Deep Learning

Paul Lehaut

October 25, 2025

Contents

1 Bases du Machine Learning

Le Machine Learning est une discipline qui vise à concevoir des algorithmes généraux applicables à de nombreux problèmes et ce en partant uniquement des données.

Le data engineering consiste en l'étude approfondie des données et à les rendre exploitables. Il s'agit d'une étape cruciale qui concentre une grande partie du travail dans les problèmes réelles.

1.1 Types d'apprentissage

Selon les scénarios, il existe différents types d'apprentissage:

- L'apprentissage supervisé: les données sont composées à la fois des données d'entrées et des sorties attendues. Le but est d'implémenter un algorithme capable de généraliser proprement pour de nouvelles entrées.
- L'apprentissage non supervisé: les données sont fournies sans les sorties attendues, il s'agit alors de trouver des structures au sein des données (typiquement à l'aide de méthodes de clustering).
- Il en existe encore d'autres type comme le reinforcement learning qui permet à un algorithme d'interagir avec son environnement.

Il faut également faire la distinction entre deux types de méthodes:

- La méthode paramétrique: elle définit un ensemble de fonctions pour lesquelles il convient de trouver les meilleurs paramètres.
- La méthode non paramétrique: elle dépend directement des données.

1.2 Apprentissage Supervisé Paramétrique

Soit X un espace d'entrée et Y un espace de sortie, on va chercher à apprendre une fonction de prédiction: $f : X \longrightarrow Y$ paramétrée par un paramètre θ. On va donc chercher les, voire les, paramètre θ qui maximise la performance de la fonction f.

Formelement, on définit une fonction de coût $l(y', y)$, qui mesure l'erreur entre la prédiction y' et la valeur attendue y, ainsi qu'une fonction de risque: $R(f) = \mathbb{E}_{(x,y)}(l(f(x), y))$.

Malheureusement, on ne connaît jamais la distribution réelle des données mais plutôt simplement un échantillon Z. On définit alors le risque empirique:

$$R_Z(f) = \frac{1}{|Z|} \sum_{(x,y)\in Z} l(f(x), y).$$

L'apprentissage consiste alors à minimiser une fonction de perte définie à partir de ce risque et, éventuellement, d'autres termes.

Pour éviter qu'un modèle over/underfit les données, il est souvent judicieux de séparer les données en trois sets: un pour l'entraînement (train set), un pour la validation du modèle (validation set) et enfin un pour estimer la performance rélle (test set).

2 Premier Réseau de Neurones

2.1 Perceptron Multicouche (MLP)

Un (MLP) est la composition de couches linéaires ($x \mapsto Wx + b$ avec W le poids et b le biais) et non-linéaires. Ces architectures sont caractérisées par leurs hyperparamètres (taille des couches, type de non-linéarité).

Théorème: Cybenko

Un MLP à 2 couches peut approximer n'importe quelle fonction continue sur un compact avec une précision arbitraire.

Définition: Fonction de perte

On définit la perte quadratique moyenne (MSE) adaptée à la classification multi-classe via codage one-hot par:

$$L_{MSE}(\theta, D) = \frac{1}{N}\sum_{i=1}^{N} ||f_\theta(x_i) - y_i||^2 \quad \text{où D correspond au dataset.}$$

Cette fonction ne cherche qu'à réduire l'erreur sur les données d'entraînement, il y a donc un risque d'overfiting, on peut donc ajouter une pénalité sur la taille des poids, on définit alors le coefficient de régularisation $\lambda > 0$, la fonction à minimiser devient:

$$L(\theta, D) = L_{MSE}(\theta, D) + \lambda||\theta||^2$$

plus λ est grand plus le réseau est contraint de rester simple.

2.2 Optimisation

Même avec un modèle simple, on obtient un problème d'optimisation non convexe, on utilise donc la descente de gradient.
Le principe de la descente gradient est le suivant: à chaque itération, on déplace le paramètre (ici θ) dans la direction opposée au gradient de la fonction de perte: $\theta \leftarrow \theta - \nu\nabla_\theta L(\theta, D)$ où ν correspond au taux d'apprentissage.
Cette technique nécessite toutefois de calculer le gradient sur toutes les données ce qui est couteux sur les grands datasets.

Pour contourner ce problème, on utilise la méthode de descente gradient stochastique (SGD) qui consiste à ne pas calculer le gradient sur l'ensemble des données mais à partir d'un échantillon.
Il en existe deux variantes:

- SGD pur: à chaque itération on prend un seul exemple aléatoire: $\theta \leftarrow \theta - \nu\nabla_\theta l(f_\theta(x_i), y_i)$.
- Mini-batch SGD: à chaque itération on prend un petit lot de données (batch) de taille B:

$$\theta \leftarrow \theta - \nu\frac{1}{B}\sum_{i=1}^{B} \nabla_\theta l(f(x_i), y_i).$$

  Cette version est plus simple en pratique car plus stable que le SGD pur et plus efficace que le gradient complet (elle est également bien adapté au GPU).

2.3 Hyperparamètres Clés de l'Optimisation

Les hyperparamètres clés sont:

- Le taux d'apprentissage: c'est le paramètre ν qui détermine la taille des pas dans l'espace des paramètres, lorsqu'il est trop grand le modèle diverge et lorsqu'il est trop petit l'apprentissage peut être très lent voire bloqué dans un minimum local. Souvent on choisit au départ $\nu \in [10^{-3}, 10^{-2}]$.
- La taille des batch B: une trop petite taille permet de mieux explorer le paysage de la perte mais augmente l'importance du bruit, lorsqu'elle est trop grande le modèle risque d'overfit. En pratique on choisit des batch de 32 ou 64 pour les petits modèles.
- Le calendrier du taux d'apprentissage: le taux d'apprentissage n'est souvent pas constant, on le fait varier de diverses façons (division toutes les x epochs, décroissance en forme de cosinus, augmentation au fur et à mesure etc). L'objectif est d'éviter de se retrouver coincé dans des minima locaux tout en assurant la convergence en fin d'entraînement.
- Le nombre d'époques (Epochs): l'algorithme voit l'ensemble des données d'entraînement en une epoch, lorsque le nombre d'epoch est trop grand le modèle risque d'overfit.

