Deep Learning Optimisé - Jean Zay

Entraînement et large batches

INSTITUT DU DÉVELOPPEMENT ET DES RESSOURCES EN INFORMATIQUE SCIENTIFIQUE

DLO-JZ

4e partie de la formation de l'IDRIS.

Diapositives commentées.

Auteur : Bruno Tessier

Octobre 2022

Chapitres :

- Loss Landscape
- Optimiseurs de descente de gradient
- Optimisation des larges *batch*

Nous allons d’abord nous intéresser à la visualisation du paysage de la *loss.* Un outil de visualisation qui, bien que difficile à mettre en place, peut nous amener des informations utiles sur l’utilité des différents modèles et paramètres sur la descente de gradient.

Loss Landscape

https://losslandscape.com/

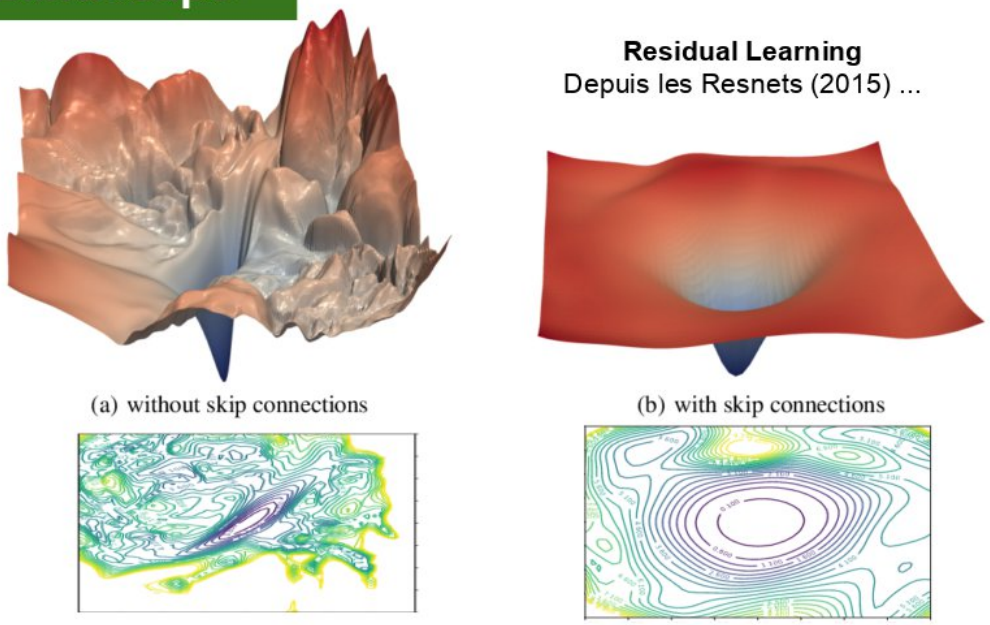
3

Nous vous invitons à regarder plus en détails le site https://losslandscape.com/ qui présente de multiples exemples de visualisation de la *loss* et qui met en avant différents papiers de recherche sur les méthodes de visualisation de *loss* et des avantages que ces visualisations peuvent apporter.

Loss Landscape

https://arxiv.org/pdf/1712.09913.pdf

Residual Learning
Depuis les Resnets (2015) ...

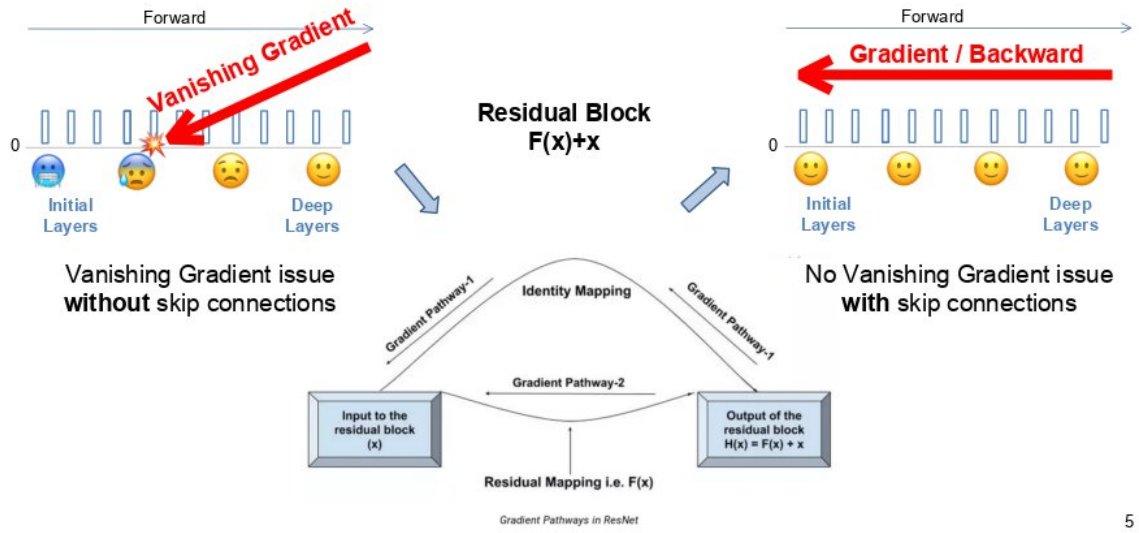


(a) without skip connections

(b) with skip connections

Par exemple, lorsqu'on compare la visualisation de la *loss* sur un apprentissage de modèles avec et sans *skip connections,* on peut voir que le fait d'ajouter des *skip connection*s (*Residual Learning*) rend le paysage de la *loss* beaucoup plus proche d'une fonction convexe, ce qui facilite grandement la descente de gradient.

Residual Learning

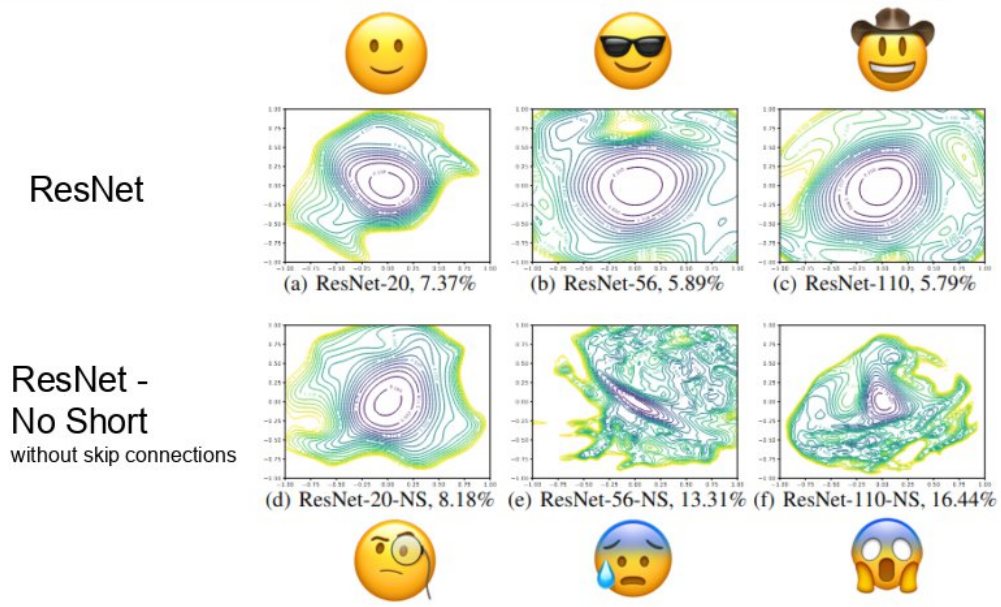


Lorsqu'un modèle commence à avoir trop de couches, on remarque que, avec le problème de *vanishing gradient* lors de la rétro-propagation, les premières couches auront un gradient nul et donc aucune modification des poids.

Pour faire face à ce problème, on utilisera le *residual learning.* Ce qui correspond à faire des liens (*skip connection*) qui vont sauter une ou plusieurs couches du réseau pour donner en entrée d'une couche éloignée leur sortie qui sera additionnée à la sortie de la couche précédente. Cela permet une diffusion du gradient sur l'ensemble des couches du modèle.

Le modèle *Resnet* a introduit cette technique qui est maintenant utilisée dans tous les modèles.

Residual Learning – impact sur la profondeur



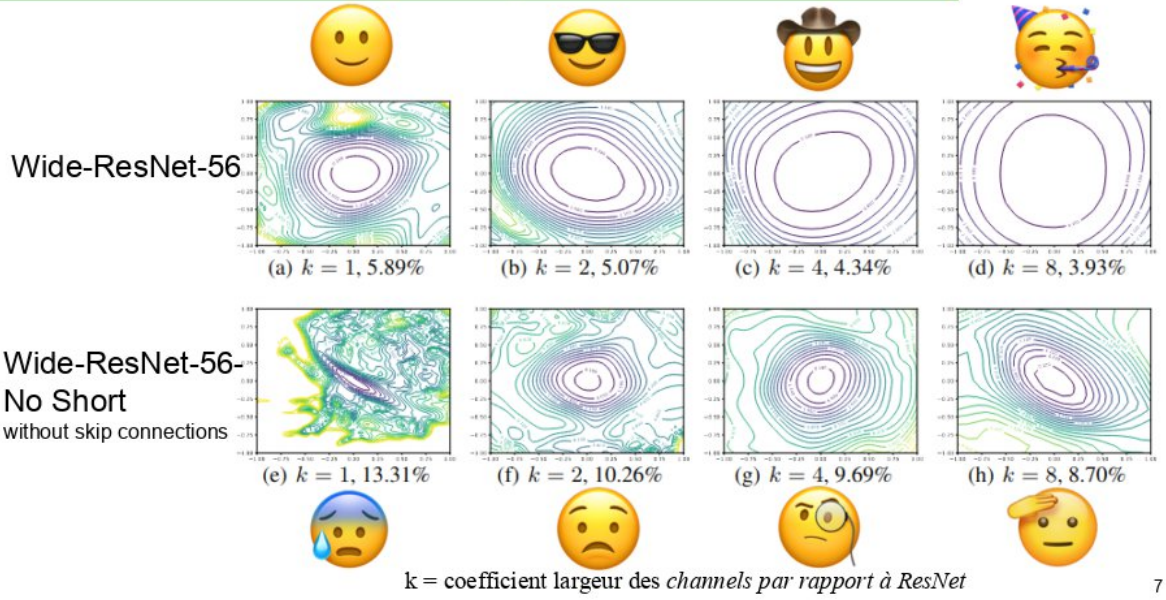
6

Lorsque qu'on utilise le modèle *Resnet* avec les *skip connections,* on peut voir que lorsqu'on augmente la taille du modèle, le paysage de *loss* devient de plus en plus proche d'une simple fonction convexe.

