UTC
PROJET TX 2024

A guinea pig for deep learning with many classes

Deseure–Charron Alexis
Calzas Vincent

Référent école
Yves Grandvalet

Table des matières

1 Introduction **1**

2 Etat de l'art **2**
2.1 Classification par des réseaux de neurones à convolution 2
2.1.1 Concept 2
2.1.1.1 Réseau de neurones (Perceptron multicouches) 2
2.1.1.2 Les caractéristiques 3
2.1.1.3 Réseau de neurones convolutif - CNN 3
2.1.2 Exemple du ResNet 4
2.2 L'impact de la régularisation sur l'apprentissage dépend des classes 5

3 La dernière représentation interne du modèle d'apprentissage profond, décisive dans la classification **7**
3.1 Notations 7
3.2 Importance de la dernière couche 7
3.3 Limitations 8
3.4 Évaluation du modèle 8
3.4.1 ImageNet 9
3.4.2 Propagation 9
3.4.3 Sauvegarde des poids 9
3.4.4 Algorithme utilisé 10

4 Apprentissage d'un modèle de régression logistique multinomiale **11**
4.1 Notations 11
4.2 Un modèle de régression logistique multinomiale 11
4.3 Entraînement 12
4.3.1 Backpropagation et descente de gradient 13
4.3.1.1 Généralisation 13
4.3.1.2 Optimisations particulières 14
4.3.2 Régularisation : weight decay 15
4.4 Algorithme pour l'entraînement du modèle 16
4.5 Interprétation des résultats 18

4.5.1 Définition 18
4.5.1.1 Taux de bonne classification 18
4.5.1.2 Fonction de coût 19
4.5.2 Analyse 20

5 Analyse des effets de la régularisation sur la dernière couche du modèle spécifiquement aux classes 26

6 Conclusion 33

7 Remerciements 34

Bibliographie 35

1. Introduction

En Deep Learning, les problèmes de classification peuvent composer plusieurs centaines voire plusieurs milliers de classes. Balestriero a montré en 2022 [1] que les effets de la régularisation et de l'augmentation de données dépendent des classes du problème, ce qui signifie que la valeur optimale de l'hyper-paramètre de régularisation (exemple : le weight decay) peut varier pour chacune des classes. L'hyper-paramètre est alors choisi en faisant un compromis entre les différentes classes en étant ainsi optimal pour aucune classe.

L'objectif de notre projet est alors d'étudier l'impact de l'ajustement de la dernière couche d'un modèle de classification en gelant les poids des couches précédentes déjà pré-entraînés. Pour cela, nous utiliserons un ResNet50 entraîné sur ImageNet afin de récupérer les sorties de l'avant-dernière couche et ainsi, à partir de ces résultats, ajuster l'entraînement d'un modèle de régression logistique correspondant à la dernière couche du modèle initial, pour étudier l'impact des régularisations.

2. Etat de l'art

2.1 Classification par des réseaux de neurones à convolution

2.1.1 Concept

2.1.1.1 Réseau de neurones (Perceptron multicouches)

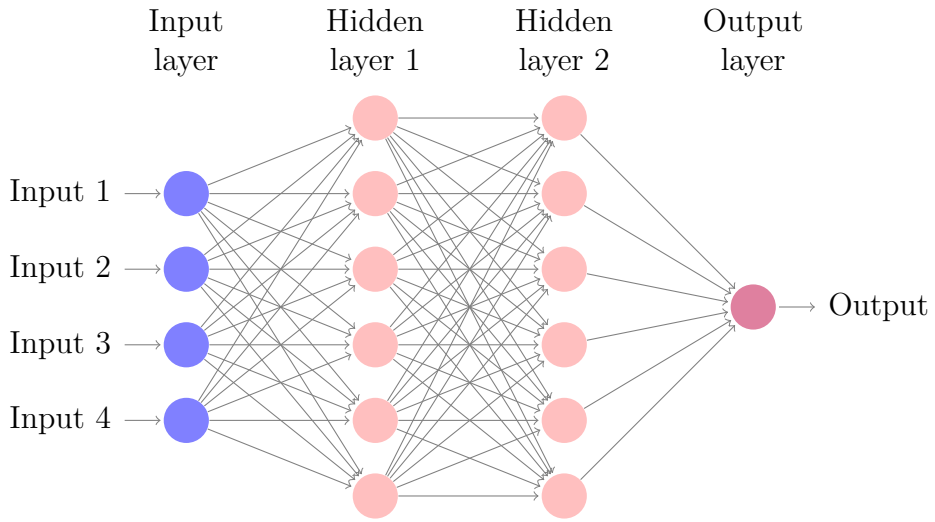


FIGURE 2.1 – Représentation schématique d'un réseau de neurones artificiels

Un réseau de neurones est un système composé d'une multitude de neurones disséminés sur plusieurs couches connectées entre-elles. Pour le problème de classification, le réseau renvoie en sortie une probabilité pour chaque classe en fonction de l'image mise en entrée.

Au sein d'une couche, chaque neurone reçoit en entrée les sorties de la couche précédente. Puis le neurone calcule une combinaison linéaire des données en entrée avec des coefficients appelés poids. Ce résultat passe ensuite dans une fonction d'activation avant d'être délivré en sortie.

La dernière couche délivre les probabilités associés à chaque classe en utilisant une fonction d'activation. Dans notre cas multi-classe : softmax.

Pour obtenir l'erreur de classification, on calcule une fonction de coût (loss). Les poids introduits précédemment sont ajustés par rétropropagation du gradient. En partant de la fin, on

calcule pour chaque couche les poids qui minimisent la fonction de perte.

2.1.1.2 Les caractéristiques

Les caractéristiques (features en anglais) sont des représentations de "zones intéressantes de l'image" comme des points, des zones ou des régions. Pour être plus précis, les caractéristiques sont des motifs qui formalisent les propriétés des éléments caractéristiques d'une classe d'images. Une caractéristique se doit d'être invariante par transformations, unique et non ambiguë et correspondre à une zone de petite taille.

2.1.1.3 Réseau de neurones convolutif - CNN

Les réseaux de neurones de convolution (CNN) sont une sous-catégorie de réseaux de neurones conçue pour traiter, entre-autres, des images en entrée, mais aussi des signaux 1D comme du texte.2.2.

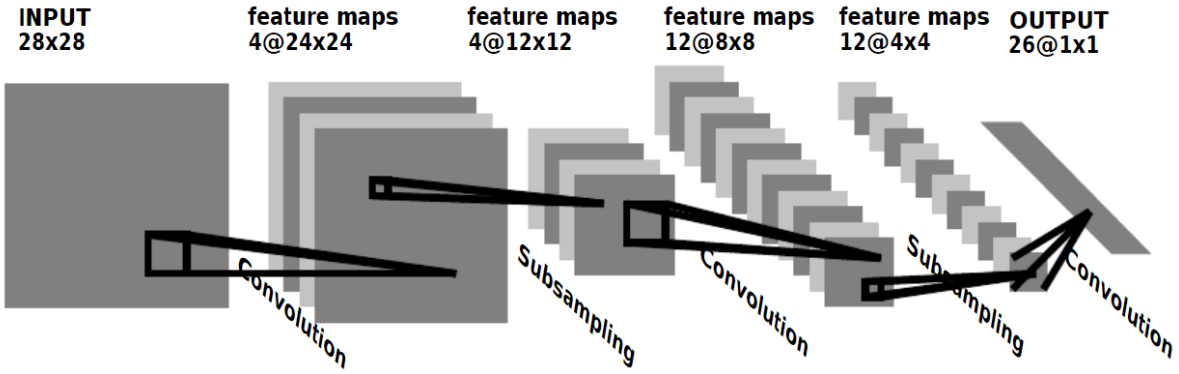


FIGURE 2.2 – Exemple de représentation schématique d'un réseau de neurones convolutif [2]

Les CNN sont formés de 3 types de couches principales :

— les couches de convolution : ce sont les bases d'un CNN, elles permettent d'extraire les caractéristiques d'une image et font l'effet de filtre sur l'image en entrée. Chaque neurone d'une couche de convolution dispose d'un noyau appliqué sur toute l'image et dont les valeurs seront optimisées par l'entraînement. Les hyperparamètres associés à ces couches sont : la profondeur de la couche (c'est à dire le nombre de noyaux de convolution qui produiront chacun une image filtrée), la taille des noyaux, le stride qui correspond au nombre de pixels entre chaque déplacement du noyau et le padding qui rajoute des pixels de valeurs nulles afin de conserver les dimensions des images obtenues

en sortie.

— les couches de pooling : elles consistent à réduire la taille spatiale des images intermédiaires afin de limiter les calculs du modèle tout en conservant l'information. Pour cela, l'image d'entrée est découpée en série de rectangles de même taille auxquels on applique une fonction sur l'ensemble de leurs pixels pour n'en obtenir qu'un seul par rectangle dans l'image finale. Les techniques de pooling les plus souvent utilisées sont le max-pooling et le average-pooling.

— les couches complètement connectées (fully-connected) : ce sont les couches finales d'un CNN. Elles prennent en entrée les réponses aux filtres aplatis en un seul et même vecteur. Ces couches suivent le même principe que celui expliqué en 2.1.1.1 et permettent de décider de la classe prédite par le modèle. Le nombre de neurones de la dernière couche sera donc égal au nombre de classes possible du dataset d'entrée. Pour obtenir une probabilité, la fonction d'activation softmax est souvent privilégiée.