3 Backpropagation

La backpropagation se base essentiellement sur la règle de dérivation en chaîne. Chaque couche est vue comme pouvant faire deux choses:

- Forward: Calculer sa sortie en fonction de son entrée et de ses paramètres.
- Backward: Calculer les gradients de la perte par rapport à ses paramètres et à son entrée.

L'algorithme se fait en deux étapes:

- Forward pass: On fait passer les données à travers le réseau pour obtenir la prédiction.
- Backward pass: On fait revenir les gradients depuis la sortie jusqu'à l'entrée.

A la fin on a les gradients de la perte L par rapport à tous les paramètres du réseau, ce qui permet de le mettre à jour avec la descente de gradient.
Par exemple pour un réseau de trois couches: $x \to h_1 \to h_2 \to y$ on calcule en forward h_1, h_2 et y, puis en backward on remonte: $\frac{\partial L}{\partial y}$, puis $\frac{\partial L}{\partial h_2}$, ensuite $\frac{\partial L}{\partial h_1}$, et enfin les dérivées par rapport aux poids.

3.1 Formalisation

Considérons un réseau à K couches, chaque couche f_k prend en entrée un vecteur x_{k-1} et des paramètres θ_k pour produire une sortie:

$$x_k = f_k(x_{k-1}, \theta_k)$$

la sortie finale du réseaux est $y = x_K$.

Soit une fonction différentiable de perte L pour la sortie y du réseau, le backward pass a pour objectif de calculer, pour tout paramètre θ_k, la dérivée $\frac{\partial L}{\partial \theta_k}$, on peut alors mettre à jour la valeur de ce paramètre:

$$\theta_k \leftarrow \theta_k - \alpha \frac{\partial L}{\partial \theta_k}(\theta_k) \quad \text{avec } \alpha \text{ le taux d'apprentissage.}$$

Pour effectuer ce calcule on calcule récurcivement $\frac{\partial L}{\partial \theta_k}$ à partir de la dernière couche:

$$\frac{\partial L}{\partial \theta_k}(\theta_k) = \frac{\partial L}{\partial x_k}(x_k) \frac{\partial f_k}{\partial \theta_k}(x_{k-1}, \theta_k)$$

or, par hypothèse, L est différentiable donc on peut calculer directement $\frac{\partial L}{\partial x_K}$ puis:

$$\frac{\partial L}{\partial x_{k-1}}(x_{k-1}) = \frac{\partial L}{\partial x_k}(x_k) \frac{\partial x_k}{\partial x_{k-1}} = \frac{\partial L}{\partial x_k}(x_k) \frac{\partial f_k}{\partial x_{k-1}}(x_{k-1}\theta_k).$$

4 Réseaux de Neurones Convolutifs (CNN): Achitectures LeNet et AlexNet

Il existe une différence fondamentale entre les CNN et les MLP, en effet un MLP considère chaque entrée comme indépendante des autres. Or, cette interprétation des données peut être problématique, dans une image par exemple les pixels proches sont corrélés, on utilise alors un CNN pour conserver cette cohérence spatiale en exploitant des opérations locales appelées convolutions.

4.1 Principe de la Convolution

Une couche de convolution apprend un ensemble de filtres (ou noyaux) qui détectent des motifs locaux dans les données d'entrée.

Définition:

Soit une entrée $X \in \mathbb{R}^{W \times H \times C_{in}}$ où W et H sont la largeur et la hauteur, C_{in} correspond au nombre de canaux (par exemple trois pour du RGB).

Un noyau ou filtre $K \in \mathbb{R}^{S \times S \times C_{in} \times C_{out}}$ (pour l'exemple typique d'un filtre en carré) où S est la taille du filtre et C_{out} le nombre de canaux de sortie.

La sortie de la couche de convolution est alors: $Y_i = K_i * X$ pour $i \in [|1, C_{out}|]$ avec $*$ la convolution en 2D.

Chaque filtre K_i balaie l'image, effectue des multiplications locales, et produit une carte d'activation (feature map) qui indique où le motif est présent.

Pour des données brutes (sans padding), la taille de sortie est $W_{out} = W - S + 1$, $H_{out} = H - S + 1$, avec un padding (ajout de zéro aux limites des données donc autour de l'image par exemple), on conserve la taille: $W_{out} = W$ et $H_{out} = H$.

On peut également implémenter un pas s (stride) afin d'échantilloner moins souvent et donc réduire la dimension de sortie: $W = \left\lfloor \frac{W-S}{s} + 1 \right\rfloor$.

En pratique, la plupart des CNN modernes utilisent des filtres 3×3, un padding de 1 et un stride de 1 (voire 2).

L'intérêt de la convolution est de diminuer significativement le nombre de paramètres du modèle. Par exemple une couche linéaire classique reliant tous les pixels entre eux aurait:

$$(W \times H \times C_{in})(W_{out} \times H_{out} \times C_{out})$$

paramètres ce qui devient vite énorme. Tandis que une convolution ne dépend que d'un petit voisinage et donc possède uniquement: $S^2 \times C_{in} \times C_{out}$ paramètres.

