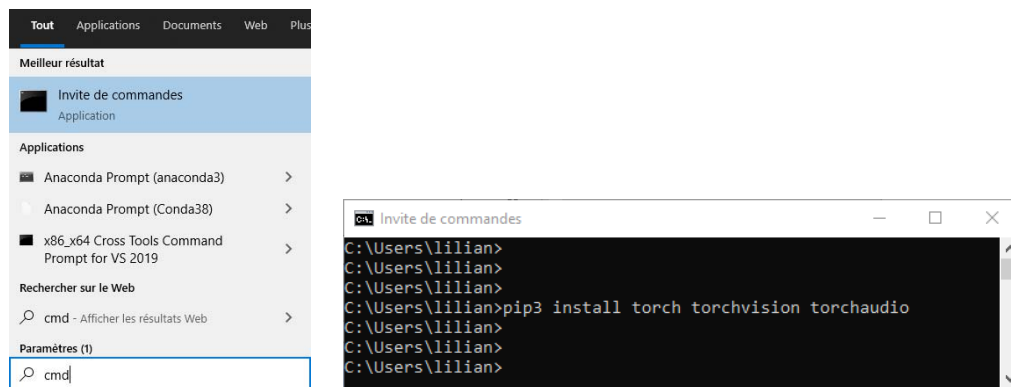
Ci-dessous, vous trouvez les cases cochées pour une installation Windows/Anaconda/Python/CPU. Ensuite, ouvrez la fenêtre « Conda Prompt » et copier-coller la ligne en face de « Run this Command » : conda install...

PyTorch Build	Stable (1.8.1)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 10.2	CUDA 11.1	ROCm 4.0 (beta)	CPU
Run this Command:	conda install pytorch torchvision torchaudio cpuonly -c pytorch			

- **Windows/Anaconda** : lancez une fenêtre Anaconda prompt et recopiez la ligne proposée dans « Run this Command » :



- **Windows/Python** : lancez une fenêtre cmd et recopiez la ligne proposée dans « Run this Command » :



- **Linux/Mac** : lancez une fenêtre shell et recopiez la ligne proposée dans « Run this Command ». Utilisez la commande **pip3** et non **pip** car par défaut sur les distributions Linux/Mac, **pip** et **python** pointent vers une version historique de Python2 présente par défaut sur l'OS. Il faut donc aussi utiliser la commande **python3** pour lancer l'interpréteur Python dans sa dernière version.
- **Plusieurs installations de Python**. Si vous voulez installer Python spécifiquement pour une des versions de Python installée sur votre système, ouvrez une fenêtre de commande (powershell/cmd/shell) et déplacez-vous dans le répertoire où se trouve installée la version de Python que vous recherchez. Utilisez ensuite la commande : `python -m pip install...` en complétant la ligne proposée après « Run this Command ». Si vous ne faites pas ainsi, l'installation s'effectuera avec la version de Python désignée par votre PATH.

3.2 Rappels et conventions

Par convention, dans ce document :

- Le terme **liste** désigne une liste Python
- Le terme **tableau** dans ce document désigne un Numpy array
- Le terme **tenseur** désigner un tenseur Pytorch

Création d'une liste en Python :

```
L = [ 1, 2, 3, 5, 8]          # liste
L = [ [1,2,3,4], [5,6,7,8]] # liste de listes
```

Pour créer un tableau, voici les différentes options :

```
A = np.zeros((3,2))          # crée un tableau de taille (3,2) rempli de 0
A = np.ones((3,2))           # crée un tableau de taille (3,2) rempli de 1
A = np.empty((3,2))          # crée un tableau sans initialiser les valeurs
```

Il est possible de convertir des listes en tableaux et réciproquement :

```
A = np.array(L)              # convertit une liste Python en Numpy Array
L = A.tolist()               # convertit un Numpy array en liste Python
```

Remarque : dans les exercices, nous allons donner des valeurs d'exemple souvent représentées par des listes de nombres. Ces listes seront converties en tenseurs pour être traitées par la librairie Pytorch qui fournira en sortie un tenseur résultat. Pour afficher des graphiques afin d'illustrer ces résultats, nous utiliserons ensuite des librairies tierces qui requièrent souvent des Numpy Array. Il faudra donc convertir le tenseur résultat vers une liste ou un tableau.

Ce document n'a pas pour but de présenter les manipulations des listes et des tableaux. Nous allons nous concentrer sur les tenseurs PyTorch.

3.3 Création et conversion de tenseurs

Pour créer un tenseur, nous utilisons la fonction `FloatTensor` qui crée par défaut un tenseur en flottant 32 bits stocké dans la RAM de l'ordinateur, pas dans le GPU. On peut initialiser un tenseur indifféremment à partir d'une liste Python ou d'un tableau Numpy :

```
import torch, numpy
ListePython = [[1,2,3], [4,5,6]]
A = torch.FloatTensor(ListePython)
print(A)
ArrayNumpy = numpy.array(ListePython)
B = torch.FloatTensor(ArrayNumpy)
print(B)
```

Ce qui donne, dans les deux cas, le même résultat :

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

On peut définir aléatoirement un tenseur avec la fonction `rand()` :

```
a = torch.rand((2,3))
==> tensor([ [0.6004, 0.9491, 0.7909],
             [0.3673, 0.5234, 0.1315] ])
```

On peut convertir un tenseur vers une liste Python ou un tableau Numpy en utilisant les fonctions `tolist()` et `numpy()`. Voici un exemple :

```
import torch
A = torch.FloatTensor([[1,2,3],[4,5,6]])
print(A.tolist())      # => liste python
print(A.numpy())       # => numpy array
```

Dans le cas particulier, où le tenseur ne contient qu'une seule valeur, on peut utiliser la fonction `item()` pour l'extraire. Voici un exemple :

```
A = torch.FloatTensor([2])
print(A.item()) # => numérique python
=> 2.0
```

3.4 Taille d'un tenseur : fonction shape

La propriété shape d'un tenseur nous permet de connaître sa taille :

```
import torch
A = torch.FloatTensor([ [[0,0,0,0],[1,1,1,1],[2,2,2,2]],
                        [[0,0,0,0],[1,1,1,1],[2,2,2,2]] ])
print(A.shape)
=> torch.Size([2, 3, 4])
```

Si nous examinons la liste de listes de listes Python : [[[0,0,0,0], [1,1,1,1], [2,2,2,2]], [[0,0,0,0], [1,1,1,1], [2,2,2,2]]], le plus bas niveau est constitué de listes de 4 entiers. Cette taille correspond à la valeur la plus à droite dans la dimension [2,3,4] du tenseur. Plus on imbrique de niveaux, plus le tenseur a de dimensions :

- Une liste de 4 entiers est associée à un tenseur de taille (4)
- 3 listes de listes de 4 entiers sont associées à un tenseur de taille (3,4).
- 2 listes de 3 listes de listes de 4 entiers sont associées à un tenseur de taille (2,3,4).

3.5 Indexer un tenseur

La syntaxe pour indexer un tenseur utilise des **virgules**. Faites attention, car la syntaxe avec des crochets existent aussi mais offre moins de possibilités. Chaque indice doit respecter la plage de valeurs correspondant à la dimension du tenseur :

$A.shape = [2, 3, 4] \Rightarrow A[i, j, k]$ avec $0 \leq i < 2$, $0 \leq j < 3$, $0 \leq k < 4$

Pour accéder à un élément du tenseur, nous écrivons :

```
A = torch.FloatTensor([ [[0,0,0,0],[1,1,1,1],[2,2,2,2]],
                        [[0,0,0,0],[1,1,1,1],[2,2,2,7]] ])
print(A[1,2,3])
=> 7
```

3.6 Correspondance matrice / tenseur

Dans le cas deux dimensions, si vous voulez faire correspondre les lignes de votre tenseur avec les lignes d'une matrice, vous devez permuter les indices x et y correspondant aux indices de colonnes et de lignes. Ainsi, il faut écrire ces indices dans l'ordre inverse :

$A[y,x]$ # élément de la matrice, colonne x et rangée y

3.7 Plage d'indices et opérateur :

Il est possible lorsque l'on modifie un tenseur de traiter en une commande plusieurs valeurs associées à une plage d'indices, on parle du mécanisme de **slicing**. Pour cela, on utilise l'opérateur « : » qui a plusieurs significations suivant le contexte. Pour ceux connaissant Matlab ou Python/Numpy, le principe de fonctionnement est similaire :

```
a = torch.tensor([[1,2,3,4],[6,7,8,9]])
print( a[0] )      # 1ere ligne
```



```
print( a[1] )      # 2eme ligne
print( a[0,1:] )   # 1ere ligne, tous les indices >=1
print( a[0,:2] )   # 1ere ligne, tous les indices < 2
print( a[0,-1] )   # 1ere ligne, 1er élément en partant de la fin
print( a[0,:] )    # 1ere ligne, et tous les éléments
```

Ce qui donne :

```
==> tensor([1, 2, 3, 4])
==> tensor([6, 7, 8, 9])
==> tensor([2, 3, 4])
==> tensor([1, 2])
==> tensor(4)
==> tensor([1, 2, 3, 4])
```

On peut étendre encore ce système pour sélectionner un sous-tenseur :

```
a = torch.tensor( [ [10,11,12,13],
                    [20,21,22,23],
                    [30,31,32,33],
                    [40,41,42,43] ] )
print( a[1:3,1:3] )    # sélection des a[i,j] avec 1≤i<3 et 1≤j<3
a[1:3,1:3] = 99        # affectation des indices sélectionnées
print(a)
```

Ce qui donne comme résultat :

```
==> tensor([[21, 22],
            [31, 32]])
==> tensor([[10, 11, 12, 13],
            [20, 99, 99, 23],
            [30, 99, 99, 33],
            [40, 41, 42, 43]])
```

Si vous voulez utiliser un pas, la syntaxe est la suivante :

Indice de début : Indice de fin : pas

```
a = torch.tensor( [ [10,11,12,13,14,15],
                    [20,21,22,23,24,25],
                    [30,31,32,33,34,35],
                    [40,41,42,43,44,45] ] )
a[:, 0:6:2 ] = 99      # affectation des colonnes 0 2 4
==> tensor([ [99, 11, 99, 13, 99, 15],
            [99, 21, 99, 23, 99, 25],
            [99, 31, 99, 33, 99, 35],
            [99, 41, 99, 43, 99, 45]])
```

3.8 Indexage complexe

Chaque nouvelle version de PyTorch propose de nouvelles fonctionnalités pour faciliter l'utilisation des tenseurs. Nous présentons certaines de ces syntaxes, pas forcément très courantes, mais qui pourront parfois vous aider.

Indexation sur une liste d'indices :

```
a = torch.tensor( [ [10,11,12,13],
                    [20,21,22,23],
                    [30,31,32,33],
                    [40,41,42,43] ] )
i = torch.LongTensor( [1,3] ) # les indices sont des entiers 64 bits
a[i,:] = 7 # affectation des lignes d'indice 1 et 3
==> tensor([ [10, 11, 12, 13],
              [ 7,  7,  7,  7],
              [30, 31, 32, 33],
              [ 7,  7,  7,  7]])
```

Indexation sur une liste de booléens :

```
a = torch.tensor( [ [10,11,12,13],
                    [20,21,22,23],
                    [30,31,32,33],
                    [40,41,42,43] ] )
i = torch.BoolTensor( [True,True,False,True])
a[i,:] = 7 # affectation sur les lignes associées à la valeur True
==> tensor([ [ 7,  7,  7,  7],
              [ 7,  7,  7,  7],
              [30, 31, 32, 33],
              [ 7,  7,  7,  7] ])
```

Ce qui peut donner des utilisations originales comme :

```
a = torch.tensor( [ [10,11,12,13],
                    [20,21,22,23],
                    [30,31,32,33],
                    [40,41,42,43] ] )
b = torch.tensor([1,2,13,14])
a[b>10,:] = 99 # affectation pour les lignes associées à un indice > 10
==> tensor( [ [10,11,12,13],
               [20,21,22,23],
```

```
[99,99,99,99],
[99,99,99,99] ] )
```

3.9 La recopie

Nous rappelons que dans le langage Python, la syntaxe « A = B » fait en sorte que la variable A désigne le même objet que la variable B. Ainsi, A ne désigne pas un nouvel objet et par conséquent toute modification sur A, se reporte sur B. Cette remarque est valable pour les tenseurs :

```
T = torch.ones((2,2))
A = T
A[0,0] = 5
print(T)
print(A)
```

Ce qui donne comme résultat :

```
==> tensor([[5., 1.],
            [1., 1.]])
==> tensor([[5., 1.],
            [1., 1.]])
```

Pour déclencher une copie d'un tenseur, il faut utiliser la fonction `clone()`. Cependant, nous travaillons dans une librairie de deep learning et un tenseur contient aussi des informations supplémentaires nécessaires pour la phase de rétropropagation. Ainsi, si vous ne voulez copier que les données, et désassocier le tenseur du graph de calcul, il est préférable d'utiliser en plus la fonction `detach()`.

```
T = torch.ones((2,2))
A = T.clone().detach() # ou clone() si besoin
A[0,0] = 5
print(T)
print(A)
```

Ce qui donne :

```
==> tensor([[1., 1.],
            [1., 1.]])
==> tensor([[5., 1.],
            [1., 1.]])
```

3.10 Les vues

Lorsque l'on manipule des tenseurs de grande taille, il est préférable de mettre en place un système qui évite les copies inutiles. Ainsi, la librairie PyTorch permet à un tenseur d'être une « vue partielle » d'un tenseur existant. Le tenseur/vue et le tenseur d'origine partagent les mêmes données.

3.10.1 Vue par slicing

Par exemple, un slicing retourne une vue :

```
T = torch.ones((100,100))
A = T[0:50,0:50] # A est une vue partielle de T
A[0,0] = 5
print(T.storage().data_ptr() == A.storage().data_ptr())
print(T[0,0])
```

Ce qui retourne :

```
==> True          # création d'une vue, A et T ont le même champ de données
==> tensor(5.)
```

3.10.2 Vue par sélection

Une autre manière d'obtenir une vue de manière implicite est de sélectionner une dimension d'un tenseur :

```
T = torch.FloatTensor([ [0,1,2,3],[1,2,3,4],[2,3,4,5]],
                        [[10,11,12,13],[11,12,13,14],[12,13,14,15]] ] )
A = T[0]
B = T[0][0]
print(A)
print(B)
print(T.storage().data_ptr() == A.storage().data_ptr())
print(T.storage().data_ptr() == B.storage().data_ptr())
```

Ce qui retourne :

```
==> tensor([[0., 1., 2., 3.],      # T[0] : sélection du tenseur T de dim (3,4)
            [1., 2., 3., 4.],
            [2., 3., 4., 5.] ])
==> tensor( [0., 1., 2., 3.])      # T[0][0] : sélection du tenseur T de dim (4)
==> True
==> True
```

3.10.3 Vue par reshape

Lorsque nous effectuons un reshape sur un tenseur, on obtient généralement une vue :

```
T = torch.ones((2,3))
A = T.reshape(6)
A[0] = 9
print(T)
print(T.storage().data_ptr() == A.storage().data_ptr())
```

Ce qui donne :

```
==> tensor([9., 1., 1., 1., 1., 1.])
```

```
==> True
```

Cependant, les choses ne sont pas toujours évidentes avec la fonction reshape, regardons la documentation :

TORCH.TENSOR.RESHAPE

`Tensor.reshape(*shape) → Tensor`

Returns a tensor with the same data and number of elements as `self` but with the specified shape. This method returns a view if `shape` is compatible with the current shape. See `torch.Tensor.view()` on when it is possible to return a view.

See `torch.reshape()`

Parameters

shape (*tuple of python:ints or int...*) – the desired shape

La dernière ligne nous indique que la fonction reshape() prend comme paramètre un tuple. Ainsi, pour redimensionner un tenseur T, l'appel à la fonction reshape s'écrit :

```
T = T.reshape((...,...))
```

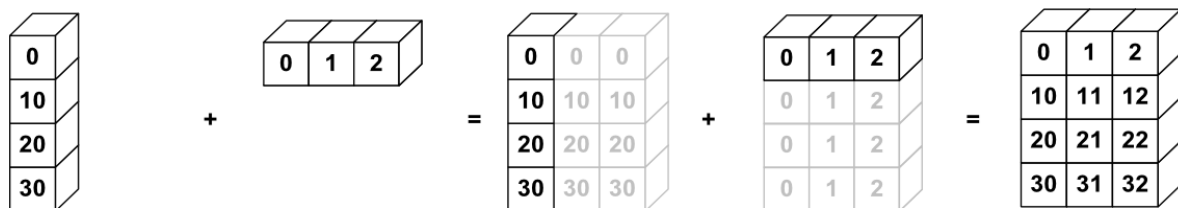
Prenons un exemple avec 6 valeurs dans un tenseur 1D : [1,2,3,4,5,6]. Nous pouvons grâce à la fonction reshape() modifier la dimension du tenseur pour :

- Une dimension (3,2) : [[1,2], [3,4], [5,6]]
- Une dimension (2,3) : [[1,2,3], [4,5,6]]
- Une dimension (2,1,3) : [[[1,2,3]], [[4,5,6]]]
- Une dimension (6,1,1) : [[[1]], [[2]], [[3]], [[4]], [[5]], [[6]]]
- Une dimension (1,1,6) : [[[1,2,3,4,5,6]]]

Remarque : l'utilisation de la fonction reshape peut retourner une vue mais aussi une copie lorsque sa mécanique interne le décide. La plupart du temps, dans les cas standards : création/chargement d'un tenseur puis application d'un reshape, une vue sera créée. Mais si vous appliquez un reshape sur un tenseur dont la taille n'est pas totalement compatible, le retour sera probablement une copie de cet élément.

4 Tenseurs et broadcasting

Pourquoi avoir choisi le terme de tenseur (tensor en anglais) plutôt que celui de matrice ? En fait, les tenseurs sont des containers de données comme les tableaux, les listes ou les dictionnaires. Ils ne sont pas spécialement liés aux applications linéaires : un tenseur n'a pas vocation à rechercher des valeurs propres ou à servir pour un changement de base. Cependant, les tenseurs héritent des opérations matricielles de base comme l'addition, mais pas seulement, car les tenseurs permettent d'étendre ces opérations grâce à la méthode du broadcasting. Par exemple, ajouter un vecteur colonne à un vecteur ligne est en math impossible, cependant, pour un tenseur cela a un sens. Voici comment on procède :



Si vous ajoutez un tenseur de taille (4,1) avec un tenseur de taille (1,3), tout se comporte comme si chaque tenseur était dupliqué autant de fois que nécessaire pour obtenir deux tenseurs de taille (4,3). Le résultat obtenu est ainsi un tenseur de taille (4,3).

4.1 Logique du broadcasting

Si intuitivement on comprend comment les opérations se déroulent, voici une manière plus rigoureuse de les expliquer. Notons (a_0, \dots, a_n) et (b_0, \dots, b_m) les dimensions des deux tenseurs en entrée. Si $n \neq m$, il suffit de compléter par la gauche avec des 0 la taille d'un des deux tenseurs afin d'obtenir $n = m$.