2.1.2 Exemple du ResNet

L'arrivée des CNN dans la résolution des problèmes de classification a été une avancée majeure dans le domaine qui a conduit à d'excellents résultats pour des réseaux de convolution très profonds. Cependant, cela a soulevé la question de l'efficacité de constamment augmenter le nombre de couches de ces réseaux afin de résoudre des problèmes de classification toujours plus complexes en nombre de classes et en proximité visuelle. En effet, pour des réseaux profonds pouvant converger, à mesure que leur profondeur augmente, leur précision est impactée négativement et cela n'est ni expliqué par des problèmes de surapprentissage, ni même par le phénomène d'explosion ou de disparition du gradient [3]. En effet, ceux-ci sont résolus par l'initialisation normalisée [5] et les couches intermédiaires de normalisation [4].

Un moyen de mieux optimiser le problème, et ainsi résoudre les pertes de précision avec la profondeur, a été l'introduction des réseaux neuronaux résiduels dont le principe est d'entraîner la sortie de la couche précédente mais également un résidu d'une couche précédente [3]. Les architectures utilisant cette méthode sont aussi appelées ResNet. Elles utilisent des blocs avec des connexions raccourcies sautant une ou plusieurs couches (figure 2.3).

De manière plus formelle, pour une couche i quelconque qui reçoit un résidu, on a donc

$$a^i = g(W^{i-1,i}a^{i-1} + b^i + W^{i-2,i}a^{i-2}) = g(Z^i + W^{i-2,i}a^{i-2})$$

avec a^i les sorties des neurones de la couche i, g la fonction d'activation pour la couche i, b^i le vecteur de biais associé à la couche et $W^{i-1,i}$ la matrice de poids entre les couches $i-1$ et i.

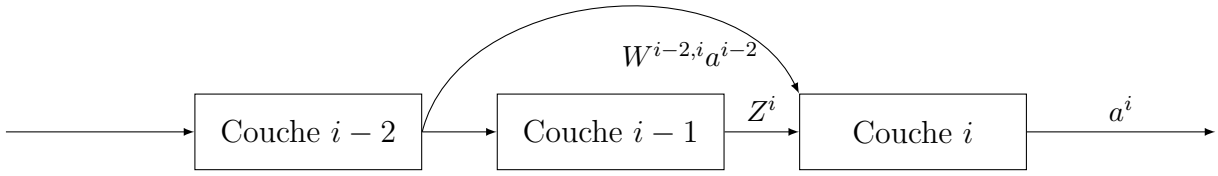


FIGURE 2.3 – Bloc d'un réseau neuronal résiduel

L'architecture originale de ResNet est le ResNet-34, qui comprend 34 couches (figure 2.4). Il a été conçu avec ses frères de 50, 101 et 151 couches pour classifier les images du célèbre jeu de données ImageNet. C'est l'architecture qui remportera le concours 2015 organisé par l'organisation derrière ImageNet.

Au cours de notre projet nous avons utilisé l'implémentation officielle sur pytorch de l'architecture de 50 couches (ResNet-50) qui dispose d'une avant-dernière couche d'average pooling (sorties de taille 2048) et d'une dernière couche complètement connectée vers les 1000 sorties (correspondant aux 1000 classes).

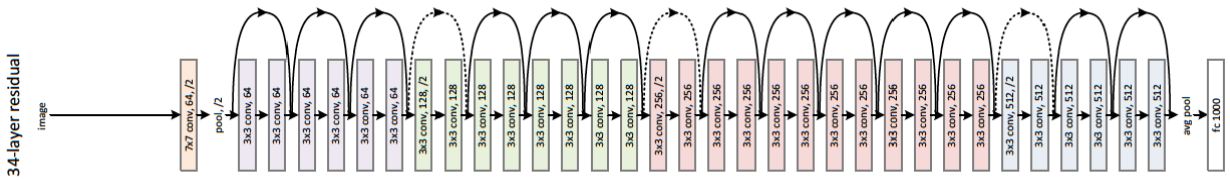


FIGURE 2.4 – Exemple d'architecture pour ImageNet : ResNet comprenant 34 couches [3]

2.2 L'impact de la régularisation sur l'apprentissage dépend des classes

Pour réduire le problème de surapprentissage et contrôler la complexité des modèles utilisés en machine learning ou en statistiques, la régularisation est fréquemment utilisée. Elle consiste à restreindre l'espace de recherche des solutions pendant la phase d'apprentissage pour imposer aux paramètres de rester petits. En deep learning, la régularisation peut être effectuée de plusieurs manières : soit explicitement en pénalisant les poids (traduit par l'utilisation d'un weight decay) ou soit implicitement en utilisant d'autres techniques comme l'augmentation de données.

En deep learning, les problèmes de classification consistent à considérer plusieurs classes auxquelles en fonction d'une entrée quelconque (image, texte) le classifieur attribuera autant de nombres en sortie qu'il y a de classes, chacun compris dans un intervalle, typiquement $[0, 1]$. Cependant, ces problèmes demandent souvent plusieurs centaines ou milliers de classes ce qui augmente considérablement la complexité des modèles.

Balestriero et al. (2022) [1] ont démontré récemment qu'en utilisant une régularisation comme le weight decay, un biais significatif est ajouté dans le modèle entraîné. Malgré l'amélioration globale de la précision du modèle, il y a alors une inégalité entre la précision relative à chacune des classes, c'est à dire que certaines classes voient leur précision diminuer arbitrairement. Le weight decay choisi fait alors le compromis entre toutes les classes mais n'est optimal pour aucune d'entre elles (illustration fidure 2.5).

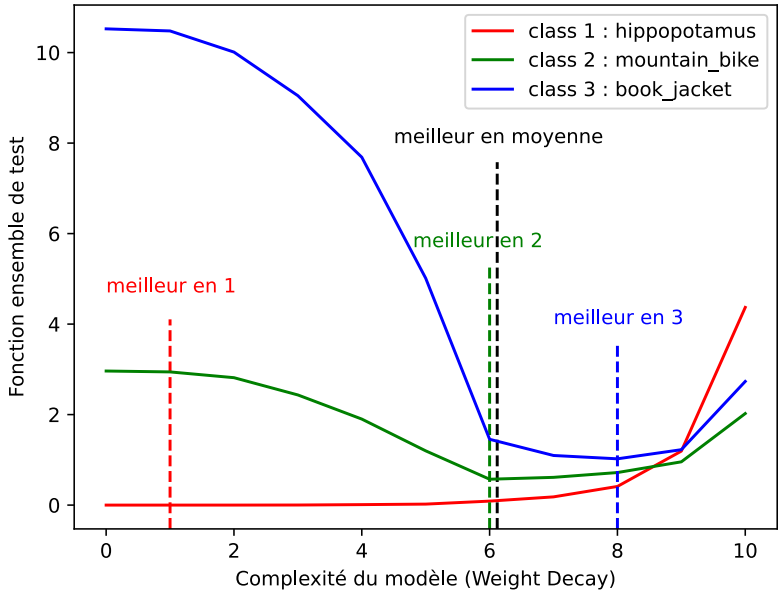


FIGURE 2.5 – Ensemble de 3 classes prisent arbitrairement pour témoigner de la performance de la validation croisée qui est plutôt bonne sur la majorité des classes mais relativement mauvaise sur les autres quand la complexité du modèle est calibrée par une régularisation (ici le weight decay)

3. La dernière représentation interne du modèle d'apprentissage profond, décisive dans la classification

3.1 Notations

Dans la suite de ce rapport nous aurons recours à des conventions de notations spécifiques, introduisons :

— K le nombre de classes possibles identifiables

— n le nombre de couches du réseau de neurones

— $w^{[i]}_{o_1,o_2} \in \mathbb{R}$ la valeur du poids liant le o_2^{eme} neurone de la couche i avec le o_1^{eme} neurone de la couche $i-1$ du réseau

— $z^{[i]}_o$ la valeur que reçoit en entrée le o^{eme} neurone de la couche i : la somme des sorties pondérées de la couche $i-1$, on notera $z^{[0]}_o$ les entrées du réseau de neurones

— $a^{[i]}_o$ la valeur de sortie d'une o^{eme} neurone de la couche i

— $\hat{y}_c$ la prédiction associée au neurone c de la dernière couche, avec $c \in [0, K]$, l'indice d'un neurone de la dernière couche (équivalent à l'indice de la classe)

3.2 Importance de la dernière couche

Dans les réseaux de neurones convolutifs, les couches complètement connectées (section 2.1.1.3) viennent après les couches de convolutions, de pooling et de correction. L'ensemble des couches complètement connectées prend donc en entrée les représentations apprises par les précédentes couches sous forme d'un unique vecteur en 1D. Ce sont ces couches où la décision finale est prise quant à la prédiction du modèle. Pour un problème de classification comme le nôtre, c'est à l'issue de cet ensemble complètement connectées que la classe à laquelle appartient l'image est prédite.

Au cours de notre étude, le modèle utilisé est le ResNet-50 (section 2.1.2) composé de couches

de convolution et de pooling suivies d'une unique dernière couche complètement connectée avec une fonction d'activation softmax permettant ainsi d'estimer la probabilité de la classe de l'image passée en entrée du modèle. Par conséquent, c'est cette couche qui est déterminante dans le résultat final et dans la bonne estimation du modèle. Les couches précédentes permettent d'extraire les caractéristiques de l'image et de les représenter de manière résumée en un seul vecteur 1D.