Une convolution permet donc de réduire significativement le poids tout en exploitant la structure spatiale.

4.2 Autres Couches Importantes

Une architecture de CNN manipule de nombreux tenseurs de différentes dimensions, des opérations pour changer ces dimensions peuvent donc s'avérer utiles, il sera par ailleurs de les interpréter comme des couches à part entières de l'architecture bien qu'elles ne puissent présenter aucun paramètre à entraîner.

On considère donc des couches de:

- mise en forme (reshaping) qui changent la structure des tenseurs,
- pooling qui réduisent la taille spatiale des cartes caractéristiques pour diminuer le coût de calcul et augmenter la robustesse aux petites translations tout en conservant les activations les plus significatives. Il en existe deux types principaux:

  - le Max Pooling qui garde la valeur maximale dans chaque région

  - le Average Pooling qui fait la moyenne par région.

  On détaille le fonctionnement de tels fonctions juste après.
- interpolation (upsampling) qui augmentent la taille spatiale

Exemple pour le pooling:

On s'intéresse à:

$$\begin{pmatrix} 1 & 3 & 2 & 0 & 1 & 2 \\ 4 & 6 & 5 & 1 & 3 & 1 \\ 7 & 2 & 8 & 2 & 0 & 4 \\ 3 & 5 & 0 & 6 & 7 & 3 \\ 2 & 1 & 9 & 8 & 2 & 4 \\ 1 & 0 & 3 & 5 & 2 & 1 \end{pmatrix}$$

à laquelle on applique MaxPool2d(kernel_size=2, stride=2), on obtient donc:

$$\begin{pmatrix} 6 & 5 & 3 \\ 7 & 8 & 7 \\ 2 & 9 & 4 \end{pmatrix}.$$

4.3 LeNet: Le Premier CNN

Proposé par Yann LeCun dans les années 90, il est utilisé pour reconnaîtres les chiffres manuscrits sur les chèques bancaires.

Sa structure générale est la suivante:

- Convolution $->$ ReLU $->$ pooling
- Convolution $->$ ReLU $->$ pooling
- Fully Connected (dense) $->$ ReLU
- Fully Connected $->$ sortie (10 classes)

Ce modèle possède peux de paramètres (quelques centaines de milliers), il s'agit du premier CNN montrant qu'on poouvait apprendre directement à partir de pixels pour faire de la reconnaissance.

4.4 AlexNet: La Révolution de 2012

Avant 2012 les approches de vision par ordinateur reposaient surtout sur des descripteurs manuels. Puis arrivent Alex Krizhevsky, Ilya Sutskever et Geoffrey Hinton qui publient AlexNet un réseau de deep learning qui écrase la concurrence (réduisant par deux la marge d'erreur).

Cette révolution s'organise autour de cinq points clés:

- un réseau très profond pour l'époque (8 couches)
- une utilisation massive du GPU pour l'entraînement
- la fonction d'activation ReLU plus facile à entraîner que le sigmoid
- la technique du dropout
- et une data augmentation

Le dropout est une technique de régularisation qui consiste à 'éteindre' aléatoirement un certain pourcentage des neurones à chaque étape d'entraînement. L'objectif de cette méthode est d'empêcher le réseaux de devenir trop dépendant d'un petit ensemble de neurones pour avoir une meilleur généralisation.
Par exemple supposons qu'on est en entrée le vecteur d'activation $x = (2, 8, -3, 11, 5)$ et pour la couche un dropout de 50%, alors on génère selon une loi de Bernoulli(0.5): $m = (1, 0, 1, 0, 1)$, alors l'entrée devient: $x' = x \odot m = (2, -3, 5)$.

Néanmoins, quand le réseau est utilisé pour faire des prédictions, tous les neurones sont activés mais les sorties sont réduites par la probabilité de maintient p: $x_{test} = px_{train}$.

On va désormais détailler l'architecture d'AlexNet. Pour ce faire on considère une taille d'entrée classique 224x224x3 ($H \times W \times C_{in}$). L'architecture est à retrouver dans AlexNet.py.

La première couche conv1:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 3,\ C_{out} = 96,\ S = 11,\ stride = 4,\ padding = 2$ activation de ReLU
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{224+2\cdot 2-11}{4} \right\rfloor + 1 = 55 = H_{out}$ la sortie totale est donc: $55^2 \times 96$.

- Nombre de paramètres, poids et biais: poids= $96 \times 3 \times 11^2 = 34848$, chaque filtre a un biais scalaire unique donc 96 biais, soit 34 944 paramètres au total.
- Il s'agit ici d'une réduction forte de la dimension spatiale dès le départ pour minimiser le coût de calcul, le filtre large permet d'observer des motifs de grande échelle.

La couche pool1:

- Type: MaxPool2d
- Paramètres de la fonction: $kernel = 3,\ stride = 2$ pas de padding.
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{55-3}{2} \right\rfloor + 1 = 27 = H_{out}$ la sortie totale est donc: $27^2 \times 96$.
- Nombre de paramètres, poids et biais: pas de paramètres.
- L'objectif des couches de pooling est ici de réduire la résolution et les invariances locales aux translations.

La seconde couche conv2:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 96,\ C_{out} = 256,\ S = 5,\ stride = 1,\ padding = 2$ activation de ReLU
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{27+2^2-5}{1} \right\rfloor + 1 = 27 = H_{out}$ la sortie totale est donc: $27^2 \times 96$.
- Nombre de paramètres, poids et biais: poids= $256 \times 96 \times 5^2 = 614400$, chaque filtre a un biais scalaire unique donc 256 biais, soit 614 656 paramètres au total.

