Optimized Deep Learning - Jean Zay

Training and large batches

INSTITUT DU DÉVELOPPEMENT ET DES RESSOURCES EN INFORMATIQUE SCIENTIFIQUE

Loss Landscape

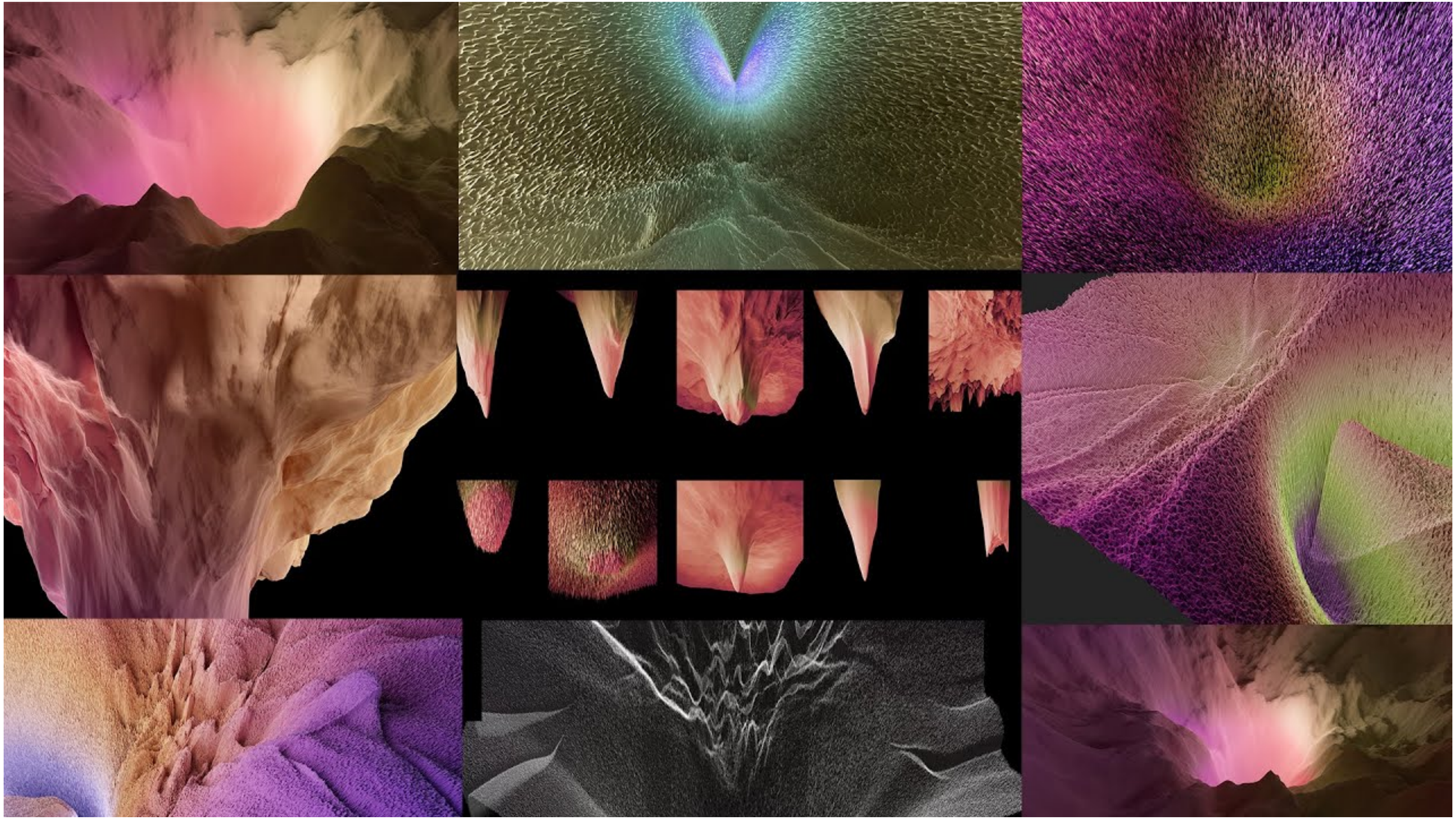
Loss Landscape ◀

Residual Learning ◀

Initialization ◀

Loss Landscape

https://losslandscape.com/

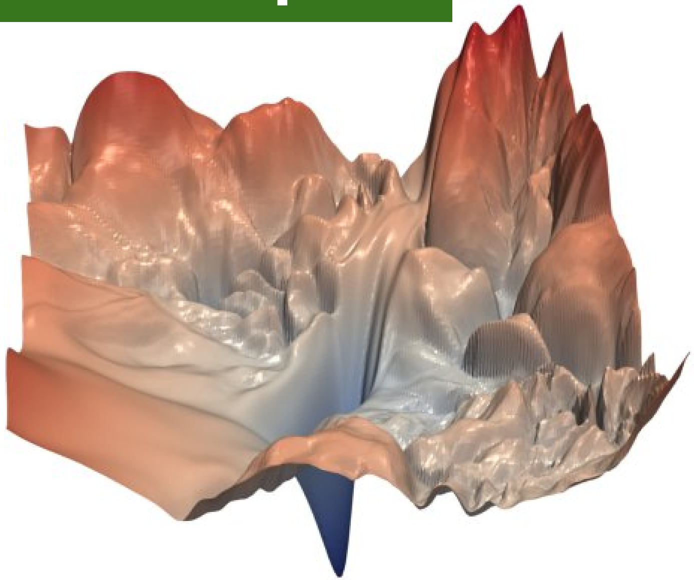


Loss Landscape

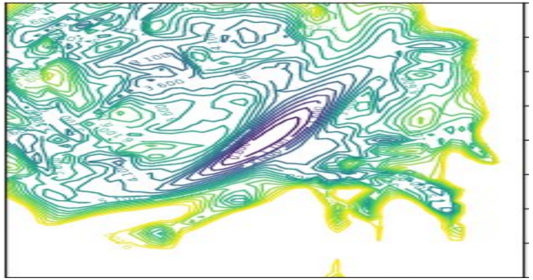
https://arxiv.org/pdf/1712.09913.pdf

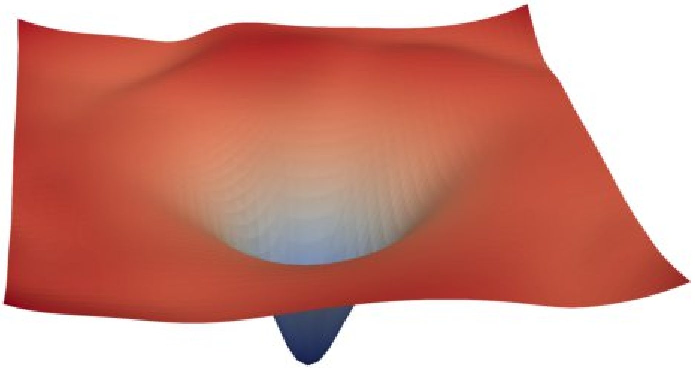
Residual Learning

Since Resnets (2015) ...