À l'inverse, lorsqu'on utilise le même modèle sans les *skip connections*, plus le modèle devient grand, plus la *loss* prend une forme qui rend difficile une descente de gradient.

Le pourcentage affiché correspond à la *loss* en fin d'apprentissage. Plus la valeur est petite, meilleur est l'apprentissage.

Residual Learning – impact sur la largeur



k = coefficient largeur des *channels par rapport à ResNet*

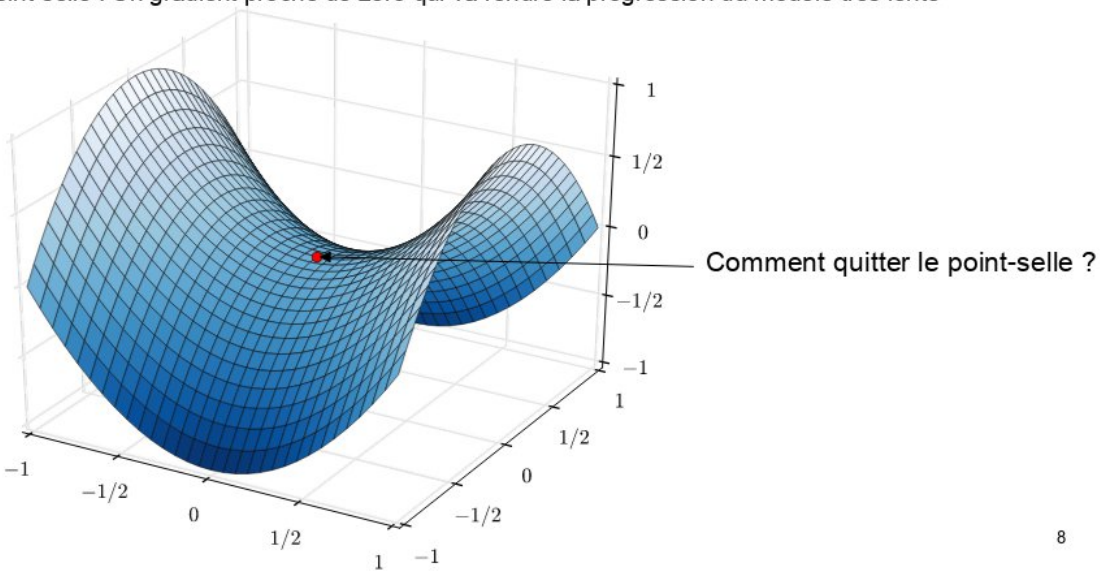
Par contre dans les deux cas (avec ou sans *skip connection*), on peut observer qu'en augmentant le nombre de filtres de convolution par couche (le coefficient de largeur des *channel*s) on augmente la convexité de la *loss* ce qui facilite l'apprentissage.

Remarque : Cela explique pourquoi les VGG pourtant profonds mais surtout très larges, marchaient avant les Resnet correctement sans *skip connection.*

Le pourcentage affiché correspond à la *loss* en fin d'apprentissage. Plus la valeur est petite, meilleur est l'apprentissage.

Problème du point-selle

Point-selle : Un gradient proche de zéro qui va rendre la progression du modèle très lente

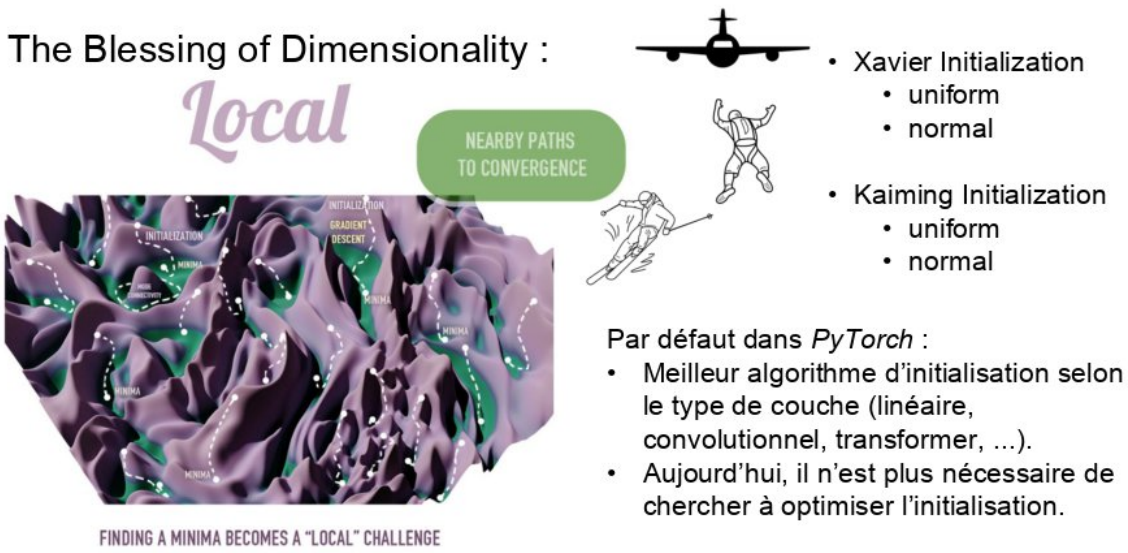


8

Un learning rate scheduler (vu par la suite) permet de diminuer le temps passé par le modèle dans les point-selles.

Ces points où le gradient est égal de zéro, mais qui ne sont pas des minimum vont coûter beaucoup d'itérations, mais en augmentant périodiquement le *learning rate* on va pouvoir sortir de ces points facilement pour converger plus rapidement.

Initialisation des paramètres du modèle

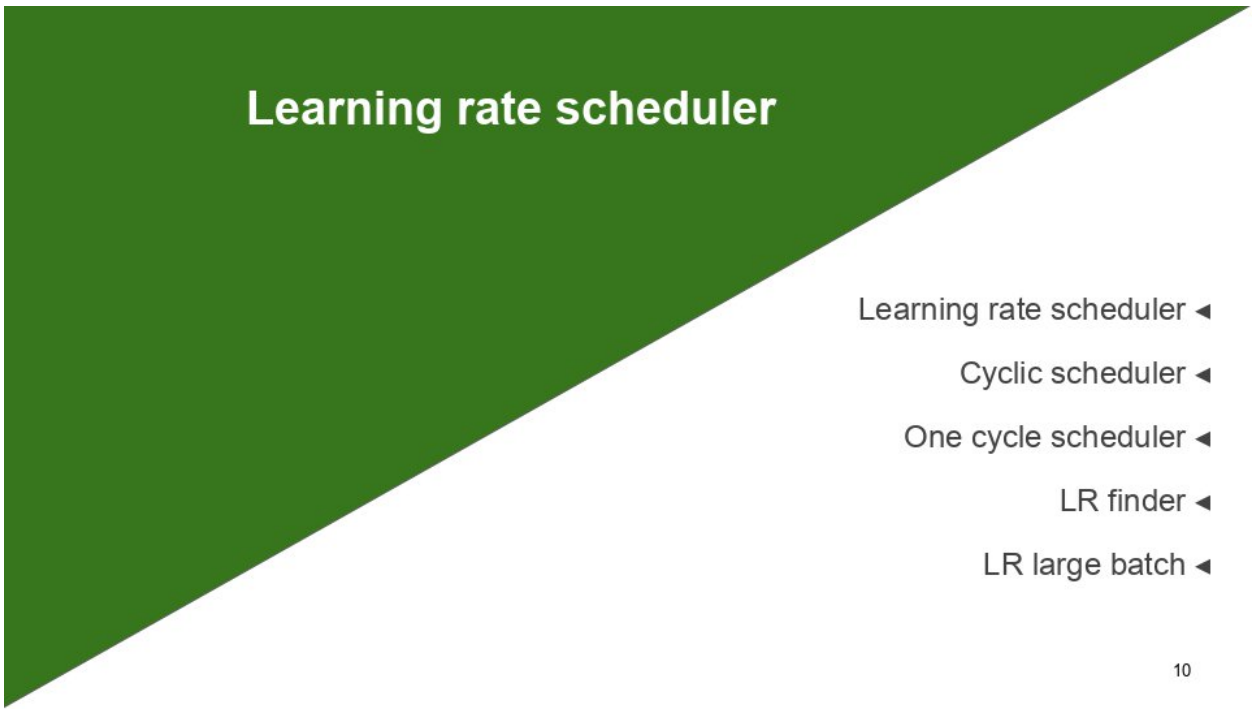


En opposition du principe de « Malédiction de la dimension » (« Curse of dimensionality »), qui explique que plus on rajoute de dimensions, plus l'apprentissage deviendra difficile (voir impossible), on observe en *Deep Learning* un effet bénéfique « Bénédiction de la dimension », lorsque l'on atteint un grand nombre de dimension.

Avec le nombre de dimension qui augmente, le nombre de minimaux locaux acceptable augmente aussi, le problème de descente de gradient devient donc un problème local où l'on cherche le plus petit minimum local proche de notre point de départ.

Le choix du point de départ correspond à l'initialisation de notre réseau de neurones. Il existe différents algorithmes qui vont essayer d'optimiser cette initialisation (Xavier, Kaiming, ...).