La couche pool2:

- Type: MaxPool2d
- Paramètres de la fonction: $kernel = 3,\ stride = 2$ pas de padding.
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{27-3}{2} \right\rfloor + 1 = 13 = H_{out}$ la sortie totale est donc: $13^2 \times 96$.
- Nombre de paramètres, poids et biais: pas de paramètres.

La troisième couche conv3:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 256,\ C_{out} = 384,\ S = 3,\ stride = 1,\ padding = 1$ activation de ReLU
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{13+2-3}{1} \right\rfloor + 1 = 13 = H_{out}$ la sortie totale est donc: $13^2 \times 384$.
- Nombre de paramètres, poids et biais: poids= $384 \times 256 \times 3^2 = 884736$, chaque filtre a un biais scalaire unique donc 384 biais, soit 885 120 paramètres au total.

La quatrième couche con4:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 384,\ C_{out} = 384,\ S = 3,\ stride = 1,\ padding = 1$ activation de ReLU
- Taille de sortie (spatial): la taille de sortie reste $13^2 \times 384$
- Nombre de paramètres, poids et biais: poids= $384 \times 384 \times 3^2 = 1327104$, chaque filtre a un biais scalaire unique donc 384 biais, soit 1 327 488 paramètres au total.

La cinquième couche conv5:

- Type: Conv2d
- Paramètres de la fonction: $C_{in} = 384,\ C_{out} = 256,\ S = 3,\ stride = 1,\ padding = 1$ activation de ReLU
- Taille de sortie (spatial): la taille de sortie est $13^2 \times 256$.
- Nombre de paramètres, poids et biais: poids= $256 \times 384 \times 3^2 = 884736$, chaque filtre a un biais scalaire unique donc 256 biais, soit 884 992 paramètres au total.

On peut observer dans les couches précédentes un enchaînement de petits filtres (3×3) capturant les motifs locaux et combinatoire.
La couche pool3:

- Type: MaxPool2d
- Paramètres de la fonction: $kernel = 3,\ stride = 2$ pas de padding.
- Taille de sortie (spatial): $W_{out} = \left\lfloor \frac{13-3}{2} \right\rfloor + 1 = 6 = H_{out}$ la sortie totale est donc: $6^2 \times 256$.
- On flatten ensuite pour la couche fully connected: vecteur de dimension $256 \times 6^2 = 9216$.

On passe désormais aux couches dites fully connected.
La couche fc1, précédée d'une couche de dropout pour éviter le surapprentissage:

- Type: dense.
- Paramètres: $C_{in} = 9216,\ C_{out} = 4096$ puis activation de ReLU.
- Nombre de paramètres poids, et biais: Poids= 4096×9216, biais=4096 donc 37 752 832 paramètres au total.

La seconde couche fc2, précédée d'une couche de dropout pour éviter le surapprentissage:

- Type: dense.
- Paramètres: $C_{in} = 4096,\ C_{out} = 4096$ puis activation de ReLU.
- Nombre de paramètres poids, et biais: Poids= 4096×4096, biais=4096 donc 16 781 312 paramètres au total.

Et enfin la dernière couche et la sortie du programme fc3:

- Type: dense.
- Paramètres: $C_{in} = 4096,\ C_{out}$ =NbClasses puis activation de ReLU.
- Nombre de paramètres poids, et biais: Poids= 4096×NbClasses, biais=NbClasses donc 4 097 000 paramètres au total pour 1000 classes.

En 2012 il était en effet courant d'enchaîner les couches fully connected pour transformer les caractéristiques spatiales en représentations globales et classifier (comme nous allons le voir c'est très coûteux en paramètres et mémoire).
Le réseaux compte donc un total d'environ 62.4M paramètres (il est intéressant de constater que 94% des paramètres se trouvent dans les couches fc dont 60% dans la première).

4.4.1 L'intérêt des Couches Conv2d, ReLU et MaxPool2d

On détaille tout d'abord chacune de ces fonctions:

- La fonction Conv2d (Convolution 2D): Son but est d'extraire les motifs locaux d'une image (bords, textures, formes, motifs répétitifs, etc), c'est à ce moment que le réseaux repère et analyse les structures visuelles.

  Le principe des convolutions est détaillé au début de cette section.

  L'intérêt d'utiliser ces filtres est ici de les appliquer partout dans l'image afin de reconnaître un motif quelque soit son emplacement.

- La fonction ReLU (Rectified Linear Unit): C'est une fonction d'activation qui a pour objectif d'introduire de la non linéarité dans le réseau. Sans cette fonction même les CNN profonds seraient équivalents à une seule convolution linéaire. La ReLU s'est imposée car il s'agit d'une activation simple, rapide et efficace.

  Mathématiquement, elle correspond à la fonction: $\max(0, x)$, cette fonction possède donc une dérivée simple: $1_{R_+^*}$, elle est rapide à entraîner, elle introduit énormément de zéros donc permet une meilleure généralisation.

- La fonction MaxPool2d: Il s'agit d'une fonction de pooling (son fonctionnement est détaillé au début de cette section).

L'intérêt de ces couches est de construire une hiérarchie de représentations.

En effet, au cours de l'entraînement le réseaux apprend automatiquement quels motifs sont utiles pour minimiser la fonction de perte. Cette apprentissage se déroule de la façon suivante:

- Initialement: Le poids de chacun des filtres est initilaisé aléatoirement, ils ne détectent rien a priori.
- Entraînement: Lorsqu'on calcule la perte (par exemple une erreur de classification) on fait ensuite un rétropropagation (backpropagation), les poids des filtres sont légèrement ajustés pour réduire l'erreur.
- Résultat: Certains filtres deviennent sensibles aux bords verticaux (par exemple $[-1,\ 0,\ 1]$), d'autre aux bords horizontaux, aux motifs à répétitions etc. Il est intéressant de constater que cette apprentissage permet au réseau de découvrir par lui même les motifs qui maximisent la bonne classification.