Tout d'abord avant une opération de broadcasting, PyTorch vérifie si les dimensions des tenseurs sont compatibles :

Les dimensions des tenseurs A et B sont compatibles si $\begin{cases} a_i = 0/1 \text{ ou} \\ b_i = 0/1 \text{ ou} \\ a_i = b_i \end{cases}$

Voici quelques exemples :

- $(3,4) + (3)$ donne $\rightarrow (a_0, a_1) = (3,4)$ et $b_0 = 3$. Nous devons ajouter un 0 sur la gauche de la dimension de b pour obtenir : $(b_0, b_1) = (0,3)$. Nous avons ainsi pour l'indice 0 : $a_0=0$ ce qui convient. Mais, pour l'indice 1 : $a_1, b_1 \neq 0,1$ et $a_1 \neq b_1$ donc les dimensions sont incompatibles.
- $(1,1,4) + (1,3)$ donne $(a_0, a_1, a_2) = (1,1,4)$ et $(b_0, b_1, b_2) = (0,1,3)$. Nous avons $a_0, a_1 = 1$ ce qui convient, mais cependant $a_2, b_2 \neq 0,1$ et $a_2 \neq b_2$ donc les dimensions sont incompatibles.

Finalement, la taille (s_0, \dots, s_n) du tenseur de sortie est donnée par la formule :

$$s_i = \max(a_i, b_i)$$

Voici quelques exemples pour trouver la taille du tenseur résultat :

- $(3,4) + (1) \rightarrow (3,4) + (0,1) \rightarrow (3,4)$

- $(3,4) + (1,1) \rightarrow (3,4)$
- $(3,1) + (4) \rightarrow (3,1) + (0,4) \rightarrow (3,4)$
- $(3,1) + (1,4) \rightarrow (3,4)$
- $(1,1,4) + (1,3,1) \rightarrow (1,3,4)$
- $(1,1,4) + (3,2,1) \rightarrow (3,2,4)$

Notons (a_0, a_1, a_2) et (b_0, b_1, b_2) les tailles des deux tenseurs en entrée A et B. Le comportement de PyTorch lorsque l'on effectue l'opération $C = A + B$ en mode broadcasting se traduit par le code suivant :

fonction : $\gamma(i, \text{size})$ retourne i si $\text{size} > 1$, 0 sinon

Avec $0 \leq i < \max(a_0, b_0)$:

Avec $0 \leq j < \max(a_1, b_1)$:

Avec $0 \leq k < \max(a_2, b_2)$:

$$C[i, j, k] = A[\gamma(i, a_0), \gamma(j, a_1), \gamma(k, a_2)] + B[\gamma(i, b_0), \gamma(j, b_1), \gamma(k, b_2)]$$

4.2 Exemple : tenseur 2D + tenseur valeur

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])
B = torch.FloatTensor([[7]]) # ou ([7])
R = A + B
print(R)
```

Le tenseur A a une dimension $(a_0, a_1) = (4, 3)$ et le tenseur B une dimension $(b_0, b_1) = (1, 1)$. Ces deux dimensions sont compatibles, car b_0 et $b_1 = 1$. Le tenseur résultat a donc une dimension de $(\max(4, 1), \max(3, 1)) = (4, 3)$. Voici comment est construit le tenseur R par broadcasting :

$$\text{Avec } 0 \leq i < 4 \text{ et } 0 \leq j < 3 : R[i, j] = A[i, j] + B[0, 0]$$

Voici le résultat obtenu :

```
tensor([[ 7.,  7.,  7.],
        [17., 17., 17.],
        [27., 27., 27.],
        [37., 37., 37.]])
```

4.3 Exemple : tenseur 2D + tenseur colonne

Voici un nouvel exemple :

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])
B = torch.FloatTensor([[1],[2],[3],[4]])
R = A + B
print(R)
```

Le tenseur A a une dimension $(a_0, a_1) = (4, 3)$ et le tenseur B une dimension $(b_0, b_1) = (4, 1)$. Ces deux tailles sont compatibles, car $a_0 = b_0$ et $b_1 = 1$. Le tenseur résultat a donc une taille de $(\max(4, 4), \max(3, 1)) = (4, 3)$. Voici comment est construit le tenseur R par broadcasting :

$$\text{Avec } 0 \leq i < 4 \text{ et } 0 \leq j < 3 : R[i, j] = A[i, j] + B[i, 0]$$

Nous avons comme résultat affiché :

```
tensor([[ 1.,  1.,  1.],
        [12., 12., 12.],
        [23., 23., 23.],
        [34., 34., 34.]])
```

4.4 Exemple : tenseur 2D + tenseur ligne

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])
B = torch.FloatTensor([0,1,2]) # ou [[0,1,2]] même résultat
R = A+B
print(R)
```

Le tenseur A a une dimension $(a_0, a_1) = (4, 3)$ et le tenseur B une dimension $(b_0, b_1) = (0, 3)$. Ces deux dimensions sont compatibles, car $b_0 = 0$ et $a_1 = b_1 > 1$. Le tenseur résultat a donc une dimension de $(\max(4, 0), \max(3, 3)) = (4, 3)$. Voici comment est construit le tenseur R par broadcasting :

$$\text{Avec } 0 \leq i < 4 \text{ et } 0 \leq j < 3 : R[i, j] = A[i, j] + B[i]$$

Voici donc le résultat affiché :

```
tensor([[ 0.,  1.,  2.],
        [10., 11., 12.],
        [20., 21., 22.],
        [30., 31., 32.]])
```

Si nous avions écrit :

```
B = torch.FloatTensor([[0,1,2]])
```

Nous aurions ajouté un tenseur 2D de taille (4,3) à un tenseur 1D de taille (1,3) ce qui produit un tenseur de taille (4,3). Le résultat est donc équivalent : même dimension et mêmes valeurs.

4.5 Exemple : tenseur colonne + tenseur ligne

```
import torch
A = torch.FloatTensor([[10],[10],[10],[10]])
B = torch.FloatTensor([0,1,2]) # ou FloatTensor([0,1,2])
R = A+B
print(R)
```


Le tenseur A a une dimension $(a_0, a_1) = (4, 1)$ et le tenseur B une dimension $(b_0, b_1) = (1, 3)$. Ces deux dimensions sont compatibles, car $b_0 = 1$ et $a_2 = 1$. Le tenseur résultat a donc une dimension de $(\max(4, 1), \max(1, 3)) = (4, 3)$. Voici comment est construit le tenseur R par broadcasting :

$$\text{Avec } 0 \leq i < 4 \text{ et } 0 \leq j < 3 : R[i, j] = A[i, 0] + B[0, j]$$

```
tensor([[10., 11., 12.],
        [10., 11., 12.],
        [10., 11., 12.],
        [10., 11., 12.]])
```

On peut se demander ce qu'il se passe si l'on ajoute un vecteur ligne à un vecteur colonne :

```
R = B + A
print(R)
```

Dans la logique du broadcasting, le résultat est identique, ce ne serait pas le cas en algèbre linéaire.

4.6 Exemple : tenseur 2D + tenseur colonne 3D

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20]])
B = torch.FloatTensor([[[4]],[[5]],[[6]]])
print(A+B)
```

La dimension du tenseur A est (3,3) et celle du tenseur B est (3,1,1).

Le tenseur A a une dimension $(a_0, a_1, a_2) = (0, 3, 3)$ et le tenseur B une dimension $(b_0, b_1, b_2) = (3, 1, 1)$. Ces deux dimensions sont compatibles, car $b_1 = b_2 = 1$ et $a_0 = 0$. Le tenseur résultat a donc une dimension de $(\max(0, 3), \max(3, 1), \max(3, 1)) = (3, 3, 3)$. Voici comment est construit le tenseur R par broadcasting :

$$\text{Avec } 0 \leq i, j, k < 3 : R[i, j, k] = A[j, k] + B[i, 0, 0]$$

```
tensor([[[ 4.,  4.,  4.],
         [14., 14., 14.],
         [24., 24., 24.]],

        [[ 5.,  5.,  5.],
         [15., 15., 15.],
         [25., 25., 25.]],

        [[ 6.,  6.,  6.],
         [16., 16., 16.],
         [26., 26., 26.]])
```

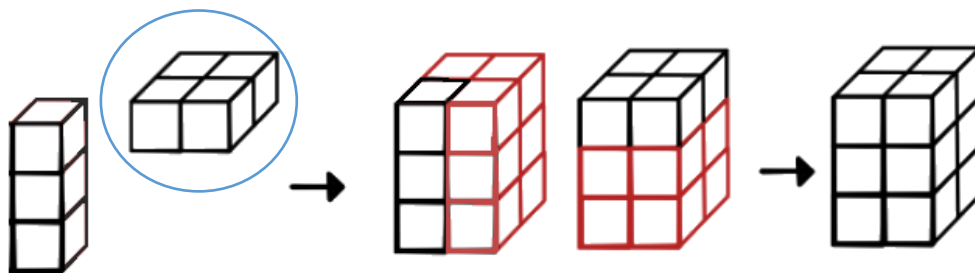
Remarque : si nous oublions par erreur une paire de crochets en écrivant le tenseur B, nous sommes d'après les règles du broadcasting dans un cas tenseur 2D + tenseur colonne : (3,3) + (3,1) ce qui donne un tenseur de taille (3,3) :

```
import torch
A = torch.FloatTensor([[0,0,0],[10,10,10],[20,20,20]])
B = torch.FloatTensor([[4],[5],[6]])
print(A+B)
tensor([ [ 4.,  4.,  4.],
         [15., 15., 15.],
         [26., 26., 26.] ])
```

Ainsi l'oubli d'une paire de crochets a totalement changé le résultat. Il faut donc faire attention, car une erreur dans les manipulations des tenseurs ne provoque pas forcément un message d'erreur. De ce fait, la souplesse et l'adaptabilité des opérations sur les tenseurs fait que l'on doit redoubler de vigilance lorsqu'on les manipule.

4.7 Représentation sous forme graphique

On trouve ce type de schéma sur certains tutoriels internet :



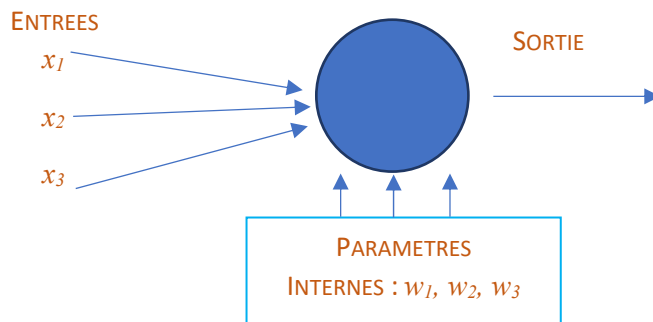
Le tenseur 2D entouré en bleu est un tenseur 2D de dimension (2,2). Il est représenté couché en travers. Le vecteur colonne sur la gauche occupe une troisième dimension, il est donc de taille (3,1,1), ce qui donne après broadcasting un tenseur de taille (3,2,2).

4.8 Exercices

Ouvrez le fichier « Ex tensor.py » et complétez chaque exercice.

5 Rôle des neurones

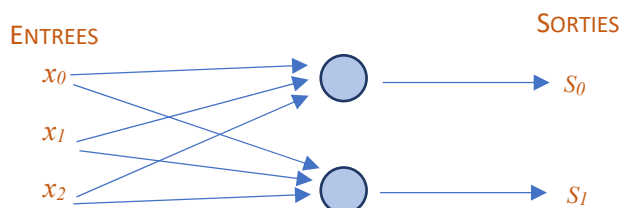
5.1 Le neurone informatique



Voici le schéma d'un neurone informatique. Il compte plusieurs entrées pour une UNIQUE sortie. Les entrées, notées x_i , ainsi que la sortie sont des valeurs numériques. Ses paramètres internes sont appelés *poids*, en anglais **weights**, ce qui donne la notation w_i . On peut parfois rajouter un paramètre optionnel à la sortie appelé biais et noté b . Un neurone effectue une seule opération que l'on peut écrire de la manière suivante :

$$Sortie = \sum_{i=0}^{n-1} x_i * w_i + b$$

Que se passe-t-il lorsque l'on met deux neurones dans la même couche ? Le deuxième neurone aura exactement les mêmes entrées donc le même nombre de poids. Il produira, comme les autres neurones, sa propre valeur de sortie. Voici le schéma correspondant :



Ce type de couche est appelée **Fully Connected** (car chaque neurone est connecté à toutes les entrées) ou encore couche **Linear**. En consultant la documentation PyTorch, nous trouvons à la page :

<https://PyTorch.org/docs/stable/generated/torch.nn.Linear.html>

LINEAR

CLASS `torch.nn.Linear(in_features, out_features, bias=True)`

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports `TensorFloat32`.

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

Vous remarquerez que la description d'une couche Linear ne fait pas mention de neurones ! Dans la documentation, on cite : la taille de l'entrée et la taille de la sortie ainsi qu'un booléen autorisant la présence d'un biais ou non. Cependant, nous savons que la taille de la sortie correspond aux nombres de neurones. Si le tenseur d'entrée est de taille 4 et si nous voulons une couche contenant deux neurones, nous créons donc une couche Linear en donnant comme paramètres : `in_features` égal à 4 et `out_features` égal à 2.

Variables

- **-Linear.weight** – the learnable weights of the module of shape `(out_features, in_features)`. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in_features}}$
- **-Linear.bias** – the learnable bias of the module of shape `(out_features)`. If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in_features}}$

Les poids et si besoin les biais des neurones sont initialisés aléatoirement à partir d'une loi uniforme dont l'intervalle est défini en fonction du nombre d'entrées.

Les poids sont stockés dans le tenseur A , les entrées dans le tenseur x et le biais dans le tenseur b . La formule : $y = xA^T + b$ décrit le calcul mis en place par la couche Linear : une multiplication matricielle est effectuée entre x et A pour construire les valeurs en sortie des neurones. Puis si besoin, un biais est ajouté à chaque valeur.

Voici donc ce que nous obtenons lorsque nous avons n_{out} neurones et n_{in} valeurs d'entrée :

- Le tenseur de sortie y et le tenseur de biais b sont de taille (n_{out})
- Le tenseur d'entrée x est de taille (n_{in})
- Le tenseur A est de taille (n_{out}, n_{in}) et correspond aux $n_{out} \times n_{in}$ poids de la couche

Voici un code qui va nous permettre de tester les paramètres internes de la couche Linear :

```
import torch

layer = torch.nn.Linear(in_features = 4, out_features = 2)
Input = torch.FloatTensor([1,2,3,4])
Output = layer(Input)
```

```
print("Weight : ", layer.weight)
print("Biais : ", layer.bias)
print("Input : ", Input)
print("Output : ", Output)

print("Weight shape : ", layer.weight.shape)
print("Biais shape : ", layer.bias.shape)
print("Input shape : ", Input.shape)
print("Output shape : ", Output.shape)
```

Nous vous conseillons de tester ce code. Voici les résultats obtenus, avec $n=2$ et $m=4$:

```
Weight : Parameter containing:
      tensor([[ 0.1304, -0.2287,  0.0151, -0.0708],
              [-0.0865, -0.3633, -0.3858,  0.1804]], requires_grad=True)
Biais : Parameter containing:
      tensor([0.1493, 0.0297], requires_grad=True)
Input : tensor([1., 2., 3., 4.])
Output : tensor([-0.4158, -1.2191], grad_fn=<AddBackward0>)

Weight shape : torch.Size([2, 4])
Biais shape : torch.Size([2])
Input shape : torch.Size([4])
Output shape : torch.Size([2])
```

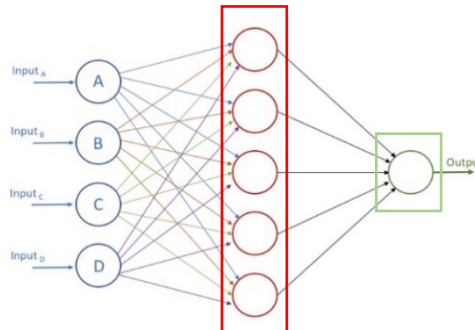
Remarque : les tenseurs stockant les poids et les biais de la couche Linear ont été initialisés directement lors de la création de cette couche. Ces deux tenseurs représentent des paramètres internes du problème d'apprentissage et nous devons donc déterminer leurs gradients. La librairie PyTorch le sait déjà et elle a ainsi annoté ces deux tenseurs avec un flag spécial : *requires_grad=True*. Le tenseur de sortie, Output, a aussi ce flag, car nous avons vu dans l'algorithme de rétropropagation que tous les résultats intermédiaires dans la chaîne de traitement vont servir à rétropropager les calculs intermédiaires du gradient. Le tenseur Input lui n'a pas ce flag car il représente des informations qui restent constantes durant la phase d'apprentissage et aucun calcul de gradient n'est nécessaire à leur niveau.

5.2 Topologie des réseaux

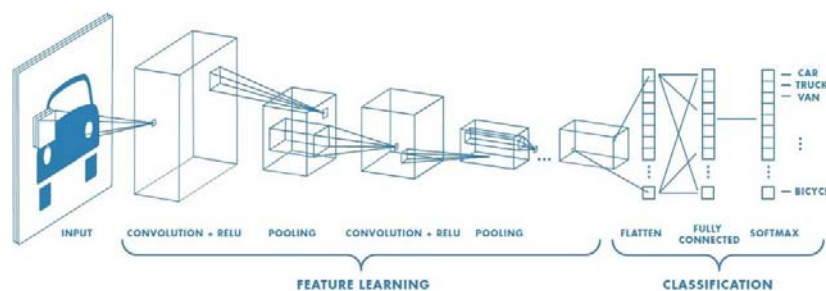
Une couche Linear peut être représentée graphiquement à partir d'un rectangle vertical, contenant des ronds correspondant aux neurones. Les entrées peuvent être aussi regroupées dans une **couche d'entrée**, « **input layer** » en anglais même si aucun neurone n'est présent dans cette couche. Cependant, la couche d'entrée ayant pour fonction de « charger » les données, cette couche a donc

un rôle « actif » et peut être considérée comme une couche à part entière même si elle n'effectue aucun calcul.

L'empilement de plusieurs couches permet de construire un réseau. Les entrées sont généralement situées sur la gauche, les couches successives sont empilées de gauche à droite et la sortie se situe à l'extrémité droite :



Un petit réseau : 4 valeurs d'entrée, puis une couche Linear au centre avec 5 neurones et une deuxième couche Linear à droite avec un seul neurone donnant une unique valeur de sortie.



Un réseau complexe : les entrées correspondent à plusieurs images.

Chaque sortie d'une couche est symbolisée par un volume pour décrire la taille de son tenseur.

Chaque couche porte un nom : CONV / POOL / FLATTEN / FULLY CONNECTED (Linear).

Lorsque qu'une fonction d'activation est présente, son nom est donné : RELU.