L'implémentation officielle du ResNet-50 fourni par Pytorch dispose d'une dernière couche de 1000 neurones correspondant au nombre de classes et d'un vecteur d'entrée de taille 2048 pour cette couche finale.

3.3 Limitations

Dans le cadre de notre étude, pour pouvoir entraîner notre modèle profond, il est nécessaire de disposer de puissants GPU pour accélérer au mieux les calculs. Cependant, dans le cas contraire, il est déraisonnable d'un point de vue temporel d'essayer cet entraînement dénué de puissants GPU.

Dans le cadre de notre étude c'est principalement la dernière couche qui nous intéresse (section 3.2) car c'est la prise de décision qu'on souhaite étudier et optimiser. Pour ce faire, deux options sont alors possibles pour utiliser le modèle pré-entraîné pour toutes les couches mis à part les dernières, complètement connectées. La première consiste à empêcher l'apprentissage sur ces couches en bloquant la rétropropagation du gradient.

Cependant, la limite de cette méthode est le coût en calcul de l'évaluation du modèle pendant l'entraînement qui n'est pas négligeable au vu du nombre important de couches du réseau. La deuxième option est celle qu'on a privilégiée, elle consiste à évaluer la sortie de la couche précédant la première couche complètement connectées du modèle pour chacune des images du dataset d'entraînement (correspondant à la sortie de l'avant dernière couche du ResNet-50). À partir de ces sorties, on entraîne un modèle de régression logistique séparément afin de simuler l'effet des couches complètement connectées en évitant les calculs redondants des premières couches pendant l'entraînement.

3.4 Évaluation du modèle

Pour obtenir le dataset composé des sorties de la couche précédant les couches complètement connectées, il est nécessaire d'évaluer notre modèle pour chaque image de l'ensemble d'ImageNet en calculant la propagation du ResNet pré-entraîné.

3.4.1 ImageNet

ImageNet est un projet créé par des chercheurs de l'Université de Stanford et de l'Université de Princeton en 2009 dont l'objectif est de constituer une base de données de plusieurs millions d'images différentes afin de permettre aux chercheurs du monde entier d'y accéder librement pour faire avancer la recherche sur les modèles de deep learning de vision. Pour atteindre cet objectif, les chercheurs ont collecté plus de 14 millions d'images à partir de diverses sources, telles que des moteurs de recherche et des sites Web d'images libres de droits.

Chaque image a été annotée manuellement avec l'une des 21 841 catégories issues de la hiérarchie WordNet, une base de données lexicale qui regroupe les mots en groupes de synonymes cognitifs (synsets) exprimant des concepts distincts. Ces synsets sont interconnectés sémantiquement par un réseau de relations conceptuelles comme la relation d'hyperonymie/hyponymie qui relie les concepts généraux aux plus spécifiques (ex : meuble -> lit -> lit superposé).

Le projet d'ImageNet a aussi donné lieu à la compétition annuelle ILSVRC (2010-2017) qui a permis de jouer un rôle essentiel dans le développement de modèles de réseau de neurones convolutifs (CNN) tels que AlexNet, VGG, et ResNet qu'on a utilisé dans le cadre de notre expérience avec sa version de 50 couches.

3.4.2 Propagation

Pour les couches complètement connectée d'un réseau de neurones (section 2.1.1.1) appelées aussi couches denses, pour obtenir la prédiction du modèle étant donné une entrée $z^{[}0]$, on effectue la propagation en avant caractérisée par l'équation suivante utilisée à chaque neurone o de la couche i du modèle :

$$z_o^{[i]} = \sum_p w_{p,o}^{[i]} a_p^{[i-1]}$$

En sortie du modèle, l'a fonction d'activation utilisée est la fonction softmax pour obtenir une probabilitée, caractérisée par l'expression suivante :

$$\hat{y}_c = \frac{\exp z_c^{[n]}}{\sum_o \exp z_o^{[n]}}$$

Le vecteur $\hat{y}$ ainsi obtenu correspond à l'ensemble des prédictions de chacune des classes sous forme de probabilité lorsqu'on est dans un problème de classification.

3.4.3 Sauvegarde des poids

Dans un premier temps, étant donné que l'entraînement et la simple évaluation de millions d'images dans un réseau de neurones profond comme un ResNet-50 est très coûteuse, le plus

efficace est d'évaluer les sorties de taille n (dans notre cas, $n = 2048$) obtenues à l'avant-dernière couche du réseau pour chacune des images de l'ensemble de données d'entraînement (dans notre cas, ImageNet), et de sauvegarder ces résultats dans un dataset qui servira par la suite à entraîner notre modèle de régression logistique. (cf. section 3.3)

En effet, étant donné que l'objectif est d'étudier l'évolution de la fonction de coût pour chacune des classes au cours de l'apprentissage, il est nécessaire de sauvegarder les modèles à différentes époques de l'apprentissage et surtout pour différents hyperparamètres de régularisation.

3.4.4 Algorithme utilisé

Pour implémenter la sauvegarde des sorties de l'avant-dernière couche de notre modèle, nous avons utilisé la librairie "pytorch", voici un exemple simplifié, extrait du code source utilisé :

```
# Instanciation du modèle ResNet-50 avec les poids par défaut
model = resnet50(weights=ResNet50_Weights.DEFAULT)

# Définition du dictionnaire de stockage des activations
activation = {}
def getActivation(name):
  '''
  Fonction pour le forward hook afin de récupérer
  les sorties de l'avant dernière couche.
  '''
  def hook(model, input, output):
    activation[name] = output.detach()
  return hook

h = model.avgpool.register_forward_hook(getActivation('avgpool'))
dataloader = DataLoader(
    dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    pin_memory=pin_memory
)
# Début de la propagation en avant sans calculer les gradients
model.eval()
with torch.no_grad():
    # Itérations sur les batchs fournis par le dataloader
    for images, labels in tqdm(dataloader, desc="Evaluation"):

        outputs = model(images) # Calcul des prédictions faites par le modèle

        # Récupération des activations de l'avant-dernière couche (average pooling)
        avgpool_list = torch.cat([avgpool_list, activation['avgpool'].squeeze()],dim=0)

torch.save(avgpool_list, f'{saving_folder}.pt')
```

4. Apprentissage d'un modèle de régression logistique multinomiale

4.1 Notations

Nous reprendrons les notations introduites en section 3.1.

4.2 Un modèle de régression logistique multinomiale

Dans le domaine de l'apprentissage automatique (Machine Learning), on distingue généralement deux grands types d'algorithmes : l'apprentissage supervisé et l'apprentissage non supervisé. L'objectif est de déterminer une fonction f qui modélise au mieux la relation entre les données d'entrée et les données de sortie, de sorte que :

$$sortie = f(entrée)$$

Dans le cas de l'apprentissage supervisé, les données d'entraînement comprennent les étiquettes (labels) correspondant aux sorties désirées. Les problèmes d'apprentissage supervisé peuvent être divisés en deux catégories principales : la régression et la classification.

La régression est utilisée lorsque la variable cible est continue, tandis que la classification est employée lorsque la variable cible est discrète, c'est-à-dire qu'elle prend un nombre fini de valeurs distinctes (modalités). Dans le cas présent, les sorties étant discrètes, un modèle de classification est approprié.

La régression logistique est un modèle de classification souvent utilisé pour modéliser la probabilité d'appartenance à chaque classe. Elle est particulièrement adaptée aux problèmes binaires (deux classes), mais peut être généralisée à des problèmes multiclasses (plusieurs modalités) sous la forme de la régression logistique multinomiale que nous utiliserons pour simuler la dernière couche du ResNet-50.

Soit $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ le vecteur d'entrée de taille N. La couche linéaire applique une transformation affine à ce vecteur d'entrée pour produire un vecteur de scores non normalisés $\mathbf{z} = (z_1, z_2, \ldots, z_K)$ de taille K, correspondant aux K classes. Cette transformation est réalisée à l'aide d'une matrice de poids $\mathbf{W}$ de taille $N \times K$ et d'un vecteur de biais $\mathbf{b}$ de taille K, selon l'équation suivante :

$$\mathbf{z} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$$

La fonction d'activation softmax est ensuite appliquée au vecteur de scores non normalisés $\mathbf{z}$ pour obtenir un vecteur de probabilités $\hat{\mathbf{y}}$ de taille K, où chaque élément $\hat{y}_i$ représente la probabilité estimée que l'entrée $\mathbf{x}$ appartienne à la classe i. La fonction softmax est définie comme suit :

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Lors de l'entraînement du modèle, la fonction de coût de l'entropie croisée est minimisée pour ajuster les paramètres $\mathbf{W}$ et $\mathbf{b}$. L'entropie croisée $\mathcal{L}$ entre les probabilités prédites $\hat{\mathbf{y}}$ et les étiquettes réelles $\mathbf{y}$ (sous forme de vecteur one-hot de taille K - vecteurs composés de une seule valeur égale à 1, les autres nulles) est définie comme :

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{K} y_j \log \hat{y}_j$$

On cherchera ainsi à minimiser cette fonction par la suite.