L'intérêt d'enchaîner ces couches est de permettre à chaque couche de prendre comme entrée les cartees d'activation de la couche précédente, les couches supérieures repèrent donc les détails de bas-niveau (bords, textures, couleurs) puis les couches suivantes les motifs intermédiaires (parties d'objets) puis les concepts entiers (objets, visages) et ainsi de suite.

4.4.2 L'intérêt des Couches Entièrements Connectées

Ces couches forment la tête du réseau, elles interprètent les caractéristiques extraites par les convolutions et prennent la décision finale de classification.

Remarque:

Dans la fonction forward(self, x) on remarque l'appele x = torch.flatten(x, 1) entre le passage des couches de convolutions aux couches entièrements connectées. En effet ces dernières prennent en entrée des vecteurs d'une seule dimension tandis que le réseau de convolution renvoi des cartes caractéristiques de dimension (256, 6, 6), l'appele x = torch.flatten(x, 1) convertit donc x en un vecteur de taille 256 x 6 x 6 = 9 216.

La première couche fc1 opère la transformation affine:

$$y = Wx + b \ \text{ avec } W \in \mathcal{M}_{4096,9216}(\mathbb{R}) \ \text{ est la matrice de poids, } b \in \mathbb{R}^{4096} \ \text{ le vecteur de biais}$$

$x \in \mathbb{R}^{9216},\ y \in \mathbb{R}^{4096}$ sont les entrée et sortie respective de la couche.

Ainsi, chaque neurone de cette couche reçoit tous les éléments d'entrée, apprend à les combiner et peut ainsi en déduire un représentation de l'image.

Les fonctions dropout et ReLU sont détaillées précédement.

Deuxième couche fc2: elle fonctionne de la même façon que la précédente avec $y = Wx + b$ (évidemment avec les

dimensions qui conviennent), néanmoins elle apprend à un niveau d'abstraction supérieur en combinant les motifs d'entrés pour en extraire des concepts.

Enfin la couche fc3 correspond à la sortie du réseau, elle convertit la représentation finale abstraite en un score de classe (dans le cas d'AlexNet). Par exemple l'interprétation des sorties de ces trois couches pourrait être:

- fc1: voit des yeux, des oreilles et des poils
- fc2: en déduit qu'il doit s'agir d'un animal à quattre pattes
- fc3: conclut que c'est un chien à 98%, un chat à 2%.

5 Batch Normalization

Il s'agit d'une technique visant à stabiliser et accélérer la convergence d'un réseau de neurones. Elle s'implémente comme une couche à part entière dans le réseau de neurones. Elle possède à la fois des paramètres estimés (comme la moyenne ou la variance) et des paramètres appris (comme un coefficient d'échelle et un biais).

Attention, son comportement diffère entre la phase d'entraînement et la phase de test (c'est une source courante d'erreurs).

5.1 Comportement durant l'Entraînement

Supposons que les entrées d'une couche BatchNorm (BN) soient des vecteurs de $\mathbb{R}^d$. On en considère alors un lot (batch) $B = \{x_1, ..., x_n\}$. On calcule, pour ce lot, sa moyenne μ_B et son écart-type σ_B, on note par ailleurs a (facteur de mise à l'échelle) et b (le biais) les paramètres appris qui sont également des vecteurs de $\mathbb{R}^d$.

La normalisation par lots agit indépendamment sur chaque dimension k du vecteur d'entrée selon la formule:

$$y_{i,k} = \frac{a_k(x_{i,k} - \mu_{B,k})}{\sqrt{\sigma^2_{B,k} + \epsilon}} + b_k$$

avec: ϵ une petite constante pour éviter une éventuelle division par zéro.

5.2 Comportement Durant le Test

Lors du test, les données ne viennent pas toujours par lot et les échantillons peuvent ne pas être indépendants, ainsi, plutôt que de calculer la moyenne et la variance du lot, BN utilise des estimations globales accumulées pendant l'entraînement. Ces moyennes et variances de référence sont mises à jour à chaque itération d'entraînement:

$$\mu_k \leftarrow m \times \mu_k + (1 - m) \times \mu_{B,k}, \quad \sigma^2_k \leftarrow m \times \sigma^2_k + (1 - m) \times \sigma^2_{B,k}$$

où m est un paramètre de momemtum proche de 1 (typiquement 0.9).

5.3 Intuition

L'idée fondamentale est qu'il est plus facile d'apprendre avec des données normalisées (moyenne nulle, variance de 1). Cette normalisation stabilise en effet la propagation des gradients, accélère la convergence et permet d'entraîner des réseaux plus profonds.

En pratique BN est placée juste avant la non-linéarité, en effet des activations comme ReLU sont très non linéaire autour de 0 mais deviennent presque linéaire ailleurs (pour des entrées très grandes ou très négatives) normaliser les données autour de zéro les gardes donc dans la zone active de la fonction.

Il est également important de noter que l'ajout de cette couche (qui est linéaire) n'augment pas la capacité de représentation du réseau (c'est-à-dire le type de fonction qu'il peut approximer), elle n'affecte que la façon dont le réseau est optimisé.