5.3 La fonction d'activation

Suffit-il d'empiler des couches pour construire un réseau ? Pas si sûr. En effet, prenons le cas de trois couches Linear sans biais. En utilisant la formule associée, nous obtenons :

$$Out_{Réseau} = Input \cdot {}^tA_1 \cdot {}^tA_2 \cdot {}^tA_3$$

Où chaque tenseur A_i correspond à une couche du réseau. Ce réseau à trois couches équivaut donc à un réseau ayant une seule couche. En effet, il suffit de créer une matrice $M = A_3 \cdot A_2 \cdot A_1$ ce qui nous permet d'écrire :

$$Out_{Réseau} = Input \cdot {}^tM$$

Pour avoir un réseau à plusieurs couches, il faut donc ajouter entre chaque couche une « non-linéarité », c'est le rôle de la fonction d'activation. Cette fonction est une fonction de $\mathbb{R} \rightarrow \mathbb{R}$ non-

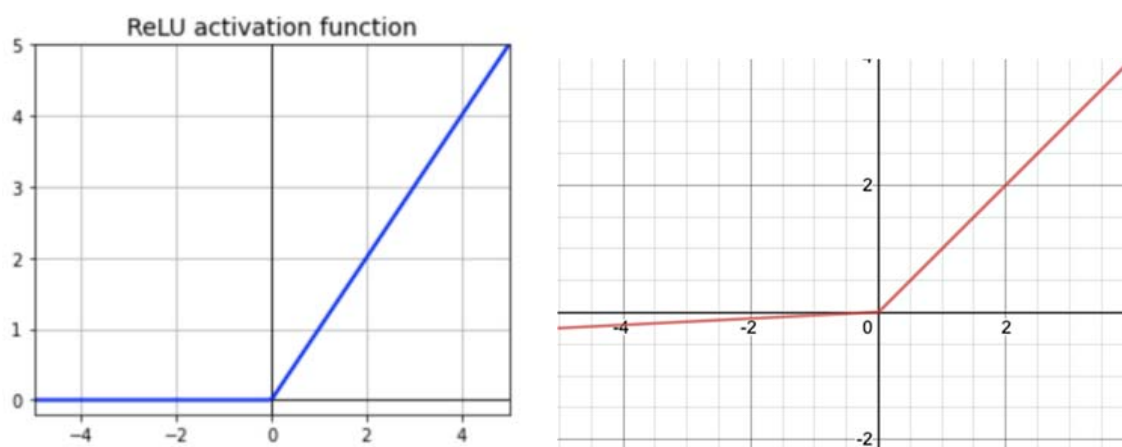
linéaire qui se branche à la sortie de chaque neurone. **Dans une même couche, tous les neurones utilisent la même fonction d'activation.** Les fonctions d'activation n'ont pas de poids.

Une fonction assez populaire aujourd'hui est la fonction *ReLU* définie ainsi :

$$ReLU(x) = \max(0, x)$$

Il s'agit d'une fonction continue, dérivable quasiment partout et peu coûteuse en calcul ! Dans les schémas des réseaux de neurones, vous trouverez ainsi des commentaires sous chaque couche du type LINEAR+ReLU pour indiquer le type de couche utilisée ainsi que sa fonction d'activation. Si cette fonction est performante, elle a le désavantage d'annuler le gradient lorsque x est négatif et lorsque le gradient est nul... impossible d'évoluer dans une direction ou une autre. Ainsi, certains auteurs préfèrent utiliser sa variante *Leaky ReLU*(x) qui masque cet inconvénient :

$$Leaky ReLU(x) = \max(\alpha x, x) \text{ avec } 0 < \alpha < 0.1$$



Avant que la fonction ReLU soit popularisée, on trouvait d'autres fonctions d'activation comme *tanh()* dont les valeurs de sortie sont entre -1 et 1 ou comme la fonction sigmoïde $\sigma(x)$ dont les valeurs de sortie sont entre 0 et 1. La fonction sigmoïde a cependant une propriété notable qui facilite les calculs : $\sigma'(x) = \sigma(x) \times [1 - \sigma(x)]$.

5.4 Créer une couche avec 1 neurone

Nous allons maintenant utiliser une couche Linear avec un seul neurone. Pour cela, nous avons deux étapes :

- En premier, nous créons une couche de type Linear nommée *layer* et nous lui donnons ses caractéristiques internes.
- Ensuite, nous utilisons cette couche *layer* comme une fonction en lui donnant un tenseur d'entrée. La couche *layer* calcule les valeurs de sortie et retourne un tenseur de sortie.

```
import torch
layer = torch.nn.Linear(1,1)
input = torch.FloatTensor([4])
output = layer(input)
print(output)
```

Ce qui donne en sortie un tenseur de taille (1) :


```
tensor([-2.5396], ...)
```

5.5 Affichage du graph associé à un neurone+ReLU

Nous allons tracer le graph d'une fonction associée à un neurone avec sa fonction d'activation ReLU.

La fonction `linspace()` de `numpy` permet d'échantillonner un nombre donné de valeurs dans un intervalle ceci avec un pas régulier. Ainsi `linspace(-2,2,5)` échantillonne 5 valeurs dans l'intervalle $[-2, 2]$ et donne comme résultat : $[-2, -1, 0, 1, 2]$. Pour l'affichage, nous allons utiliser la librairie `matplotlib` et plus particulièrement les fonctions de tracé issues de `pyplot` (`plt`). La fonction `plt.plot()` prend en paramètres une liste d'abscisses x_i et une liste d'ordonnée y_i et trace l'ensemble des points aux coordonnées (x_i, y_i) . La fonction `plt.axis('equal')` force les échelles des deux axes à être égales et pour terminer la fonction `plt.show()` crée la fenêtre d'affichage à l'écran et lance un rendu.

Pour tracer le graph de fonction associée à un neurone, une première approche consiste à faire une boucle pour calculer chaque valeur de sortie. Examinons le code :

```
import torch, numpy, matplotlib.pyplot as plt

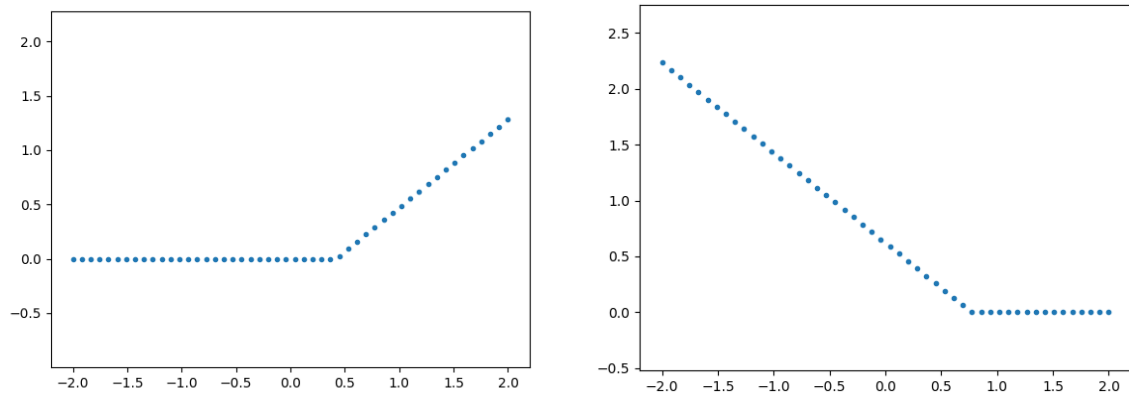
layer = torch.nn.Linear(1,1)    # creation de la couche Linear
activ = torch.nn.ReLU()         # fonction d'activation ReLU
Lx = numpy.linspace(-2,2,50)    # échantillonnage de 50 valeurs dans [-2,2]
Ly = []

for x in Lx:
    input = torch.FloatTensor([x]) # création d'un tenseur de taille 1
    v1 = layer(input)              # utilisation du neurone
    v2 = activ(v1)                 # application de la fnt activation ReLU
    Ly.append(v2.item())           # on stocke le résultat dans la liste

plt.plot(Lx, Ly, '.')            # dessine un ensemble de points
plt.axis('equal')                # repère orthonormé
plt.show()                       # ouvre la fenêtre d'affichage
```

La première ligne : `layer = torch.nn.Linear(1,1)` crée une couche Linear avec un seul neurone. La deuxième ligne : `activ = torch.nn.ReLU()` nous permet de créer une fonction d'activation. Résumons la chaîne de calcul. Nous avons échantillonné différentes valeurs d'entrée stockées dans le tableau `Lx`. Pour chacune de ces valeurs, nous créons un tenseur *input* de dimension (1). Nous passons ce tenseur à la première couche en écrivant : `v1 = layer(input)` ; le résultat est ainsi stocké dans le tenseur `v1`. Nous écrivons ensuite : `v2 = activ(v1)` pour appliquer la fonction d'activation *ReLU* sur le tenseur `v1` et créer le tenseur `v2`. La valeur de l'ordonnée y est récupérée dans le tenseur `v2` en utilisant la fonction `item()`.

Voici le tracé de la sortie d'une couche Linear à 1 neurone avec une fonction d'activation ReLU. Avec une seule valeur en entrée, le neurone effectue le calcul $y = a.x + b$ qui correspond graphiquement à une droite. En appliquant ensuite la fonction `ReLU()`, cela a pour effet de supprimer toutes les valeurs négatives. Nous obtenons un graph correspondant à la fonction $y = \max(0, a.x + b)$ donnant à cette forme de droite tronquée :



Visualisation du signal de sortie d'un neurone avec une fonction d'activation ReLU

5.6 Couche Linear en mode parallèle

Pour traiter plusieurs images, un réflexe serait de créer une boucle et de traiter les images itérativement en les injectant une à une dans la couche Linear. Cependant, la couche Linear peut gérer elle-même cette boucle en interne. Pour cela, au lieu d'injecter itérativement k tenseurs image de dimension (n) , nous injectons en une fois un seul tenseur stockant l'ensemble des images et qui sera de dimension (k,n) . Cette deuxième approche permet d'effectuer les calculs des différentes images en parallèle sur plusieurs cœurs CPU ou GPU. A l'inverse, l'approche basée sur n appels itératifs ne permet pas cette optimisation. En résumé, voici comment se comporte une couche Linear :

Si une couche Linear traite un tenseur de taille (n) pour produire un tenseur de taille (m)

ALORS

Cette couche peut traiter un tenseur de taille (k,n) pour produire un tenseur de taille (k,m)

Nous trouvons cette information dans la documentation officielle de PyTorch au paragraphe Shape : <https://PyTorch.org/docs/stable/generated/torch.nn.Linear.html>

Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out_features}$.

5.7 Exercice

Ouvrez le fichier « Ex Graph Linear.py ». Il reprend le code du tracé d'une couche Linear avec un seul neurone comme vu précédemment.

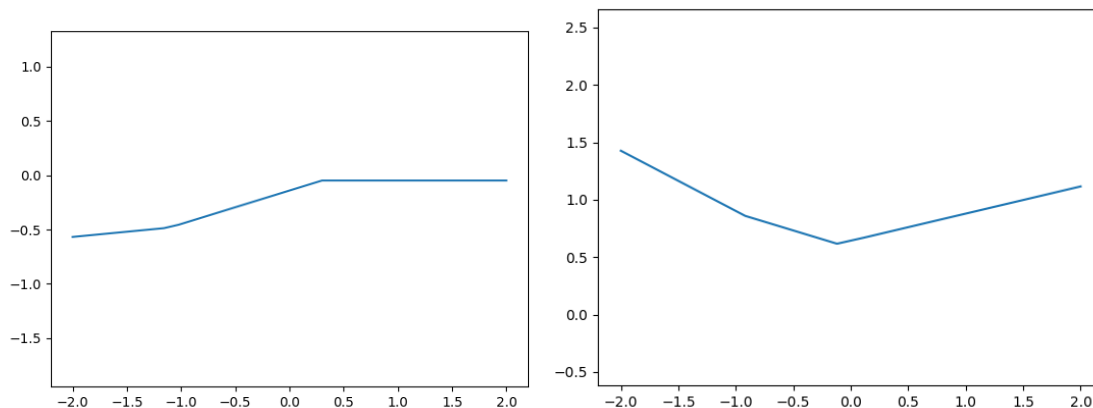
Etape 1 : Retirez la boucle for et utilisez le mécanisme de parallélisation vu ci-dessus.

Etape 2 : Améliorez le réseau : la première couche doit contenir 3 neurones et la deuxième 1 neurone unique. La fonction d'activation en sortie de la première couche sera la fonction ReLU. Conserver le mécanisme de parallélisation et tracez le graphique associé.

Question subsidiaire : Quelle est la dimension du tenseur en entrée de la deuxième couche ?

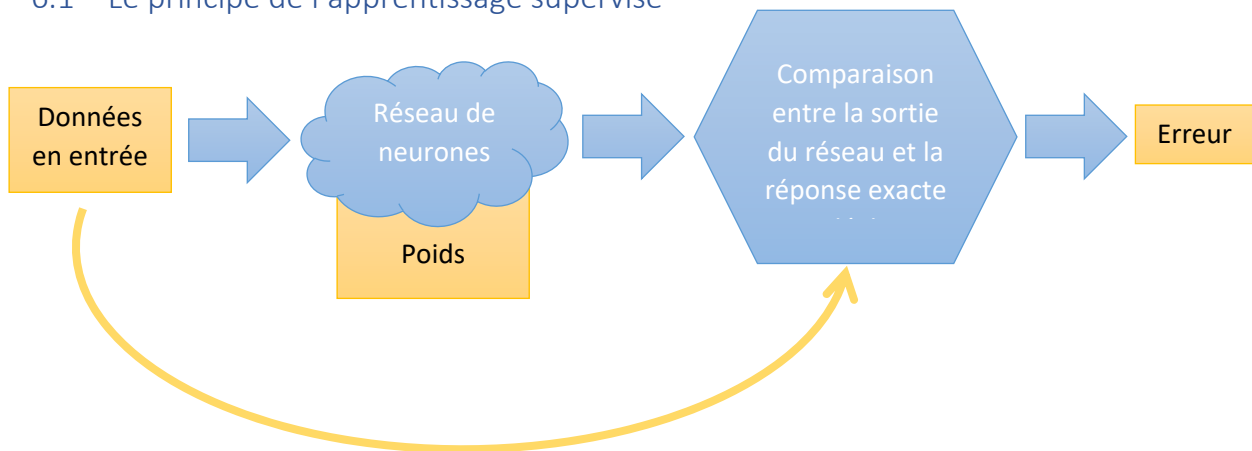
Pour extraire les données d'un tenseur vers un tableau numpy, il faut d'abord le dissocier du graph de calcul en utilisant la fonction `detach()`. Vous pouvez ensuite utiliser la fonction `.numpy()` pour convertir ce tenseur en tableau numpy.

Vous allez obtenir des tracés correspondant à des fonctions linéaires par morceaux. Cela est normal, car la sortie d'un neurone + ReLU est une fonction linéaire par morceau et la combinaison linéaire de plusieurs fonctions linéaires par morceaux est une fonction linéaire par morceaux. De manière pragmatique, le tracé d'un neurone + ReLU crée une seule discontinuité (une cassure) sur le tracé. Avec trois neurones sur la première couche, nous pouvons donc avoir un maximum de trois cassures sur la courbe finale, ce que nous constatons sur nos tracés :



6 L'apprentissage

6.1 Le principe de l'apprentissage supervisé



Nous disposons de données sous forme de tenseurs (une image, un son, un texte...) et nous voulons que notre réseau calcule lorsqu'on lui présente un échantillon la réponse idéale : un score, une température, une probabilité, un âge... Dans l'apprentissage supervisé, nous connaissons la réponse exacte associée à chaque échantillon. En comparant la sortie du réseau de neurones et la réponse attendue, on peut alors mesurer l'équivalent d'une erreur sous la forme d'un nombre réel.

Un tel système peut apprendre ! Pourquoi ? Les données en entrée sont représentées par des tenseurs constants car ces données ne changent pas et restent fixes pendant toute la phase d'apprentissage. Seuls les poids du réseau peuvent varier. On peut ainsi modéliser le réseau comme une fonction de $\mathbb{R}^n \rightarrow \mathbb{R}$ qui associe à un ensemble de poids donnés un nombre réel correspondant à l'erreur.

Plus l'erreur est importante, plus la réponse du réseau s'est éloignée de la réponse idéale. Si l'erreur vaut 0 cela sous-entend que toutes les réponses sont exactes. Ainsi, faire apprendre un réseau de neurones consiste à trouver des valeurs pour les poids qui minimisent l'erreur de sortie associée à l'ensemble des échantillons en entrée. On se ramène donc à un problème d'optimisation où l'on recherche le minimum d'une fonction.

Comment trouver des valeurs pour les poids qui permettent d'obtenir une erreur minimale ? Cette question est difficile car plusieurs minima locaux peuvent exister et lorsque nous détectons un minimum, nous ne savons pas s'il en existe un meilleur. Cependant, si le minimum trouvé est associé à une très faible erreur sur le set de validation, nous pouvons considérer que l'apprentissage s'est bien passé.

6.2 Une méthode d'optimisation naïve

Comment faire évoluer les valeurs W des poids pour diminuer l'erreur du réseau ? Générons au hasard un nouveau vecteur de poids W' proche de W . Calculons alors la nouvelle valeur de l'erreur associée à ces nouveaux poids W' . Si cette nouvelle erreur est plus faible, alors nous remplaçons les poids actuels par ceux de W' . En répétant ce processus, l'erreur diminue ! Voici l'algorithme de principe de cette méthode naïve :

```

Initialiser les poids W au hasard
ErrCourante = N(W,Data)  # N représente la fonction d'apprentissage
  
```

```

Tant qu'il nous reste du temps :
    Choisir des poids W2 proches des valeurs de W
    Err = N(W2,Data)
    Si Err < ErrCourante :
        W = W2
        ErrCourante = Err

```

Cet algorithme fonctionne correctement et nous permet de minimiser l'erreur. Cependant, il n'est pas optimal. En effet, sa performance dépend de sa probabilité à générer de nouveaux poids diminuant l'erreur courante. Lorsque le nombre de poids augmente, les choses deviennent plus difficiles.

6.3 Optimisation par la méthode du gradient

Heureusement, les mathématiques nous viennent en aide et nous donne un moyen de calculer une variation des poids W qui nous permettent de diminuer l'erreur quasiment à chaque fois. Et cerise sur le gâteau, PyTorch, ainsi que toutes les bibliothèques de deep-learning, savent calculer ceci ! La variation idéale s'appelle le gradient, noté $\text{grad } W$ ou encore ∇W . L'algorithme de descente du gradient utilise la formule suivante :

$$W' = W - \text{pas} \times \text{grad } W$$

Le choix de la valeur du pas se fera pour l'instant par essais successifs. Si cette valeur est trop petite, l'erreur va diminuer très lentement. Si cette valeur est trop importante, l'erreur lorsqu'elle approche d'un optimum se mettra à osciller en faisant des rebonds, comme ici : 2, 1.8, 2.5, 2.3, 2.1, 1.5, 2.7, 2.3...

La méthode du gradient s'effectue en deux temps :

- En premier, on effectue la passe Forward où on évalue l'erreur courante en propageant les calculs de l'entrée du réseau vers la sortie. La bibliothèque PyTorch en profite pour stocker des informations intermédiaires utiles au futur calcul du gradient.
- En second, on effectue la passe Backward où on calcule le gradient des poids en rétropropageant les calculs de la sortie du réseau vers l'entrée. Ainsi les gradients des poids présents à chaque étage de la chaîne de calcul sont déterminés.

Nous obtenons donc comme algorithme de principe pour la méthode du gradient sur un réseau :

```

Initialiser aléatoirement les poids W
Choisir une valeur pour le pas
Tant qu'il nous reste du temps :
    ErrCourante = N(W,Data)    # passe Forward
    grad = PyTorch_grad()      # passe Backward
    W = W - grad * pas          # mise à jour des poids

```

6.4 Exemple : mise en place de la méthode du gradient à pas constant

Nous allons programmer une version simple de l'algorithme de la descente du gradient sans utiliser les fonctions avancées de PyTorch. Pour cela, nous traitons une fonction d'apprentissage $f(x, a)$ où x représente les valeurs d'entrée et a un paramètre interne (poids) de la fonction.