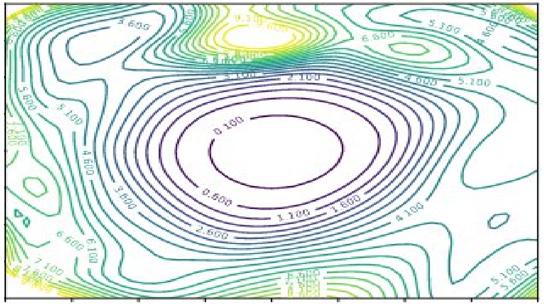


(a) without skip connections

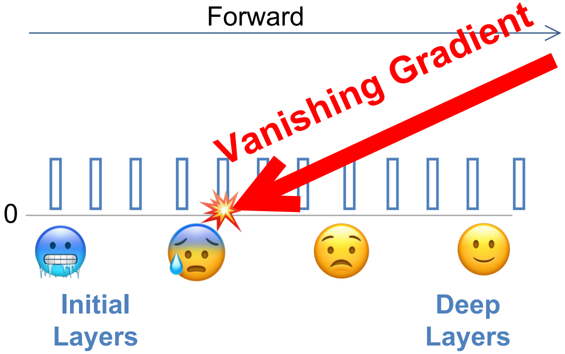




(b) with skip connections

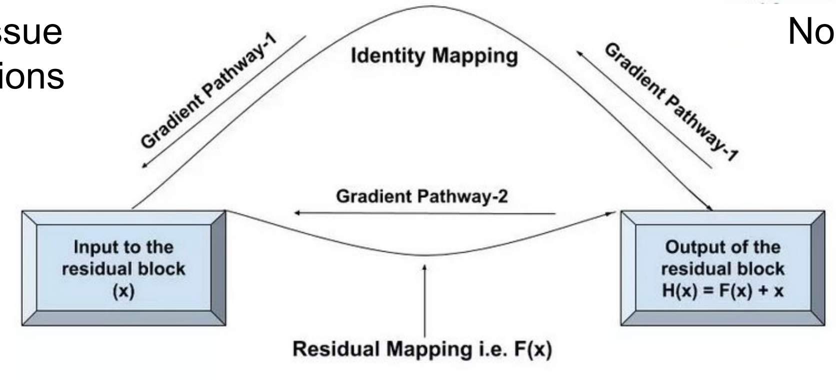


Residual Learning

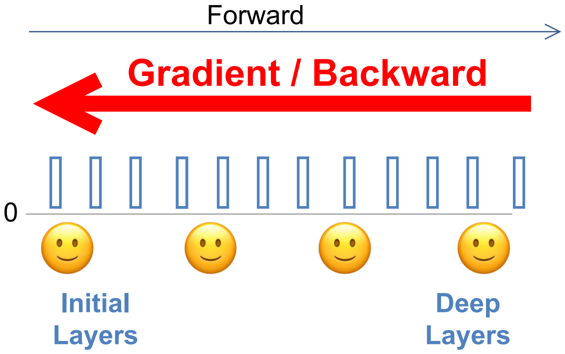


Vanishing Gradient issue
without skip connections

Residual Block
F(x)+x



Gradient Pathways in ResNet

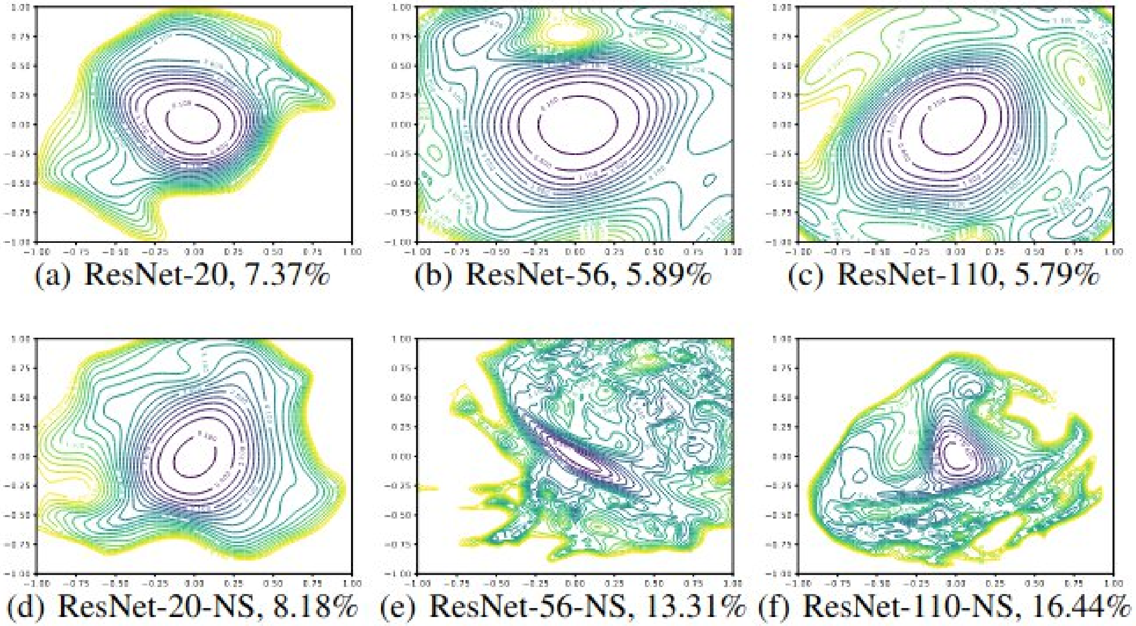


No Vanishing Gradient issue
with skip connections

Residual Learning – depth impact

ResNet

ResNet -
No Short
without skip connections

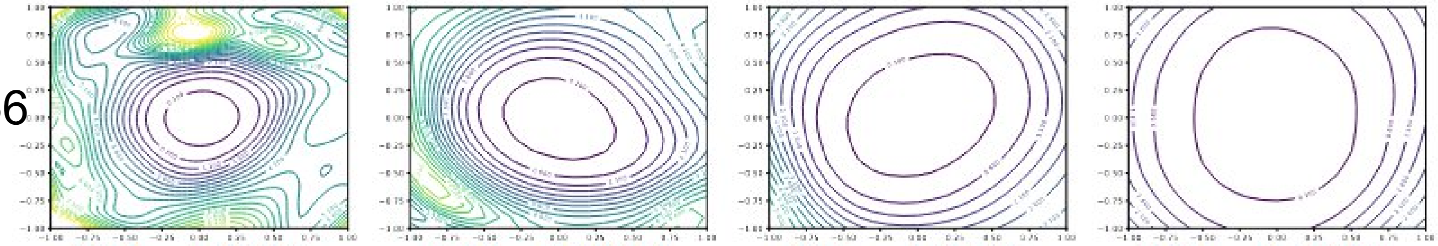


(a) ResNet-20, 7.37% (b) ResNet-56, 5.89% (c) ResNet-110, 5.79%

(d) ResNet-20-NS, 8.18% (e) ResNet-56-NS, 13.31% (f) ResNet-110-NS, 16.44%

Residual Learning – width impact

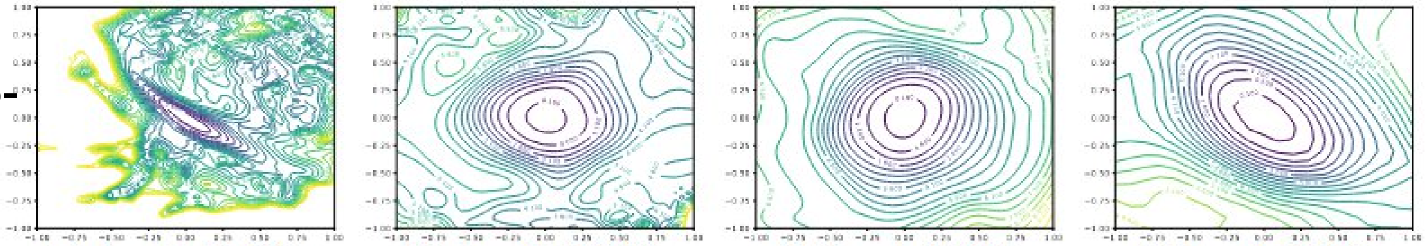
Wide-ResNet-56



(a) $k = 1$, 5.89% (b) $k = 2$, 5.07% (c) $k = 4$, 4.34% (d) $k = 8$, 3.93%

Wide-ResNet-56-No Short

without skip connections

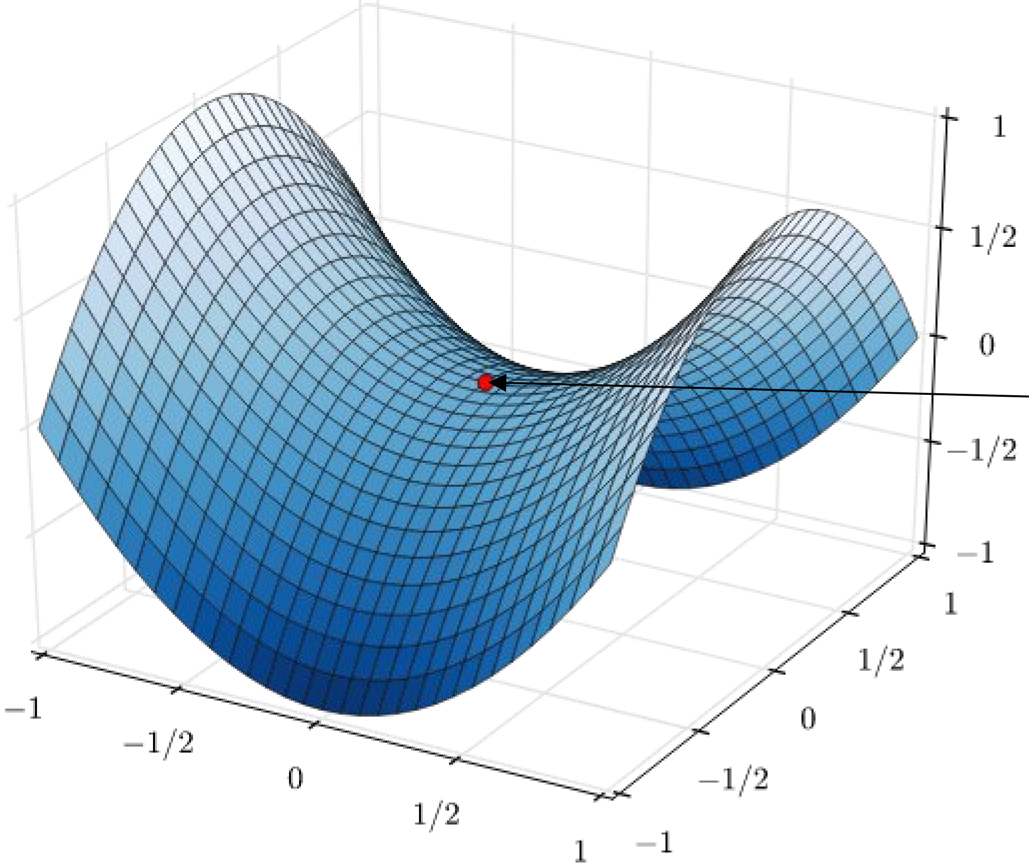


(e) $k = 1$, 13.31% (f) $k = 2$, 10.26% (g) $k = 4$, 9.69% (h) $k = 8$, 8.70%

k = channel width coefficient compared to ResNet

Saddle point Problem

Saddle point: A gradient close to zero which will make the progression of the model very slow



How do I leave the saddle point?

Model Parameters Initialization

The Blessing of Dimensionality :

FINDING A MINIMA BECOMES A “LOCAL” CHALLENGE

- Xavier Initialization
  - uniform
  - normal

- Kaiming Initialization
  - uniform
  - normal

By default in PyTorch:

- Best initialization algorithm depending on the type of layer (linear, convolutional, transform, ...).
- Today, it is no longer necessary to try to optimize initialization.