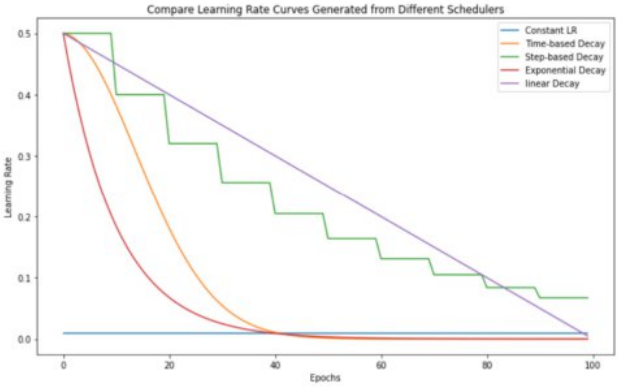
Dans Pytorch, l'initialisation sera par défaut optimisée avec ces différents algorithmes selon les types de couche du réseau. Par exemple, on a d'après la documentation pour les couches linéaires et les couches convolutionnelles : Kaiming uniform, pour les Transformer : Xavier uniform, ... etc

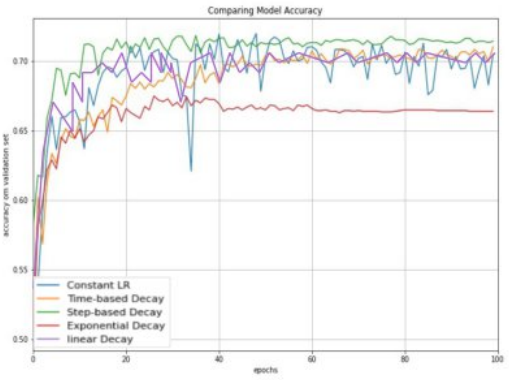


Dans ce chapitre, nous allons nous intéresser aux problématiques que l'on rencontre quand la taille de batch devient trop grande et nous allons étudier les différentes solutions qui existent pour faire face à ces problématiques.

Learning Rate Scheduler

Learning rate decay

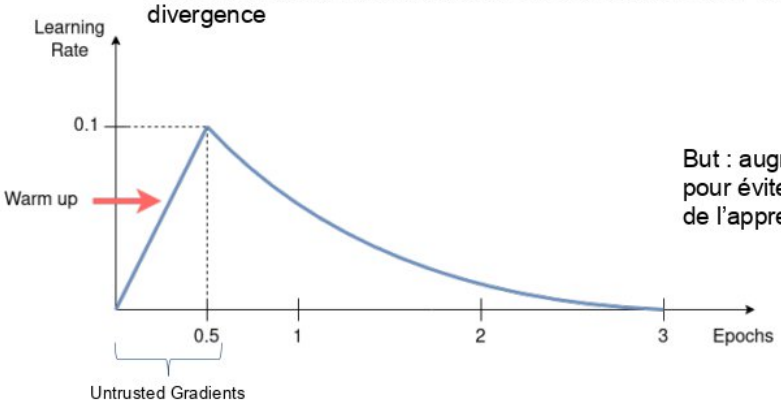




Learning Rate Scheduler

WARMUP pour *large batches*

Problèmes : Les premières itérations ont trop d'effet sur le modèle (loss importantes, gradients élevés, biais, …), un *learning rate* élevé peut provoquer une forte instabilité ou une divergence



But : augmenter petit à petit le *learning rate* pour éviter le risque de divergence au début de l'apprentissage

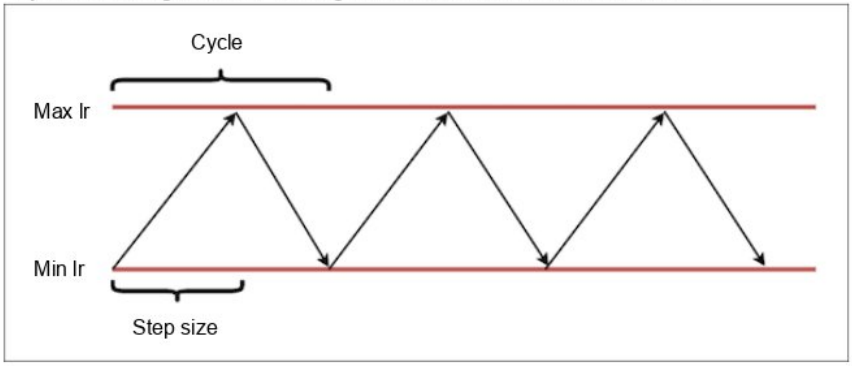
12

Les modèles ayant généralement une initialisation aléatoire des poids, les premières itérations peuvent avoir un trop grand effet sur les mises à jour du modèle (loss importantes, gradients élevés, biais,…). Cet effet est d'autant plus présent avec de grandes tailles de *batch.*

Pour essayer de réduire les effets de ces premières itérations, on peut ajouter en plus du *learning rate scheduler* un phase de *warmup* qui va graduellement augmenter le *learning rate* au début de l'apprentissage.

Cyclic Learning Rate Scheduler

Cyclical Learning Rates for Training Neural Networks - Leslie N. Smith 2017



Paramètres :

- Step_size = x * epoch ($2 \leq x \leq 10$)
- Base_lr -> valeur minimum de convergence
- max_lr -> valeur maximale avant divergence

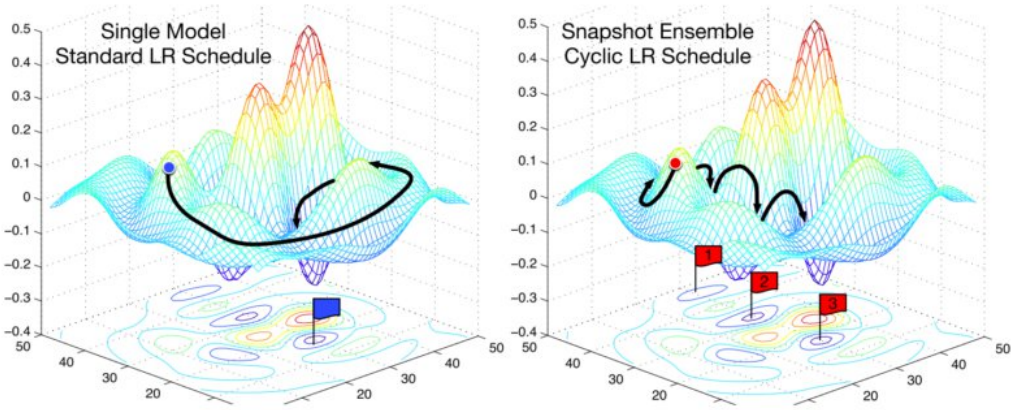
Succession de *warmups* et de *learning rate decays*

13

En restant sur l'idée du learning rate qui va être modifié dynamiquement au cours de l'apprentissage, Leslie N. Smith a introduit en 2017 le concept de *Cyclic Learning Rate Scheduler.*

L'idée est de faire des cycles de phase de *Warmup* et de diminution du *learning rate.* Le *learning rate* va varier entre une valeur minimale et maximale et va répéter ces cycles qui vont durer un nombre d'itération proportionnel au nombre d'itération par *epoch*.

Cyclic Learning Rate Scheduler



SNAPSHOT ENSEMBLES: TRAIN 1, GET M FOR FREE
Gao Huang, Yixuan Li, Geoff Pleiss

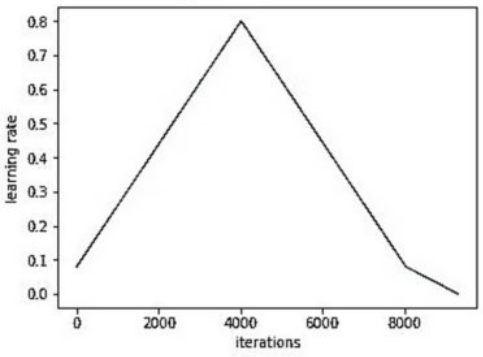
14

Cette technique a pour premier avantage de forcer le modèle à sortir facilement de possibles minimum locaux pour essayer d'en trouver un meilleur à chaque fois que le *learning rate* augmente.

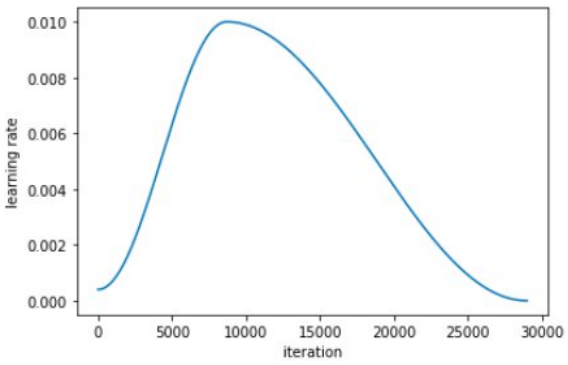
One Cycle Learning Rate

Un seul cycle suffit ! A disciplined approach to neural network hyper-parameters - Leslie N. Smith

Proposition initiale



cosine annealing : Recommandation par FastAI

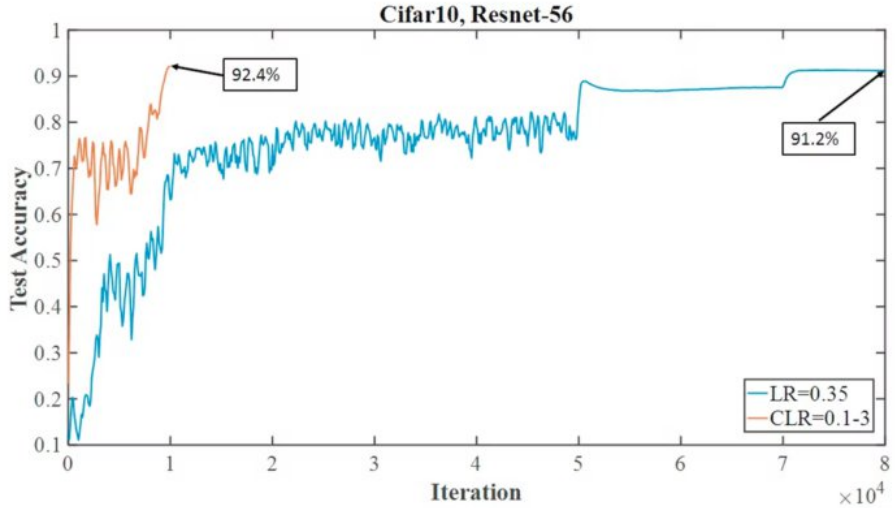


15

En 2018, Leslie N. Smith s'est rendu compte qu'au lieu de faire plusieurs cycles, il est plus efficace de ne faire qu'un seul grand cycle d'augmentation et diminution du *learning rate* que l'on appelle le *One Cycle Learning Rate*. Cela amène un effet de super convergence où l'on va forcer le modèle à converger rapidement avec un *learning rate* élevé après une phase d'augmentation progressive.