Nous notons X le tenseur contenant les n valeurs en entrée et Ref le tenseur des n réponses de référence. Une fonction d'erreur $Err(estim, ref)$ calcule l'erreur entre l'estimation et la valeur de référence.

Il reste quelques points importants à rappeler sous PyTorch :

- Pour que PyTorch calcule le gradient du paramètre interne a , il faut explicitement déclarer que nous avons besoin de connaître son gradient. Pour cela, nous écrivons :

```
a.requires_grad = True
```

- L'erreur totale correspond à la somme des erreurs associées à chaque échantillon. Une fois cette erreur connue et stockée dans un tenseur nommé $ErrTot$, il faudra utiliser la syntaxe suivante pour déclencher la passe Backward :

```
ErrTot.backward()
```

- Pour accéder à la valeur du gradient d'un tenseur T , on utilise la syntaxe :

```
T.grad
```

- En appliquant la formule de l'algorithme de descente : $a = a - \text{pas} * \nabla a$, on effectue une opération sur le tenseur a ce qui a pour effet de modifier son gradient ∇a durant ce calcul ce qui représente une source d'erreur. Ainsi, avant de faire évoluer les valeurs du tenseur a , il faut préciser au moteur PyTorch qu'il ne doit pas, durant un court instant, calculer d'informations relatives au gradient et à la rétropropagation. Pour cela, on utilise la syntaxe :

```
with torch.no_grad() :
```

qui désactive les calculs de gradient dans le bloc de code associé au with.

- Une fois tous les paramètres internes modifiés, il faut réinitialiser le gradient du tenseur a pour la prochaine passe Forward. Pour cela, on utilise la fonction suivante :

```
a.grad.zero_()
```

Ainsi, nous pouvons construire l'algorithme de principe suivant :

```
Fonction Descente du gradient(f,Err,X,Ref,pas) :  
  
    a = torch.FloatTensor(0)  
    a.require_grad = True  
  
    Tant qu'il nous reste du temps :  
  
        # passe Forward  
        ErrTot = torch.FloatTensor(0)  
        Pour i = 0 à n-1  
            y = f(X[i],a)  
            erreur = Err(y,Ref[i])  
            ErrTot += erreur  
        print(ErrTot)  
  
        # passe Backward  
        ErrTot.Backward()  
  
        # algorithme de descente du gradient à pas constant  
        with torch.no_grad() :  
            a -= pas * a.grad  
            a.grad.zero_()
```

NB : on peut remarquer que le tenseur ErrTot est créé sans positionner son flag `require_grad` à `True` alors que ce tenseur intervient dans la rétropropagation. Cela est possible car la librairie PyTorch propage le flag `require_grad`. Ainsi nous avons :

- A la ligne : `y = f(X[i],a)`, le tenseur `a` avec le flag `require_grad` à `True` se combine avec le tenseur `X` dont le flag est à `False`. Ainsi le tenseur résultat `y` a son flag `require_grad` à `True`.
- La tenseur erreur est créé à partir d'un tenseur `Ref` avec son flag à `False` et d'un tenseur `y` avec son Flag à `True`, il aura donc son flag à `True`.
- Le tenseur `ErrTot` est créé avec un flag `require_grad` à `False`. Cependant, à la ligne : « `ErrTot += erreur` », nous le combinons avec un tenseur avec son flag `require_grad` à `True`, ainsi son flag est modifié et passe à `True`.

6.5 Exercice : apprentissage d'une fonction booléenne

On se propose de faire apprendre à une fonction le comportement de l'opérateur booléen \neq (différent) donnant comme résultat :

→ 1 si différent → 0 si égal

La fonction d'apprentissage choisie est :

$$f(x,y,a) = \min(a \times |x-y|, 1)$$

avec a paramètre d'apprentissage et x et y les deux valeurs en entrée.

L'apprentissage doit s'effectuer sur le set d'échantillons suivant :

x	y	Réponse de référence
4	2	1
6	-3	1
1	1	0
3	3	0

Rappel : si l'apprentissage réussit, l'évaluation en dehors de ces échantillons peut cependant être erronée.

Voici les consignes :

- Prenez comme fonction d'erreur : $Err(estim, ref) = (estim - ref)^2$
- Dans cet exercice, vous devez mettre en place l'algorithme de descente du gradient avec un pas fixe correctement choisi.
- Le gradient sera calculé par la librairie PyTorch.
- Vous devez utiliser les fonctions fournies par PyTorch : `torch.abs(.)`, `torch.min(.)`, `torch.sum(.)` sinon la librairie ne sera pas en mesure d'effectuer la rétropropagation.
- Trouvez une valeur pour le pas qui permettent de converger en moins de 100 itérations.

Ouvrez et complétez le fichier d'exercice «Ex grad intro.py».

Pour aller plus loin : on se propose de supprimer la boucle itérant sur les 4 échantillons. Pour cela, il faudra utiliser un unique tenseur stockant les informations des 4 expériences en parallèle. L'erreur totale sera calculée en sommant l'ensemble des valeurs du tenseur stockant les résultats.

7 Utilisation des optimiseurs PyTorch

7.1 SGD

La méthode Stochastic Gradient Descent correspond à notre méthode de la descente de gradient à pas fixe. Voici un extrait de la documentation PyTorch :

SGD

```
CLASS torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0,
weight_decay=0, nesterov=False) [SOURCE]
```

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from [On the importance of initialization and momentum in deep learning](#).

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate

Le premier paramètre de SGD contient la liste des paramètres internes à optimiser. Le second paramètre intitulé « learning rate » correspond au pas dans l'équation de la descente du gradient. Que signifie le terme stochastique ? Pour l'instant nos échantillons sont de petite taille et nous pouvons calculer l'erreur sur la totalité des données en entrée. Cependant, lorsque la taille des données commence à devenir importante, une méthode consiste à sélectionner un sous-ensemble de l'ensemble des informations en entrée pour estimer le gradient. Cette sélection étant faite aléatoirement à chaque itération, on qualifie le gradient de stochastique.

7.2 Pour aller plus loin

Dans la méthode de descente du gradient, vous remarquez que l'on utilise un pas constant ceci pour tous les paramètres internes du système. Cependant, différents paramètres internes peuvent avoir besoin d'être mis à jour à des rythmes différents, en particulier dans les cas de données éparses, où certains paramètres sont plus activement impliqués dans l'extraction de caractéristiques comparés à d'autres. Pour améliorer cette faiblesse, l'optimiseur Adagrad introduit l'idée de mise à jour des paramètres internes avec un pas propre à chaque paramètre :

$$\beta_i^{t+1} = \beta_i^t - \frac{\alpha}{\sqrt{SSG_i^t + \epsilon}} * \frac{\delta L(X, y, \beta)}{\delta \beta_i^t}$$

L'indice i dans l'équation désigne le i -ème paramètre interne et l'indice t représente le numéro de l'itération. La grandeur SSG est la somme des gradients au carré pour le i -ème paramètre à partir de l'itération 0 jusqu'à l'itération t . La grandeur ϵ est utilisée pour indiquer une petite valeur ajoutée à SSG pour éviter une division par zéro. En divisant le pas noté ici α par la racine carrée de SSG , on s'assure que le taux d'apprentissage pour les paramètres changeant fréquemment diminue plus rapidement que le taux d'apprentissage pour les paramètres rarement mis à jour.

7.3 Les optimiseurs avancés

Dans la librairie PyTorch (<https://PyTorch.org/docs/stable/optim.html>), on peut trouver une dizaine d'optimiseurs avancés. Ils ont pour objectif d'apporter une amélioration par rapport à une faiblesse d'un optimiseur plus classique. L'inconvénient est l'apparition de multiples paramètres plus ou moins faciles à exploiter. La différence d'un optimiseur à l'autre est parfois subtile, chacun étant une variation ou le mélange d'un ou plusieurs voisins.

Si nous recherchons une faiblesse dans Adagrad, on peut facilement remarquer que le dénominateur sous le terme α (correspondant au pas constant) a une valeur qui ne cesse d'augmenter car ce terme cumule au fil des itérations de plus en plus de valeurs positives. Cela peut ainsi faire tendre le pas pondéré vers une valeur infinitésimale. Pour résoudre ce problème, on peut limiter l'emprise de ce terme en le moyennant avec le carré du dernier gradient connu :

$$SSG_i^t = \gamma * SSG_i^{t-1} + (1 - \gamma) * \left(\frac{\delta L(X, y, \beta)}{\delta \beta_i^t} \right)^2$$

Le facteur γ , compris entre 0 et 1, introduit un facteur d'affaiblissement (decay factor) des valeurs antérieures, ce qui évite ainsi la saturation à l'infini de ce terme. Cette idée sera le point de départ de plusieurs autres optimiseurs comme Adadelta, RMSprop ou Adam...

Par curiosité, ceux qui voudront tester un autre optimiseur que SGD, pourront prendre **Adam** car cet optimiseur est connu pour avoir souvent de très bonnes performances.

7.4 Mise en place de la forme standard

Beaucoup de tutoriaux sur internet ont tendance à instancier un réseau à partir d'un objet de type Sequential(...). Cet objet permet de donner la description de chaque couche du réseau pour le créer. Après cela, il est transmis à un objet optimiseur qui déroule la séquence de minimisation. Cette approche a le mérite de fournir des programmes très compacts. Elle est pratique et simple pour les non-informaticiens cherchant à mettre en place rapidement et facilement un réseau de neurones et ceci en un minimum de lignes de code.

Cependant, cette approche a un problème important : elle ne permet aucun débogage ou, en tout cas, elle le complexifie. Le suivi de l'évolution de la convergence n'est pas facilité non plus. Nous recommandons donc dans la suite des exercices de créer votre réseau en utilisant la forme type présentée sur la page suivante :

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class Net(nn.Module) :
    def __init__(self):
        super().__init__()
        self.couche1 = ...
        self.couche2 = ...
        self.couche3 = ...

    def forward(self,x):
        x = self.couche1(x)
        x = F.relu(x)
        x = self.couche2(x)
        x = F.max_pool2D(x,2)
        ...
        return x

model = Net()
optim = optim.SGD(model.parameters(), lr=0.5)

for i in range(10000):
    optim.zero_grad()                # remet à zéro le calcul du gradient
    X = input(...)                    # choisit les data
    Y = answers(...)
    predictions = model(X)            # démarrage de la passe Forward
    loss = F.loss(predictions, Y)     # choisit une fonction de loss de PyTorch
    loss.backward()                   # effectue la rétropropagation
    optim.step()                      # algorithme de descente

# n'oubliez pas de rajouter des print pour vérifier un maximum d'info
```

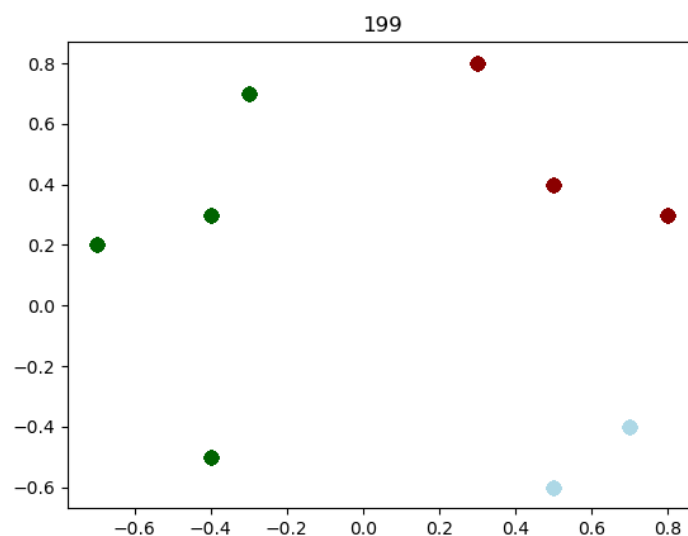
7.5 Exercices

7.5.1 Exercice 1 : apprentissage de catégories

L'exercice consiste à faire deviner à un réseau la catégorie d'un échantillon. Il existe trois catégories numérotées 0,1 et 2.

Chaque échantillon contient deux paramètres (p_1, p_2) et la catégorie associée. Nous représentons donc un échantillon à l'écran aux coordonnées (p_1, p_2) avec un code couleur correspondant à sa catégorie:

- 0 : rouge
- 1 : vert
- 2 : bleu



Nous allons pour commencer utiliser **une unique couche Linear pour prédire trois scores**. Chaque score correspond au score de chaque catégorie. Le plus haut score donne la catégorie prédite par le réseau.

Par exemple, un input est injecté dans la couche Linear. La sortie donne pour valeurs : (1,5,3). Le plus haut score est 5 et il correspond au score de la catégorie 1. Cette catégorie correspond donc à la catégorie prédite par le réseau pour cet échantillon.

Ouvrez le fichier : « Ex apprentissage de forme V1.py ». Vous trouverez dans le code les diverses fonctions Matplotlib utilisées pour le tracé.

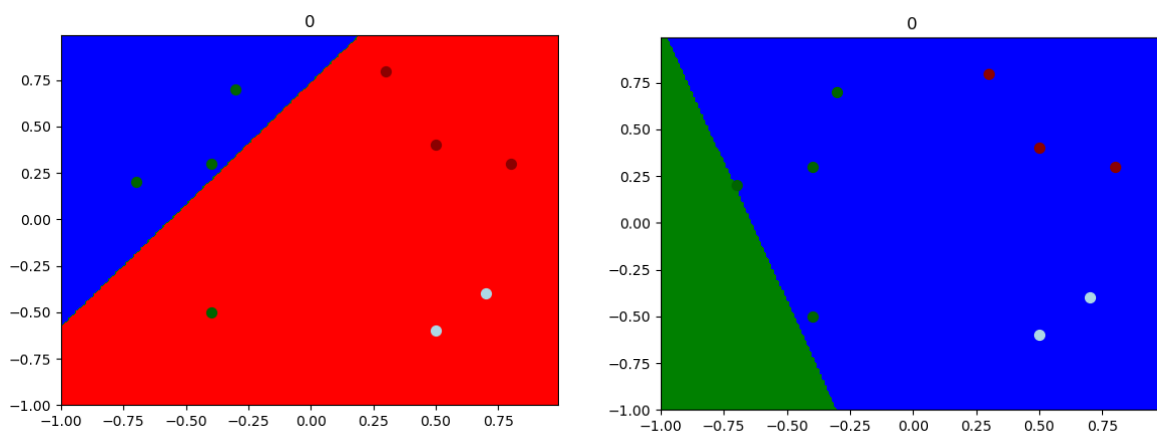
Etape 1 : Mise en place de l'affichage interactif

Créez la couche Linear décrite ci-dessus. La première partie de l'exercice va consister à visualiser en couleur la catégorie de chaque point du graphique associée à la réponse de cette couche :

- Construisez un tenseur qui contient toutes les coordonnées (x,y) des pixels du graphique, pour cela, utilisez les deux tableaux XXXX et YYYY stockant ces valeurs. Si le graphique a une taille de 200x200 pixels, la taille du tenseur T correspondant sera : (200,200,2).

- Construisez le tenseur des scores contenant les scores de chaque pixel. Ce tenseur doit être le résultat du tenseur T injecté dans la couche Linear. Sa taille sera donc de (200,200,3) : 3 scores pour chaque pixel.
- Utilisez la fonction `torch.argmax(axis=???)` pour retourner l'indice du score maximal. Ainsi, nous obtenons la catégorie prédite pour chaque pixel en repérant le maximum des trois scores.
- Convertissez ce tenseur contenant les catégories prédites en un tableau Numpy ayant les mêmes dimensions que le tableau XXXX.
- Complétez la fonction : `ComputeCatPerPixel()` servant à effectuer l'ensemble de ces traitements.

Au lancement du programme, la fonction `ComputeCatPerPixel()` retourne par défaut la catégorie 1, le fond est donc vert uniforme. Maintenant, vous devriez obtenir des écrans de ce type :



Etape 2 : Mise en place de l'apprentissage

Nous vous demandons pour l'instant de traiter les 10 échantillons itérativement. L'erreur totale correspondra à l'erreur générée par chaque échantillon. Créez votre réseau en le mettant sous la forme standard :

- Création d'une classe Python héritant de `nn.Module()`
- Création de la fonction forward qui prend un input (x,y) et retourne trois scores
- Création de la fonction d'erreur qui prend en paramètres :
 - Un tenseur *Scores* contenant les trois scores des différentes catégories
 - La catégorie exacte de l'échantillon : *id_cat*
 - Pour calculer l'erreur, on utilise la formule suivante :

$$Err(Scores, id_cat) = \sum_{j=0}^2 \max(0, Scores[j] - Scores[id_cat])$$

Comment interpréter cette formule ?

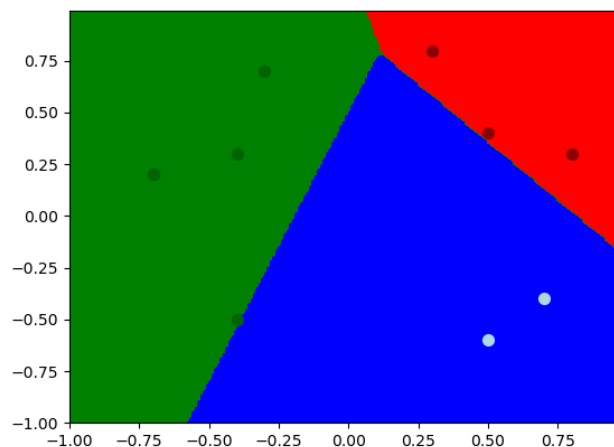
- La grandeur $Scores[j] - Scores[id_cat]$ nous donne l'écart entre le score de la bonne catégorie et le score de la j -ième catégorie.
- Si j correspond à *id_cat*, la contribution à l'erreur vaut 0.
- Lorsque $j \neq id_cat$:

- Si cet écart est positif, cela sous-entend que $Scores[j] > Scores[id_cat]$ et le plus grand score ne correspond pas à la bonne catégorie et ainsi l'erreur augmente.
- Si cet écart est négatif, cela sous-entend que $Scores[j] < Scores[id_cat]$ et le score de la bonne catégorie est supérieur au score courant. Dans ce cas, tout va bien et aucune contribution à l'erreur n'est ajoutée car la fonction max retourne la valeur 0.

Lors de la phase d'apprentissage, pour voir défiler l'animation où l'on voit les zones associées aux catégories s'adapter en live aux points d'entrée, votre code doit avoir la structure suivante :