Learning Rate Scheduler

Learning Rate scheduler ◂

Cyclic scheduler ◂

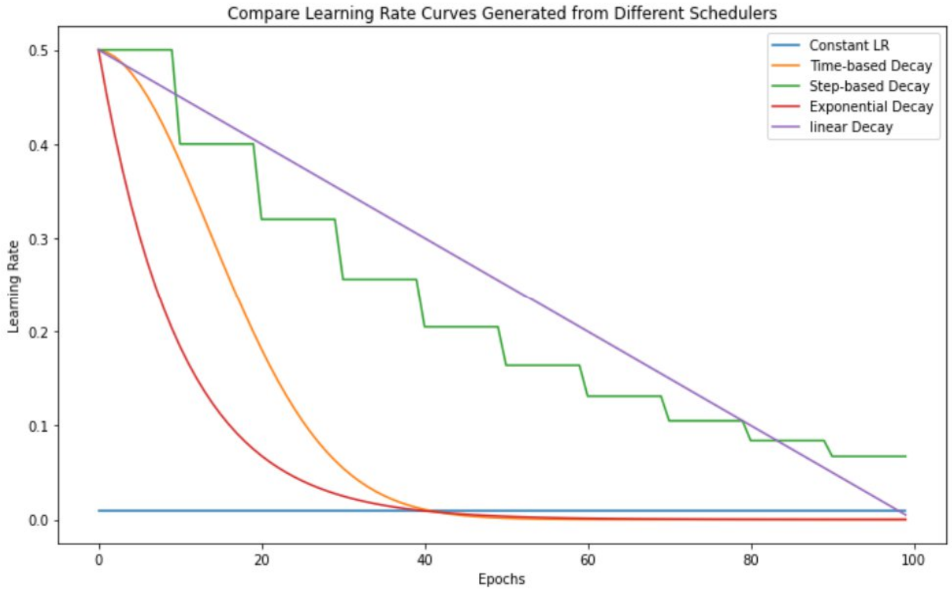
One Cycle scheduler ◂

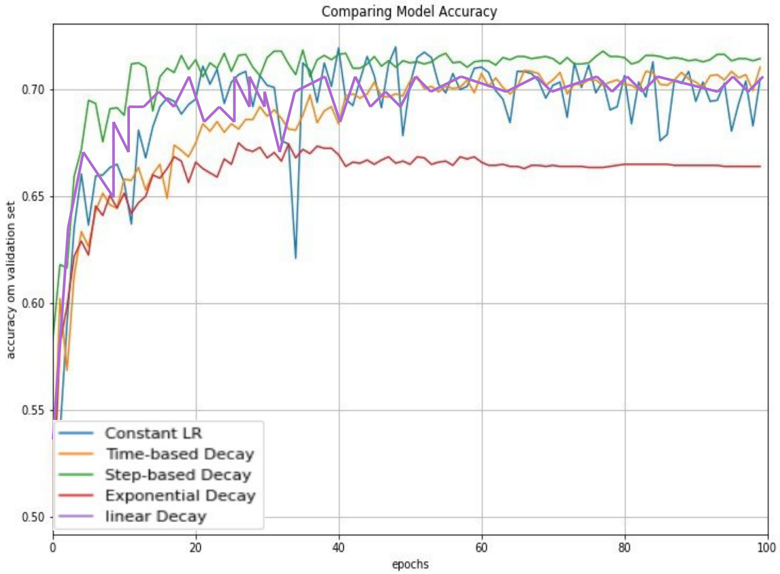
ScheduleFree ◂

LR Finder◂

Learning Rate Scheduler

Learning rate decay

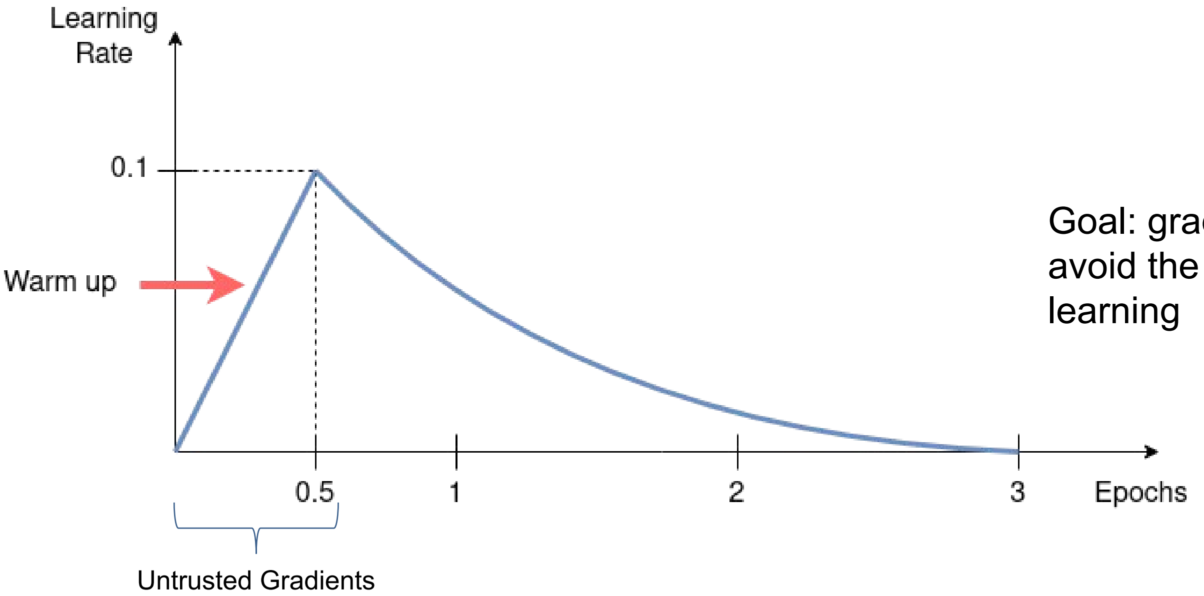




Learning Rate Scheduler

WARMUP for *large batches*

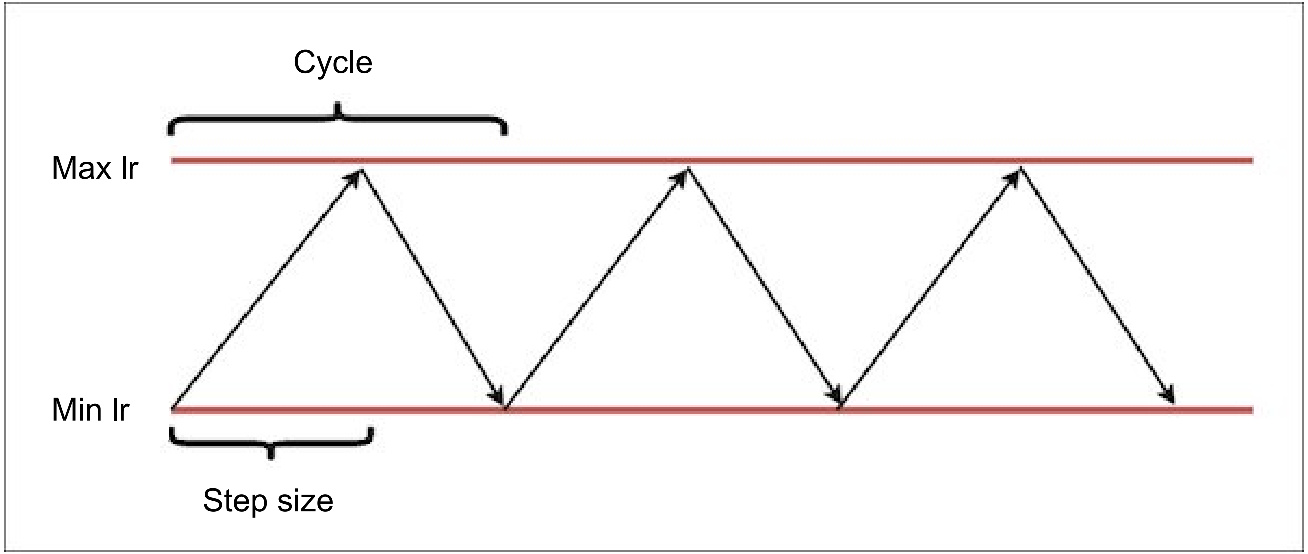
Problems: The first iterations have too much effect on the model (significant losses, high gradients, bias, etc.), a high learning rate can cause strong instability or divergence



Goal: gradually increase the learning rate to avoid the risk of divergence at the start of learning

Cyclic Learning Rate Scheduler

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

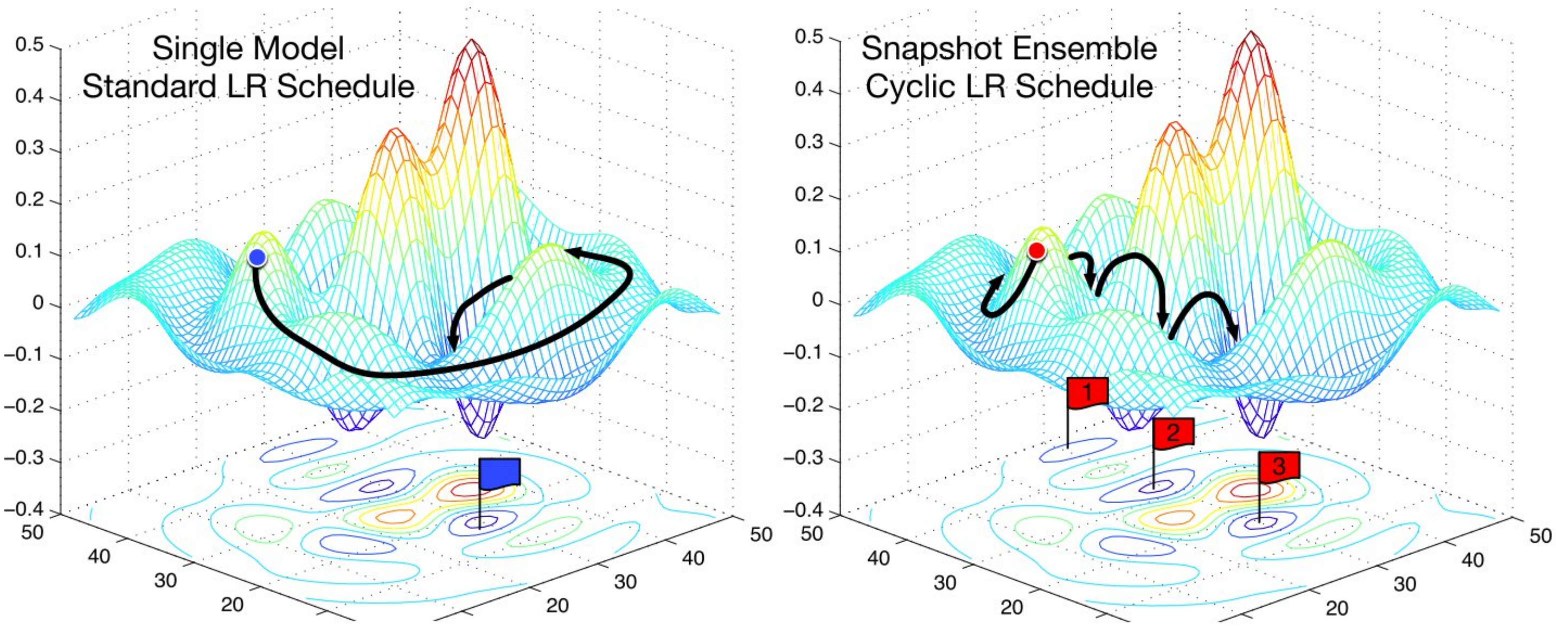


Paramètres :

- Step_size = x * epoch ($2 \leq x \leq 10$)
- Base_lr -> min convergence value
- max_lr -> max convergence value

Succession of warmups and learning rate decays

Cyclic Learning Rate Scheduler



SNAPSHOT ENSEMBLES: TRAIN 1, GET M FOR FREE

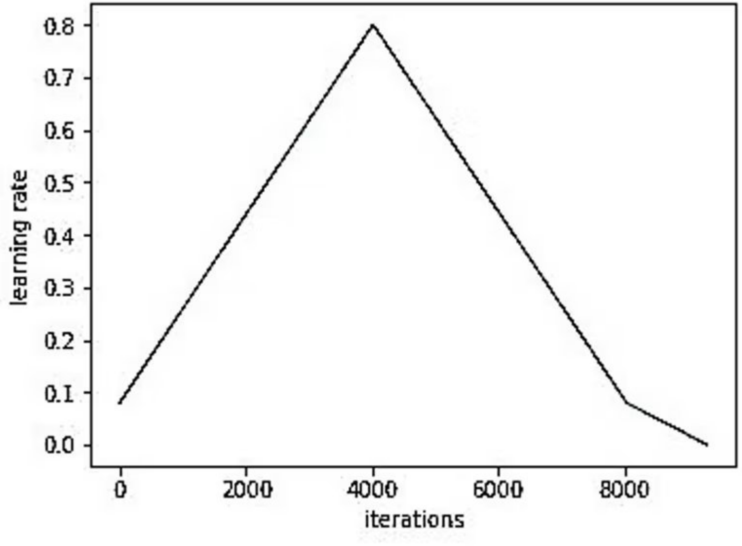
Gao Huang, Yixuan Li, Geoff Pleiss

One Cycle Learning Rate

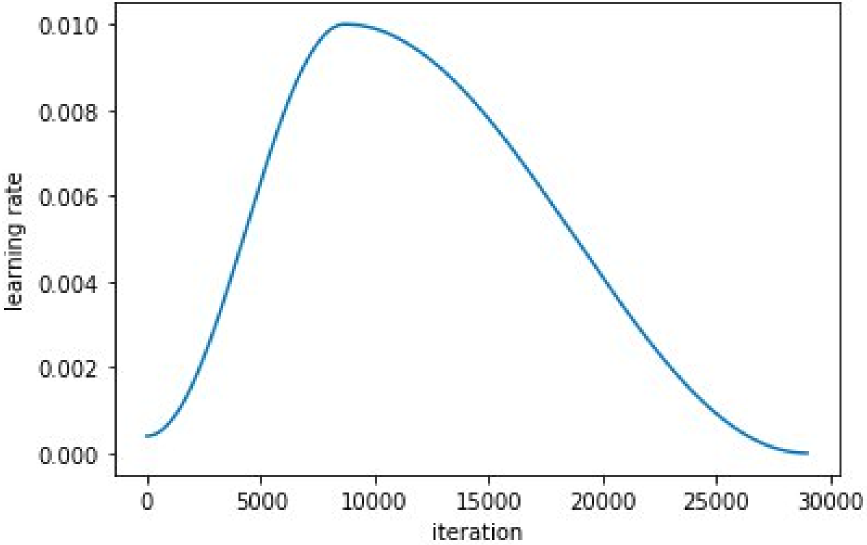
One cycle is enough!