6 Classification à K-classes et par Estimation du Maximum de Vraisemblance (MLE)

6.1 Principe de l'Estimation du Maximum de Vraisemblance

Il s'agit d'un principe fondamental de statistique, il consiste à choisir les paramètres d'un modèle de sort que ce modèle rende les données observées aussi probables que possible, on explicitera plus tard.

On suppose pour ce faire que: X est une VA représentant les observations (par exemple une image), Y une VA représentant les classes et que la probabilité conditionnelle $\mathbb{P}(Y = y|X = x)$ est modélisée par un réseau de neurones paramétré par θ, noté $p_\theta(y|x)$. Ce qu'on veut intuitivement c'est que, si l'observation de x implique y, alors $p_\theta(y_i|x_i)$ soit grande. Alors, pour un jeu de données $D = \{(x_1, y_1), ..., (x_n, y_n)\}$, on suppose que les échantillons sont indépendants et identiquement distribués, la probabilité d'observer tout le dataset selon le modèle est:

$$p(D) = \prod_{i=1}^{n} p_\theta(y_i|x_i)p(x_i).$$

Remarque:

Comme $p(x_i)$ ne dépend pas de θ on peut l'ignorer pour l'optimisation.

On cherche donc à maximiser la vraisemblance: $\theta^* = \arg\max_\theta \prod_{i=1}^{n} p_\theta(y_i|x_i)$, en pratique, on travaillera plutôt sur le problème de minimiser la perte de log-vraisemblance négative:

$$L(\theta) = -\sum_{i=1}^{n} \log(p_\theta(y_i|x_i))$$

en effet la fonction log permet d'accentuer la perte considérée pour une petite baisse de probabilité.

6.2 Entropie Croisée

Pour une classification à K classes où chaque classe c est représentée par une vecteur one-hot: $y_i \in \mathbb{R}^k$ ($y_{i,c} = 1$ si c est la classe correcte pour x_i, 0 sinon), le réseau prédit un vectuer de probabilité $\tilde{y_i} = p_\theta(y|x_i)$ obtenu par une couche softmax (décrite juste après). La perte d'entropie croisée s'écrit alors:

$$L = -\frac{1}{n} \sum_{i=1}^{n} \sum_{c=1}^{K} y_{i,c} \log(\tilde{y_i}_c)$$

il s'agit en fait de la log vraisemblance-négative, mais écrite explicitement pour les sorties multinomiales.

6.3 La Couche Softmax

La softmax transforme les parties brutes du réseau (appelées logits) en probabilités normalisées. Si le réseau renvoie pour une entrée x un vecteur de score $z = (z_1, ..., z_K)$ la softmax est définie par:

$$p_\theta(y = c|x) = \frac{e^{z_c}}{\sum_{k=1}^{K} e^{z_k}}$$

cette transformation garantit les propriétés d'une distribution de probabilité à p_θ (probabilités positives de somme 1).

L'intuition globale peut se formuler de la façon suivante: le modèle produit des scores pour chaque classe, le softmax les transforme en probabilités, la cross-entropy compare ces probabilités à la vérité (le vecteur one-hot) et la backpropagation ajuste les poids pour maximiser la probabilité des bonnes classes.

Remarque:

PyTorch combine le plus souvent la softmax et la cross-entropy en une seule fonction (nn.CrossEntropyLoss()) qui calcule:

$$L = -\log(\frac{e^z y_i}{\sum_{k=1}^{K} e^{z_k}}).$$

7 Réseaux de Neurones Convolutifs Classiques: ResNet

Les réseaux résiduels (ResNet) forment une famille d'architectures convolutionnelles introduite en 2015 qui ont démontré d'excellentes performances sur une grande variété de tâches de vision par ordinateur tout en restant relativement simples à entraîner et à comprendre. Ils sont construits de manière modulaire ce qui permet de définir des réseaux de profondeurs variables avec un nombre de paramètres ajustables.

Aujourd'hui, bien que cette architecture ne soit plus la meilleure, elle reste une référence standard et robuste. Elle a également inspiré de nombreuses variantes: Wide-ResNet (qui étend la largeur du réseau et établi que la largeur peut être aussi importante que la profondeur et développé notamment à l'ENPC) et ResNeXt (rend le modèle plus efficace à coût de calcul équivalent) en 2016 et ConvNeXt (2022).

Avant ResNet, entraîner des réseaux profonds était difficile (même avec quelques dizaines de couches seulement), souvent un réseau plus profond performait moins bien qu'un réseau plus petit. Il ne s'agissait alors par d'un problème de sur-apprentissage, mais d'un problème d'optimisation: les gradients devenaient trop instables en se propageant. En effet, on rappelle que le gradient est calculé par en multipliant plusieurs dérivées intermédiaires entre les couches:

$$\frac{\partial L}{\partial x_0} = \frac{\partial L}{\partial x_n} \times ... \times \frac{\partial x_1}{\partial x_0}$$

où chaque terme du produit correspond à une matrice Jacobienne. Lorsqu'on les multiplie toutes, deux phénomènes apparaissent:

- Le gradient s'annule: si les dérivées sont légèrement inférieure à 1 alors le gradient tend vers 0 en remontant les couches et les premières couches n'apprenent donc presque plus rien.
- Ou au contraire le gradient explose, la perte diverge et l'entraînement devient chaotique.

ResNet a alors résolu ce problème avec une idée simple mais révolutionnaire: ajouter des connexions résiduelles (skip connections) qui facilitent l'apprentissage dans les réseaux très profonds.

7.1 Architecture

L'idée des connections résiduelles est assez simple: par défaut, une partie d'un réseau (si elle ne change pas la taille des caractéristiques) devrait apprendre l'identité. C'est-à-dire que, si une couche n'apporte pas d'amélioration, elle ne doit rien déformer. Ainsi, il devient plus facile d'entraîner des réseaux profonds car le chemin des informations est toujours accessible.