4.3 Entraînement

À partir des sorties de l'avant-dernière couche du modèle initial (ResNet-50), on constitue un nouvel ensemble de données (voir section 3.4.3) composé d'autant de vecteurs qu'il y avait d'images initialement. Par conséquent, on défini un modèle de régression logistique multinomial composé d'une seule couche de sorties complètement connectée (explications section 3.3) qu'on entraîne avec le nouveau dataset des vecteurs de l'avant-dernière couche obtenu. L'ajustement des valeurs des poids permet ensuite de minimiser la fonction de perte et ainsi obtenir un taux de bonne classification optimal.

L'entraînement d'un tel modèle consiste à trouver la valeur des poids permettant d'avoir une fonction de coûts minimale. Dans notre cas, on utilisera l'entropie croisée $\mathcal{L}$ définit pour le vecteur $\mathbf{y}$ correspondant au label de taille K associée au vecteur d'entrée $\mathbf{x}$ et $\hat{\mathbf{y}}$ la prédiction obtenue par le modèle (section 4.2) :

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{K} y_j \log \hat{y}_j$$

avec y_j et $\hat{y}_j$ les valeurs d'indice j des vecteurs $\mathbf{y}$ et $\hat{\mathbf{y}}$ respectivement.

Ensuite, afin de limiter le surapprentissage mais également dans le cadre de notre projet, nous rajoutons un weight decay en utilisant la régularisation L2 (ie. section 4.3.2) afin d'obtenir la fonction de coût $L2$ que l'algorithme minimisera :

$$L2(\mathbf{y}, \hat{\mathbf{y}}) = \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

avec $\lambda \geq 0$ l'hyperparamètre de régularisation, et $\mathbf{w}$ le vecteur de poids de l'unique couche du réseau.

L'intérêt de cette méthode est d'étudier l'impact de la technique de régularisation utilisant un weight decay sur l'apprentissage de notre modèle pour différentes classes et pour différentes valeurs de cet hyperparamètre.

Au cours de l'entraînement, il est également intéressant de recueillir les valeurs des erreurs et de la fonction optimisée à chaque époque afin de détecter de potentiels phénomènes de surapprentissage ou de sous-apprentissage.

4.3.1 Backpropagation et descente de gradient

4.3.1.1 Généralisation

Le processus d'entraînement et d'ajustement des poids d'un réseau de neurones à propagation avant (légèrement différent pour un réseau neuronal résiduel) est caractérisé d'abord par l'algorithme de backpropagation qui calcule les gradients de la fonction de coût en utilisant la règle de dérivation en chaîne. Pour cela, l'algorithme propage l'erreur en arrière et calcule le gradient de la fonction de coût. Les explications mathématiques du processus sont décrites ci-dessous.

Soit $\mathcal{L}$ la fonction de coût. On a pour la dernière couche :

$$\frac{\partial \mathcal{L}}{\partial z_o^{[n]}} = \sum_c \frac{\partial \mathcal{L}}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial z_o^{[n]}} = \hat{y}_o - y_o$$

car

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_c} = -\frac{y_c}{\hat{y}_c} \qquad \frac{\partial \hat{y}_c}{\partial z_o^{[n]}} = \begin{cases} \frac{e^{z_c^{[n]}}}{\sum_o e^{z_o^{[n]}}} - \frac{e^{z_c^{[n]}} e^{z_c^{[n]}}}{(\sum_o e^{z_o^{[n]}})^2} = \hat{y}_c(1 - \hat{y}_c) & \text{si } c = o \\ 0 - \frac{e^{z_c^{[n]}} e^{z_c^{[n]}}}{(\sum_o e^{z_o^{[n]}})^2} = -\hat{y}_c \hat{y}_o & \text{si } c \neq o \end{cases}$$

donc,

$$\frac{\partial \mathcal{L}}{\partial w_{p,o}^{[n]}} = \frac{\partial \mathcal{L}}{\partial z_o^{[n]}} \frac{\partial z_o^{[n]}}{\partial w_{p,o}^{[n]}} = (\hat{y}_o - y_o) a_p^{[n-1]}$$

On a pour la i^{eme} couche :

$$\frac{\partial \mathcal{L}}{\partial w_{q,p}^{[i]}} = \sum_o \frac{\partial \mathcal{L}}{\partial z_o^{[i+1]}} \frac{\partial z_o^{[i+1]}}{\partial a_p^{[i]}} \frac{\partial a_p^{[i]}}{\partial z_p^{[i]}} \frac{\partial z_p^{[i]}}{\partial w_{q,p}^{[i]}} = \sum_o (\hat{y}_o - y_o) w_{p,o}^{[i+1]} a_p^{[i]} (1 - a_p^{[i]}) a_q^{[i-1]}$$

A l'issu de l'algorithme de backpropagation, après avoir calculé le gradient, on utilise l'algorithme d'optimisation de descente de gradient pour trouver les nouveaux poids qui minimiseront la fonction de coût (en réalité on utilise la descente de gradient stochastique par minibatch (SGD) qui, au lieu de parcourir tous les échantillons de l'ensemble d'apprentissage pour effectuer une seule mise à jour des paramètres, utilise uniquement un sous-ensemble ou batch afin de réduire considérablement le temps de calcul tout en conservant des performances acceptables). La formule qui caractérise la descente de gradient est la suivante,

$$w_{o,p}^{[i]} = w_{o,p}^{[i]} - \alpha \frac{\partial \mathcal{L}}{\partial w_{o,p}^{[i]}}$$

avec α le pas de gradient (learning rate) tel que $\alpha \geq 0$.

4.3.1.2 Optimisations particulières

La première optimisation qu'on utilise communément est l'usage d'un step scheduler pour le learning rate qui va permettre de réduire la valeur du learning rate d'un facteur à chaque intervalle d'époques de l'apprentissage afin d'améliorer la stabilité et la précision du modèle en le faisant converger plus en douceur. Grâce à cette méthode, on permet au modèle d'apprendre suffisamment rapidement pendant les premières époques de l'apprentissage puis en réduisant l'amplitude des mises à jour à mesure que le modèle converge, cela permet d'améliorer encore plus précisément les poids au lieu d'osciller autour de l'optimum.

Une autre optimisation intéressante pour l'apprentissage est l'usage de la descente de gradient avec un "momentum". L'objectif de cette technique est d'accélérer le processus d'apprentissage pour atteindre le point de convergence. En effet, le problème avec l'algorithme de la descente de gradient est que sur les pentes douces, le gradient devient très faible et les mises à jour deviennent lentes. L'utilisation d'un momentum consiste à utiliser la mise à jour précédente d'un poids multipliée par un nouvel hyperparamètre auquel on ajoute le terme calculé avec la backpropagation. Ce comportement ce traduit ainsi par les équations :

$$(b_{o,p}^{[i]})_t = \mu \cdot (b_{o,p}^{[i]})_{t-1} + \frac{\partial \mathcal{L}}{\partial w_{o,p}^{[i]}}$$

$$w_{o,p}^{[i]} = w_{o,p}^{[i]} - \alpha(b_{o,p}^{[i]})_t$$

avec μ le nouvel hyperparamètre associé au momentum et $(b_{o,p}^{[i]})_t$ le terme modifié de direction pour le poids $w_{o,p}^{[i]}$ à une période $t \in \mathbb{N}$ donnée de l'apprentissage.

4.3.2 Régularisation : weight decay

Le weight decay est une méthode de régularisation qui est utilisée pour pénaliser la fonction de coût pour réduire l'amplitude des poids et les empêcher de devenir trop importants. Cela permet de limiter le surapprentissage. En effet, des poids élevés provoquent une variance élevée en sortie. Pour cela, la fonction de coût est pénalisée. Il se décline de deux façons :

1. L1 Regularisation :

   Combat le sur-apprentissage en réduisant les poids de certaines caractéristiques à 0. Cela revient à sélectionner certaines caractéristiques et en supprimer d'autres.

   $$L1\ Regularisation = (fonction\ de\ perte) + \lambda \sum_{i=1}^{n} \sum_{p=1}^{m_i} \sum_{o=1}^{m_{i-1}} |w_{o,p}^{[i]}|$$

   avec n le nombre de couche, m_i le nombre de neurone de la couche i et $\lambda \geq 0$

2. L2 Regularisation :

   Combat le sur-apprentissage en réduisant les poids de certains caractéristiques mais les poids restent supérieur à 0. Ainsi les caractéristiques moins importantes auront toujours une une petite influence sur la prédiction.

   $$L2\ Regularisation = (fonction\ de\ perte) + \lambda \sum_{i=1}^{n} \|\mathbf{w}^{[i]}\|^2$$

   avec $\lambda \geq 0$ et $\mathbf{w}^{[i]}$ le vecteur de poids de la couche i du modèle.

3. Comparaison des 2 méthodes :

   — La L1 Regularisation renvoie une solution clairsemée alors que la L2 Regularisation une solution dense.
   — La L1 Regularisation offre une solution plus robuste alors que L2 Regularisation offre une solution moins coûteuse en calcul.

   Pour notre problème nous opterons pour la régularisation L2 implémentée nativement de cette manière avec l'optimiseur SGD de pytorch.