La proposition initiale est une augmentation et diminution linéaire, mais plus tard FastAI a proposé une variation plus efficace avec une forme sinusoïdale. Dans les deux cas, le cycle se termine par une grande diminution du *learning rate* dans les dernières itérations.

One Cycle Learning Rate - Super convergence



Une convergence plus rapide pour une précision finale équivalente

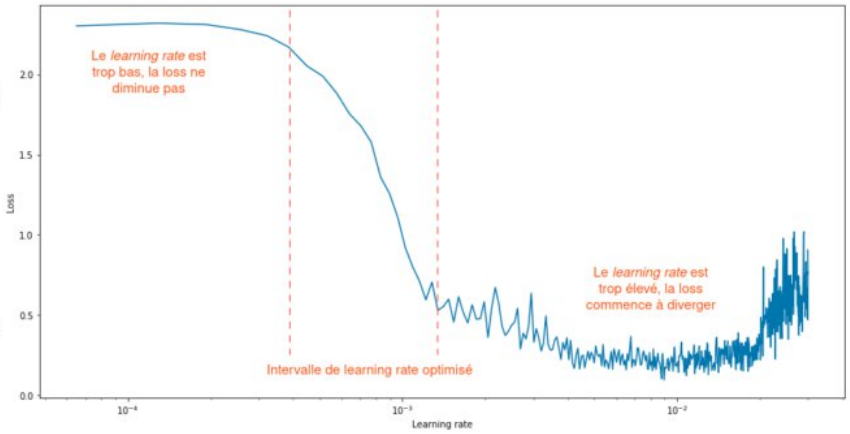
16

Dans son papier, Leslie N. Smith montre qu'avec un réseau Resnet 56 et la base de données CIFAR 10, on peut converger plus rapidement avec un *Cyclic Learning Rate* qu'avec un *learning rate* constant tout en obtenant une précision similaire.

Learning Rate Finder

But : Trouver les valeurs de *learning rate* optimales pour son modèle, particulièrement pour la valeur maximale d'un *cyclic scheduler*

- Faire tourner son modèle sur quelques *epochs* en faisant augmenter son *learning rate*
- Début de baisse de la *loss* → *Learning rate* minimal
- Début de variation de la *loss* → ***Learning rate*** **maximal**



17

Pour trouver les valeurs minimales et maximales du *cyclic learning rate* on utilise la technique du *learning rate finder.*

On va entraîner un modèle sur quelques *epochs* en commençant avec une valeur de *learning rate* très faible qui va augmenter petit à petit jusqu'à atteindre une valeur très grande.

En traçant la courbe de *loss* sur ce petit entraînement, on va pouvoir observer deux choses :

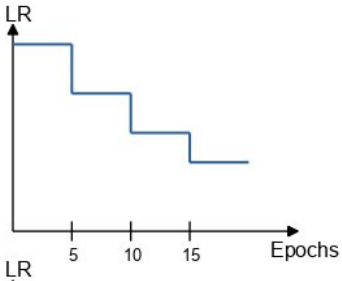
- Au début, la *loss* ne diminue pas jusqu'à que l'on atteigne une valeur de *learning rate* assez grande pour avoir un effet positif sur le modèle. Cette valeur sera la valeur minimale.

La *loss* diminue de manière assez stable jusqu'à ce qu'elle commence à beaucoup varier et diverger. On peut donc trouver la valeur maximale de *learning rate* qui a un effet positif sur la loss sans la faire diverger.

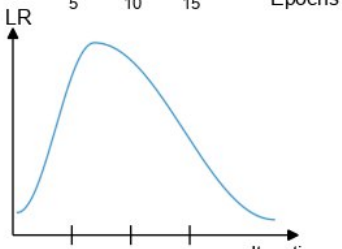
Learning Rate Scheduler

Chaque *scheduler* a ses propres paramètres

```
import torch.optim as opt

scheduler = opt.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

for epoch in range(100):
        train(...)
        validate(...)
        scheduler.step()
```




```
import torch.optim as opt

scheduler = opt.lr_scheduler.CyclicLR(optimizer, base_lr=0.01, max_lr=0.1)


for epoch in range(10):
        for batch in data_loader:
                train_batch(...)
                scheduler.step()
```



18

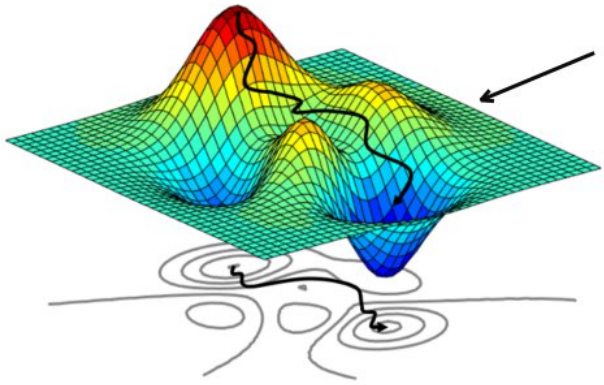
Tous les types de *learning rate scheduler* que nous avons vu dans ce cours sont implémentés dans la librairie Pytorch.

Il faut noter que les *cyclic learning rate schedulers* font leurs mises à jour à chaque nouveau batch contrairement aux autres qui font leurs mises à jour à chaque *epoch*.

Dans ce chapitre, nous allons étudier le fonctionnement des deux optimiseurs les plus utilisés et l'importance de leurs paramètres pour un apprentissage.

Optimiseur - SGD

L'optimiseur est l'algorithme qui pilote la descente de gradient et la recherche de minimum avec pour but d'optimiser le temps d'apprentissage et la métrique finale.



SGD = *Stochastic Gradient Descent*
Calcul du Gradient et mise à jour des poids
à chaque *batch*

\+ Taille de *batch et learning rate* adaptables selon les besoins contradictoires :

- D'exploration pour trouver le meilleur minimum local
- D'accélération de la descente de gradient

20

Un optimiseur est tout simplement l'algorithme qui va essayer de réguler la descente de gradient et la recherche de minimum dans la fonction de *loss*. L'objectif d'un optimiseur est de faire converger la descente de gradient vers le minimum global (ou meilleur local) en un minimum de temps.

SGD with Momentum

$$m_0 = 0$$

$$m_i = \beta * m_{i-1} + (1 - \beta) * g_i$$

$$\theta_i = \theta_{i-1} - \alpha * m_i$$

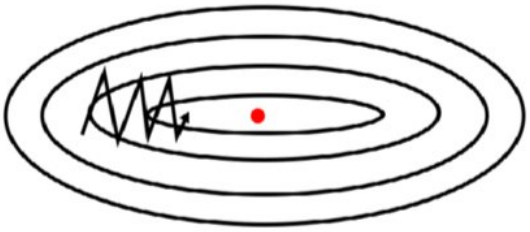
Coefficient de *momentum*

Objectif : Prendre en considération les gradients précédents pour une descente de gradient plus rapide.

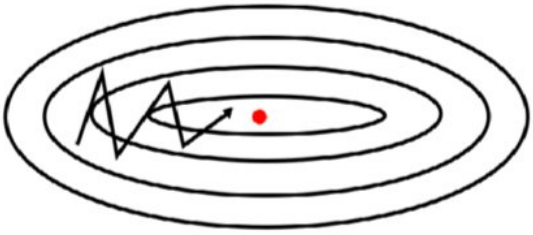
Valeur initiale conseillée : 0,9

$$0.85 < \beta < 0.95$$

SGD without momentum



SGD with momentum



+ Permet de converger plus rapidement

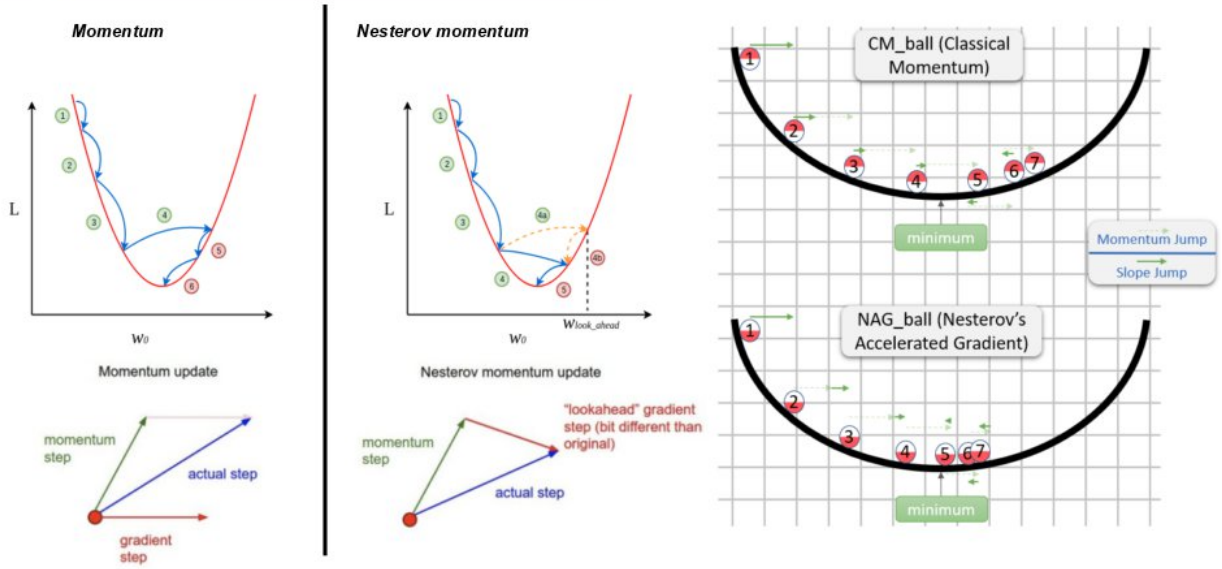
- Pas de garantie que le *momentum* nous amène dans la bonne direction

21

Pour utiliser le *momentum* les optimiseurs rajoute une valeur m, qui va prendre en mémoire les gradients précédents à l'aide d'une moyenne glissante. Cette valeur peut prendre une importance plus ou moins grande en fonction du coefficient *de momentum* que l'on va donner à notre optimiseur.