A disciplined approach to neural network hyper-parameters - [Leslie N. Smith]

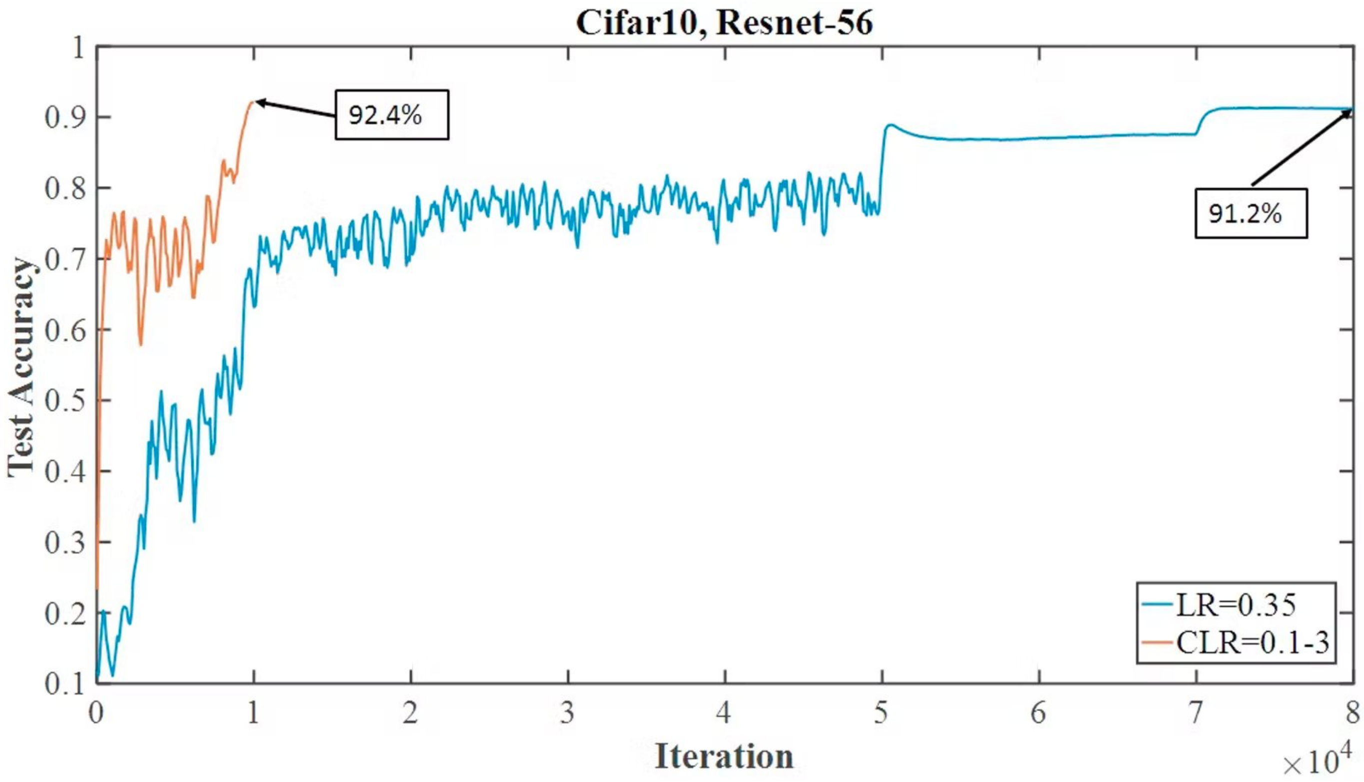
Proposition initiale



cosine annealing : Recommandation par FastAI



One Cycle Learning Rate - *Super convergence*



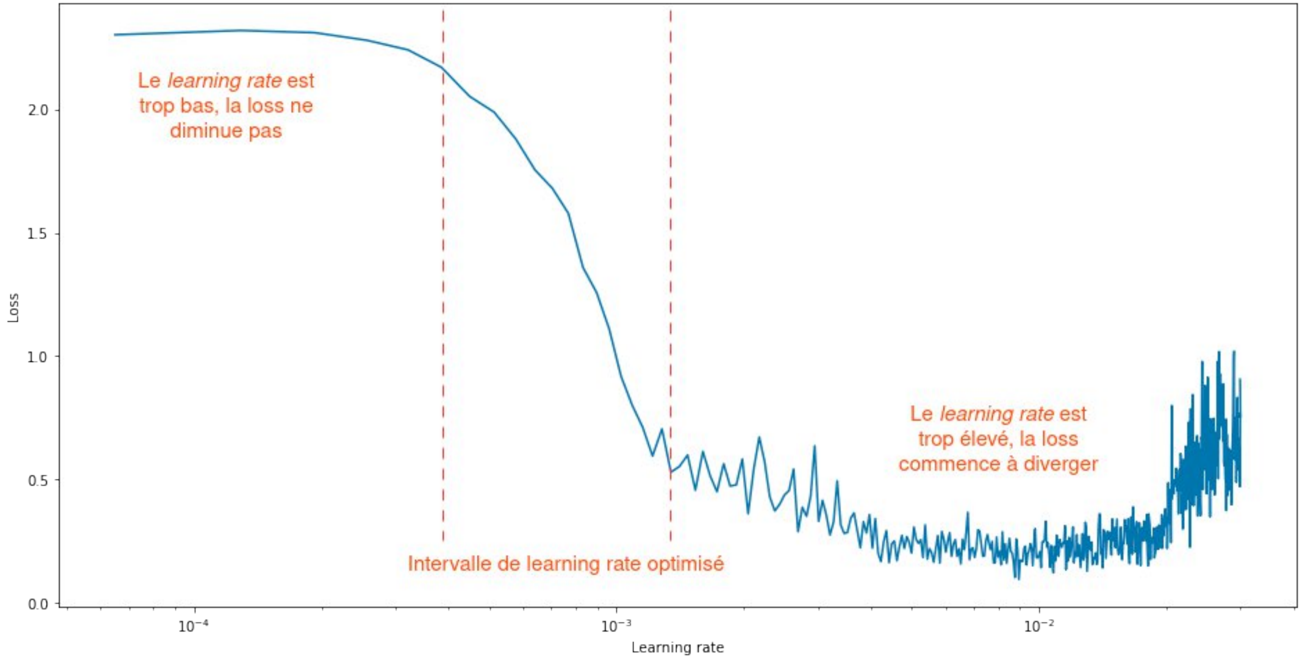
Faster convergence for equivalent final precision

Learning Rate Finder

Goal: Find the **optimal learning rate** values for your model, particularly for **the maximum value** of a *cyclic scheduler*

Run your model over a few epochs by increasing its learning rate

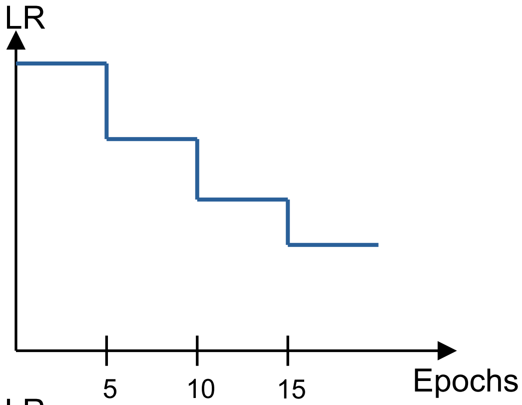
- **Start of loss reduction → Minimal learning rate**
- **Start of loss variation → Maximum learning rate**



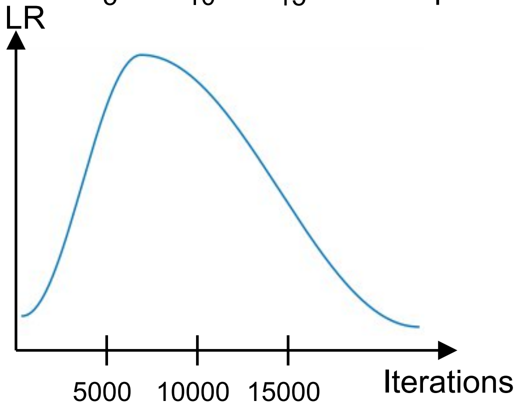
Learning Rate Scheduler

Each **scheduler** has *its own settings*

```python
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()
```




```python
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()
```



The Road Less Scheduled : ScheduleFree

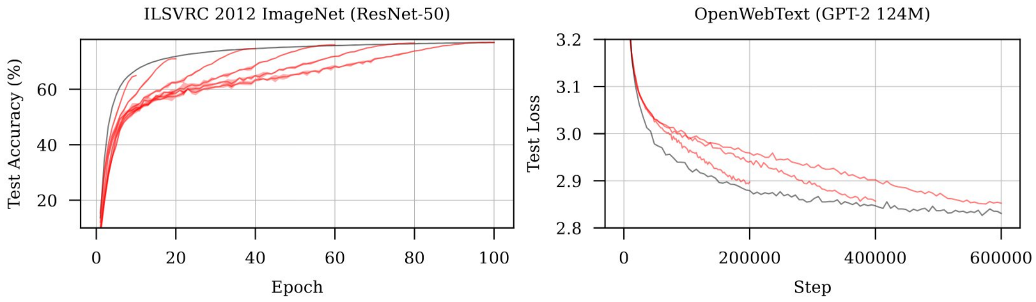


Figure 1: Schedule-Free methods (black) closely track the Pareto frontier of loss v.s. training time in a single run. Both Schedule-Free SGD (left) and AdamW (right) match or exceed the performance of cosine learning rate schedules of varying lengths (red).