4.4 Algorithme pour l'entraînement du modèle

Pour implémenter l'apprentissage et l'instanciation du modèle de régression logistique, toujours avec pytorch, nous avons définit une classe personnalisée, voici un exemple simplifié, extrait et adapté du code source réellement utilisé :

```
# Initialisation d'une classe personnalisée pour avoir un dataset optimisé sur les
# sorties de l'avant dernière couche du ResNet-50 sur ImageNet (voir code source)
dataset = PenultimateOutputsDataset(outputs_folder)
train_loader = DataLoader(dataset, batch_size=batch_size, pin_memory=True)

torch.manual_seed(0) # Reproductibilité de l'expérience dans l'initialisation aléatoire des poids

class LogisticRegression(nn.Module):
    '''
    Implémentation du modèle de régression logistique avec torch
    '''
    def __init__(self, input_size, num_classes):
        super(LogisticRegression, self).__init__()
        # Définition de la couche linéaire
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        # Définition de la propagation en avant
        out = self.linear(x)
        return out

# Itérations sur les différents weights decay
for weight_decay_parameter in weight_decay_parameters :

    # Définition du modèle
    model = LogisticRegression(input_size=input_size, num_classes=num_classes)

    # Définition de la fonction de loss et de l'optimiseur
    loss_function = nn.CrossEntropyLoss() # softmax intégrée à la fonction
    optimizer = torch.optim.SGD(
        model.parameters(),
        lr=learning_rate,
        momentum=momentum,
        weight_decay=weight_decay_parameter
    )

    # Définition du scheduler pour faire varier le learning rate
    scheduler = torch.optim.lr_scheduler.StepLR(
        optimizer,
        step_size=lr_decay_step,
        gamma=decay_rate
    )
```

```
# Initialisation des différentes variables pour stocker les éléments intéressants
accuracy_history = []
L2_regularisation_history = torch.tensor([])
mean_loss_class = torch.tensor([])
weights_epoch = torch.tensor([])

# Obtention du nombre d'images par classe
nb_image_class = torch.bincount(dataset.labels).to(DEVICE)

for epoch in range(num_epochs):

    # Initialisation du nombre de prédictions correctes
    correct_predictions = 0

    # Initialisation du vecteur de la somme des valeurs de la fonction de coût pour chaque classe
    class_loss = torch.zeros(1000)

    for inputs, labels in train_loader:

        # Forward pass
        outputs = model(inputs)
        loss = loss_function(outputs, labels)
        L2_regularisation = loss + weight_decay_parameter * \
        sum(t.pow(2).sum() for t in model.state_dict().values()) / 2

        # Calcul de la fonction de coût pour chaque image et ajout dans class_loss
        input_loss = nn.functional.cross_entropy(outputs,labels,reduction='none') + \
        weight_decay_parameter * sum(t.pow(2).sum() for t in model.state_dict().values()) / 2
        # On obtient ainsi la valeur de la fonction de coût pour chaque classe
        class_loss.index_add_(0,labels,input_loss)

        # Ajout des prédictions correctes au total
        _, predicted = torch.max(outputs, 1)
        correct_predictions += (predicted == labels).sum().item()

        # Backpropagation et optimisation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Modification du lerning rate
    scheduler.step()

    # Sauvegarde du modèle intermédiaire
    # (models_path normalement défini et modifié pour chaque weight decay)
    torch.save(model.state_dict(), f'{models_path}/epoch-{epoch}.pt')

    # Calcul du taux de bonne classification
```

```
        accuracy = 100 * correct_predictions / len(dataset)

        # Sauvegarde des différentes mesures
        accuracy_history.append(accuracy)
        L2_regularisation_history = torch.cat(
            (L2_regularisation_history, L2_regularisation.unsqueeze(0)),
            dim=0
        )
        mean_loss_class = torch.cat(
            (mean_loss_class,(class_loss/nb_image_class).detach().unsqueeze(0)),
            dim=0
        )
        weights_epoch = torch.cat(
            (weights_epoch, sum(t.abs().sum() for t in model.state_dict().values()).unsqueeze(0)),
            dim=0
        )

    # Sauvegarde de toutes les données importantes liées à l'entraînement du modèle
    # (data_path normalement défini et modifié pour chaque weight decay)
    torch.save(accuracy_history, f'{data_path}/accuracy.pt')
    torch.save(loss_history, f'{data_path}/loss.pt')
    torch.save(L2_regularisation_history, f'{data_path}/L2_regularisation.pt')
    torch.save(mean_loss_class, f'{data_path}/mean_loss_class.pt')
    torch.save(weights_epoch, f'{data_path}/weights.pt')
```

4.5 Interprétation des résultats

Pour visualiser les différences sur les prédictions selon le choix du paramètre du weigh decay (λ), on trace la fonction de coût et le taux de bonne classification en fonction des époques du modèle.

4.5.1 Définition

4.5.1.1 Taux de bonne classification

Le taux de bonne classification est un facteur déterminant dans le choix du modèle. En effet, plus le taux de bonne classification est grande : meilleur est notre modèle. Pour chaque modèle, elle se calcule de cette facon :

$$Taux\ de\ bonne\ classification = \frac{1}{n_d} \sum_{i=1}^{n_d} \mathbb{1}_{y_i = \hat{y}_i}$$

avec $n_d \in \mathbb{N}$ le nombre d'images.

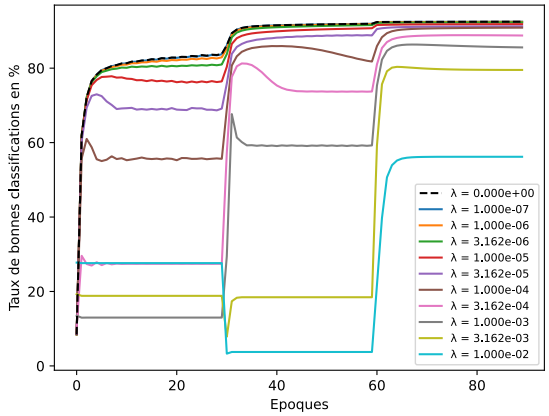
4.5.1.2 Fonction de coût

La fonction de coût est aussi un facteur déterminant dans le choix du modèle. De plus, elle est utilisée dans la descente de gradient (Voir section 4.3.1) et dans l'apprentissage de notre modèle. Pour chaque modèle, elle se calcule de cette façon :

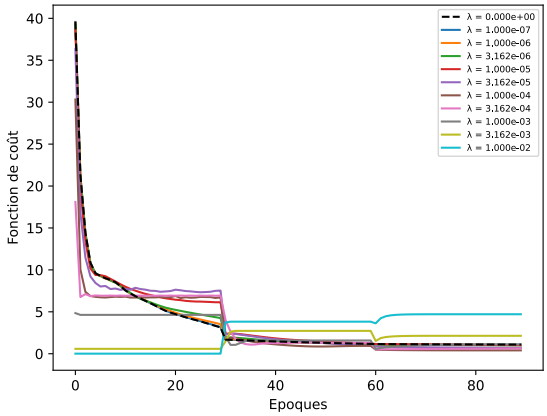
$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{K} y_j \log \hat{y}_j$$

avec y_j et $\hat{y}_j$ les valeurs d'indice j des vecteurs $\mathbf{y}$ et $\hat{\mathbf{y}}$ respectivement.

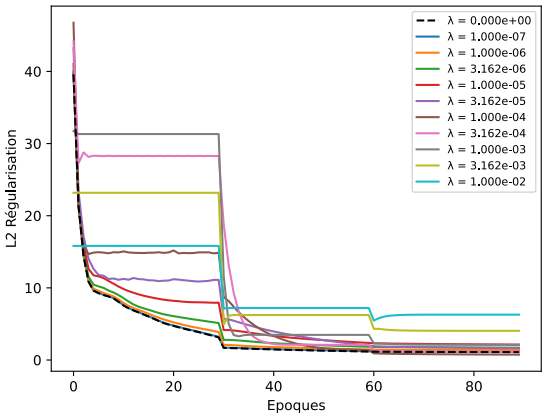
4.5.2 Analyse



(a) Taux de bonne classification en fonction des époques



(b) Fonction de coût en fonction des époques



(c) L2 Regularisation en fonction des époques

FIGURE 4.1 – Évolution des résultats pour l'ensemble d'entraînement sur les modèles obtenus à l'issu de chaque époque de l'entraînement

En premier lieu, la figure 4.1(c) témoigne du bon déroulement de l'apprentissage. En effet, la régularisation L2, tracée en ordonnée, est la fonction minimisée pendant le processus d'entraînement qui va décroître au cours de l'apprentissage. Les paliers témoignent de la variation brutale du pas de gradient qui permet à l'algorithme de sortir d'un potentiel état oscillant autour de l'optimum pour ainsi diminuer directement la valeur de l'erreur. Les valeurs λ de

weight decay les plus élevés témoignent d'une importante pénalisation qu'on retrouve dans nos expériences avec l'écart entre les valeurs de la fonction de régularisation. Cette pénalisation se traduit par des performances réduites au cours de l'apprentissage pour éviter au modèle d'être ajusté spécifiquement pour l'ensemble d'entraînement. En comparant la figure 4.1(c) à la figure 4.1(b), on en déduit la présence du terme de régularisation qui augmente les valeurs à la dernière époque. Cependant, contrairement à la figure 4.1(c) les courbes de la figure 4.1(b) ne sont pas décroissantes pour l'ensemble des modèles. En effet, on observe pour les valeurs de pénalisation les plus élevées, un comportement particulier de l'évolution de la fonction de coût que nous expliqueront ensuite.