À l'aide de cette mémoire des gradients, lorsque il y aura une forte pente de gradient les mises à jour des poids seront plus importantes ce qui nous permettra de converger plus rapidement.

Types de *Momentum*



But : Vérifier que le momentum nous amène dans la bonne direction

Dans certains cas où le *momentum* prend trop d'ampleur, il arrive qu'une mise à jour du gradient soit trop importante et rate la valeur minimale de loss que l'on cherche à trouver.

Pour contrer ce genre de problème le Nesterov momentum ne va pas calculer le gradient de la position où il se trouve, mais le gradient d'une position intermédiaire (*lookahead*) calculée en appliquant le *momentum* actuel. Si le *momentum* est trop élevé et nous fait dépasser le minimum que l'on souhaite atteindre, le gradient modifié devrait atténuer la mise à jour des poids effectuée.

Optimiseurs adaptatifs

Plutôt que de piloter la descente de gradient manuellement avec le *learning rate* …

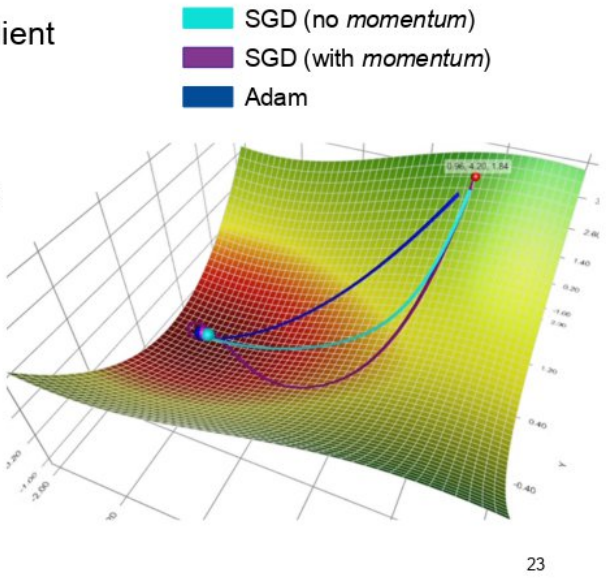
Nous pouvons adapter le *learning rate* **pour chaque poids du modèle** en fonction du gradient, du gradient^2, ou de la norme des poids de la couche !!

Exemples :

- AdaGrad,
- AdaDelta,
- RMSprop
- **Adam**

Spécialisés pour les batches larges :

- **LARS**
- **LAMB**



23

Les optimiseurs adaptatifs vont regarder l'évolution du gradient et des poids des différentes couches du modèle pour essayer de trouver quel est le *learning rate* adapté à chaque situation.

Adam

Adam : Adaptative moment estimation

$$m_i = \beta_1 * m_{i-1} + (1 - \beta_1) * g_i$$

Premier moment : moyenne glissante

$$v_i = \beta_2 * v_{i-1} + (1 - \beta_2) * g_i^2$$

Second moment : variance non centrée glissante

$$\hat{m}_i = \frac{m_i}{1 - \beta_1^i}$$

$$\hat{v}_i = \frac{v_i}{1 - \beta_2^i}$$

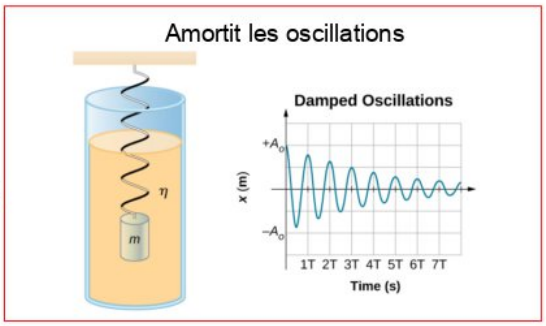
Correction des biais des premières itérations

$$\theta_i = \theta_{i-1} - \frac{\alpha}{\sqrt{\hat{v}_i + \epsilon}} * \hat{m}_i$$

Paramètres :

α — Taux de régression ($\beta_1 = 0.9$ & $\beta_2 = 0.999$)

ϵ — Très petite valeur pour éviter une division par zéro



But : Adapter l'importance des mise à jours de poids en fonction des précédents gradients et de la variabilité du gradient.

24

L'optimiseur Adam (*Adaptative Moment Estimation*) va reprendre l'idée du *momentum* en gardant une moyenne glissante des gradients précédents (premier moment) qui sera plus au moins importante en fonction de la valeur du coefficient β1.

Adam ajoute une autre valeur (second moment) qui va être la moyenne glissante des précédents gradients au carré dont on va calculer la racine carrée ce qui va correspondre à la variance non centrée glissante des précédents gradients.

Une opération supplémentaire va avoir lieu pour réduire les biais des premières itérations où les moyennes glissantes sont faites sur un petit nombre de valeurs.

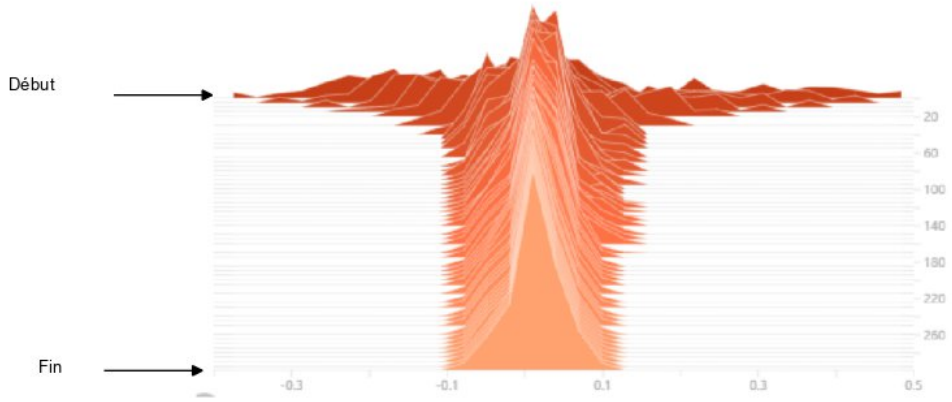
Pour la mise à jour, on va soustraire aux poids du réseau le premier moment m divisé par la racine carrée du deuxième moment v le tout multiplié par le *learning rate* donné en entrée.

Cet optimiseur a donc pour avantage d'intégrer le momentum, mais aussi un autre paramètre qui va empêcher au momentum de prendre trop d'importance dans la mise à jour des poids. Ces deux outils vont permettre à Adam d'amortir les effets d'oscillation dans la descente de gradient.

Weight decay

Un réseau de neurones qui converge et généralise correctement (ni sous-apprentissage ni sur-apprentissage) a généralement **des poids qui tendent vers 0.**

Distribution des poids durant l'apprentissage :



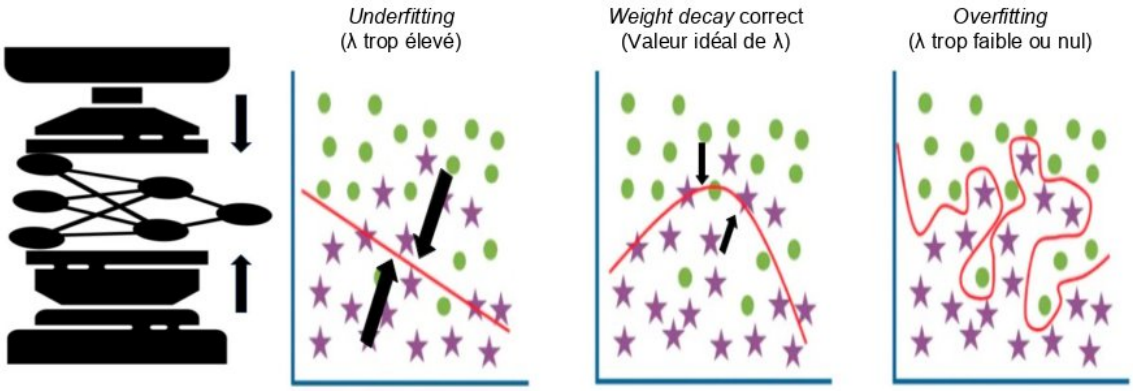
25

Lors des entraînements de réseaux de neurones, on remarque généralement que les modèles qui généralisent bien ont des poids qui vont tendre de plus en plus vers zéro.

Weight decay

Préférable à la régularisation L2 standard définie dans la fonction de perte

λ : paramètre du *weight decay (généralement entre 0 et 0.1)*



La technique de *weight decay*, définie dans *l'optimizer* permet de forcer les poids à converger vers des valeurs proches de zéro.

26

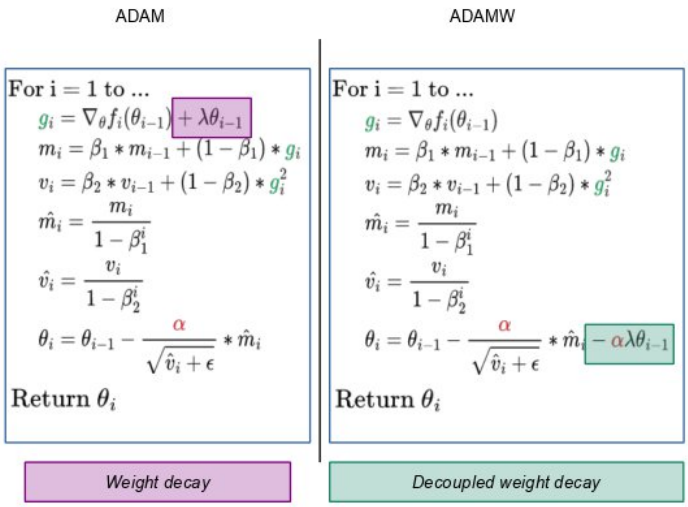
Le *weight decay* va forcer les poids du modèle a converger vers zéro plus rapidement. Dans ce but les optimiseurs vont prendre un paramètre supplémentaire λ qui va représenter la proportion de réduction des poids à chaque mise à jour.