```
for iteration in range(...):
    ...
    Calcul ...
    ...
    # affichage
    print("Iteration : ",iteration, " ErrorTot : ",ErrTot.item())
    DessineFond()                # dessine les zones
    DessinePoints()              # dessine les données sous forme de points
    plt.title(str(iteration))     # insère le n° de l'itération dans le titre
    plt.pause(2)                 # pause en secondes pour obtenir un refresh
    plt.show(block=False)        # affichage sans blocage du thread
```

Voici le résultat obtenu par cette première version :



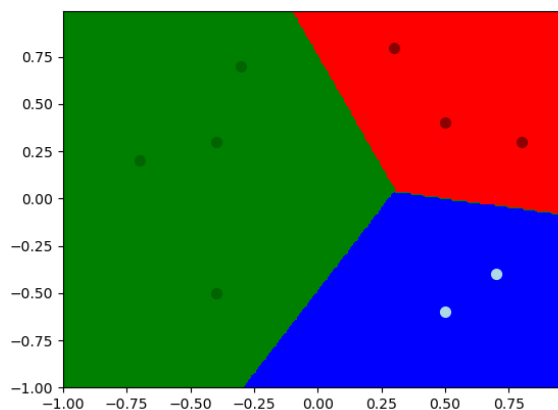
Etape 3 : amélioration de la fonction d'erreur

Vous remarquez que le résultat de l'étape 2 nous permet d'atteindre une erreur nulle car chaque point se situe dans la bonne zone de prédiction. Cependant, comme vous pouvez le remarquer, certains points se trouvent vraiment proches des frontières. Ainsi, si l'on donne en entrée un point très proche du point vert sur le bas du graphique, ce nouveau point peut se situer dans la zone bleue ce qui serait interprété comme une mauvaise prédiction. Nous aurions aimé une disposition des

zones plus équilibrée où les points de référence se trouvent plus à l'intérieur et non aux bords. Pour cela, nous allons modifier la formule de calcul d'erreur pour qu'elle intègre un paramètre supplémentaire permettant d'augmenter l'écart entre le score de la bonne catégorie et les autres scores :

$$Err(Scores, id_cat) = \sum_{j=0}^{nb_cat-1} \max(0, Scores[j] - (Scores[id_cat] - \delta))$$

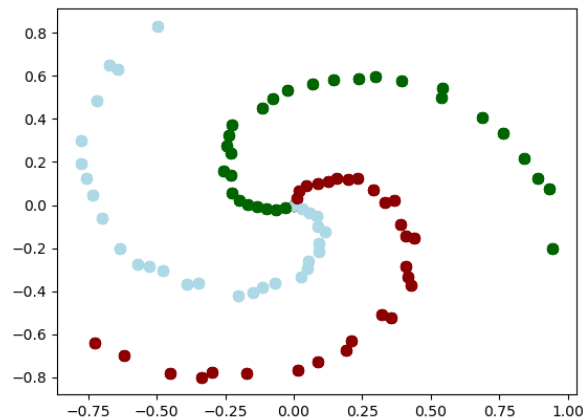
Le paramètre δ permet d'augmenter l'écart entre le score de la bonne catégorie et les autres scores. En effet, pour que l'erreur soit nulle, il faut que $Scores[j] - (Scores[id_cat] - \delta) < 0$ soit $Scores[j] < Scores[id_cat] - \delta$. Autrement dit, l'erreur devient nulle, lorsque l'écart entre le score de la bonne catégorie et les autres scores est d'au moins δ . Dans la suite, on pourra prendre $\delta = 1$ ou 2.



Cette modification se traduit graphiquement par l'éloignement des points de référence des frontières, ces points se retrouvent maintenant plus à l'intérieur des zones de prédiction.

7.5.2 Exercice 2 : apprentissage de formes

On se propose de poursuivre l'exercice précédent sur un ensemble de points plus complexes décrivant une forme en hélice :



Etape 4 : mise en place d'un réseau à deux couches

Reprenez le code de l'exercice précédent et créez un nouveau fichier pour ne pas écraser votre travail. Remplacez la portion de code correspondant à la création des points par le code se trouvant dans le fichier « Ex Appr formes V2.py ».

On se propose de travailler avec 2 couches de neurones :

- Couche 1 : Linear + ReLU
- Couche 2 : Linear

La couche 2 est maintenant chargée de calculer les trois scores en sortie du réseau.

Quelques remarques :

- La fonction ReLU est fournie dans PyTorch : `torch.nn.functional.relu`
- Lancez un affichage toutes les 20 itérations, car ce dernier consomme des ressources

Pour la couche 1, comparez la qualité des résultats obtenus avec 10 / 30 / 100 / 300 / 500 et 1000 neurones. Choisissez un nombre de neurones minimal mais suffisant pour apprendre la forme de l'hélice.

Etape 5 : utilisation du broadcasting

Cette partie est difficile et nécessite que vous soyez à l'aise avec les manipulations de tenseurs.

L'objectif de cette étape est de retirer toute boucle de la passe Forward en utilisant le mécanisme de parallélisation des couches Linear et les opérations sur les tenseurs (max sur un tenseur par exemple).

Voici quelques conseils :

- Préparez un tenseur contenant les données d'entrée et un autre contenant les catégories de référence. Pour les construire, vous pouvez utiliser des boucles 😊 car cette opération est effectuée une unique fois en début de programme.
- Pour extraire les scores en sortie du réseau qui correspondent aux bonnes catégories (`scores[Id_cat]`), vous pouvez utiliser la syntaxe : `Scores[I, CAT]` où `I` est un tenseur d'indices d'entiers 64bits allant de 0 à `nb_points-1` et où le tenseur `CAT` contient les catégories codées sous la forme 0 1 2.
- Pour créer le tenseur d'indices `I`, utilisez la syntaxe : `torch.arange(0,nb, dtype=torch.int64)`

8 Apprentissage supervisé et classification

Dans ce chapitre, nous présentons les différentes notions utiles à la mise en place d'un processus d'apprentissage supervisé pour le problème de la classification. Le problème de classification d'images consiste à deviner la catégorie d'une image parmi une liste de catégories connues (chat ou chien par exemple). Pour cela, on dispose d'une base d'images (un set d'apprentissage) fournissant des milliers d'échantillons utilisés dans l'apprentissage. La base fournit aussi pour chaque échantillon la réponse exacte devant être prédite par le réseau, ainsi on parle d'apprentissage supervisé.

8.1 Base d'entraînement et base de validation

8.1.1 Le surapprentissage

Une des bêtes noires du machine learning est sans aucun doute le surapprentissage (en anglais overfitting). Voici le scénario associé : vous avez entraîné un réseau qui obtient un excellent taux de bonnes prédictions, par exemple 99%, c'est un miracle ! Mais en production, lorsque vous utilisez le réseau préentraîné sur de nouvelles données, les performances sont très médiocres. Cette situation peut être synonyme de surapprentissage. En effet, si trop de neurones sont présents dans le réseau, il peut apparaître un phénomène d'apprentissage par cœur des données d'entraînement : le réseau arrive à identifier chaque échantillon et à fournir la réponse associée. Il n'y a donc pas apprentissage au sens strict où le système aurait recherché des caractéristiques particulières dans les données d'entrée permettant de fournir une prédiction de qualité.

Pour prévenir le phénomène de surapprentissage, on utilise un deuxième ensemble d'échantillons utilisé uniquement durant une deuxième phase de test/validation. On n'utilisera jamais les échantillons du groupe de test/validation durant la phase d'apprentissage. Ainsi, une fois la phase d'apprentissage terminée, les performances du réseau sont évaluées à partir d'échantillons lui étant inconnus. Cette approche est très fiable et permet de détecter le phénomène de surapprentissage.

Suivant la base que vous utilisez, les deux sets peuvent directement être fournis : le set d'entraînement (train) et le set de validation (test). Les deux sets sont organisés exactement de la même manière, le set d'entraînement est généralement 5 fois plus grand que le set de validation. Vous pourrez trouver d'autres bases fournissant un seul set de données. Ce sera donc à vous de le diviser pour construire un set d'entraînement et un set de validation avec une répartition 80% 20% par exemple.

8.1.2 Chargement des sets d'exemples de PyTorch

Vous pouvez trouver la liste des bases d'apprentissage disponibles depuis la librairie PyTorch à la page :

<https://pytorch.org/vision/stable/datasets.html>

On peut trouver plus d'une trentaine de bases sur des sujets très variés. Les bases fournies par les librairies d'apprentissage sont très pratiques, car elles font gagner un temps précieux en nous évitant le décodage de données binaires stockées dans un obscur format. Voici, par exemple, la syntaxe utilisée pour charger les informations contenues dans le set d'entraînement MNIST :

```
TrainSet = datasets.MNIST('../data', train=True, download=True)
```

Examinons la documentation de datasets.MNIST :

MNIST

```
CLASS torchvision.datasets.MNIST(root: str, train: bool = True, transform:
Optional[Callable] = None, target_transform: Optional[Callable] = None, download:
bool = False) [SOURCE]
```

MNIST Dataset.

Parameters

- **root** (*string*) – Root directory of dataset where `MNIST/processed/training.pt` and `MNIST/processed/test.pt` exist.
- **train** (*bool, optional*) – If True, creates dataset from `training.pt`, otherwise from `test.pt`.
- **download** (*bool, optional*) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.

Voici la description des paramètres qui nous intéressent :

- Le premier paramètre (`root`) correspond au répertoire qui va accueillir les données téléchargées. Vous pouvez utiliser `./data/` pour indiquer la racine de votre espace de travail.
- Le booléen `train` permet d'indiquer si l'on désire télécharger le set de données d'entraînement ou de validation.
- Le booléen `download` sera généralement positionné à True pour télécharger les données depuis internet, ceci uniquement au premier lancement. Les données une fois téléchargées seront conservées pour les fois suivantes.

8.2 Préparation des données

8.2.1 Uniformisation des échantillons

De nombreux sets sont maintenant disponibles sur internet. Cependant, parfois les formats utilisés dans les bases sont très hétérogènes. Il faut donc examiner au cas par cas chaque set et éventuellement homogénéiser l'ensemble des données, car, rappelons-le, les inputs d'un réseau doivent toujours être de même taille. Voici une présentation des différents problèmes pouvant être rencontrés :

- Résolutions hétérogènes des données :
 - Images de résolutions différentes pouvant être en format portrait ou paysage.
 - Données texte représentant des phrases contenant un nombre variable de mot.
- Absence de données :
 - Lorsque l'on reçoit des infos depuis des capteurs distants (sismographes par exemple), parfois un capteur peut ne plus émettre d'informations pendant une heure. Comment traiter ces cas ? Faut-il faire des blocs d'1 heure et rejeter les blocs

contenant des pannes ? Faut-il remplir de zéro les zones sans informations et espérer que le réseau sache traiter ces imperfections ?

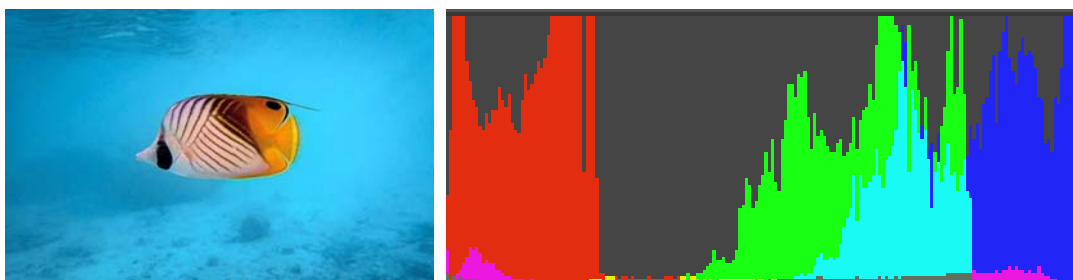
- Avec un flux vidéo qui transite sur internet, mais c'est aussi vrai pour une webcam, le framerate n'est pas fixe, il oscille entre 10 et 20 images par seconde par exemple. Par moment, il peut aussi chuter à 3 images par seconde. Le flux de données est donc irrégulier.
- Format des données :
 - Pixels d'une image stockés en x/y ou y/x
 - Codage des couleurs en RVB, BVR ou YCbCr
 - Images/Audio en 16 bits ou 8 bits
 - Enregistrement de sons en 44kHz ou 22kHz
- Echantillons triés par catégories (d'abord les tigres ensuite les avions...) ou échantillons classés aléatoirement ? Il serait préférable lors de l'apprentissage de présenter au réseau des images de différentes catégories.

L'intérêt des datasets fournis dans les bibliothèques d'apprentissage comme PyTorch est que les données sont déjà uniformisées. Cela permet d'économiser une semaine de travail ou plus. A titre d'exemple, dans la base d'images CIFAR10, quelle que soit l'orientation (portrait, paysage) et la résolution des images d'origine, les images ont toutes été redimensionnées vers une résolution de 32x32 pixels.

Remarque : une taille de 32x32 peut choquer, ne serait-ce que par la petitesse de la résolution. Mais plus la taille des images en entrée du réseau est grande, plus de couches de traitement seront nécessaires ce qui allongera forcément la durée d'apprentissage sans pour autant améliorer la performance.

8.2.2 Normalisation des entrées

Si nous prenons l'exemple des images RGB, trois plans de couleur sont disponibles. Si nous traitons des images de la faune marine par exemple, on obtient un histogramme comme ci-dessous :



Dans l'histogramme, les valeurs faibles sont comptabilisées sur la gauche et les niveaux hauts sur la droite. Ainsi, dans cette image les rouges sont très présents dans la zone des 0 à 20% et ils sont donc peu présents dans l'image. Les bleus sont évidemment présents dans les hautes valeurs entre 70% et 100%. La présence de vert dans les hautes valeurs peut surprendre, mais il faut remarquer que l'eau est plutôt d'un bleu céleste (38, 196, 236) ce qui correspond à l'histogramme que l'on retrouve ici.

Alors quel est le problème docteur ? Que les valeurs des rouges soient basses et que celles des bleues soient hautes peut être corrigé par le réseau, il suffit pour cela que les poids associés aux valeurs du rouge soient plus importants afin de rivaliser avec les valeurs des bleus déjà hautes. Effectivement, cela est possible, mais au démarrage de la phase d'apprentissage, les poids des

couches sont initialisés aléatoirement dans l'intervalle [0,1]. Ainsi, pour que les poids associés au rouge deviennent, en moyenne, 4 à 5 fois plus importants pour compenser le faible niveau du rouge, cela va nécessiter du temps et ce temps s'ajoute au temps global d'apprentissage.

Ainsi, il est d'usage de normaliser les plans R,V,B de toutes les images pour obtenir une distribution statistique ayant une moyenne de 0 et une variance de 1. Attention, cette normalisation s'effectue pour chaque canal en tenant compte de la totalité des images. Attention, la normalisation ne s'effectue pas pour chaque image. Ainsi, la normalisation du set d'images équivaut au code suivant :

```
# IMG : tenseur des images (60 000, 3, 32, 32) -> 60 000 images 32x32 en RVB
for c in range(3) :                # pour chaque canal R/V/B
    mean = IMG[:,c,:,:].mean()     # calcul de la moyenne
    std = IMG[:,c,:,:].std()        # calcul de la déviation standard
    IMG[:,c,:,:] = (IMG[:,c,:,:] - mean) / std    # normalisation
```

Cette normalisation doit être effectuée pour tous les datasets y compris les datasets fournies par PyTorch. Cependant, comme les bases sont de plus en plus grandes (ImageNet contient plusieurs millions d'images), l'évaluation de 3 moyennes et de 3 déviations standards peut prendre un certain temps. Ainsi, ces valeurs sont souvent écrites directement dans le code de chargement de la base à travers un exemple.

8.2.3 Construire une liste de transformations

Nous utilisons `transforms.Compose(...)` pour construire une liste de traitements à appliquer sur les données en entrée. Ensuite, le paramètre *transform* présent dans les constructeurs des datasets de PyTorch permet de passer cette série de transformations :

```
from torchvision import transforms
moy, dev = 0.1307, 0.3081
TRS = transforms.Compose([transforms.ToTensor(), transforms.Normalize(moy,dev)])
TrainSet = datasets.MNIST('./data', train=True, download=True, transform=TRS)
```

Les opérations de transformation sont appliquées dans l'ordre de la liste. La première opération consiste à convertir le tenseur des images utilisant des uint8 vers un tenseur en Float32. De plus, PyTorch effectue un scaling pour ramener les valeurs dans la plage [0,1]. Voici un extrait de la documentation en ligne ci-dessous :

CLASS `torchvision.transforms.ToTensor` [SOURCE]

Convert a `PIL Image` or `numpy.ndarray` to tensor. This transform does not support torchscript.

Converts a `PIL Image` or `numpy.ndarray` (H x W x C) in the range [0, 255] to a `torch.FloatTensor` of shape (C x H x W) in the range [0.0, 1.0] if the `PIL Image` belongs to one of the modes (L, LA, P, I, F, RGB, YCbCr, RGBA, CMYK, 1) or if the `numpy.ndarray` has `dtype = np.uint8`

In the other cases, tensors are returned without scaling.

• NOTE

Because the input image is scaled to [0.0, 1.0], this transformation should not be used when transforming target image masks. See the [references](#) for implementing the transforms for image masks.

La deuxième opération consiste à normaliser la distribution statistique des données. Pour les images en niveau de gris, nous donnons une seule valeur moyenne et une seule déviation standard. Voici un extrait de la documentation en ligne ci-dessous :

CLASS `torchvision.transforms.Normalize(mean, std, inplace=False)` [SOURCE]

Normalize a tensor image with mean and standard deviation. This transform does not support `PIL Image`. Given mean: `(mean[1], ..., mean[n])` and std: `(std[1], ..., std[n])` for `n` channels, this transform will normalize each channel of the input `torch.*Tensor` i.e., `output[channel] = (input[channel] - mean[channel]) / std[channel]`

• NOTE

This transform acts out of place, i.e., it does not mutate the input tensor.

Parameters

- **mean** (*sequence*) – Sequence of means for each channel.
- **std** (*sequence*) – Sequence of standard deviations for each channel.

Dans la documentation PyTorch, vous pouvez trouver l'ensemble des transformations disponibles :

<https://pytorch.org/vision/stable/transforms.html>

Remarque : cette série de transformation doit être effectuée en double pour construire le set d'images de validation.

8.3 La gestion du flux de traitement

8.3.1 Notions de batch, d'époque et d'itérations

Le calcul de l'erreur totale sur l'ensemble des images peut être une opération très longue lorsque la base contient plusieurs centaines de milliers d'images. L'usage a montré qu'il était suffisant et plus efficace pour l'apprentissage de prendre un nombre limité d'images et de calculer un gradient

approché à chaque itération du pas du gradient. Ce lot d'images contenant un nombre constant d'images s'appelle en anglais un **batch**. Le set d'images est ainsi découpé en lots. Lorsque on a parcouru une fois la totalité des images, on dit que l'on a traité une **epoch**. Le nombre d'itérations utilisées durant une epoch est donné par la formule suivante :

$$\text{Nombre d'itérations dans une epoch} = \frac{\text{Nombres d'images dans la base}}{\text{Nombre d'images dans le batch}}$$

Une epoch est toujours constituée du même nombre d'itérations. Si nous avons un set de 50 000 images et une taille de lot de 100 images, il faudra donc 500 itérations pour compléter une epoch. La taille idéale du lot d'images n'est pas connue d'avance. Elle doit être déterminée empiriquement par essais successifs.

8.3.2 Traitement par lots

Une fois les données chargées, PyTorch dispose d'un outil permettant de construire un itérateur sur les lots d'images, il s'agit du DataLoader qui reçoit en paramètres la base à traiter ainsi que la taille des lots. L'itérateur met ainsi à disposition un lot d'images et les catégories associées à travers l'utilisation d'une boucle for :

```
loader = torch.utils.data.DataLoader(TrainSet, batch_size)
for (data, target) in loader:
    ...
```

8.3.3 Dans la pratique

Si nous examinons l'objet retourné par le dataset MNIST(...), on trouve dans ses attributs : la liste des transformations choisies et les données originales du dataset :