En vue des résultats observés au cours de l'apprentissage, même si le taux de bonne classification et la fonction de coût semblent être les plus optimaux sur l'ensemble d'entraînement pour un $\lambda = 0$ (figure 4.1), ce n'est plus le cas lorsqu'on évalue l'ensemble de test sur nos modèles pré-entraînés et sauvegardés à chaque époque de l'apprentissage (figure 4.2). Avec un $\lambda = 0$, notre modèle a sur-appris car aucune pénalisation ne lui a été appliquée au cours de son entraînement ce qui a conduit a l'ajustement spécifique de ses poids uniquement sur l'ensemble des données d'entraînement. Par conséquent, cet apprentissage "par cœur", trop spécialisé à un même ensemble de données pendant l'entraînement se caractérise par de mauvaises performances sur un autre ensemble de données comme notre ensemble de test. Avec les modèles sauvegardés qui correspondent à la dernière époque, la valeur de la fonction de coût pour le modèle ayant eu un $\lambda = 0$ se retrouve alors maximale par rapport à celle des autres modèles (ayant d'autres valeurs de λ) tandis que le taux d'erreur de classification se situe uniquement devant les modèles entraînés avec $\lambda = 1 \cdot 10^{-2}$ et $\lambda = 3 \cdot 10^{-3}$ (figure 4.2)

Par conséquent, pour déterminer le λ optimal, on s'intéresse surtout à l'ensemble de test qui correspond à la véritable performance d'un modèle sur des données quelconques. Par observations des résultats numériques mais aussi des représentations graphiques de la figure 4.2, on peut considérer que le meilleur λ se situe dans l'intervalle $[1 \cdot 10^{-4}, 1 \cdot 10^{-3}]$.

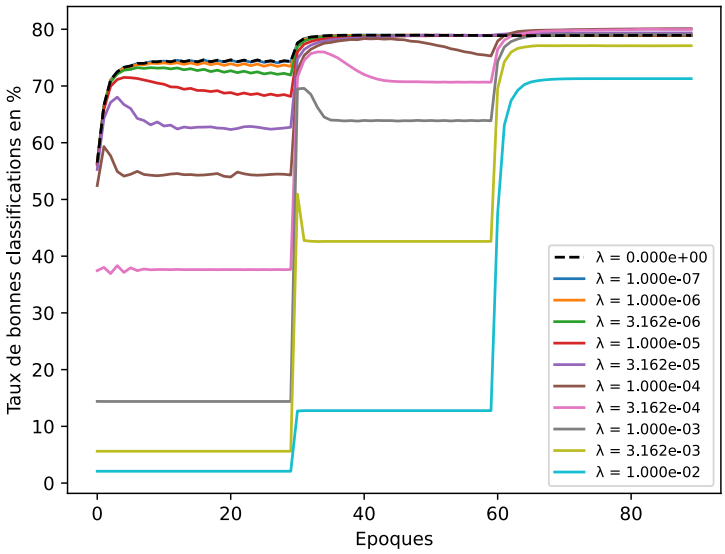
Pour mieux se représenter les valeurs de λ et la performance des modèles associés pour chacune des classes, nous analyserons différentes représentations dans la section 5.

En s'intéressant aux modèles pour des valeurs de λ en dehors de l'intervalle $[1 \cdot 10^{-4}, 1 \cdot 10^{-3}]$, on remarque bien que :

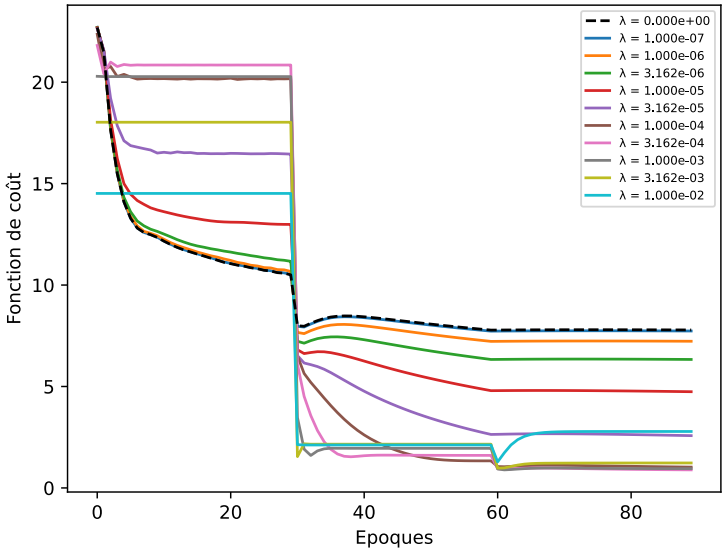
— Pour des modèles avec un $\lambda < 1.10^{-4}$, le taux de bonnes classifications et la fonction de coût (supérieur ou égale à 80% et inférieure ou égale à 3 respectivement) semblent performants sur l'ensemble d'entraînement mais subissent de la même manière que pour un $\lambda = 0$, le phénomène de surapprentissage sur l'ensemble de test qui se traduit par un taux de bonnes classifications qui ne sont plus les meilleurs et une valeur de la fonction

de coût bien au dessus de ce qu'on retrouvait sur l'ensemble d'apprentissage (fonction de coût supérieure ou égale à 5%). Mis à part le modèle ayant un λ nulle, donc aucune pénalité, les autres modèles avec un $\lambda < 1.10^{-4}$ en ont une qui est trop faible pour suffisamment pénaliser l'optimisation et éviter le surapprentissage.

— Pour des modèles avec un $\lambda > 1.10^{-3}$, le taux de bonne classification et la fonction de coût (inférieur strictement de 80% et supérieur ou égale à 3 respectivement) performent évidemment peu au cours l'apprentissage par rapport aux modèles ayant des valeurs de λ inférieurs. En effet, la pénalité étant largement plus importante, l'optimisation des poids se retrouve fortement impactée. Cela se traduit par des valeurs de la fonction de coût croissantes, ou croissantes par moments, car l'algorithme d'optimisation se retrouve contraint de diminuer la norme des poids qui ont un impact bien plus important dans la fonction optimisée (régularisation L2) que n'a la fonction de coût à elle seule. Par conséquent, même sur l'ensemble de test (figure 4.2), le taux de bonne classification reste peu compétitif (toutefois supérieur) bien que la fonction de coût est diminuée (fonction de coût strictement inférieure à 3% mais reste toutefois pas optimale). Ces modèles sont alors dans une situation de sous apprentissage où la pénalisation trop importante a trop contraint le modèle l'obligeant à perdre de l'information importante.



(a) Taux de bonne classification en fonction des époques



(b) Fonction de coût en fonction des époques

FIGURE 4.2 – Évolution des résultats pour l'ensemble de test sur les modèles obtenus à l'issu de chaque époque de l'entraînement

Comme expliqué précédemment, les paliers observables pour chacune des courbes des figures

4.1 et 4.2 toutes les 30 époques sont dû à l'usage d'un "scheduler" qui permet de changer le pas de gradient au cours de l'apprentissage. Ce changement permet d'améliorer la stabilité et le taux de bonne classification du modèle en le faisant converger plus en douceur. Ces paliers peuvent s'expliquer par le nombre relativement important d'époques pour chaque valeur de pas de gradient (30 époques), ce qui conduit rapidement à une stagnation (oscillation) autour d'un optimum à cause de la valeur trop importante du pas. Ainsi, le changement direct du pas de gradient d'un facteur 10 permet de débloquer la situation dès la première époque (plus de 1 287 000 entrées) et ainsi améliorer et préciser significativement l'optimisation.

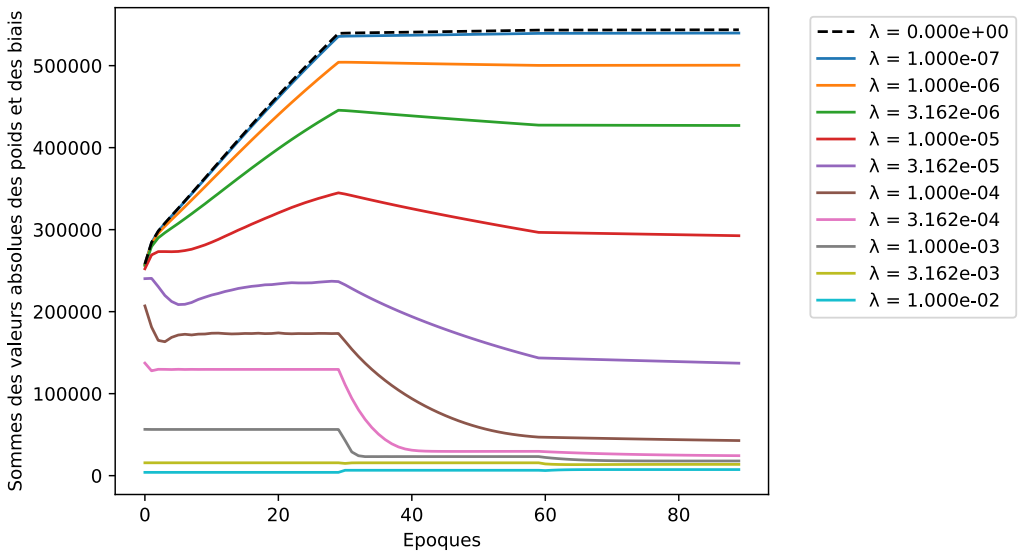


FIGURE 4.3 – Évolution de la somme des normes des paramètres (poids et biais) du modèle en fonction des époques. L'initialisation est faîtes selon une loi uniforme avec une graine de génération définie à l'avance pour la reproductibilité de l'expérience. Les valeurs à l'époques 0 correspondent aux valeurs des poids après la première époque (ce ne sont pas les valeurs d'initialisation).