Algorithm 1 Schedule-Free AdamW

Input: x_1, learning rate γ, decay λ, warmup steps $T_{\text{warmup}}, \beta_1, \beta_2, \epsilon$
$z_1 = x_1$
$v_0 = 0$
for $t = 1$ **to** T **do**
  $y_t = (1 - \beta_1)z_t + \beta_1 x_t$
  $g_t \in \partial f(y_t, \zeta_t)$
  $v_t = \beta_2 v_{t-1} + (1 - \beta_2)g_t^2$
  $\hat{v}_t = v_t/(1 - \beta_2^t)$
  $\gamma_t = \gamma \min(1, t/T_{\text{warmup}})$
  $z_{t+1} = z_t - \gamma_t g_t/(\sqrt{\hat{v}_t} + \epsilon) - \gamma_t \lambda y_t$
  $c_{t+1} = \frac{\gamma_t^2}{\sum_{i=1}^t \gamma_i^2}$
  $x_{t+1} = (1 - c_{t+1})\, x_t + c_{t+1} z_{t+1}$
end for
Return x_{T+1}

https://arxiv.org/pdf/2405.15682
https://github.com/facebookresearch/schedule_free

Gradient Descent Optimizer

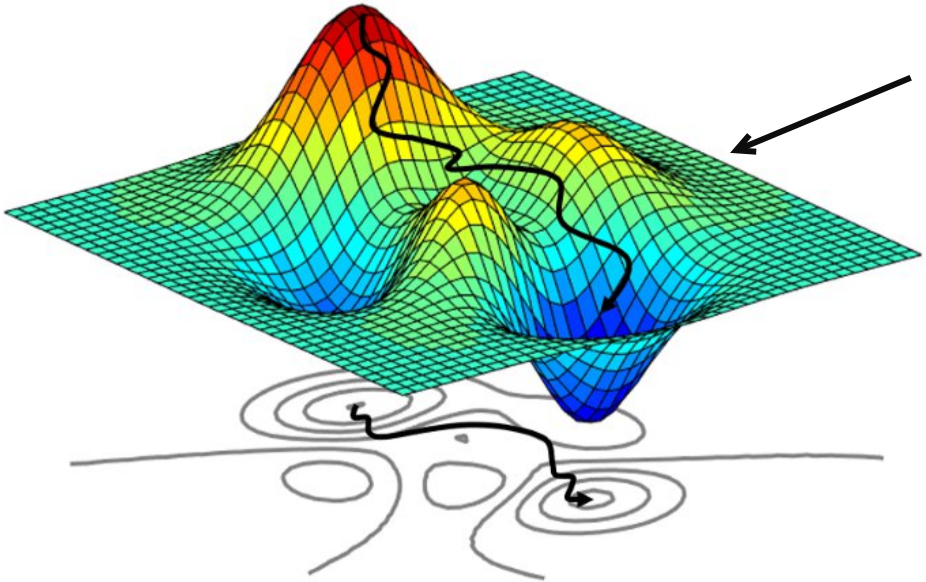
SGD ◄

ADAM◄

ADAMW ◄

Optimizer - SGD

The **optimizer** is the algorithm that **controls the gradient descent** and **the minimum search** with the aim of optimizing the learning time and the final metric.



***SGD = S**tochastic **G**radient **D**escent*

Calculating the Gradient and updating the weights at each batch

\+ Batch size and learning rate adaptable according to conflicting needs:

- Exploration to find the best local minimum
- Acceleration of gradient descent

SGD *with Momentum*

Momentum coefficient

$$m_0 = 0$$

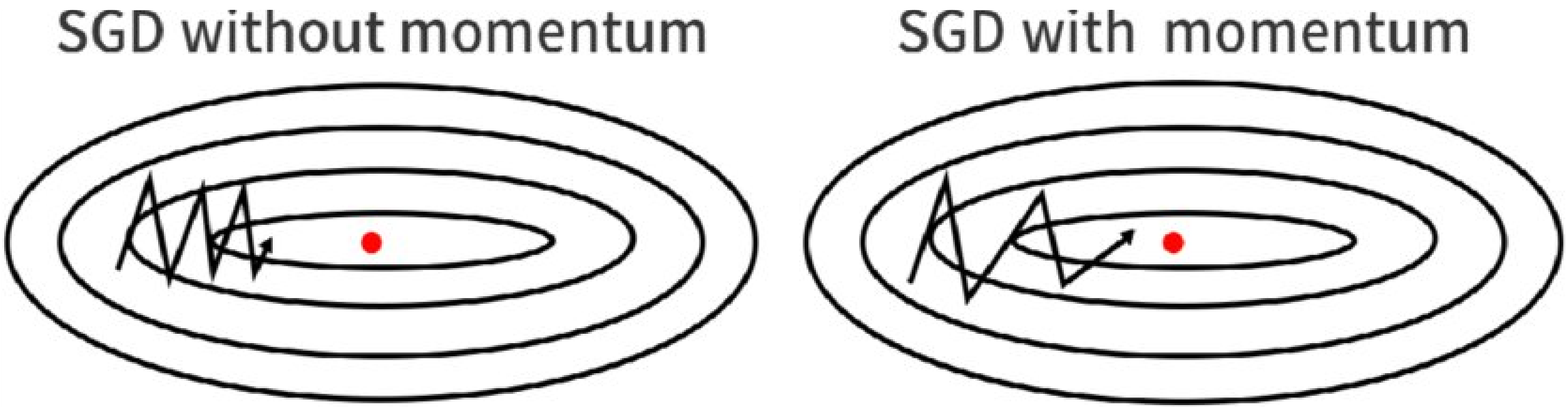
$$m_i = \boxed{\beta} * m_{i-1} + (1-\beta) * g_i$$

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

Goal: Take **previous gradients** into consideration for **faster** gradient descent.

Recommended initial value: *0.9*

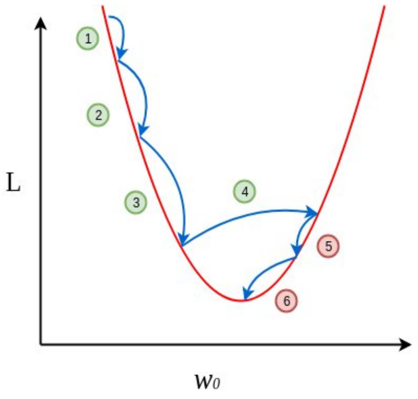
$$0.85 < \beta < 0.95$$



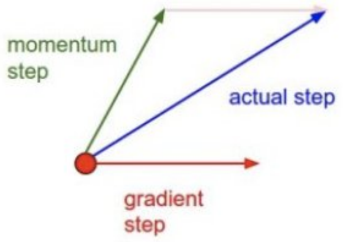
+ Allows you to **converge more quickly**
- **No guarantee** that momentum will take us in the right direction

Momentum type

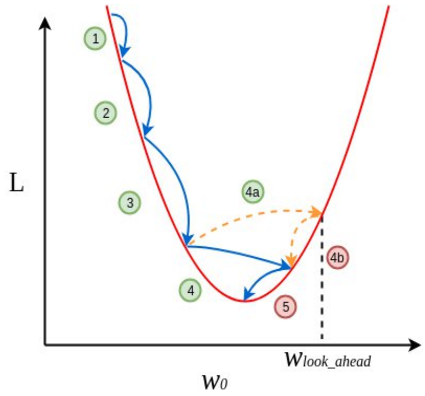
Momentum



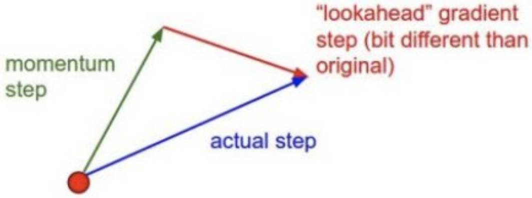
Momentum update

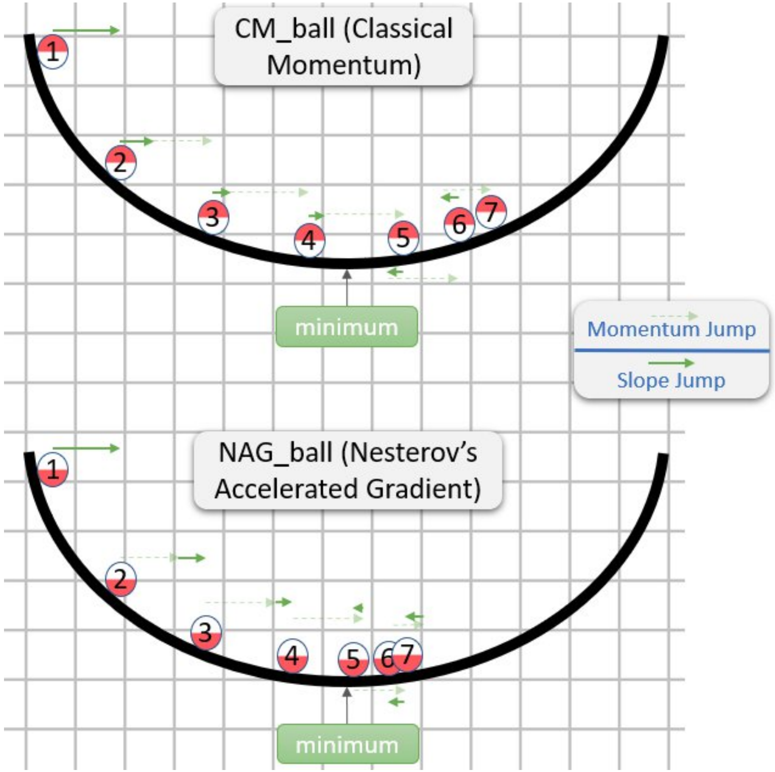


Nesterov momentum

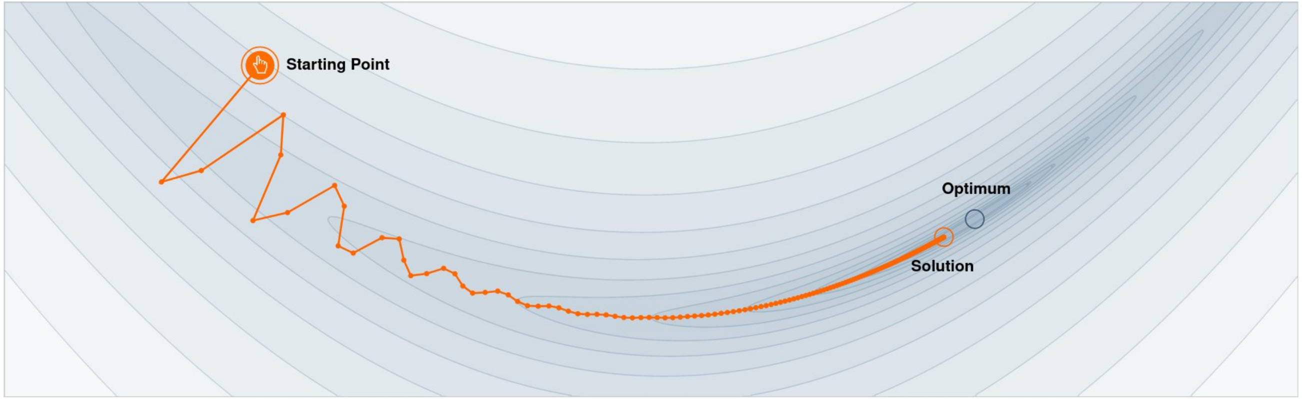


Nesterov momentum update





Why Momentum Works ?



Step-size α = 0.02

0 | 0.003 | 0.006

Momentum β = 0.99

0.00 | 0.500 | 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

GABRIEL GOH
UC Davis

April. 4
2017

Citation:
Goh, 2017

https://distill.pub/2017/momentum/

Adaptive Optimizers

Rather than controlling the gradient descent manually with the learning rate...

... We can adapt the *learning rate* **for each weight** of the model according to the **gradient**, the **gradient2**, or the **norm of the weights** of the layer!!!