Si λ est trop faible ou nul les poids vont moins tendre vers zéro et il est possible que le modèle devienne trop adapté aux données d'entraînement et donne de mauvais résultats avec les données de test (*Overfitting*).

À l'inverse si λ est trop élevé le modèle va commencer à trop généraliser et aura une mauvaise précision.

Lorsqu'il est bien configuré le *weight decay* va réduire le temps d'entraînement en accélérant la vitesse de convergence des poids vers zéro.

Weight decay and decoupled weight decay



Évolution du *weight decay*: *Decoupled weight decay* (découplé du momentum !!)

- SGD et Adam avec le *weight decay*
- SGDW et AdamW avec le *decoupled weight decay*

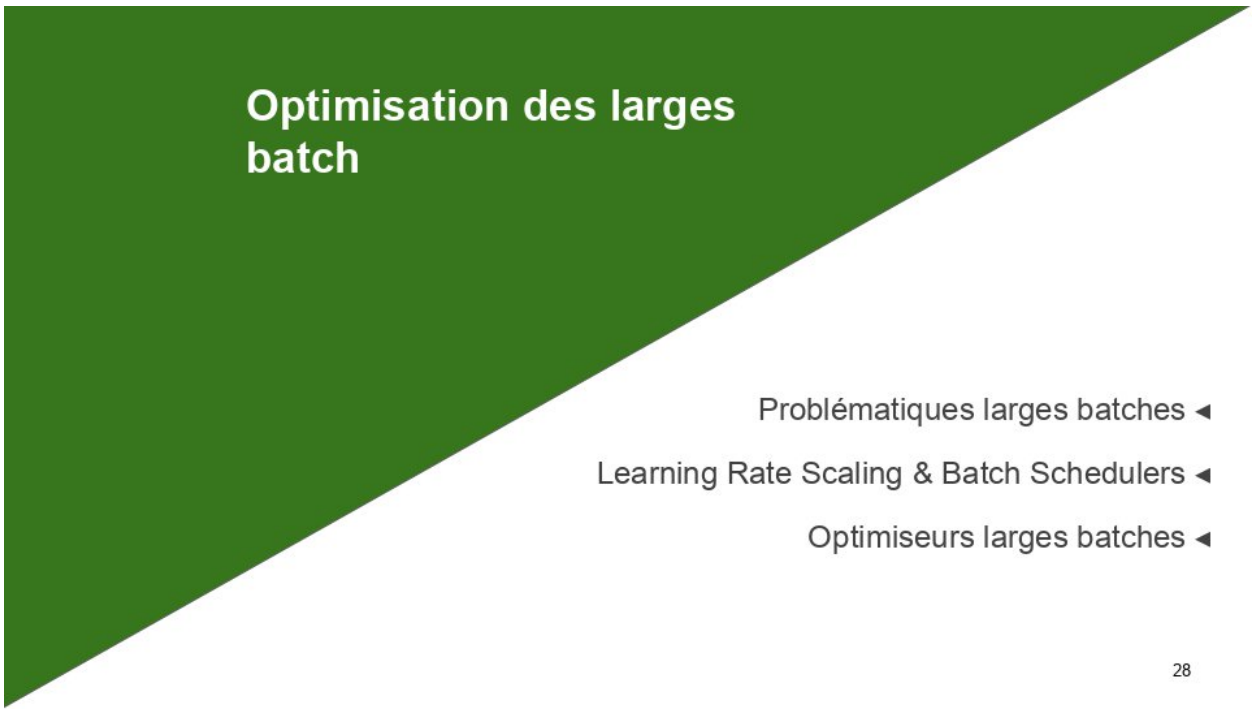
SGD et SGDW sont à peu près équivalents en performance.

Cependant AdamW est notablement meilleur que Adam !!

27

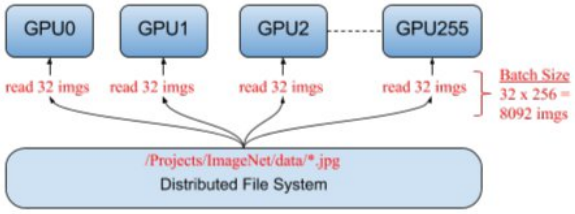
Il existe deux implémentations différentes du *weight decay*. Dans la version initiale, on va ajouter au gradient la valeur des poids multipliée par le coefficient de *weight decay.* Cette méthode est utilisée dans les optimiseurs SGD et ADAM.

Avec le *decoupled weight decay* on va plutôt soustraire directement la valeur des poids multipliée par le coefficient de *weight decay* au niveau de la mise à jour des poids.



Dans ce chapitre, nous allons nous intéresser aux problématiques que l'on rencontre quand la taille de batch devient trop grande et nous allons étudier les différentes solutions qui existent pour faire face à ces problématiques.

Large Batches avec le parallélisme de données



Parallélisme de données : La parallélisation implique une grande taille de *batch*

Problème : Les *batches* trop grands (> 512) ont tendance à engendrer de moins bonnes performances

29

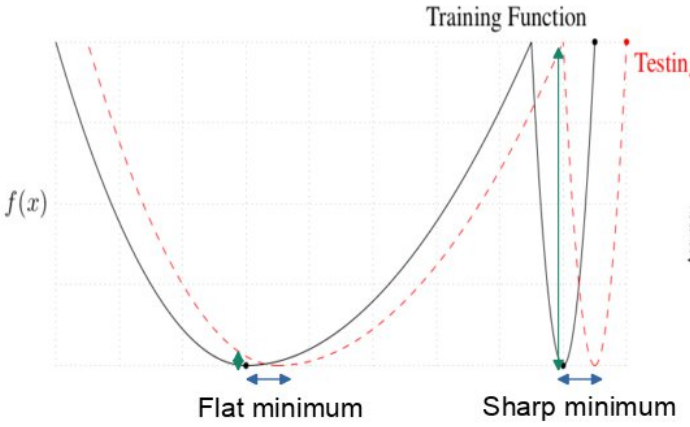
Lorsqu'on utilise le *data parallelism* la taille de *batch* globale va devenir la taille de *batch* initiale multipliée par le nombre de GPU, cette taille globale peut devenir rapidement importante.

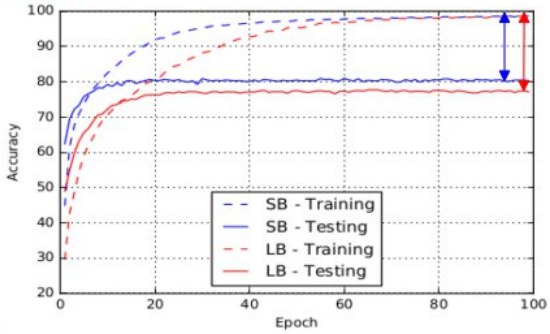
Le problème étant que lorsque la taille de batch devient trop grande, on observe que les modèles ont tendance à moins bien généraliser.

Large Batches

On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, Ping Tak Peter Tang





Comparaison d'entraînement d'un réseau convolutionnel avec des petits batch (SB) et large batch (LB) sur CIFAR 10

Plus le *batch* est grand, plus le modèle a tendance à converger vers des minimums pentus et étroits. 30

Dans le papier *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*, les auteurs expliquent que ce problème de mauvaise généralisation est causé par le fait que plus la taille de *batch* est grande plus le modèle va avoir tendance à converger vers des minimums pentus (*sharp minimum*) et étroits au lieu de minimums larges et plats (*flat minimum*).

Les *sharp minima* vont moins bien généraliser, car contrairement aux *flat minimuma* une petite différence entre les données d'entraînement et les données de test vont amener une grosse différence sur la sortie du modèle.

Large Batches : Learning rate scaling

Lorsqu'on augmente considérablement la taille du *batch* global, il est souvent nécessaire mettre le *learning rate* à l'échelle :

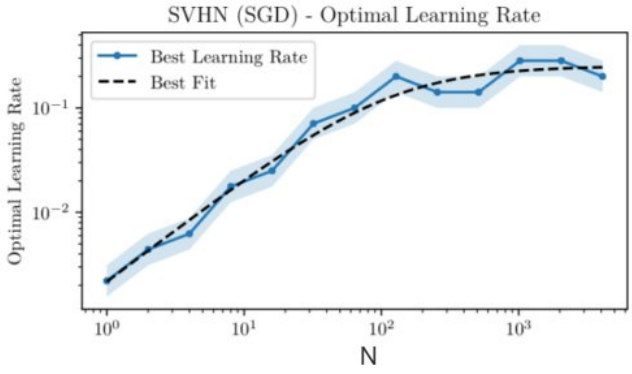
N = Nombre de processus parallèles

Croissance linéaire du *learning rate* :

$$\alpha \rightarrow N * \alpha$$

Croissance en racine carrée du *learning rate* :

$$\alpha \rightarrow \sqrt{N} * \alpha$$



An Empirical Model of Large-Batch Training
Sam McCandlish, Jared Kaplan, Dario Amodei

Optimal : croissance linéaire au début puis en racine carrée (recommandé par OpenAI)

Lorsqu'on a déjà un *learning rate* adapté à notre modèle, mais que l'on souhaite mettre notre apprentissage à l'échelle sur plus de GPU (ce qui peut considérablement augmenter notre taille te *batch)* il est nécessaire de faire une mise à l'échelle de notre *learning rate*.