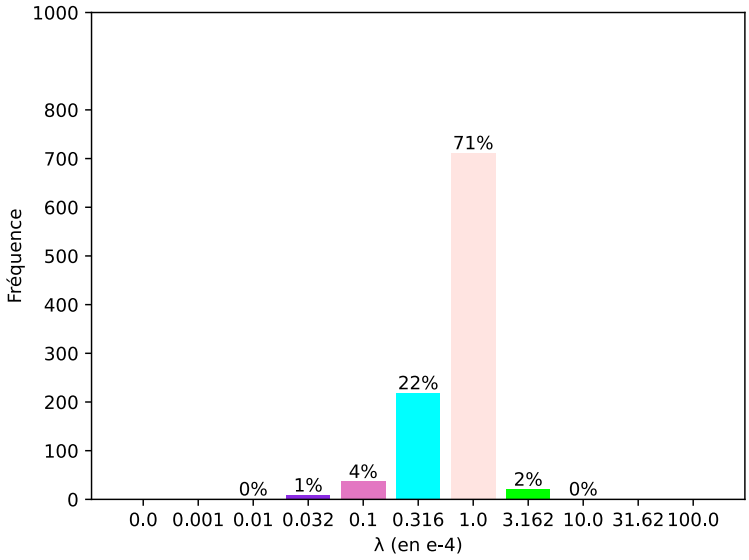
Pour bien visualiser l'effet de la régularisation qui applique des valeurs de pénalités proportionnelles aux poids, on peut tracer l'évolution de la somme des normes de chacun des poids et biais des différents modèles (pour les différentes valeurs de λ). La représentation de la figure 4.3 nous donne un indicateur intéressant sur le comportement de l'apprentissage vis à vis de la régularisation. En effet, pour les valeurs de λ les plus faibles ($\lambda \leq 1 \cdot 10^{-5}$), la somme des valeurs absolues des poids explose car la valeur de l'hyperparamètre est trop faible pour limiter cette croissance. La norme des poids étant un indicateur de complexité et de stabilité du modèle, une explosion de leurs valeurs témoigne d'un modèle en situation de surappren-

tissage dont la représentation devient trop complexe et spécifique à classifier uniquement les entrées issues de l'ensemble d'entraînement. A l'inverse pour les valeurs de l'hyperparamètre λ les plus importantes, la somme des poids devient très inférieur aux exemples précédemment cités ($\lambda \leq 1 \cdot 10^{-5}$), cela témoigne de la pénalité importante appliquée au cours de l'apprentissage qui force l'algorithme d'optimisation à réduire la norme des poids car ils occupent une part plus importante de la fonction optimisée (régularisation L2). Cette situation est donc relative à un sous apprentissage du modèle car ce dernier n'est pas assez complexe pour classifier de manières optimales les données. Ce phénomène est explicite sur la figure 4.1(b) où pour $\lambda = 1 \cdot 10^{-2}$, la valeur de la fonction de coût croit en fonction des époques au lieux de diminuer. Cela s'explique en partie par le fait que dans la fonction optimisée qui est le L2 régularisation, le λ trop important fait que diminuer les poids à un impacte plus important que simplement essayer de trouver les valeurs optimales pour minimiser la fonction de coût.

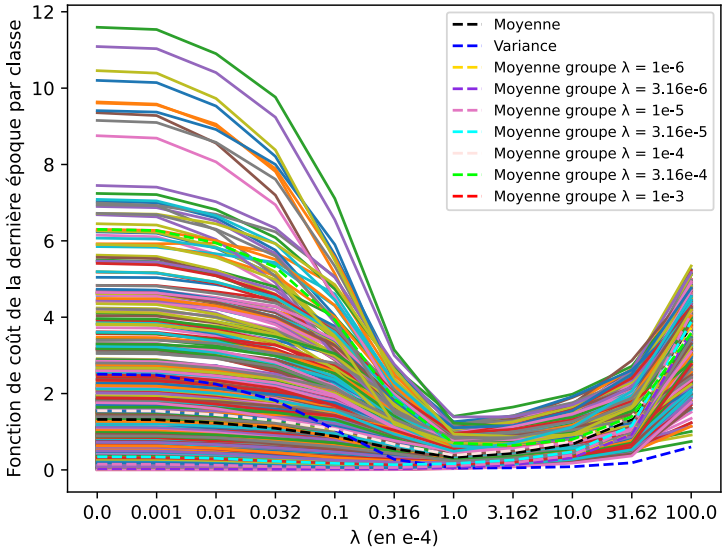
5. Analyse des effets de la régularisation sur la dernière couche du modèle spécifiquement aux classes

Après avoir analyser l'évolution de l'apprentissage au cours de l'entraînement, et de l'impact de la régularisation au cours de cette phase, nous nous intéresserons dans cette partie à l'impacte de la régularisation pour les classes comme éléments distincts en fixant cette fois si nos expérimentations aux résultats obtenus à l'issue de la dernière époque de l'apprentissage.

Comme nous l'avions introduit avec la figure 2.5, en traçant les valeurs moyennes de la fonction de coût à l'issue de la dernière époque pour chaque classe et chaque hyper-paramètre de régularisation (weight decay), on se rend compte que ces classes ne sont pas égales quant à leur apprentissage face à la régularisation qui leur est imposée. Par exemple, sur cette dernière figure on remarque que la classe "hippopotamus" possède une valeur optimale de sa fonction de coût pour un weight decay de $1 \cdot 10^{-7}$ alors que la classe "book_jacket", à l'inverse voit sa valeur optimale de la fonction de coût pour une valeur du weight decay de $1 \cdot 10^{-3}$ mais contrairement à la classe "hippopotamus" pour un weight decay de $1 \cdot 10^{-7}$, sa valeur moyenne pour la fonction de coût n'est plus du tout optimale. Par conséquent, pour choisir une valeur de régularisation, en gardant le même processus d'apprentissage, il est nécessaire de faire des compromis entre les classes pour trouver une valeur bonne en moyenne mais optimale pour peu de classes voir aucune (on verra par la suite que plus de la moitié des classes semblent s'accorder sur un weight decay mais cela s'explique aussi par nos moyens limités en puissance de calcul qui nous ont contraints à n'entraîner le modèle que pour 11 valeurs différentes de l'hyperparamètre de régularisation se traduisant par une perte de précision). Afin de résoudre ce problème, on aurait pu imaginer un algorithme appliquant une pénalité différente en fonction des classes afin d'obtenir les optimaux pour chacune des classes de notre problème de classification. Bien que cela sorte du cadre de notre projet, il pourrait faire l'objet de futures recherches sur le sujet.



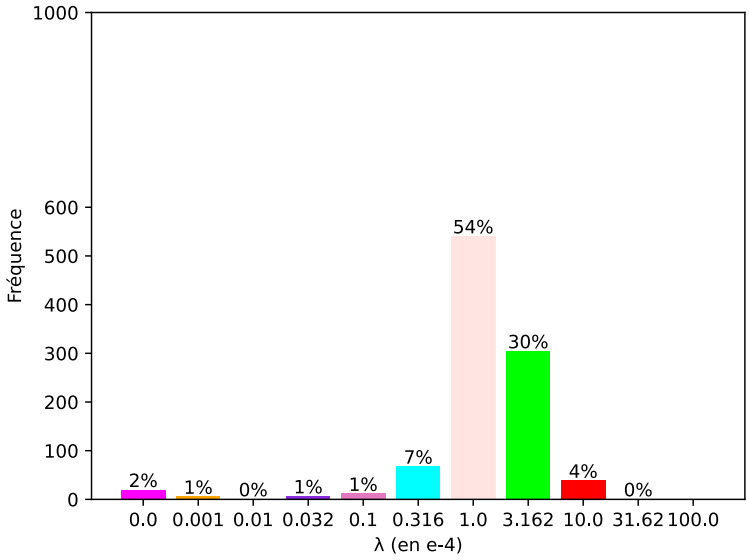
(a) Diagramme en barre du nombre de classes ayant leur minimum par λ



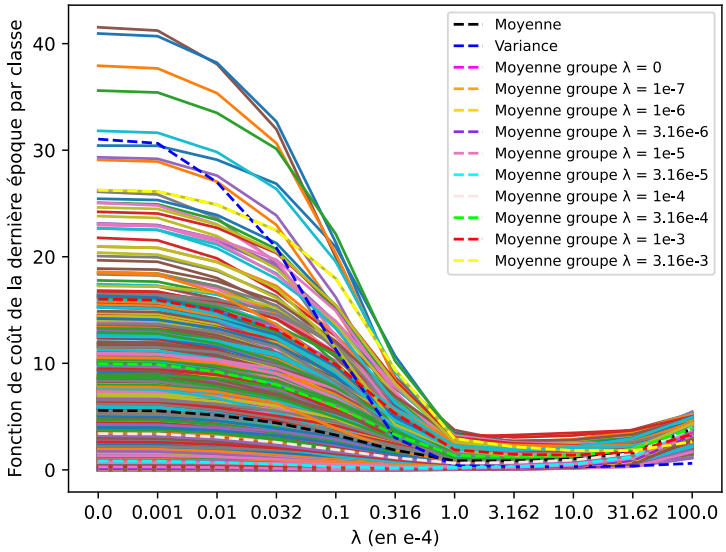
(b) Fonction de coût en fonction du paramètre du weigh decay λ pour chaque classe

FIGURE 5.1 – Représentation de la répartition des valeurs minimales de la fonction de coût des classes parmi les valeurs de λ testées pour l'ensemble d'apprentissage.