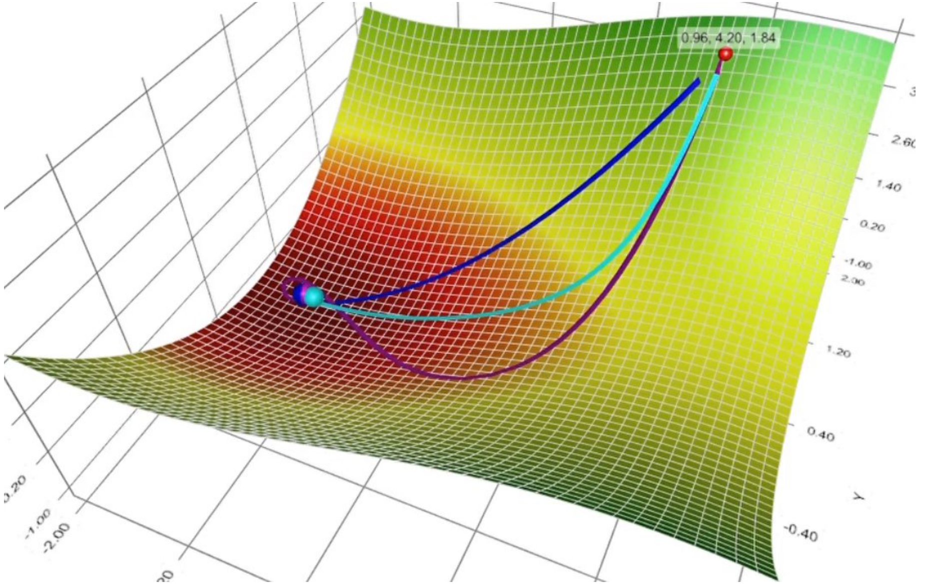
Examples :

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

Specialized for larges batches :

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

SGD (no *momentum*)
SGD (with *momentum*)
Adam



Adam

Adam : Adaptative moment estimation

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

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

First moment : sliding mean

Second moment : sliding non-centered variance

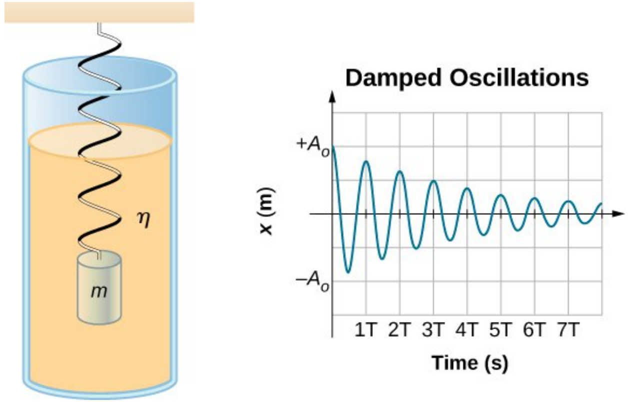
$$\hat{m}_i = \frac{m_i}{1 - \beta_1^i}$$

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

Correction of biases of the first iterations

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

Dampens oscillations



Parameters:

β_1 & β_2 = Regression rate (β_1 = 0.9 & β_2 = 0.999)
ε = Very small value to avoid division by zero

Goal: **Adapt the importance of weight** updates based on previous gradients and gradient variability.

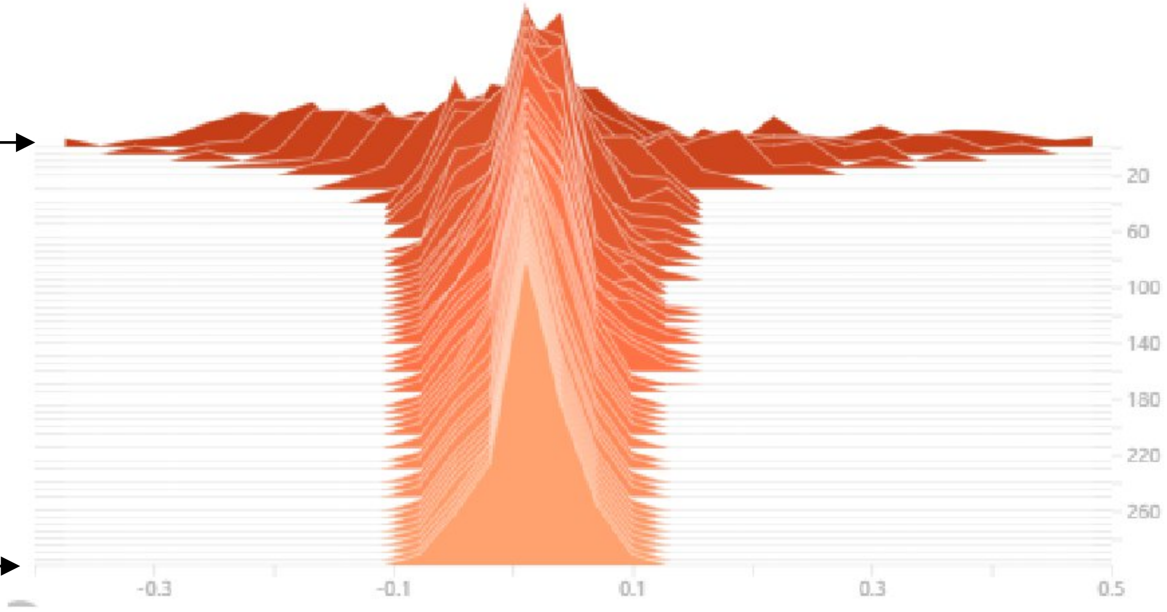
Weight decay

A neural network that **converges and generalizes correctly*** generally has weights **that tend to 0**. *(neither underfitting nor overfitting)

Distribution of weights during learning:

Beginning

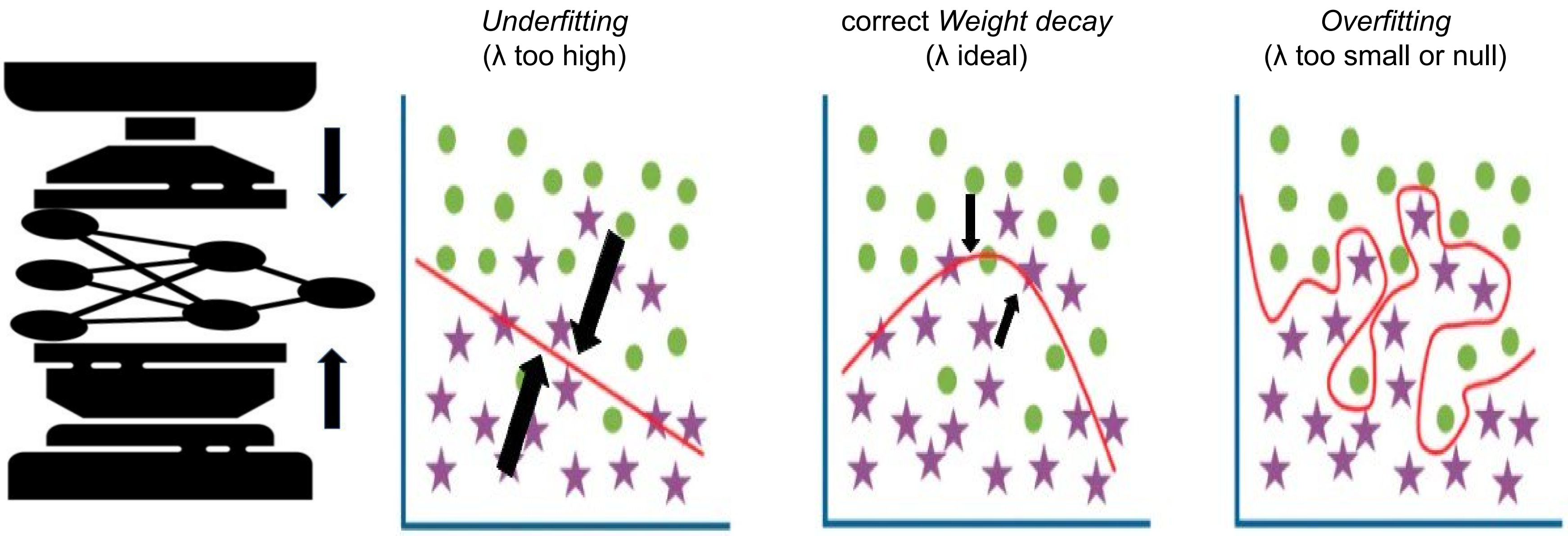
End



Weight decay

Preferable to standard L2 regularization defined in loss function

λ : *weight decay* parameter *(between 0 and 0.1)*



The weight decay technique, defined in the optimizer, makes it possible to force the weights to converge towards values close to zero.

Weight decay and decoupled weight decay

ADAM

For i = 1 to ...

$$g_i = \nabla_\theta f_i(\theta_{i-1}) \boxed{+ \lambda\theta_{i-1}}$$
$$m_i = \beta_1 * m_{i-1} + (1-\beta_1) * g_i$$
$$v_i = \beta_2 * v_{i-1} + (1-\beta_2) * g_i^2$$
$$\hat{m}_i = \frac{m_i}{1-\beta_1^i}$$
$$\hat{v}_i = \frac{v_i}{1-\beta_2^i}$$
$$\theta_i = \theta_{i-1} - \frac{\alpha}{\sqrt{\hat{v}_i + \epsilon}} * \hat{m}_i$$

Return θ_i

Weight decay

ADAMW

For i = 1 to ...

$$g_i = \nabla_\theta f_i(\theta_{i-1})$$
$$m_i = \beta_1 * m_{i-1} + (1-\beta_1) * g_i$$
$$v_i = \beta_2 * v_{i-1} + (1-\beta_2) * g_i^2$$
$$\hat{m}_i = \frac{m_i}{1-\beta_1^i}$$
$$\hat{v}_i = \frac{v_i}{1-\beta_2^i}$$
$$\theta_i = \theta_{i-1} - \frac{\alpha}{\sqrt{\hat{v}_i + \epsilon}} * \hat{m}_i \boxed{- \alpha\lambda\theta_{i-1}}$$

Return θ_i

Decoupled weight decay

Evolution of weight decay:
Decoupled weight decay (decoupled from momentum!!)

- *SGD* and *Adam* with weight decay
- *SGDW* and *AdamW* with decoupled weight decay

SGD and SGDW are **roughly equivalent** in performance.