```
TrainSet = datasets.MNIST('../data', train=True, download=True, transform=TRS)
print(TrainSet)
>> Dataset MNIST
      Number of datapoints: 60000
      Root location: ../data
      Split: Train
      StandardTransform
Transform:
  Compose(
    ToTensor()
    Normalize(mean=0, std=0.5)
  )
Print(TrainSet.dataset.data[0])
>> tensor([[ 0,  0,  0,  0... , dtype=torch.uint8])
```

Les données des images sont stockées sous la forme d'uint8. Il semble que PyTorch n'est pas encore appliqué les transformations pour construire un tenseur contenant les images étalonnées en Float32. Ce choix technique peut se comprendre car les images stockées en 8 bits tiennent 4 fois moins de

place qu'en Float32. Ainsi, ce sera lors du traitement d'un nouveau lot, que les images retenues vont être converties. Cela permet d'économiser la mémoire des GPU (entre 10 et 20 Go), mémoire servant à la fois pour stocker les poids du réseau et les images de la base.

En examinant le tenseur associé au lot courant, nous obtenons les informations suivantes :

```
loader = torch.utils.data.DataLoader(TrainSet, batch_size = 100)
for (data, target) in loader:
    print(data.shape)
    print(data.dtype)
    print(data.mean())

>> torch.Size([100, 1, 28, 28])
>> torch.float32
>> tensor(-0.0242)
```

La taille du tenseur courant correspond à un lot de 100 images de résolution 28x28 pixels en niveaux de gris. En demandant la moyenne des images du lot courant avec la ligne `data.mean()`, nous constatons que cette valeur n'est pas nulle alors que la base d'images a été normalisée. Pourquoi ? Nous rappelons que la normalisation a lieu sur l'ensemble des images pour que la valeur moyenne des images soient nulles. Or, dans le lot courant, la valeur moyenne est calculée sur les 100 images présentes dans ce lot et non la totalité, ce qui explique que la moyenne affichée ne soit pas égale à 0.


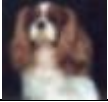

8.4 La classification d'images

8.4.1 Fonction d'erreur

Le problème de classification d'images consiste à associer une image à une catégorie parmi une liste de catégories connues. Parmi les bases les plus classiques, on trouve MNIST avec 10 catégories correspondant aux chiffres de 0 à 9, on trouve aussi la base CIFAR10 avec 10 catégories d'animaux : avion, voiture, oiseau, chat, daim, chien, grenouille, cheval, navire et camion. Un réseau de neurones devant classer ces images construit 10 valeurs notées S_0 à S_9 correspondant à un score associé à chaque catégorie, le score le plus haut indiquant la catégorie prédite. La base donne pour chaque image le numéro, noté k , de sa catégorie réelle. Ensuite, on peut utiliser la fonction d'erreur vue précédemment :

$$Erreur = \left[\sum_{i=0}^9 \max(0, (\Delta + S_i) - S_k) \right] - \Delta$$

La soustraction par un terme Δ permet de corriger la valeur obtenue avec $(\Delta + S_i) - S_k$ lorsque $i=k$. Ainsi, lorsque le score de la bonne catégorie dépasse de Δ les autres scores, la prédiction est considérée comme parfaite et l'erreur associée est nulle. Voici un exemple avec 3 images et 3 catégories. Le réseau en traitant ce lot de trois images génère un tenseur de scores 3x3. Dans le tableau ci-dessous, les scores associés aux bonnes catégories ont été soulignés. On peut ainsi calculer l'erreur introduite par chaque image en prenant $\Delta = 1$ et il nous suffit de sommer chaque erreur pour obtenir l'erreur totale.

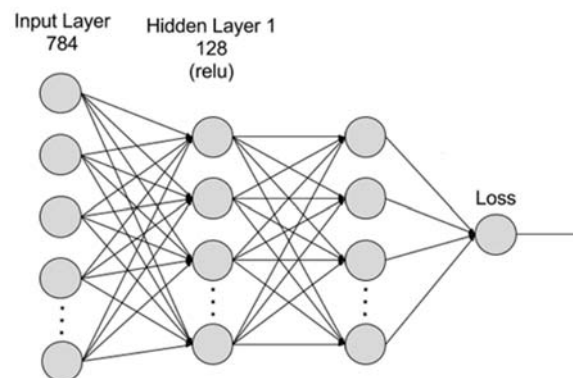
	Chien	Oiseau	Cheval		Erreur ($\Delta = 1$)	ErrTot
	5	4	<u>2</u>	→	$4+3+1-1 = 7$	15
	<u>4</u>	2	8	→	$1+0+5-1 = 5$	
	4	<u>5</u>	1	→	$0+1+0-1=0$	

8.5 Démarrage de l'exercice

Ouvrez le fichier « Classification.py ». Les réponses aux questions se feront dans le code source, en début du document. Examinez la structure du code et répondez aux questions suivantes :

- Qu1 : quel est le % de bonnes prédictions obtenu au lancement du programme, pourquoi ?
- Qu2 : quel est le % de bonnes prédictions obtenu avec 1 couche Linear ?
- Qu3 : pourquoi le test_loader n'est pas découpé en batch ?
- Qu4 : pourquoi la couche Linear comporte-t-elle 784 entrées ?
- Qu5 : pourquoi la couche Linear comporte-t-elle 10 sorties ?

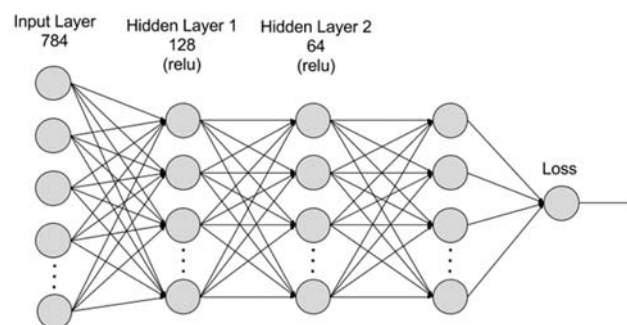
On se propose de faire évoluer le réseau actuel vers la topologie suivante :



Modifiez la classe Net pour que le réseau ait maintenant 2 couches Linear avec 128 neurones sur la première couche. La fonction d'activation sera ReLU.

- Question 6 : quelles sont les tailles des deux couches Linear ?
- Question 7 : quel est l'ordre de grandeur du nombre de poids utilisés dans ce réseau ?
- Question 8 : quel est le % de bonnes prédictions obtenu avec 2 couches Linear ?

Peut-on encore améliorer les performances en ajoutant une couche Linear supplémentaire ? La couche contenant le plus de poids est la couche gérant les entrées. Les couches proches de la sortie nécessitent beaucoup moins de poids, essayons d'ajouter une 3ème couche Linear.



Modifiez la classe Net pour que le réseau ait maintenant 3 couches Linear avec 128 neurones sur la première couche et 64 neurones sur la seconde. La fonction d'activation sera ReLU.

- Question 9 : obtient-on un réel gain sur la qualité des prédictions ?

9 Probabilités et entropie

9.1 La fonction Softmax

Nous allons maintenant utiliser une nouvelle fonction très célèbre dans le monde de l'apprentissage : la fonction Softmax. Elle permet à partir des valeurs de sortie d'un réseau fournissant 1 score par catégorie à prédire, de construire une probabilité d'appartenance pour chaque catégorie. Ainsi, la fonction Softmax transforme un tenseur de k valeurs réelles en un tenseur de k probabilités. La formule utilisée pour obtenir la i -ème valeur en sortie de la fonction Softmax pour un tenseur d'entrée $X=(x_j)_{0 \leq j < n}$ est donnée par :

$$\text{Softmax}(i, X) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Au niveau des calculs effectués, on peut découper cette formule en trois étapes comme dans le tableau ci-dessous :

	<i>Input</i>	<i>exp(x_i)</i>	<i>Total</i>	<i>exp(x_i)/Total</i>
x_0	-1	0,368	23,172	1,59%
x_1	1	2,718		11,73%
x_2	3	20,086		86,68%

- En premier lieu, chaque valeur présente dans le vecteur d'entrée est injectée dans une fonction exponentielle pour obtenir une série de valeurs intermédiaires.
- En second lieu, l'ensemble des valeurs intermédiaires sont sommées pour obtenir un total.
- Finalement, la série des valeurs intermédiaires est normalisée en divisant chaque valeur intermédiaire par le total. On obtient une série de probabilités.

On peut remarquer que les valeurs en sortie de la fonction Softmax sont positives et que leur somme est égale à 1. Ainsi la fonction Softmax à partir d'un tenseur d'entrée quelconque construit un tenseur de sortie correspondant à une distribution de probabilités.

La fonction Softmax présente certains avantages :

- Qu'elle que soit le signe des valeurs en entrée, l'exponentiation va produire des valeurs positives. On n'a donc pas de contraintes à imposer sur le signe des valeurs produites par les couches précédentes du réseau.
- Qu'elle que soit l'ordre de grandeur des valeurs d'entrée, la fonction Softmax construit une distribution de probabilités. Ainsi, que les valeurs d'entrée aient un ordre de grandeur autour de 1/10 ou de 1000 000, la fonction Softmax produit le résultat escompté.

Voici ce qu'indique la documentation PyTorch pour la fonction Softmax :

TORCH.NN.FUNCTIONAL.SOFTMAX

```
torch.nn.functional.softmax(input, dim=None, _stacklevel=3, dtype=None) [SOURCE]
```

Applies a softmax function.

Softmax is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

It is applied to all slices along dim, and will re-scale them so that the elements lie in the range $[0, 1]$ and sum to 1.

La fonction Softmax associe un ensemble de valeurs des probabilités. La fonction exponentielle étant croissante, cela sous-entend que plus une valeur d'entrée est grande plus sa probabilité augmente. Ainsi, lorsqu'une valeur d'entrée augmente par rapport aux autres, alors, sa probabilité augmente et par rééquilibrage les autres probabilités se voient diminuer. Prenons un exemple sur un cas simple :

Input	exp(...)	Sigma	prob
-1	0,368	23,172	1,59%
1	2,718	23,172	11,73%
3	20,086	23,172	86,68%

Si la deuxième valeur en entrée passe à 2, nous obtenons une nouvelle série de probabilités :

input	exp(...)	Sigma	prob
-1	0,368	27,842	1,32%
2	7,389	27,842	26,54%
3	20,086	27,842	72,14%

En passant de 1 à 2, la probabilité augmente de 11,75% à 26,54%. Les deux autres probabilités dont les valeurs d'entrée sont inchangées diminuent, il y a eu une forme de rééquilibrage.

Lorsque l'on utilise la fonction Softmax, on peut voir les valeurs d'entrée comme des intensités. Plus l'intensité est importante, plus la probabilité associée est importante. Ces valeurs n'ont pas d'unités, pas de plage de valeurs limitée, elles peuvent donc prendre une valeur quelconque mais dans tous les cas la fonction Softmax s'adapte et construit intelligemment une liste de probabilités cohérentes.

Propriété : les probabilités issues de la fonction Softmax sont invariantes par ajout d'une constante.

Cela propriété vient de la définition de la fonction Softmax :

Posons : $x'_i = x_i + c$ pour tout i , nous avons :

$$\text{Softmax}(i, X') = \frac{\exp(x_i + c)}{\sum_j \exp(x_j + c)} = \frac{\exp(x_i) \cdot \exp(c)}{\sum_j \exp(x_j) \cdot \exp(c)} = \text{Softmax}(i, X)$$

Pour tester cette propriété, décalons de 10 les valeurs de l'exemple précédent. Nous remarquons alors que les probabilités obtenues restent inchangées :

input	exp(...)	Sigma	prob
9	8103,084	510390,618	1,59%
11	59874,142	510390,618	11,73%
13	442413,392	510390,618	86,68%

Ou encore :

input	exp(...)	Sigma	prob
-11	1,6702E-05	1,0520E-03	1,59%
-9	1,2341E-04	1,0520E-03	11,73%
-7	9,1188E-04	1,0520E-03	86,68%

Peut-être pensez qu'une formule plus simple comme : $p(x_i) = x_i / \sum x_j$ auraient suffi, la somme des valeurs construites valant bien 1. Comparons cependant ces deux approches :

- Avec cette formule simplifiée, les valeurs d'entrées doivent être positives sinon on pourrait obtenir des probabilités négatives ce qui est impossible. Cette contrainte est difficilement satisfaisable à la sortie d'un réseau. En comparaison, la fonction Softmax s'adapte à tout type de valeurs, positives comme négatives et produit toujours des résultats positifs.
- Avec la formule simplifiée, il faut être sûr que les valeurs d'entrée soient non nulles sinon on obtient une division par zéro. Avec des entrées nulles, la fonction Softmax construit des probabilités équiprobables car les entrées sont équivalentes.
- Avec la formule simplifiée, les valeurs (1,1,2) donnent les probabilités en sortie de 25%, 25% et 50% alors qu'avec (10,10,11) les résultats sont proches de l'équiprobabilité. L'ordre de grandeur des valeurs d'entrée influence donc beaucoup le comportement de la fonction. Ainsi, la formule simplifiée produit des probabilités très différentes lorsque les valeurs se décalent. Il n'en ait rien pour la fonction Softmax qui corrige ce défaut. Ainsi pour la fonction Softmax seul compte l'écart entre la plus grande valeur d'entrée et les autres.

Si la formule simplifiée semble plus facile à mettre en œuvre au premier abord, son utilisation s'avère bien plus contraignante.

9.2 La notion d'entropie

Nous introduisons une notion importante pour la suite : l'entropie. L'entropie est une valeur caractérisant le niveau de désorganisation ou d'imprédictibilité d'un système. Pour une distribution de probabilité discrète $X(\Omega) = (x_i)_{0 \leq i < n}$ associée aux probabilités $(p_i)_{0 \leq i < n}$, l'entropie est définie ainsi :

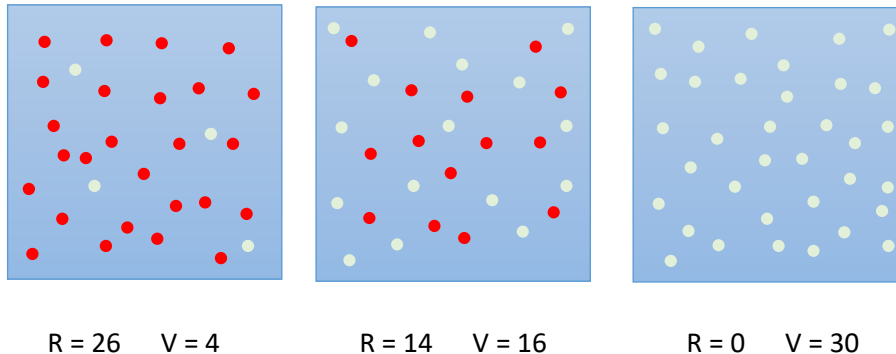
$$H(X) = - \sum_{p_i > 0} p_i \cdot \log_2(p_i)$$

Nous avons : $H(X) \geq 0$ car les probabilités étant des valeurs dans $[0,1]$, les termes en log sont tous négatifs, ainsi le signe – avant le grand sigma donne un résultat positif.

L'entropie $H(X)$ n'est pas une valeur bornée. En effet, si nous avons n valeurs équidistribuées, nous obtenons :

$$H(X) = - \sum_n \frac{1}{n} \cdot \log_2 \left(\frac{1}{n} \right) = \log_2(n)$$

Examinons un cas pratique de trois boîtes contenant des billes rouges et des billes vertes :



Dans la boîte de gauche, la probabilité de tirer une boule rouge est de 26/30 et celle de tirer une bille verte est de 4/30. Ainsi, la probabilité de tirer une boule rouge est plus certaine.

Dans la boîte au centre, la probabilité de tirer une boule rouge est de 14/30 et celle de tirer une bille verte de 16/30. Il y a presque une chance sur deux de tirer une boule rouge et une boule verte. Il devient difficile d'être sûr de la couleur tirée.

Dans la boîte de droite, nous sommes certains que la boule tirée sera de couleur verte. La probabilité de tirer une boule verte est de 1 contre 0 pour une boule rouge.

Calculons les valeurs de l'entropie pour chacune des situations :

- Boîte de gauche : $-26/30 \cdot \log_2(26/30) - 4/30 \cdot \log_2(4/30) = 0.3926$
- Boîte au centre : $-14/30 \cdot \log_2(14/30) - 16/30 \cdot \log_2(16/30) = 0.6909 < \log(2)=0.6931$
- Boîte de droite : $-30/30 \cdot \log_2(30/30) = 0$

L'entropie de la boîte de gauche est inférieure à la boîte du centre car en tirant une boule dans cette boîte nous sommes presque sûr qu'elle sera rouge. Dans la boîte au centre, nous ne sommes sûr de rien, elle peut être rouge comme verte et l'entropie calculée est proche de l'entropie maximale possible : $\log(2)$. La boîte de droite correspond à une situation idéale où nous sommes absolument certains que toute boule tirée sera verte, en conséquence l'entropie est nulle.

9.3 Exprimer l'erreur grâce à l'entropie

Nous allons utiliser la définition de l'entropie croisée pour construire une fonction d'erreur. Dans la littérature anglo-saxonne on trouve plusieurs terminologies : **Cross-Entropy Loss**, **Logarithmic Loss** ou **Logistic Loss**. L'entropie croisée pour deux distributions discrètes p et q est définie de la façon suivante :

$$CrossEntropyLoss(p, q) = - \sum_{p_i > 0} p_i \cdot \log_2(q_i)$$

Quand on compare une distribution de probabilités q avec une distribution de probabilités p , l'entropie croisée atteint son minimum lorsque les deux distributions sont égales i.e. $p_i = q_i$. Ce minimum correspond à la valeur $H(p) = H(q)$. Supposons qu'une distribution soit connue et que nous cherchions à faire en sorte qu'une deuxième distribution apprenne cette distribution de « référence » afin d'être la plus proche possible. Alors nous pouvons formuler ce problème comme un problème de minimisation de l'entropie croisée.

Dans le problème de la classification, si nous avons trois catégories, lorsque nous savons que la réponse doit correspondre à la première catégorie, cela équivaut à dire que nous aimerions que les probabilités obtenues par la fonction Softmax correspondent à la distribution (1,0,0). Ainsi, si l'on nomme $icat$ le numéro de la catégorie de l'échantillon courant, l'expression de l'entropie croisée précédente se réécrit de manière équivalente en :

$$CrossEntropyLoss(p, icat) = -\log_2(p_{icat})$$

De l'expression précédente, il ne reste qu'un seul terme correspondant à la probabilité non nulle. Ainsi en minimisant la fonction CrossEntropyLoss, on force les poids du réseau à s'optimiser pour que les probabilités en sortie de la fonction Softmax correspondent à une probabilité de 1 pour les catégories désirées.

Voici les informations données dans la documentation PyTorch :

CROSSENTROPYLOSS