La représentation sur la même figure des moyennes des valeurs de la fonction de coût pour chacune des classes (obtenues grâce aux modèles entraînés à l'issu de la 90ème époque), permet d'avoir une vue d'ensemble sur l'apprentissage des classes. La figure 5.1(b) témoigne de l'importance de la régularisation pour l'uniformité de l'apprentissage entre les classes. En effet, on remarque que pour des valeurs de λ plus proche de zéro, la variance empirique (en pointillés bleus sur la figure) augmente fortement. Par conséquent, cette dispersion relativement élevée pour des régularisations d'hyperparamètres inférieures, même sur l'ensemble d'entraînement, conduit à une performance moyenne inférieure à celle pour des valeurs plus importantes de λ. L'explication derrière cette dispersion pourrait s'expliquer par la complexité du modèle qui, avec le nombre important de classe, ne permet pas à l'optimiseur, dans les conditions de nos expériences, de maximiser toutes les classes en même temps et de la présence, par conséquent, de classes aberrantes qui augmentent fortement la valeur des indicateurs de dispersion et de moyenne. En traçant le diagramme en barre (figure 5.1(a)) pour représenter le nombre de classes qui atteignent leur minimum moyen de fonction de coût pour chacune des valeurs de λ testée, déjà dans l'entraînement, la valeur $\lambda = 1 \cdot 10^{-4}$ s'impose avec 71% des classes qui atteignent leur optimum. Sur la figure 5.1(b) la courbe en pointillés de la même couleur représente la valeur moyenne de ce groupe.



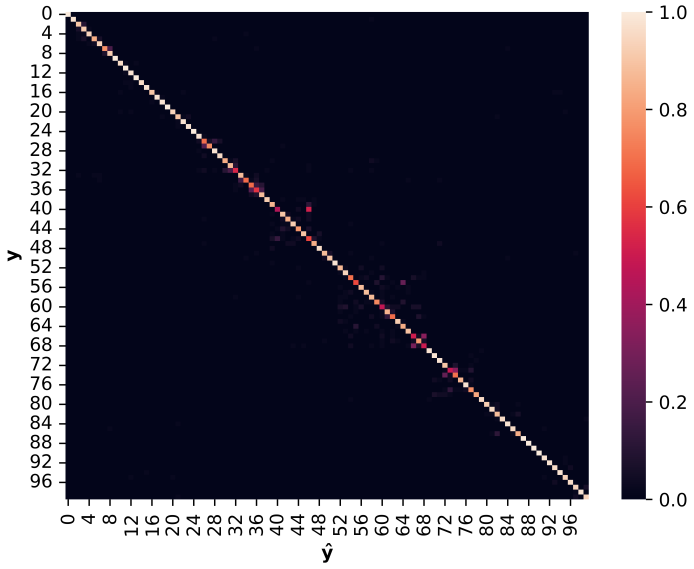
(a) Diagramme en barre du nombre de classes ayant leur minimum par λ



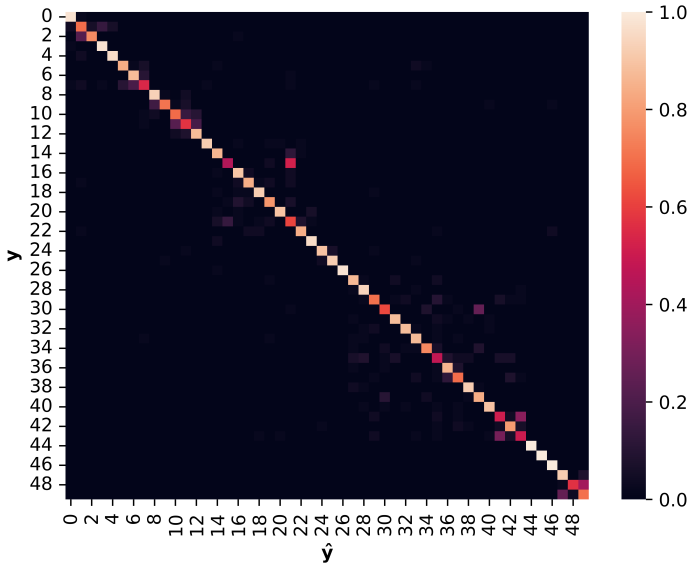
(b) Fonction de coût en fonction du paramètre du weigh decay λ pour chaque classe

FIGURE 5.2 – Représentation de la répartition des valeurs minimales de la fonction de coût des classes parmi les valeurs de λ testées pour l'ensemble de test.

Les figures 5.2(a) et 5.2(b) représente exactement la même chose que la figure 5.1 mais pour l'évaluation du modèle sur notre ensemble de données de test. Le diagramme en barre témoigne directement d'une meilleure dispersion des λ optimums presque tous représentés avec notre ensemble de test. La deuxième figure témoigne de l'explosion de la valeur de la fonction de coût pour certaines classes pour les modèles avec des valeurs de λ les plus faibles. On remarque que l'échelle a ainsi été augmentée et que la dispersion et la moyenne pour des $\lambda \geq 3.16 \cdot 10^{-5}$ bien que évidemment supérieure à celle observée sur l'ensemble d'entraînement, reste largement inférieure aux résultats retrouvés pour des valeurs de l'hyperparamètre de régularisation inférieures. Finalement, cela confirme le caractère normalisateur du weight decay qui permet de limiter au maximum les classes n'apprenant pas ou peu malgré des valeurs nécessairement moins bonnes pour une minorité de classes (moins de 50 %). Le diagramme en barre sur l'ensemble de test nous indique également que pour chercher une valeur de λ pour la majoritée des classes, il faudrait répéter l'expérience pour $\lambda \in [1 \cdot 10^{-4}, 3.162 \cdot 10^{-4}]$.



(a) Matrice de confusion pour des labels compris entre 0 et 100



(b) Matrice de confusion pour des labels compris entre 25 et 75

FIGURE 5.3 – Représentation de la matrice de confusion associées à $\lambda = 10^{-4}$

Finalement, pour étudier les potentielles erreurs de classification liées à des similarités visuelles entre les classes, il faut s'intéresser à la construction de la base de donnée ImageNet. Comme

introduit dans la section 3.4.1, ImageNet voit ses classes ordonnées selon l'architecture WordNet qui établit un lien sémantique entre les différentes classes. Ainsi, dans les ensembles de données d'ImageNet, les labels sont rangés par similitudes sémantiques au lieu d'être classés par ordre alphabétique par exemple. En traçant la matrice de confusion (figure 5.3), on pourra ainsi détecter ces potentiels erreurs liés à cette proximité visuelle. Cependant, le nombre de classes étant important et le modèle suffisamment précis, il est difficile de réellement distinguer d'importantes confusions. En se concentrant sur de plus petites zones comme sur la figure 5.3(b), on distingue toutefois certains clusters d'erreur probablement dû à cette proximité visuelle inter-classes.

6. Conclusion

Dans cette étude, notre objectif initial était de former un modèle en explorant différentes valeurs de l'hyperparamètre de régularisation (le weight decay). Par la suite, nous avons cherché à identifier le meilleur modèle, c'est-à-dire le paramètre de weight decay optimal, pour résoudre notre problème d'optimisation global ainsi que les meilleures valeurs du weight decay pour la reconnaisance de chaque classe. Cette démarche s'est avérée cruciale car l'apprentissage des différentes classes se déroule à des rythmes disparates. Afin d'atténuer cet effet, nous avons appliqué une pénalisation sur l'apprentissage de nos modèles en ajustant le paramètre λ de façon optimale.

Les valeurs du λ optimal pour le problème d'optimisation globale ainsi que pour la reconnaissance de chaque classe ont été déterminées de plusieurs manières :

- Taux de bonne classification en fonction des époques sur l'ensemble de test
- Fonction de coût en fonction des époques sur l'ensemble de test
- Fonction de coût en fonction de λ sur l'ensemble de test
- Valeurs absolues des poids et des biais en fonction des époques
- La répartition des minimums des fonctions de coût pour chaque valeur de λ

A partir de ces valeurs de λ, nous avons pu représenter la fonction de coût associée à l'utilisation de l'optimisateur global du problème.

7. Remerciements

Finalement, ce projet de TX a été une occasion unique pour nous de nous former sur de multiples notions du deep learning, et plus particulièrement sur les réseaux de neurones convolutifs (CNN) et la régularisation dans le processus d'apprentissage.

Nous tenons à remercier tout particulièrement M. Yves Grandvalet pour son encadrement précieux et sa disponibilité tout au long de ce projet.

Bibliographie

[1] Randall Balestriero, Leon Bottou, and Yann LeCun. The effects of regularization and data augmentation are class dependent. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 37878–37891. Curran Associates, Inc., 2022.

[2] Y. Bengio and Yann Lecun. Convolutional networks for images, speech, and time-series. *The Handbook of Brain Theory and Neural Networks*, 11 1997.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization : Accelerating deep network training by reducing internal covariate shift, 2015.

[5] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.