However **AdamW is noticeably better than Adam!!**

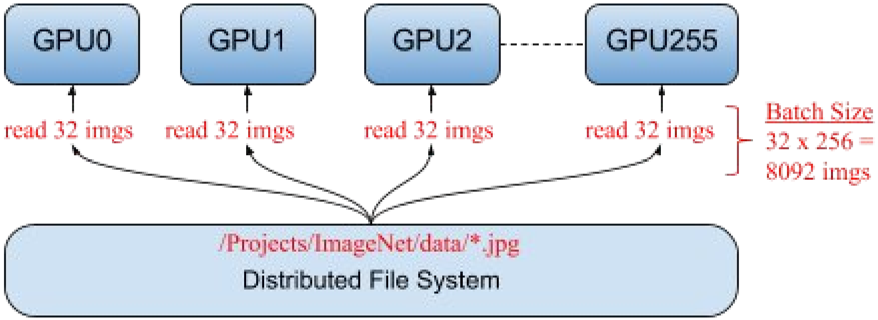
Optimization of large batches

Large batches issues ◂

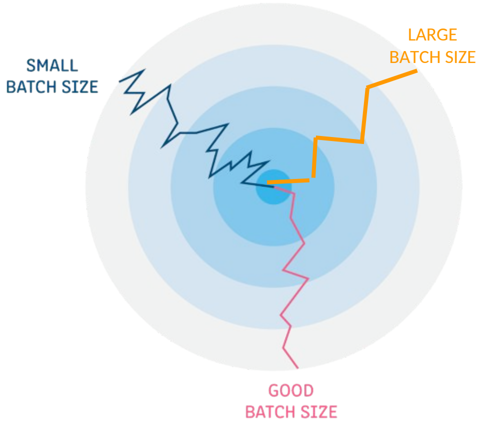
Learning Rate Scaling & Batch Schedulers ◂

Large batches optimizers ◂

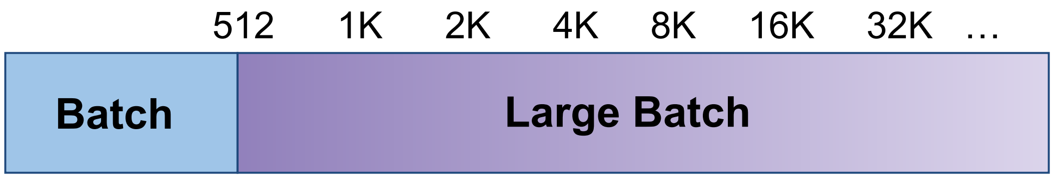
Large Batches with Data Parallelism



Data Parallelism: This parallelism generates very large batches



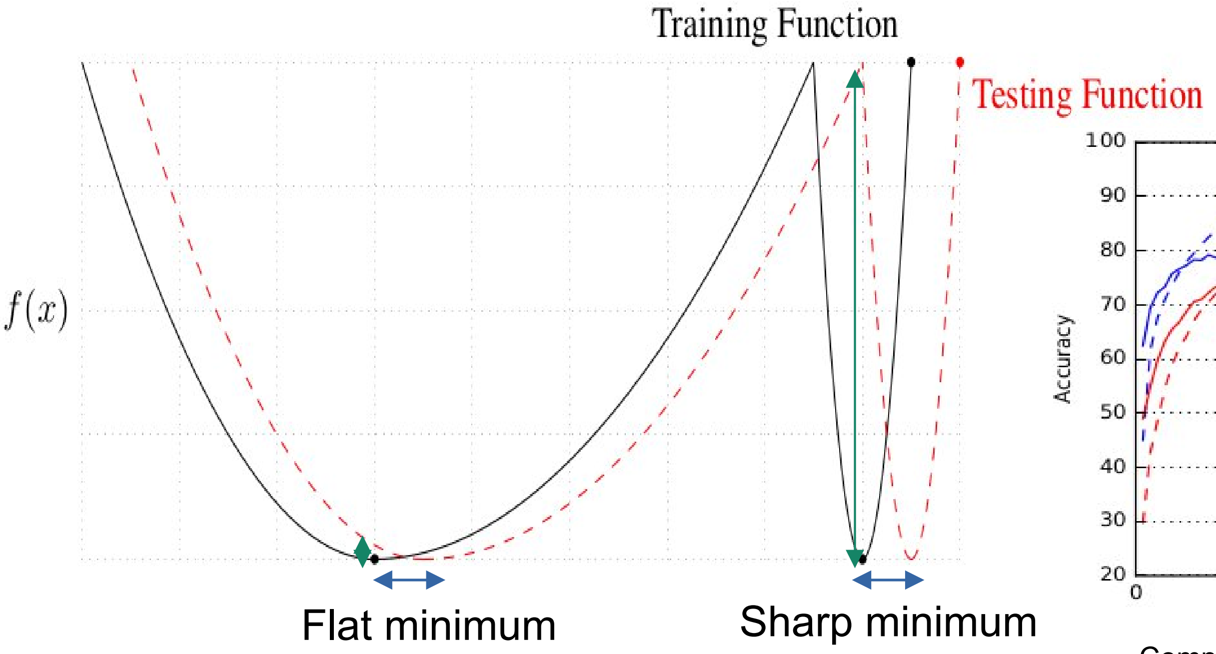
Problem: *Batch* that are too large (> 512) tend to result in **poorer performance**

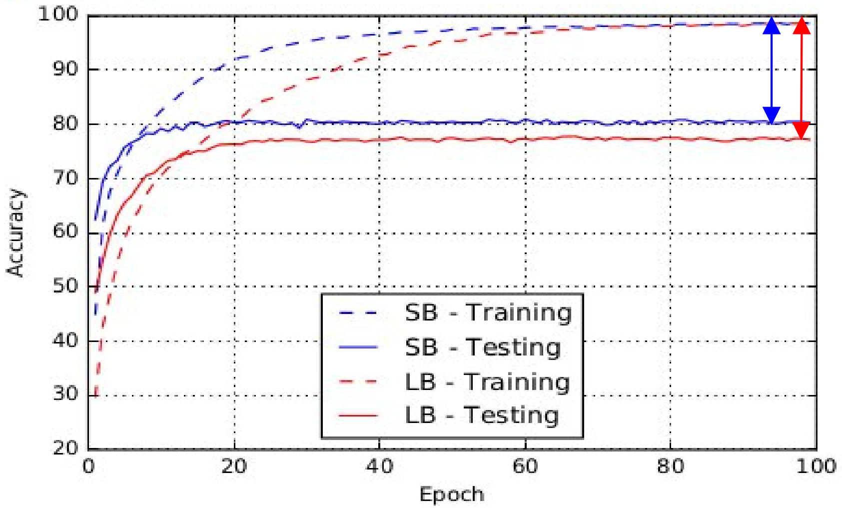


Large Batches

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





Comparison of training a convolutional network with small batch (SB) and large batch (LB) on CIFAR 10

The *larger the batch*, the more the model tends to converge towards **steep and narrow minima**.

Large Batches : Learning rate scaling

When the *size of the global* batch **is considerably increased**, it is often *necessary* to **scale the learning rate**:

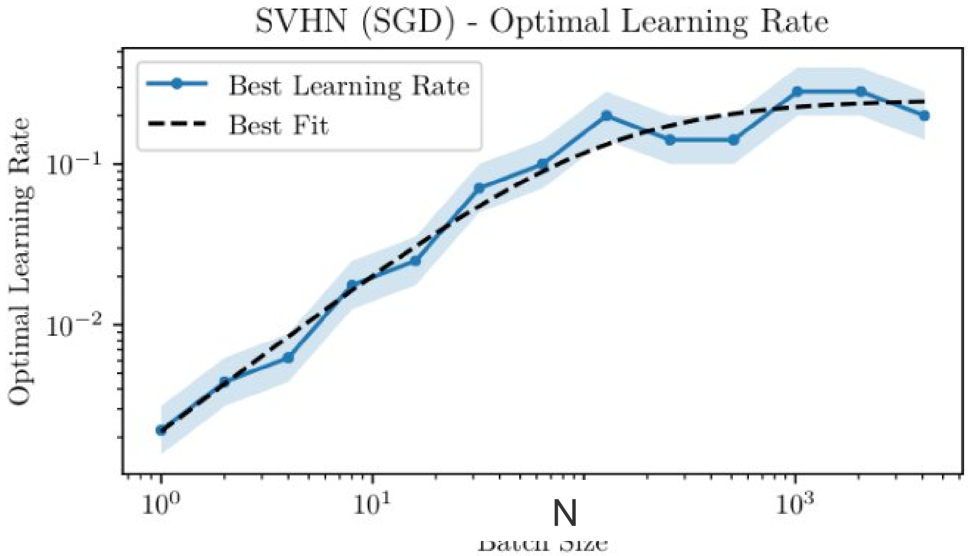
N = Number of parallel processes

Linear growth of *learning rate*:

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

Square root growth of *learning rate*:

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



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

Optimal: **linear growth** at first then **square root** (recommended by OpenAI)