Formelement, au lieu de passer directement de la sortie $f(x)$ d'un bloc au suivant, on ajoute l'entrée x: Sortie du bloc=$f(x) + x$. Cette addition crée un chemin direct pour les gradients évitant leur disparition.

Le bloque de base d'un ResNet (voir ResNet.py) est conçu pour conserver le nombre de canaux (feature maps). Il contient ici deux couches de convolution suivie d'une normalisation puis d'une activation ReLU, le tout encapsulé dans une connexion résiduelle.

L'architecture générale empile les bloques de base avec les quelques exceptions suivantes:

- La première couche est une couche de convolution classique avec un kernel carré 7x7, un stride de 2, 64 features de sortie suivie d'une BN, ReLU et d'un maxpooling sur un kernel 3x3 et une stride de 2.
- Le dernier bloque de convolution est suivi d'un global average pooling et d'une couche linéaire (entièrement connectée) produisant la sortie finale.
- Entre les deux, on retrouve quatre bloques qui incluent eux-mêmes le même nombre de bloques, le nombre de features dans chacun de ces blocs est respectivement de 64, 128,256 et 512.
- Les différents bloques sont connectés par un bloque basique légèrement modifié où la première convolution a une stride de 2 mais doubles le nombre de features et la connection résiduelle est remplacée par une convolution avec un kernel unitaire et une stride de 2 qui doulbe le nombre de features, puis une BN.

8 Modèle de Langage et Architecture Transformer

8.1 Encodage de Texte pour le Deep Learning

Le texte est naturellement difficile à traduire en une représentation continue de taille fixe. En effet il possède une dimension, un ordre de lecture et une longueur variable. Il est donc souvent représenté comme une séquence d'objets élémentaires: les tokens (ou jetons).

On nomme ainsi tokenisation l'étape de pré-traitement consistant à transformer le texte en une suite de tokens pour l'essentielle des approches NLP. Les tokens peuvent être des caractères, des ensemble de caractères voire des mots complets. On peut par ailleurs définir des jetons spéciaux pour la fin d'un mot, d'une phrase ou d'un paragraphe.

Il faut néanmoins souligner que la tokenisation à partir de caractères nécessite un nombre important de tokes même pour les textes courts et que les caractères seuls n'ont que peut de signification, l'apprentissage est donc difficile. La tokenisation basée sur les mots permet de résoudre ces problèmes mais se heurte au nombre incalculable de mots courants.

Pour l'encodage, puisqu'il n'est pas logique d'encoder les tokens selon l'ordre alphabétique ou dans un dictionnaire, on utilise typiquement la méthode du one-hot encoding. En générale, on encode les jetons dans un espace de dimension inférieure ce qui est mathématiquement équivalent à un encodage one-hot suivi d'une couche ou opération linéaire, le principe est le suivant:

- On considère une vocabulaire V de taille n qu'on veut encoder dans un espace de plus petite dimension d.
- On commence par représenter le token j par le vecteur e_j où seule la j-ième composante vaut 1.
- On utilise alors la matrice de poids du modèle $M \in \mathcal{M}_{d,n}$ de sorte que $j' = Me_j$.

Un progrès majeur a été réalisé en utilisant une tokenisation intermédiaire entre les caractères et les mots (tokenisation de sous-mots). Une tokenisation de sous-mots populaire est le Byte Payr Encoding (BPE):

- Chaque caractère est commence par être un token.
- On compte le nombre d'occurence de chaque paire de jetons successifs dans le texte.
- On défini un nouveau token pour la paire la plus courante.
- On itère jusqu'à obtenir un nombre défini de tokens.

On utilise typiquement 30 à 40 000 jetons BPE dans les réseaux profonds modernes, ils sont ensuites projetés linéairement dans un espace dont la dimension varie typiquement de 512 à 4096.

8.2 Mécanisme d'Attention

Il s'agit d'une opération fondamentale dans les architectures Transformer.

La couche d'attention est une opération sur les tokens, elle nécessite: N jetons de requête (query tokens) $Q_i \in \mathbb{R}^d$, M jetons de clé (key tokens) $T_i \in \mathbb{R}^d$ et M jetons de valeur (value tokens) associés $V_i \in \mathbb{R}^{d_V}$. Le processus se déroule alors comme suit:

- Calcul des poids d'attention: on compare tout d'abord chaque jeton de requête à tous les jetons clefs pour calculer m poids pour les n jetons de requête, on appelle ces poids attention.
- Calcul du jeton de sortie: on utilise les poids pour calculer une somme pondérée des tokens de valeur.
- Formelement: on pose $K = (T_1, ..., T_M)$ et $V = (V_1, ..., V_M)$ de sorte qu'un token de sorti soit:

$$O_i = \text{softmax}(\frac{Q_i K^T}{\sqrt{d}})V.$$

Pour calculer les ensembles de tokens on peut ou bien tous les calculer comme des projections linéaires des jetons d'entrée, alors $N = M$, les poids des couches linéaires sont les paramètres appris, cette méthode est celle de l'auto-attention (Self-attention). Ou bien on peut utiliser deux ensembles de jetons d'entrée, un pour les jetons de requête et un second pour les jetons de clé et de valeur, c'est l'attention croisée (Cross-attention).

Dans l'attention standard, un jeton de sortie est une moyenne pondérée des jetons de valeur, utilisant des poids identiques pour toutes les dimensions du vecteur de valeur. L'attention multi-têtes (Multi-head attention) permet d'obtenir plusieurs moyennes pondérées différements selon la dimension.