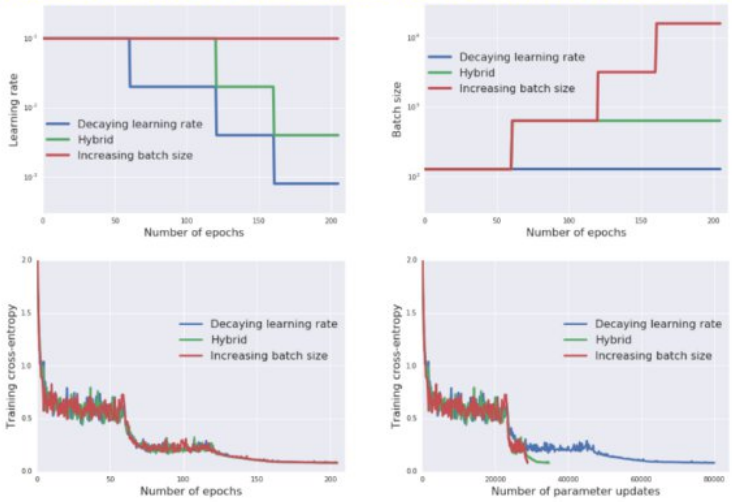
Pour cela, deux approches sont possibles : une croissance linéaire (multiplication par le nombre de processus) ou une croissance en racine carrée (multiplication par la racine carrée du nombre de processus).

Dans le but de tester empiriquement laquelle de ces deux approches est préférable, des chercheurs de OpenAI ont tracé une courbe des *learning rate* optimaux en fonction du nombre de GPU avec SGD. On peut voir que pour un petit nombre de GPU la valeur augmente linéairement, mais au bout d'un certain nombre de GPU on est plus proche d'un croissance en racine carrée.

Batch Size Scheduler

=> Alternative au *Learning Rate Scheduler*

DON'T DECAY THE LEARNING RATE, INCREASE THE BATCH SIZE



A ne pas utiliser pour *Imagenet Race* !!

Cependant souvent utilisé pour l'apprentissage des très gros modèles, par exemple BLOOM utilise :

- *AdamW*
- un *LR Scheduler* (Warmup + Decay)
- **un *Batch Size scheduler* (*linear increasing*)**

Une technique alternative au *learning rate scheduler* est d'augmenter dynamiquement la taille du batch pendant l'apprentissage. Cette technique, qui permet d'atteindre les même performances qu'avec un *learning rate scheduler*, aurait l'avantage de faire moins de mise à jour des paramètres, ce qui permet d'entraîner son modèle plus rapidement.

Large Batches

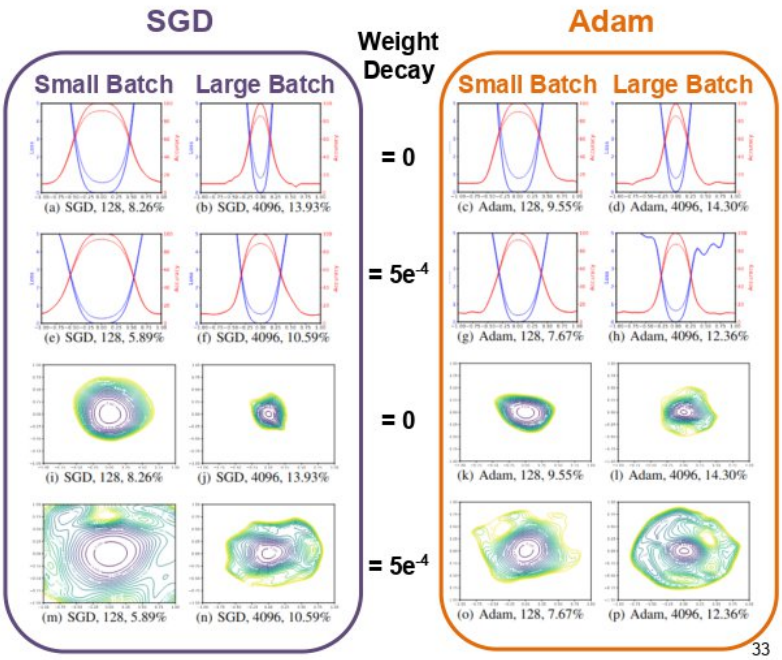
Tendances :

Flat Minimum	Sharp Minimum
- Test Loss	+ Test Loss
Slow Descent	Fast Descent

Small Batch ⟵ ⟶ Large Batch

SGD ⟵ ⟶ ADAM

Weight Decay ⟵ ⟶ Sans W Decay



33

En observant la visualisation 2D de la *loss* de différents apprentissages, on peut voir les effets des optimiseurs et paramètres utilisés lors de ces apprentissages.

On observe que des petites tailles de batch ont tendance à faire des *flat minimum* ce qui peut causer un apprentissage plus lent en contrepartie d'une petite différence entre l'apprentissage et les tests.
A l'inverse, une grande taille de batch va amener plus de *sharp minimum* qui vont amener une plus grosse différence entre les *loss* d'apprentissage et de test mais un apprentissage plus rapide.

L'optimiseur SGD tend vers des *flat minimum* tandis que ADAM tend plutôt vers des *sharp minimum.*

L'ajout d'un *weight decay* corrige l'effet des différentes tendances et réduit les chances de tomber dans un *sharp minimum.*

Optimiseurs *Large Batches* - LARS

LARS pour "Layer-wise Adaptive Rate Scaling."

Adaptation de *SGD with momentum* avec l'ajout d'un trust ratio pour chaque couche qui dépend de l 'évolution du gradient de la couche

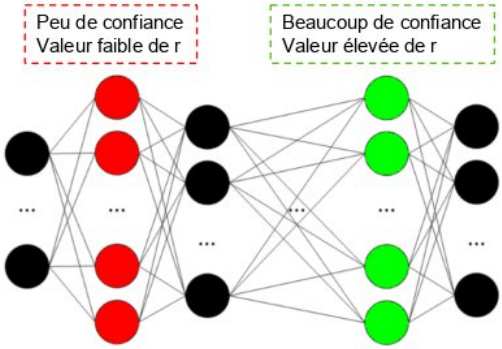
r = Trust ratio
l = Numéro de couche

$$m_i = \beta * m_{i-1} + (1-\beta) * (g_i + \lambda\theta_{i-1})$$

Weight decay

$$r_1 = \left\|\theta_{i-1}^l\right\|_2$$
$$r_2 = \left\|m_i^l\right\|_2$$
$$r = r_1 / r_2$$
$$\alpha^l = r * \alpha$$

$$\theta_i^l = \theta_{i-1}^l - \alpha^l * m_i^l$$



But : Adapter l'importance des mise à jours des poids en fonction d'un *trust ratio* calculé pour chaque couche du réseau.

34

L'optimiseur LARS (*Layer-wise Adaptative Rate Scaling*) est assez similaire à SGD, mais va rajouter une variable R appelé le ratio de confiance. Ce ratio va être recalculé à chaque mise à jour du modèle pour chaque couche de neurone.

La variable R1 représente la norme des poids de la couche et R2 représente la norme des changement des poids de la couche avec le *weight decay.*

En multipliant le *learning rate* par le ratio R1/R2 on va pouvoir adapter la vitesse de modification des poids de chaque couche en fonction de la confiance qu'on leur donne.

Optimiseurs *Large Batches* - LAMB

LAMB pour "Layer-wise Adaptive Moments optimizer for Batch training."

Adaptation de Adam avec l'ajout d'un trust ratio pour chaque couche qui dépend de l 'évolution du gradient de la couche

r = Trust ratio
l = Numéro de couche

$$r_1 = \| \theta_{i-1}^l \|_2$$

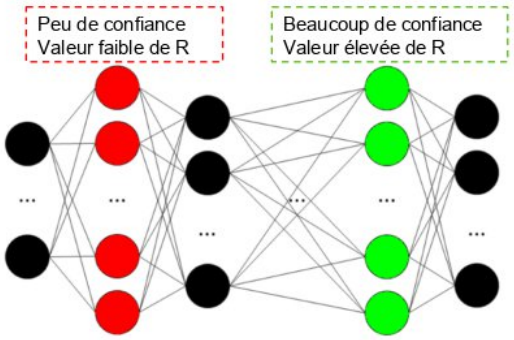
$$r_2 = \| \frac{\hat{m}_i^l}{\sqrt{\hat{v}_i^l + \epsilon}} + \lambda \theta_{i-1}^l \|_2$$

$$r = r_1 / r_2$$

$$\alpha^l = r * \alpha$$

$$\theta_i^l = \theta_{i-1}^l - \alpha^l * \left(\frac{\hat{m}_i^l}{\sqrt{\hat{v}_i^l + \epsilon}} + \lambda \theta_{i-1}^l \right)$$

Decoupled weight decay



But : Adapter l'importance des mise à jours des poids en fonction d'un *trust ratio* calculé pour chaque couche du réseau.

35

L'optimiseur LAMB (*Layer-wise Adaptative Moment optimizer for Bacth training*) est assez similaire à Adam, mais va rajouter une variable R appelée le ratio de confiance. Ce ratio va être recalculé à chaque mise à jour du modèle pour chaque couche de neurone.

La variable R1 représente la norme des poids de la couche et R2 représente la norme des changement des poids de la couche avec le *weight decay.*

En multipliant le *learning rate* par le ratio R1/R2 on va pouvoir adapter la vitesse de modification des poids de chaque couche en fonction de la confiance qu'on leur donne.