```
CLASS torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=- 100,
    reduce=None, reduction='mean', label_smoothing=0.0) [SOURCE]
```

This criterion computes the cross entropy loss between input and target.

9.4 Instabilité numérique

Cette section est réservée aux curieux et elle peut être sautée pour ceux qui sont en retard.

A la question : pourquoi avons-nous choisi telle ou telle fonction / algo / paramètre ? Nous répondons habituellement par une de ces trois réponses :

- On utilise moins d'itérations : couche dropout, normalisation des inputs...
- On effectue autant d'itérations, mais les calculs sont plus économiques et consomment moins de ressources CPU/GPU : ReLU (plutôt que tanh)...
- Les prédictions sur l'ensemble de validation sont meilleures : couche Conv

Nous allons introduire un nouveau couple de fonctions : LogSoftmax et NegativeLogLikelihood Loss qui vont remplacer le couple précédent : Softmax et Logarithmic Loss. Cette modification ne s'est pas imposée cette fois par un gain de performance mais pour corriger des instabilités numériques. Mais où se situent les risques lorsque l'on évalue la fonction Softmax et Logarithmic Loss pourtant assez robustes en apparence ? Pour rappel, voici la définition de la fonction Softmax :

$$p_i = \text{Softmax}(i, X) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Le premier problème provient des exponentielles présentes dans la fonction. Car avec une exponentielle, les grandeurs associées peuvent grimper vite, très vite. Or, les calculs dans le réseau sont faits en Float32 pouvant au maximum représenter la valeur 3.402823E+38. Ainsi, il suffit qu'en entrée de la fonction Softmax une valeur soit supérieure à $\ln(3.402823E+38) \sim 88.72$ pour que le résultat de l'exponentielle soit égal à $+\infty$ et c'est l'accident. La suite des calculs sera forcément erronée. Voici un exemple :

x_i	$\exp(x_i)$	Float 32
80	5,540622E+34	5,540622E+34
89	4,489613E+38	$+\infty$
90	1,220403E+39	$+\infty$

Pour éviter le problème de saturation à l'infini, nous allons soustraire à l'ensemble des valeurs x_i le maximum des x_i noté c pour obtenir une série de nouvelles valeurs notées x'_i . Nous avons précédemment vu dans le chapitre sur la fonction Softmax que cela ne changerait pas sa valeur finale. Ainsi, nous évitons avec cette astuce toute forme de saturation :

x_i	c	$x'_i = x_i - c$	$\exp(x_i)$	$\sum \exp(x_i)$	Prob
80	90	-10	4,54E-05	1,367925	0,003%
89		-1	0,367879	1,367925	26,893%
90		0	1	1,367925	73,103%

Une fois cette première astuce utilisée, nous allons rencontrer un autre problème pour les exposants avec une valeur négative. En effet, avec des nombres en Float32, la valeur minimale que l'on puisse représenter est -3.402823466 E+38. Ainsi, lorsqu'il existe une valeur x_i telle que $x_i < c-82$, nous allons obtenir un résultat nul :

x_i	c	$x'_i = x_i - c$	$\exp(x_i)$	$\sum \exp(x_i)$	Prob
2	90	-88	0	1,367879	0,000%
89		-1	0,367879	1,367879	26,894%
90		0	1	1,367879	73,106%

En soit, ce résultat n'est pas incohérent car la probabilité est quasi nulle. Cependant, nous injectons ce résultat dans la formule de la CrossEntropyLoss avec une fonction logarithmique et l'évaluation de $\log(0)$ produira une erreur.

Nous allons donc utiliser une deuxième astuce pour gérer ce cas. Pour cela, nous allons construire une formule effectuant le calcul du Softmax puis du Log dans la même expression :

$$\text{LogSoftmax}(i, X') = \log \left(\frac{\exp(x'_i)}{\sum_j \exp(x'_j)} \right)$$

En développant cette expression nous obtenons :

$$\text{LogSoftmax}(i, X') = x'_i - \log \left(\sum_j \exp(x'_j) \right)$$

Par définition des x'_j , une de ces valeurs est nulle et comme l'exponentielle est toujours positive et que $\exp(0)=1$, le résultat du grand sigma est supérieur ou égal à 1. Ainsi le logarithmique ne reçoit jamais une valeur nulle en paramètre et son évaluation n'est plus problématique.

On peut maintenant définir la NegativeLogLikelihood Loss qui correspond à la fonction Logarithmic Loss privée de son logarithme :

$$\text{Negative Log Likelihood Loss}(X, \text{icat}) = -x_{\text{icat}}$$

9.5 Suite de l'exercice

9.5.1 Les modules PyTorch

Sous PyTorch, les fonctions sont parfois disponibles à deux endroits différents :

- Dans `torch.nn.functional` : dans ce cas, il s'agit d'une fonction.
- Dans `torch.nn` : dans ce cas, il s'agit d'un objet foncteur, qu'il faut d'abord instancier et ensuite utiliser comme une fonction. L'objet foncteur permet de stocker des informations supplémentaires et de préserver des données dans le temps : états, poids ; ce qui est impossible avec une fonction.

Dans le cas où vous avez le choix, préférez la version présente dans `torch.nn.functional` pour éviter tout comportement inattendu. D'autres fois, les deux versions disponibles seront légèrement différentes : une version aura plus d'options ou sera plus souple... le choix se portera alors sur l'option la plus pratique... En résumé, rien d'évident sur ce point.

9.5.2 Implémentation

Nous allons reprendre le code de l'exercice de classification et implanter les deux nouvelles fonctions : `Softmax` et `CrossEntropy`. Pour cela, vous utiliserez :

- La fonction **Softmax** disponible dans `torch.nn.functional`
- L'objet **CrossEntropy** de `torch.nn`, (examinez l'exemple dans la documentation). Cette fonction est aussi disponible dans `torch.nn.functional` mais son usage est plus contraignant.

Si vous avez lu la section sur les instabilités numériques, vous pouvez utiliser :

- La fonction **log_softmax** disponible dans `torch.nn.functional`
- La fonction **nll_loss** disponible dans `torch.nn.functional`

Répondez à la question suivante :

- Question 10 : pourquoi ne change-t-on pas le code de la fonction `TestOK()` servant à déterminer le % de prédictions correctes sur le `TestSet` suite à cette modification ?

10 Les réseaux convolutionnels

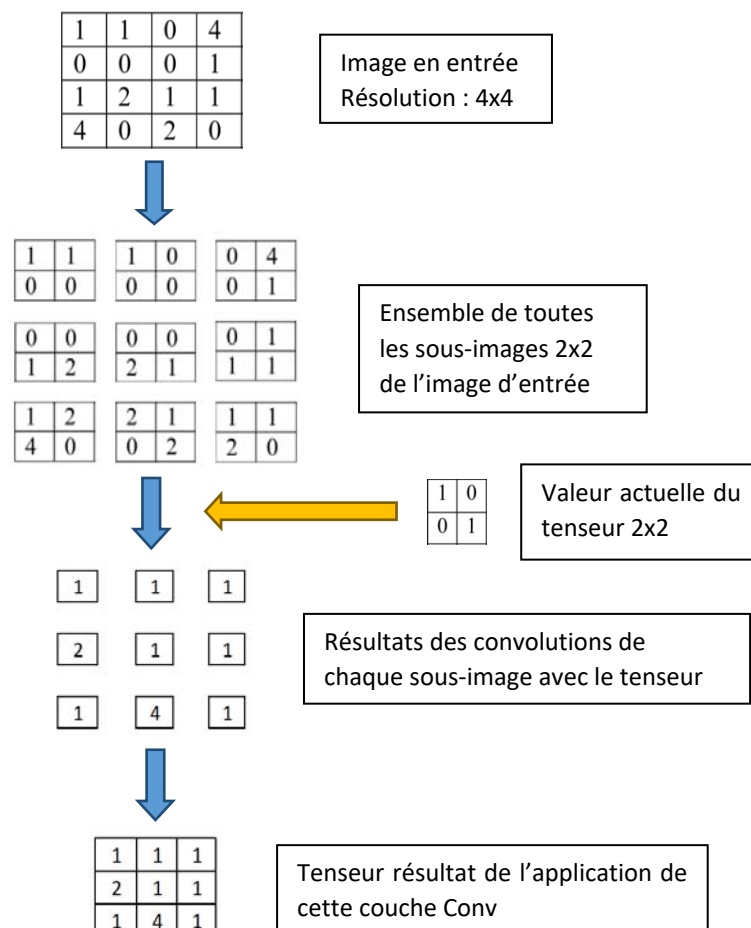
10.1 Les couches Conv et Dropout

10.1.1 La convolution 2D

Pour calculer la convolution entre un tenseur 2D de taille (u,v) et une image représentée par un tenseur de taille (n,m) , on parcourt chaque pixel de l'image et lorsqu'on peut extraire une sous-image de taille (u,v) depuis le pixel courant, on effectue la multiplication membre à membre entre cette portion d'image et le tenseur de convolution. L'ensemble des résultats obtenus forme un tenseur de taille $(n-u+1, m-v+1)$. Voici le code de principe correspondant :

```
def Conv2D(Tconv, ImageTensor) :
    sizeT      = Tconv.shape
    sizeImage  = ImageTensor.shape
    # tenseur résultat :
    R = torch.FloatTensor(size = (sizeImage[0]-sizeT[0]+1, sizeImage[1]-sizeT[1]+1))
    for x in range(0, R.shape[0]):
        for y in range(0, R.shape[1]):
            exImage = ExtractImage(ImageTensor,x,y,sizeT)
            R[x,y]  = torch.mul(Tconv, exImage)    # multiplication membre à membre
```

Voici un cas pratique utilisant une image 4x4 et un tenseur 2x2 :



Dans une image, les zones de transition sont souvent « douces ». Il n'est pas toujours nécessaire de calculer la convolution en chaque point de l'image, mais seulement tous les k pixels. Cela permet de plus de réduire la taille du tenseur de sortie d'un facteur k^2 . Le **pas** ou **stride** désigne ainsi le « saut » entre chaque convolution.

Les bords peuvent aussi poser problème. En effet, la taille du tenseur de sortie diminue légèrement en fonction de la taille du tenseur de convolution. Pour éviter cela, on peut élargir virtuellement l'image d'entrée avec des 0. Ce paramètre de **marge** ou **zero padding** permet ainsi de régler finement la taille du tenseur de sortie.

10.1.2 La convolution 2D multicanaux

Nous avons présenté la convolution dans le cas d'une image en niveaux de gris. Pour une image RVB, trois canaux sont présents. Notons $*$ l'opération de convolution 2D. Si on utilisait le même tenseur de convolution pour les trois canaux couleur, on obtiendrait :

$$T_{out} = \sum_{k=0}^2 Tconv * Image[k] = Tconv * \sum_{k=0}^2 Image[k]$$

Cette opération reviendrait à d'abord sommer les canaux couleurs $\sum_{k=0}^2 Image[k]$, ce qui équivaut à convertir l'image RVB en niveaux de gris, pour ensuite appliquer une convolution. Ainsi, l'information couleur est perdue, il faut donc choisir une autre approche.

Ainsi, lorsque l'on traite une image RVB avec trois canaux, on va utiliser 3 tenseurs pour la convolution ; chacun étant dédié à une couche couleur. Les trois tenseurs doivent être exactement de même dimension. La mise en place de cette technique se traduit par la formule :

$$T_{out} = \sum_{k=0}^2 Tconv[k] * Image[k]$$

La notation $Image[k]$ désigne la k -ième couche du tenseur image de taille (k, L, H) . Les k tenseurs de taille (u, v) utilisés pour la convolution par canal sont simplement regroupés dans un même tenseur de taille (k, u, v) ; l'écriture $Tconv[k]$ désigne ainsi le k -ième tenseur utilisé pour la k -ième couche.

10.1.3 La couche Conv2D

Nous présentons maintenant les couches convolutionnelles, en abrégé Conv, ayant permis aux réseaux de neurones d'atteindre un tout autre niveau de performance. Une couche Conv gère un groupe de **caractéristiques (feature en anglais, ou encore kernel)** correspondant à des tenseurs 2D de même taille. Le nombre de features s'appelle **la profondeur de la couche Conv**. La sortie d'une couche Conv correspond à un tenseur contenant les différentes convolution 2D entre chaque feature et l'image d'entrée. A noter que l'on parle de **CNN, Convolutional Neural Network**, lorsqu'au moins une couche Conv est présente dans le réseau. Un réseau à base de couches Linear ne permet pas d'obtenir un CNN !

La couche Conv participe aux calculs Forward du réseau et son résultat contribue à l'erreur finale. Ainsi les valeurs contenues dans ses features se comportent comme des poids qui vont ainsi être optimisés par la méthode du gradient. Le pas, la marge et la profondeur de la couche CONV sont des hyperparamètres d'apprentissage du réseau : ils sont choisis par l'utilisateur en début d'apprentissage et n'évoluent pas durant l'apprentissage.

Il reste à gérer le nombre de channels en entrée et en sortie. Examinons les informations données par la documentation en ligne à l'adresse :

<https://pytorch.org/docs/stable/nn.html#convolution-layers>

CONV2D

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
                      dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None) [SOURCE]
```

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{out}, H_{out}, W_{out})$ can be precisely described as:

$$\text{out}(N_i, C_{out_j}) = \text{bias}(C_{out_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) \star \text{input}(N_i, k)$$

La formule n'est pas évidente à saisir à la première lecture. Si l'on note la convolution multicanale par le symbole \otimes , on peut la réécrire en utilisant le pseudo-code suivant :

```
Images   : tenseur [N,Cin,L,H]           # n images LxH avec Cin canaux
Features : tenseur [Cout,Cin,u,v]         # Cout features de taille (Cin,u,v)
Out       : tenseur [N,Cout,L-u+1,H-v+1]  # résultats
for i in range(N) :                       # boucle sur les images
    for j in range(Cout) :                 # boucle sur les features
        Out[i,j,:,:] = b[j] + Features[j]  $\otimes$  images[i]
```

A retenir :

- L'utilisation de n features produit un résultat ayant n canaux
- Si k canaux sont présents en entrée, les features (u,v) sont en fait de taille (k,u,v)

Voici un exemple :

```
RVB = torch.FloatTensor( size = ( 1000,3,28,28) )
C1 = torch.nn.Conv2d(in_channels=3,out_channels=32,(5,5))
R = C1(RVB)
print(R.shape)
print(C1.weight.shape)

>> torch.Size([1000, 32, 24, 24])
>> torch.Size([32, 3, 5, 5])
```

Dans cet exemple, la couche Conv2D gère 32 features de taille 5x5, cette couche prend en entrée un tenseur de 1000 images RVB 28x28. En sortie de cette couche, le tenseur résultat a une taille de [1000,32,24,24] correspondant aux 1000 images et 32 features. La taille 24x24 vient de la taille des images en entrée 28x28 réduite de 4 = 5-1, avec 5 correspond à la taille des features. Le tenseur des poids est de taille [32,3,5,5] correspondant aux 32 features de taille 5x5 traitant des images à 3 canaux RVB. Cette couche utilise $32 \times 3 \times 5 \times 5 + 32 = 2\,432$ poids. On peut remarquer que cela est beaucoup moins qu'une couche Linear placée en début de réseau.

10.1.4 Interprétation des features

Mais à quoi peuvent correspondre les features une fois appris ? Il s'agit ici d'une question difficile. Cependant, on a réussi à déterminer que pour la couche traitant l'image d'entrée, ces features correspondent à des détecteurs de contours. Voici un exemple de résultat pour la classification des chiffres manuscrits :



Ainsi les features cherchent à détecter, dans les images traitées, des « caractéristiques » permettant de décrire l'information présente dans l'image. Si une couche Conv a seulement 5 features, elle va devoir en choisir 5 qui représentent au mieux l'information recherchée dans l'image.

10.1.5 La Couche Pool

L'objectif d'une couche Pool est de réduire la quantité d'informations. Par exemple, pour un tenseur 2D, on peut choisir de le diviser en blocs de taille 2x2 et d'effectuer pour chaque bloc une opération du type : moyennage, min ou max... Le tenseur résultat aura donc une taille 4 fois moindre. Les couches Pool permettent de réduire le flux de données d'une couche à l'autre. Vous pouvez trouver plus d'informations dans la documentation PyTorch ici :

<https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html>

MAXPOOL2D

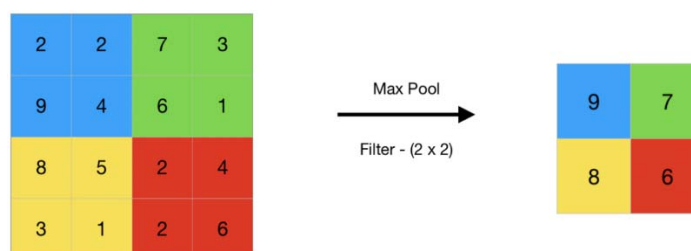
```
CLASS torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1,
    return_indices=False, ceil_mode=False) [SOURCE]
```

Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W) , output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0, \dots, kH-1} \max_{n=0, \dots, kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

Nous présentons un exemple de MaxPool-2D avec une réduction 2x2 :



10.1.6 La couche Dropout

Cette couche permet de mettre à zéro aléatoirement certaines valeurs d'un tenseur ceci avec une probabilité donnée. Il n'y a pas réduction de la taille des données, mais simplement une simulation d'une perte d'information. L'intérêt est d'améliorer l'indépendance entre les différentes features. En effet, supposons que le réseau doive trouver deux features A et B. Sans la technique de Dropout, il pourrait aussi bien sélectionner les features A et A+B, car les traitements sur les couches suivantes pourraient compenser ce choix pour reconstruire les features A et B idéaux. En désactivant de temps à autre la feature A ou la feature B, cela force le feature restant à converger sans tenir compte des valeurs du feature voisin.

Les couches Dropout ne sont utiles que durant la phase d'apprentissage et non durant la phase de validation. Ainsi, il faudra écrire :

- Au début du traitement du lot : `MonReseau.train()` pour activer les couches Dropout
- Au début de la phase de validation : `MonReseau.eval()` pour désactiver les couches Dropout

Les choses ne se résument pas seulement à une activation/désactivation. En effet, si l'on choisit une probabilité p d'effacer une donnée, cela veut dire que la somme des valeurs en sortie est abaissée (en moyenne) d'un facteur $1-p$ par rapport à l'entrée. Pour compenser cette perte, un facteur $1/(1-p)$ est appliquée aux valeurs conservées pour préserver le niveau moyen des valeurs d'entrée.

Voici un extrait des informations que l'on peut trouver dans la documentation de PyTorch à l'adresse :

<https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html>

DROPOUT

CLASS `torch.nn.Dropout(p=0.5, inplace=False)` [SOURCE]

During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.

This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper [Improving neural networks by preventing co-adaptation of feature detectors](#).

Furthermore, the outputs are scaled by a factor of $\frac{1}{1-p}$ during training. This means that during evaluation the module simply computes an identity function.