8.3 Attention Is All You Need: Introduction de l'Architecture Transformer

L'architecture que l'on introduit ici est toujours la base de la conception Transformers, développée initialement pour le texte elle s'applique avec succès à de nombreux domaines.

L'architecture globale commence par une couche d'encodage linéaire des jetons à laquelle est associé un encodage positionnel. Les jetons résultants sont l'entrée d'un encodeur Transformer (cette partie est parfois utilisée seule), ils sont alors utilisés par un décodeur Transformer qui prend un ensemble différent de jeton en entrée.

L'objectif est à l'origine de prédire le token suivant dans un tâche de traduction (par exemple du français vers l'anglais):

- Les tokens de la phrase en français sont donnés à l'encodeur.
- Les k premiers jetons de la phrase anglaise sont donnés au décodeur.
- Le réseau est entraîné pour que la sortie du décodeur associée au k-ième jeton soit le $(k+1)$-ième jeton de la phrase anglaise.
- Au moment du test, le réseau décodeur peut être utilisé récurcivement pour prédire l'entièreté de la phrase anglaise.

La formation se fait par classification du jeton suivant avec une variation de l'entropie croisée: l'entropie croisée avec lissage d'étiquette: on décale la cible de l'encodage one-hot vers une valeur proche de 1 et une petite valeur uniforme pour le reste (afin d'éviter des prédictions trop sûres du réseau). Pour des raisons d'efficacité, on utilise une stratégie de masquage dans le décodeur, permettant à chaque jeton de n'extraire des informations que des jetons passés.

Le résultat d'une couche d'attention est indépendant de l'ordre des tokens, cela pose un problème puisque, dans un texte, l'ordre des mots est fondamental. La solution la plus simple et efficace est alors d'ajouter aux tokens une information sur leur position dans la phrase. Empiriquement, une bonne façon de faire est d'utiliser des encodages positionnels sinusoïdaux et cosinusoïdaux. L'encodage positionnel $PE(pos) \in \mathbb{R}^d$ est défini pour un position pos du jeton (premier jeton, deuxième jeton etc) et pour $i \in \{1, ..., \frac{d}{2}\}$ par:

$$PE(pos)_{2i} = \sin(pos/10000^{2i/d}) \quad \text{et} \quad PE(pos)_{2i+1} = \cos(pos/10000^{2i/d})$$

$10000^{2i/d}$ constitue la longueur d'onde des composantes 2i et 2i+1 (les valeurs des premières composantes sont donc fortement dépendantes de pos, puis de moins en moins). Grâce aux identités trigonométriques l'encodage de chaque position $pos + k$ peut être exprimé comme une transformation linéaire de l'encodage de pos: $PE(pos + k) = M_k PE(pos)$. Cela signifie que le modèle n'a pas besoin d'apprendre les position individuelles absolues mais simplement les matrices de passage d'une position à l'autre qui est une connaissance beaucoup plus générales.

L'entrée du modèle est alors composée de l'encodage du token et l'encodage de sa position.

Un encodeur Transformer prend en entrée N jetons et produit N jetons de sortie en appliquant successivement K blocs Transformer de même architecture:

- Auto-Attetion Multi-Têtes: des couches linéaires prédisent les jetons de clé, de requête et de valeur pour chaque jeton d'entrée.
- Connexion Résiduelle et Normalisation de Couche: la sortie est ajoutée au jeton d'entrée de manière résiduelle et on applique une normalisation de couche au résultat.
- Réseau Feed-Forward: les jetons de sortie sont alors traités indépendamment par un réseau feed-forward (MLP à deux couches) avec une autre connexion résiduelle, suivie d'une normalisation de couche.

Le décodeur est similaire à l'encodeur mais il rajoute une couche d'attention croisée dans chaque bloc entre le réseau d'auto-attention et le réseau feed-forward. Les jetons de requête sont calculés à partir des jetons du décodeur, tandis que les clés et les valeurs sont calculées à partir des jetons de sortie de l'encodeur.

Les LLMs sont des exemples d'applications modernes du Transformer.

9 A Retenir Des Quizzs

9.1 Quizz 1

- Comment calculer le nombre de paramètres d'un réseaux de neurones: pour chaque couche on a le calcule suivant: $y = Wx$ avec y la sortie en dimension k, x l'entrée de dimension d et donc W la matrice de paramètres dans $\mathcal{M}_{k,d}(\mathbb{R})$, cette couche a donc $k * d$ paramètres.
  Par exemple pour un réseau en trois couches dont les deux premières couches contiennent respectivement 200 et 100 neurones, pour une entrée de dimension 10 et une sortie de dimension 1, on a:

$$\mathbb{R}^{10} \longrightarrow_{W_1} \mathbb{R}^{200} \longrightarrow_{W_2} \mathbb{R}^{100} \longrightarrow_{W_3} \mathbb{R}^{1} \quad \text{on a donc } 10 \cdot 200 + 200 \cdot 100 + 100 \cdot 1 \quad \text{paramètres.}$$

- Calcul de la mémoire nécessaire pour implémenter le forward/backward pass: pour un réseau dont les paramètres sont stoqués sur M bits et dont les données à chaque couche pour un unique échantillon sont stoquées sur m bits, la mémoire nécessaire pour implémenter le forward pass sur un batch de taille B est

$$M + B \cdot m \cdot \text{nombre de couches.}$$

  Pour le backward pass, il faut également compter les paramètres du gradient qui sont d'environ un par paramètre du réseau donc la mémoire nécessaire est:

$$M + B \cdot m \cdot \text{nombre de couches.}$$