Implémentation des optimiseurs

Chaque optimiseur a ses propres paramètres

SGD

```
import torch.optim as opt

SGD_optimizer = opt.SGD(params, lr, momentum=0, weight_decay=0, nesterov=False, ...)
```

ADAMW

```
import torch.optim as opt

ADAM_optimizer = opt.AdamW(params, lr=0.001, betas=(0.9, 0.999), weight_decay=0.05,..)
```

LAMB

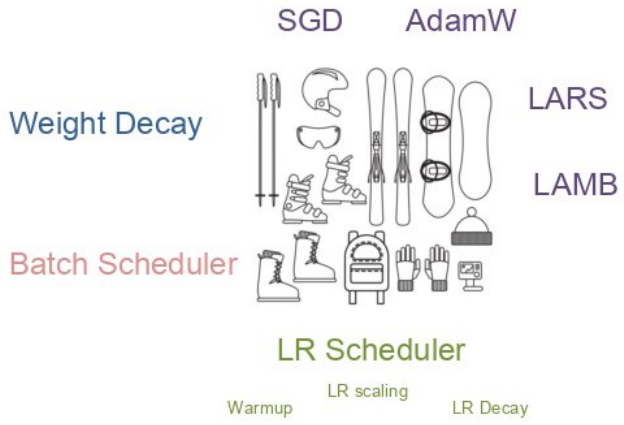
```
from apex.optimizers import FusedLamb

LAMB_optimizer = FusedLamb(params, lr=0.001, betas=(0.9, 0.999), weight_decay=0,
adam_w_mode=True)
```

LARC

Optimisation apex de LARS

```
import torch.optim as opt
from apex.parallel.LARC import LARC

base_optimizer = opt.SGD(params, lr=0.001, momentum=0.9, weight_decay=0)
optimizer = LARC(base_optimizer)
scheduler = opt.lr_scheduler.CyclicLR(base_optimizer, base_lr=0.01, max_lr=0.1)
```

Large Batches Rider



37

Quand on fait du *Data Parallelism*, gérer des gros *batches* est inévitable. Nous allons donc devoir affronter ce problème de *sharp minima*..

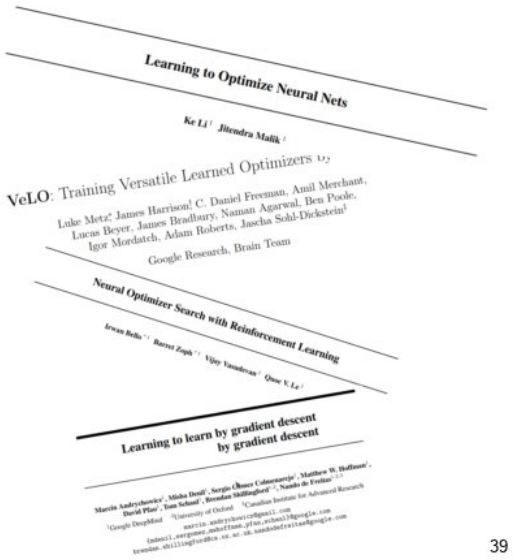
Il existe différents outils pour faire face à cette problématique.

En plus du *weight decay* et des optimiseurs adaptatifs que nous avons vu, il est recommandé d'utiliser des *learning rate schedulers* ainsi que des optimiseur adaptés au *large batches* comme LARS et LAMB que nous allons voir plus tard.

Dans cette partie, nous allons aborder les nouvelles approches existantes concernant les optimiseurs.

En particulier, nous prendrons le temps d'aborder une solution relativement récente : LION ; afin d'en comprendre le fonctionnement, les limites et avantages.

Nouvelle tendance : l’apprentissage d’optimiseur


Learning to Optimize Neural Nets

Ke Li [1] Jitendra Malik [1]

VeLO: Training Versatile Learned Optimizers [1]

Luke Metz, James Harrison, C. Daniel Freeman, Amil Merchant, Lucas Beyer, James Bradbury, Naman Agarwal, Ben Poole, Igor Mordatch, Adam Roberts, Jascha Sohl-Dickstein

Google Research, Brain Team

Neural Optimizer Search with Reinforcement Learning

Learning to learn by gradient descent by gradient descent

39

Il existe aujourd’hui un grand nombre d’optimiseurs. On peut les regrouper en famille et/ou selon des critères selon qu’ils intègrent.

Optimiseurs adaptatifs (Adagrad, RMSprop, Adam): Ajustent le taux d'apprentissage pour chaque paramètre individuellement.

Optimiseurs avec régularisation (AdamW): Intègrent la régularisation directement dans l'optimisation pour prévenir le surapprentissage.

Approches alternatives d'optimisation:

Algorithmes génétiques / de recherche basés sur la population: Explorent l'espace des paramètres en utilisant des concepts de la génétique.

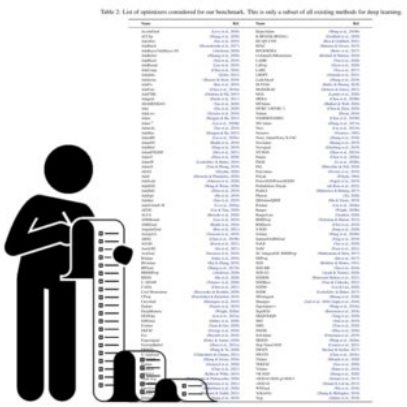
Méthodes quasi-Newtoniennes : Approximent l'inverse de la matrice hessienne pour l'optimisation.

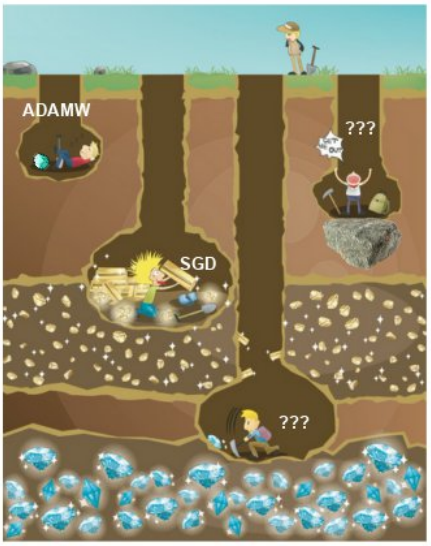
Optimisation bayésienne : Utilise des techniques probabilistes pour prédire la performance d'un modèle.

Réseaux de neurones évolutionnaires (NeuroEvolution) : Utilisent les algorithmes évolutionnaires pour optimiser les architectures de réseaux de neurones.

Chacune de ces méthodes a ses avantages et inconvénients, et leur efficacité peut dépendre de la tâche spécifique à accomplir. Nous allons maintenant explorer en détail un optimiseur récent, analysant ses forces et faiblesses.

Gouffre des nouveaux optimiseur





Schmidt, Robin M., Frank Schneider, and Philipp Hennig. "Descending through a crowded valley-benchmarking deep learning optimizers." *International Conference on Machine Learning*. PMLR, 2021.

L'évaluation des optimiseurs en deep learning ne se limite pas à leur performance brute sur les benchmarks.

Plusieurs autres facteurs entrent en jeu :

- Recherche d'hyperparamètres : Quel effort a été nécessaire pour atteindre les performances annoncées ? Quelle est la sensibilité de l'optimiseur à ces paramètres ?
- Robustesse : L'optimiseur est-il stable face aux variations aléatoires, telles que le changement de "seed" ?
- Objectifs d'entraînement : Cherche-t-on la rapidité d'apprentissage ou la qualité maximale des résultats ?
- Efficacité : Un optimiseur plus rapide mais nécessitant plus d'itérations, ou atteignant de meilleurs résultats mais étant plus lent, est-il réellement plus efficace ?
- Généralisabilité : L'optimiseur est-il performant sur une variété de tâches, de modèles et d'architectures, ou ses performances sont-elles limitées à des cas spécifiques ?

Ces réflexions nous permettent d'aborder l'évaluation des optimiseurs de manière plus complète et nuancée.

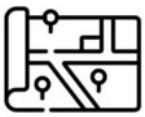
LION : un nouvel optimiseur

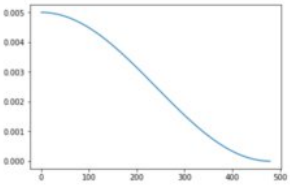
Algorithm 1 AdamW Optimizer

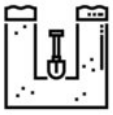
given $\beta_1, \beta_2, \epsilon, \lambda, \eta, f$
initialize $\theta_0, m_0 \leftarrow 0, v_0 \leftarrow 0, t \leftarrow 0$
while θ_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
  update EMA of g_t **and** g_t^2
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  bias correction
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
  update model parameters
  $\theta_t \leftarrow \theta_{t-1} - \eta_t (\hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) + \lambda \theta_{t-1})$
end while
return θ_t

Algorithm 2 Lion Optimizer (ours)

given $\beta_1, \beta_2, \lambda, \eta, f$
initialize $\theta_0, m_0 \leftarrow 0$
while θ_t not converged **do**
  $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
  update model parameters
  $c_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $\theta_t \leftarrow \theta_{t-1} - \eta_t (\text{sign}(c_t) + \lambda \theta_{t-1})$
  update EMA of g_t
  $m_t \leftarrow \beta_2 m_{t-1} + (1 - \beta_2) g_t$
end while
return θ_t







Chen, Xiangning, et al. "Symbolic discovery of optimization algorithms." *arXiv preprint arXiv:2302.06675* (2023).

L'optimiseur LION est un algorithme d'optimisation efficace pour l'entraînement de réseaux de neurones profonds, se distinguant par sa focus sur le "momentum" et son utilisation moindre de ressources, malgré des questions de robustesse. Il effectue une mise à jour uniforme de chaque paramètre, contrairement aux optimiseurs adaptatifs qui varient ces mises à jour selon les paramètres.

Par rapport à d'autres optimiseurs comme Adam, LION est plus efficient en termes de mémoire et semble offrir des solutions plus robustes via une exploration initiale. Il a prouvé son efficacité sur plusieurs tâches, atteignant des scores élevés sur divers ensembles de données et montrant une forte convergence initiale pour des entraînements plus courts.

Toutefois, LION est sensible à ses quelques hyper-paramètres, spécialement à la stratégie du learning rate scheduler affectant la convergence du modèle. Bien qu'il surpasse des optimiseurs comme Adam sur des tâches avec de grands modèles et de petits ensembles de données, son efficacité diminue sur des problèmes avec une grande quantité de données et des règles de régularisation.

TP : Learning rate + Optimiseurs

Objectifs :

- Modifier le learning rate scheduler
- Modifier l'optimiseur
- Faire des entraînements avec des large batches

Depuis JupyterHub :

– Lancer une instance SLURM CPU
– Allez dans le dossier tp_optimiseurs
– Ouvrez le notebook DLO-JZ_Optimiseurs