Parameters

- **p** – probability of an element to be zeroed. Default: 0.5
- **inplace** – If set to `True`, will do this operation in-place. Default: `False`

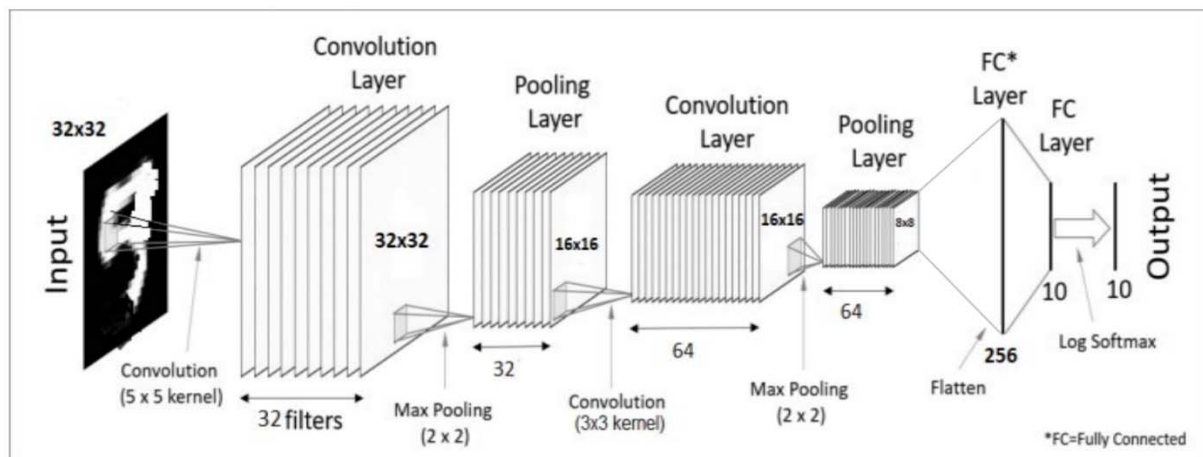
Voici un exemple de l'application d'une couche Dropout avec une probabilité de 0.5 sur un tenseur rempli de valeurs 1. Vous remarquerez que sur les 10 valeurs présentes 6 ont été mises aléatoirement à zéro. Les valeurs de sortie sont multipliées par $2 = 1/p$. Si vous lancez plusieurs fois ce test, vous obtiendrez des résultats différents.

```
import torch
m = torch.nn.Dropout(p=0.5)
input = torch.ones(10)
output = m(input)
print(input)
print(output)
```

```
>> tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
>> tensor([0., 0., 0., 0., 2., 2., 0., 0., 2., 2.])
```

10.1.7 Lecture des illustrations

Les articles sur les réseaux de neurones ont fait apparaître un art graphique dédié à la représentation des différents réseaux ! Les styles peuvent varier, quoique les empilements de rectangles ou de cube soit assez présents mais la tendance générale reste au style épuré. Ainsi, pour décrypter ces diagrammes, il faut connaître les spécificités de chaque couche pour déduire les informations implicites. Voici un exemple :



Analysons ce réseau de gauche à droite :

- En entrée, on trouve une image 2D de résolution 32x32.
- La sortie de la première couche est le résultat d'une convolution 2D donnant un tenseur de taille (32,32,32). Il y a donc 32 features dans cette couche Conv. Les features étant de taille 5x5, la résolution de sortie aurait dû être de 28x28 et non 32x32, un paramètre de zéro padding a dû être utilisé pour compenser la perte de résolution.
- La couche de Max-Pooling réduit la résolution par 2, ce qui donne un tenseur (32,16,16). Il y a toujours 32 canaux car l'opération de MaxPooling2D ne diminue pas le nombre de canal.
- Une nouvelle couche Conv de 64 features est appliquée. L'entrée étant de (32,16,16), la sortie 64x16x16 et le kernel de 3x3, la couche Conv2D a été construite avec les paramètres (32,64,(3,3)). Cette couche traite donc le tenseur d'entrée comme une image 16x16 avec 32 canaux.
- La couche de Max-Pooling suivante réduit la résolution par 2, ce qui donne un tenseur (64,8,8) en sortie. Le nombre de canaux reste inchangé.
- Le tenseur [64,8,8] subit un reshape pour être mis sous la forme d'un tenseur 1D de taille 64x8x8 = 4096, cette opération est indiquée par la flèche flatten (aplatissage).
- On trouve ensuite une couche Linear (aussi appelée Fully Connected). La grandeur 256 indique sûrement le nombre de neurones présents dans cette couche. La fonction d'activation n'est pas précisée, on peut supposer qu'il s'agit d'une fonction ReLU.
- La dernière couche correspond à une deuxième couche Linear. Elle contient 10 neurones qui calculent 10 scores correspondant aux 10 catégories étudiées.
- La fonction d'erreur indiquée par Log Softmax correspond au critère Softmax et entropie.

10.2 Un peu d'histoire

10.2.1 La quête de performance

Dans le domaine de la classification d'images, une course à la performance s'est instaurée pour atteindre les taux d'erreur les plus faibles sur les bases existantes.

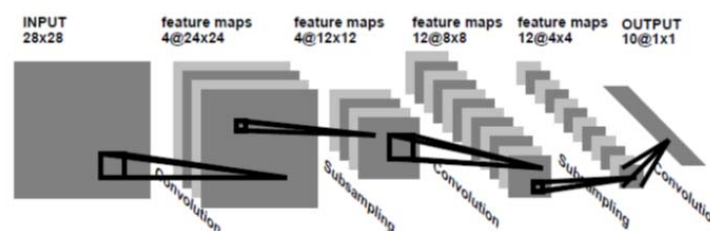
En ce qui concerne le problème de la classification des chiffres manuscrits, plusieurs algorithmes ont été mis en place pour passer de 98% de précision à 99.85% aujourd'hui, gagnant chaque fois quelques dixièmes de pourcent. Cela peut sembler superflu mais il n'en est rien. En effet, l'application historique de la reconnaissance de chiffres manuscrits était la lecture de codes zip sur les enveloppes postales. Si l'on part du principe qu'une machine de tri traite 100 000 enveloppes par jour et qu'il y a 5 chiffres à analyser, un faible taux d'erreur de 2% génère à grande échelle beaucoup d'erreurs.

On peut aussi citer la base d'images annotées ImageNet et son concours ImageNet Large Scale Visual Recognition Challenge (ILSVRC) organisé de 2010 à 2017. Chaque année sont produits des centaines de réseaux permettant de gagner quelques dixièmes de pourcent de précision. Cela peut aussi paraître futile mais à l'échelle de la conduite automatique, où l'algorithme peut fonctionner 8h par jour sur plusieurs jours et dans plusieurs millions de voitures, la recherche d'un taux d'erreur le plus faible possible peut se comprendre.

10.2.2 LeNet

La reconnaissance des chiffres manuscrits est un problème historique de la classification d'images et le réseau LeNet, spécialement créé pour cette problématique, fait partie des classiques aujourd'hui. LeNet est un réseau neuronal convolutif proposée par Yann LeCun et al. en 1989 a été construit en utilisant des couches Conv, Pool et Linear qui représentent aujourd'hui des couches incontournables pour la création de CNN. Nous présentons quatre étapes de l'évolution de ce réseau dans la suite.

10.2.3 LeNet-1



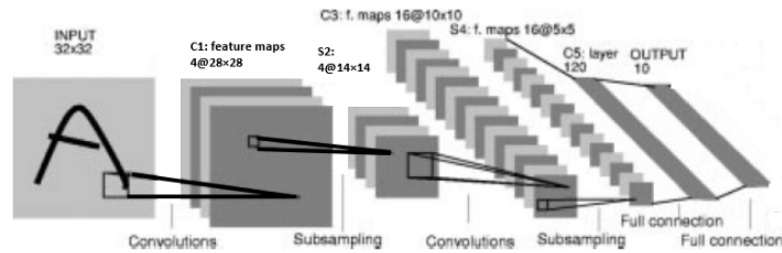
Topologie :

- Input : 28×28 input image
- Conv : 4 features 5×5 → (4,24,24)
- Average Pooling 2×2 → (4,12,12)
- Conv : 12 features 5×5 → (12,8,8)
- Average Pooling 2×2 → (12,4,4)
- Linear – 10 neurones

LeNet-1 atteint un niveau d'erreur de 1,7% sur les données de test. Lorsque LeNet-1 a été mis en place, les auteurs ont utilisé des couches average-Pool 2x2. Aujourd'hui, on trouve plus souvent des couches max-Pool 2x2 car à l'usage, il semble qu'elles accélèrent l'apprentissage. Le fait de choisir les valeurs les plus fortes permet au réseau de se focaliser sur les features les plus importants.

Les fonctions d'activation utilisées à l'époque étaient tanh et sigmoïde. La fonction ReLU n'existait pas encore. Aujourd'hui, il semble que la fonction ReLU soit largement préférée car elle accélère grandement la convergence durant l'entraînement.

10.2.4 LeNet-4

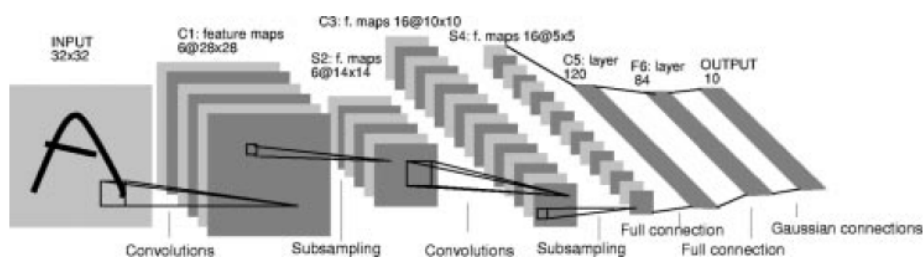


Topologie :

- Input : 32x32 input image
- Conv : 4 features 5x5 → (4,28,28)
- Average Pooling 2x2 → (4,14,14)
- Conv : 16 features 5x5 → (16,10,10)
- Average Pooling 2x2 → (16,5,5)
- Linear 120 neurones
- Linear 120 neurones

Avec plus de features et une couche Linear supplémentaires, le taux d'erreur descend à 1,1%.

10.2.5 LeNet-5



LeNet-5, plus communément appelé "LeNet" amène finalement peu de modifications par rapport à LeNet-4.

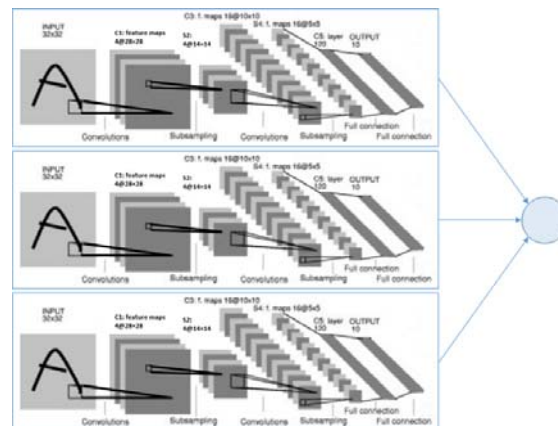
Topologie :

- Input : 32x32 input image
- Conv : 6 features 5x5 → (6,28,28)
- Average Pooling 2x2 → (6,14,14)
- Conv : 16 features 5x5 → (16,10,10)
- Average Pooling layers 2x2 → (16,5,5)

- Linear - 120 neurones
- Linear – 84 neurones
- Linear – 10 neurones

En augmentant le nombre de features et en ajoutant une couche Linear, le taux d'erreur atteint cette fois 0.95% lors des tests.

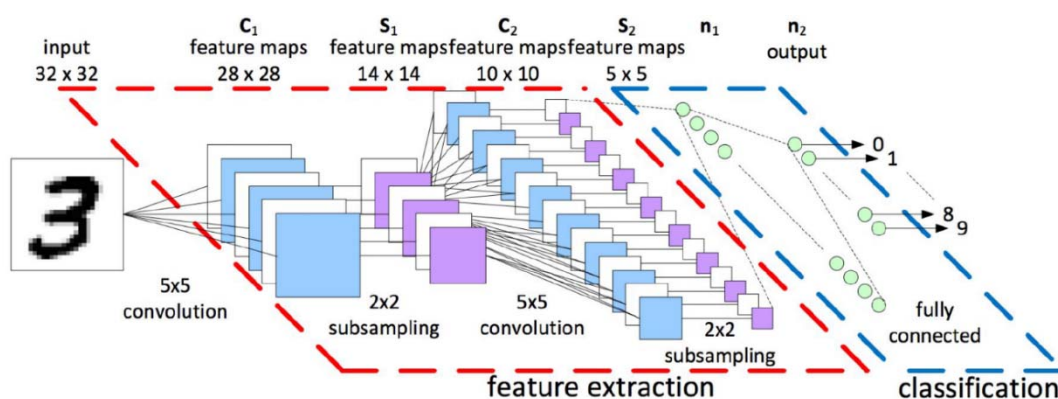
10.2.6 Boosted – LeNet 4



Le boosting est une technique consistant à combiner les résultats de plusieurs réseaux pour obtenir une réponse plus précise. Vous utilisez aussi cette technique lorsque vous demandez un conseil à différents experts pour avoir plusieurs avis. Dans cette version, la fusion des résultats consistait simplement à cumuler les scores obtenus par chaque réseau.

En utilisant la méthode du boosting sur 3 réseaux de type LeNet-4, le taux d'erreur est tombé à 0.7% soit une meilleure précision que LeNet-5. Par la suite, la technique du boosting a été largement utilisée pour d'autres réseaux afin d'améliorer les meilleurs résultats connus.

10.2.7 Extraction de features et classification



La partie gauche du réseau, proche de l'entrée, cherche à évaluer des critères (features en anglais) permettant à la partie droite du réseau de faciliter sa décision. Ainsi, les réseaux utilisés en classification d'images ont tendance à se séparer en deux parties :

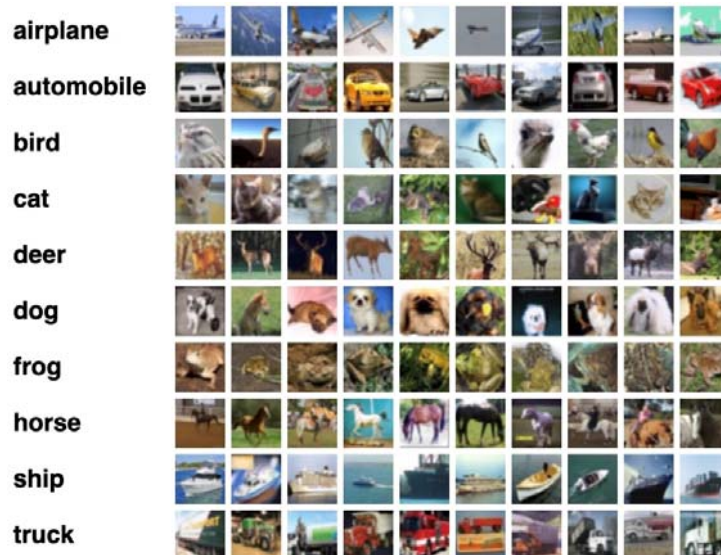
- La partie gauche, proche des données d'entrée, a pour objectif d'estimer et de trouver un ensemble de critères « feature extraction » ceci en empilant plusieurs couches Conv.

- La partie droite doit construire sa réponse et apprendre à classifier l'image à partir des critères évalués par la partie gauche du réseau. Pour mettre en place cette partie, on utilise plusieurs couches Linear empilées.

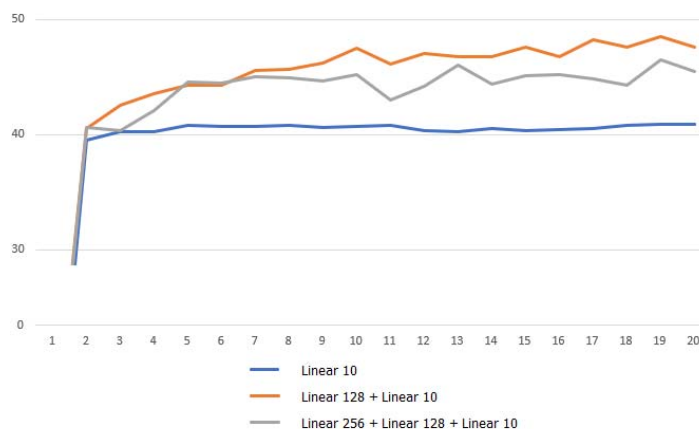
10.3 Exercice

10.3.1 Mise en place

Ouvrez le fichier nommé « Ex Convolution.py ». Nous allons maintenant travailler sur la base d'images de CIFAR10 définissant 10 classes d'images :



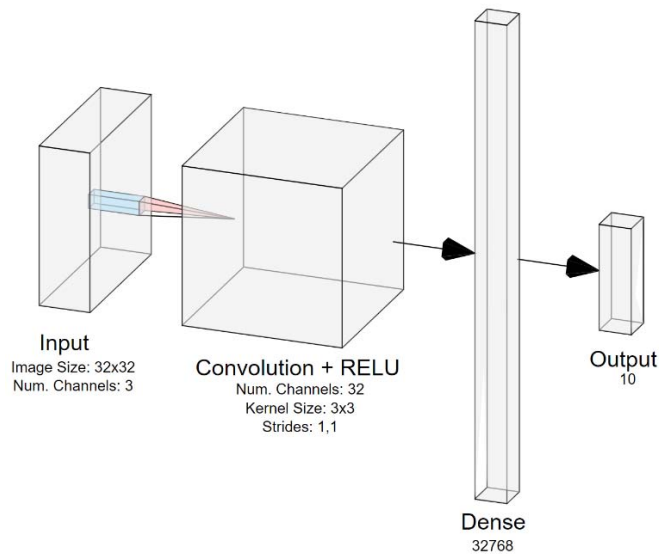
Le réseau présent dans l'exemple utilise une première couche Linear suivie d'une deuxième couche Linear de 10 neurones. En lançant le programme, normalement, le taux maximal de prédiction devrait être autour des 48%. En comparaison, un réseau avec une seule couche Linear de 10 neurones n'atteint que 41%, il y a donc eu un gain en rajoutant une couche. Par contre, l'utilisation de 3 couches Linear 256/128/10 n'augmente pas les performances. Nous présentons dans le graphique ci-dessous les résultats obtenus sur 20 epochs :



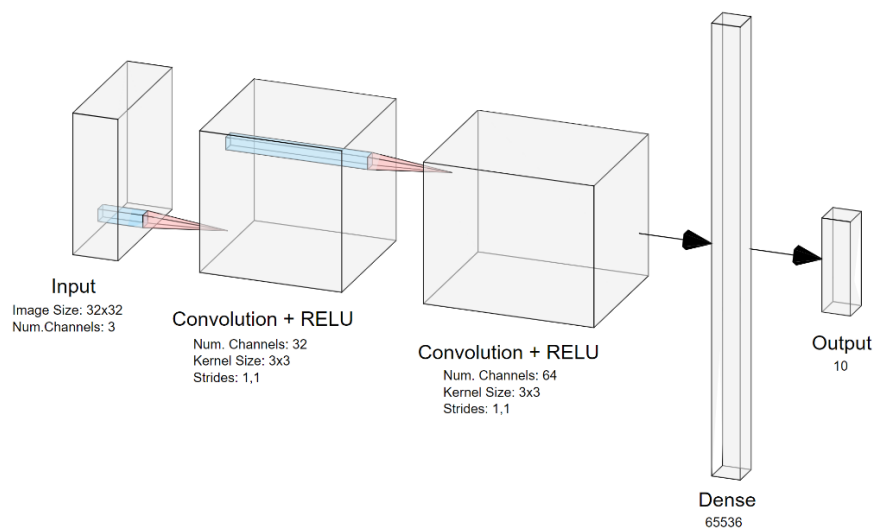
Vous remarquez que le taux de prédiction est largement en dessous des 98% obtenus sur la base MNIST. Pour améliorer les performances, vous allez mettre en place des couches convolutionnelles qui vont permettre d'améliorer les prédictions.

10.3.2 A votre tour

On vous propose d'implémenter deux réseaux à partir de leurs descriptions graphiques ci-dessous.



Réseau 1 :



Réseau 2 :

Il vous sera demandé d'envoyer le code des deux réseaux (dans un seul fichier) par mail avec le graphique des performances obtenues pour comparaison. Il est conseillé d'aller jusqu'à 80-120 epochs.