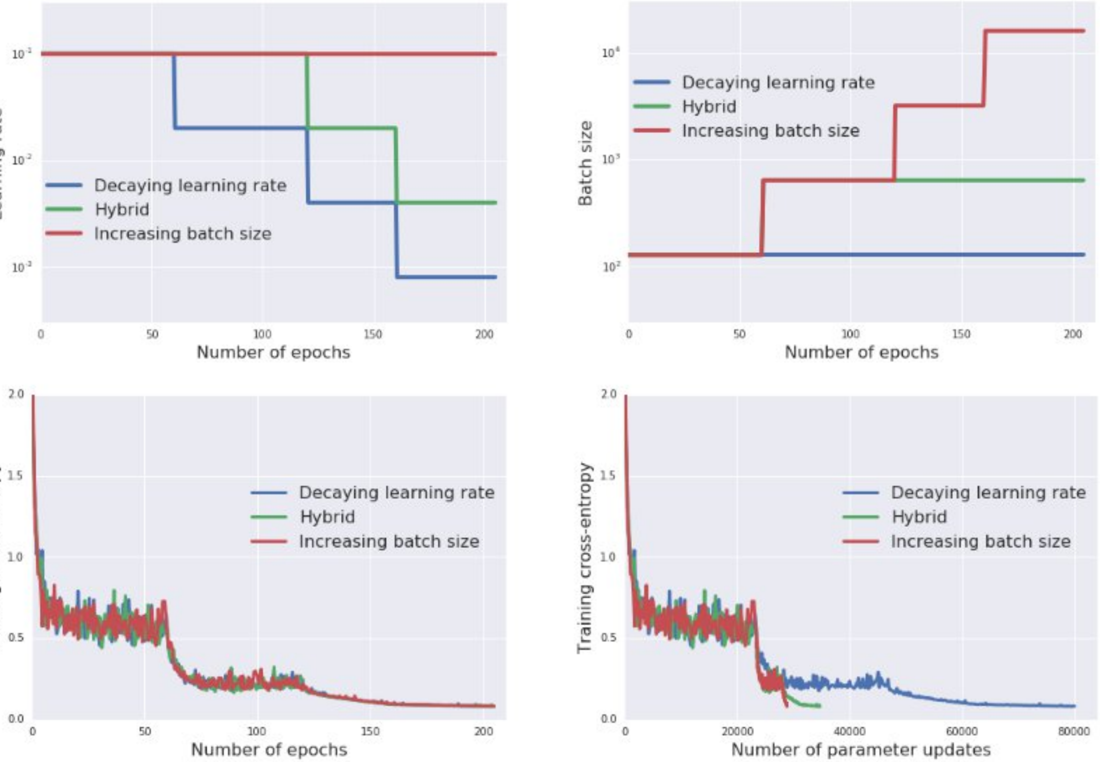
Batch Size Scheduler

=> Alternative to Learning Rate Scheduler

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

High Learning Rate

Low Learning Rate



Large Batch

Small Batch

Large Batches

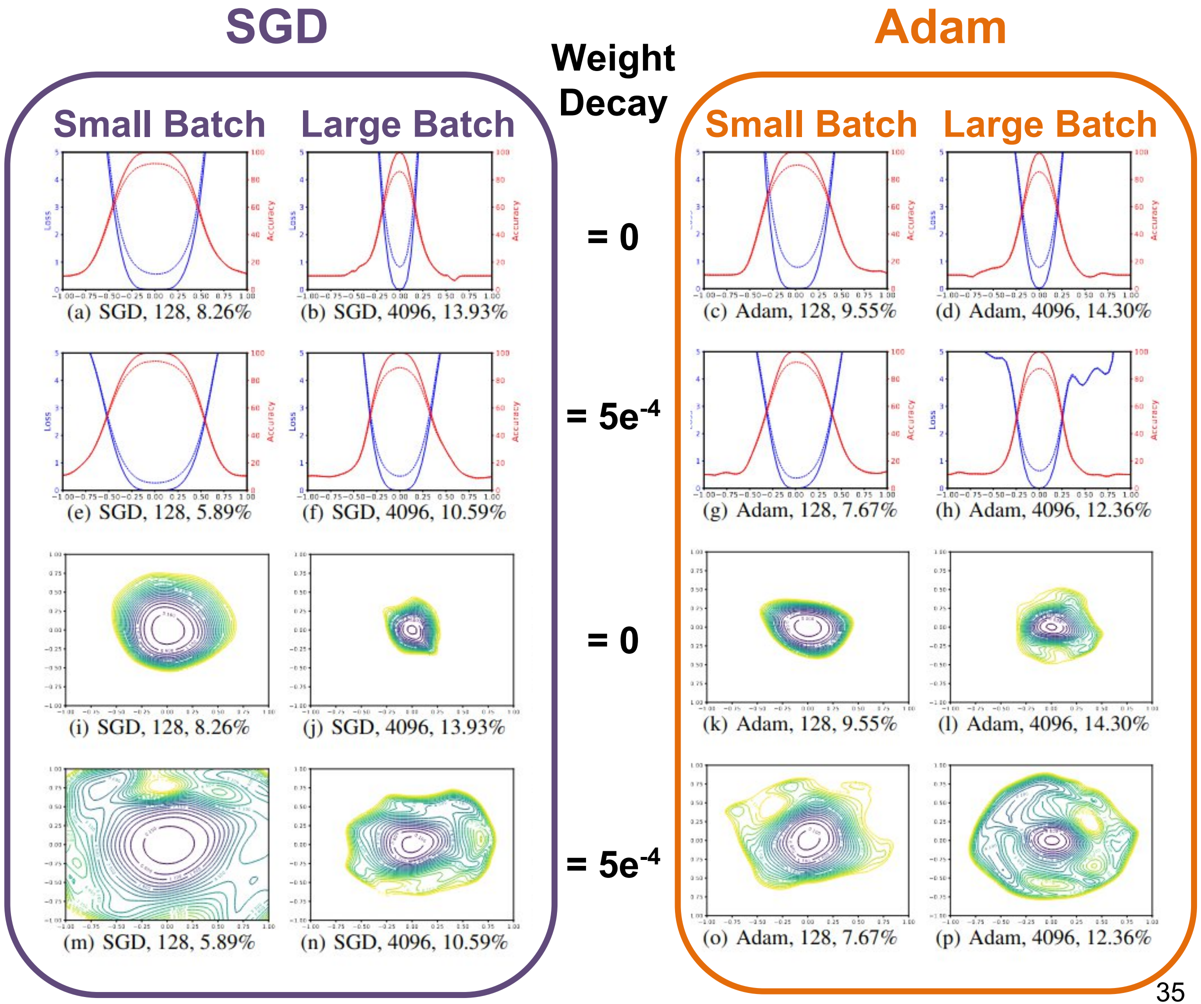
Trends :

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

Small Batch Large Batch

SGD ADAM

Weight Decay w/o W.Decay



Large Batches Optimisers - LARS

LARS = “Layer-wise Adaptive Rate Scaling.”

Adaptation of SGD with momentum with the addition of a **trust ratio** for **each layer** which depends on the evolution of the layer's gradient

r = Trust ratio
l = layer number

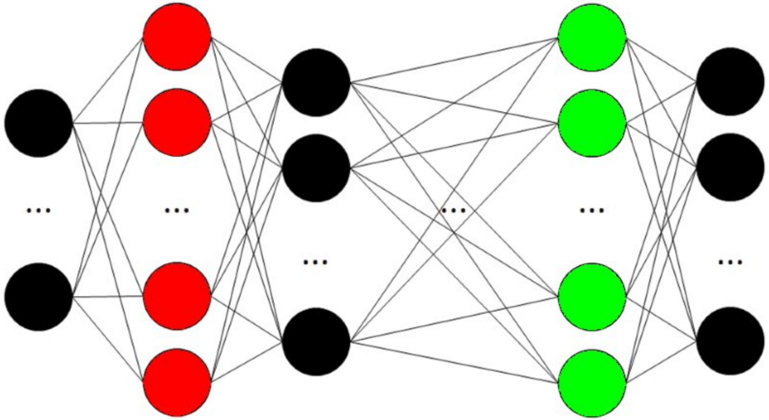
$$m_i = \beta * m_{i-1} + (1-\beta) * (g_i + \underbrace{\lambda\theta_{i-1}}_{\text{Weight decay}})$$

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

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

Low confidence
Low value of **r**

High confidence
High value of **r**



Goal: Adapt the importance of weight updates based on a **trust ratio** calculated for each layer of the network.

Large Batches Optimisers - LAMB

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

Adaptation of ADAM with momentum with the addition of a **trust ratio** for **each layer** which depends on the evolution of the layer's gradient

r = Trust ratio
l = layer number

$$r_1 = \| \theta^l_{i-1} \|_2$$

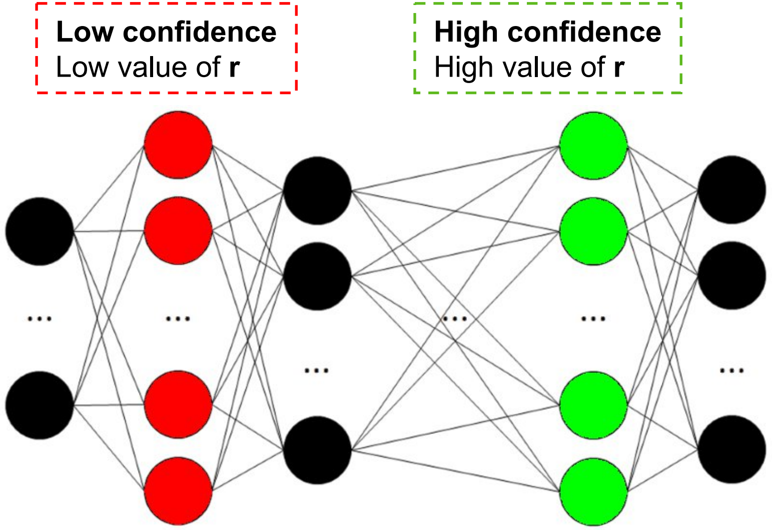
$$r_2 = \| \frac{\hat{m}^l_i}{\sqrt{\hat{v}^l_i + \epsilon}} + \lambda\theta^l_{i-1} \|_2$$

$$r = r_1 / r_2$$

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

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

Decoupled weight decay



Goal: Adapt the importance of weight updates based on a **trust ratio** calculated for each layer of the network.

Optimizer implementation

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

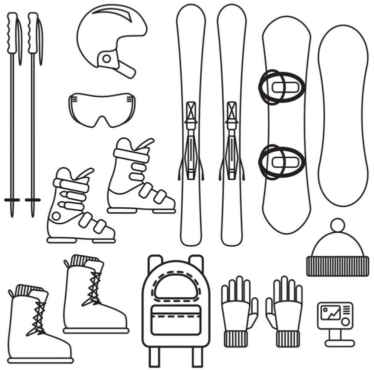
LARS optimisation from APEX

```
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

SGD AdamW

Weight Decay



LARS

LAMB

Batch Scheduler

LR Scheduler

LR scaling

Warmup

LR Decay

BLOOM example

BLOOM
176B params · 59 languages · Open-access

95281 steps (116.8 days)

AdamW,

β1=0.9, β2=0.95, eps=1e−8

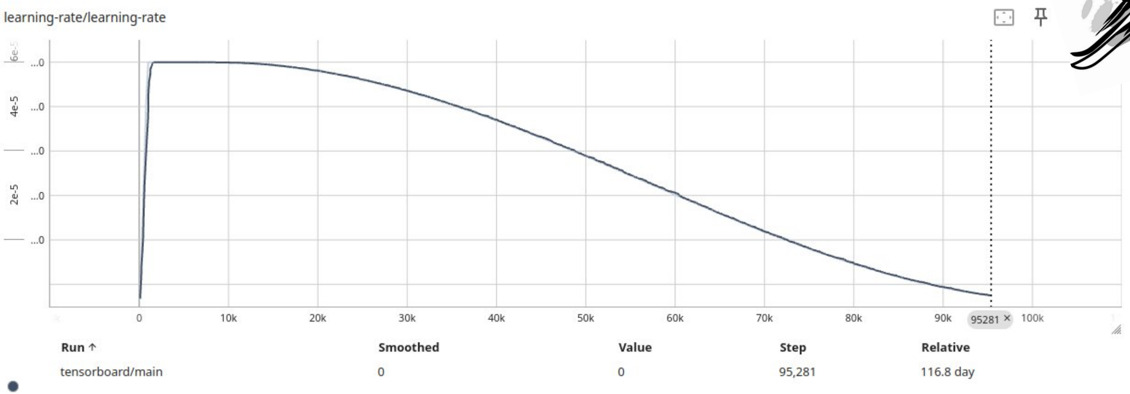
Weight Decay of 0.1

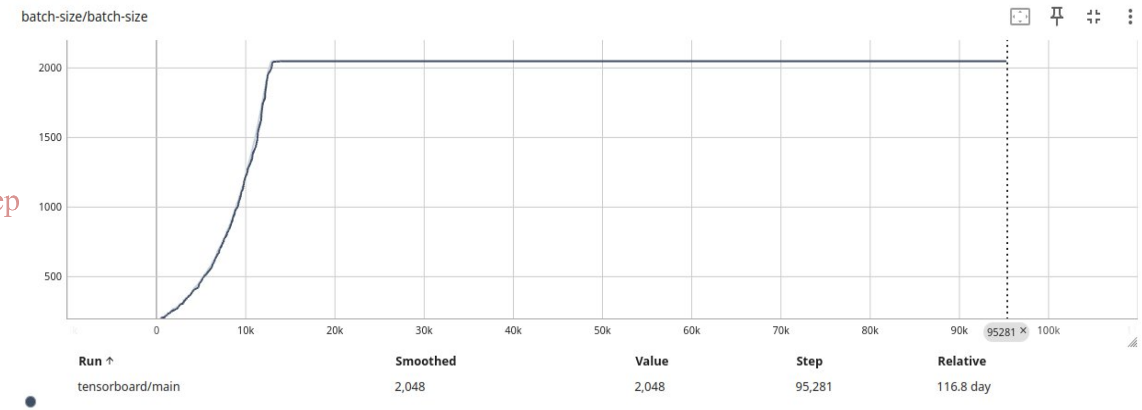
LR Scheduler

- peak=6e-5
- warmup over 375M tokens
- cosine decay for learning rate down to 10% of its value, over 410B tokens

Batch Scheduler

- start from 32k tokens (GBS=16)
- increase linearly to 4.2M tokens/step (GBS=2048) over ~20B tokens
- then continue at 4.2M tokens/step (GBS=2048) for 430B tokens





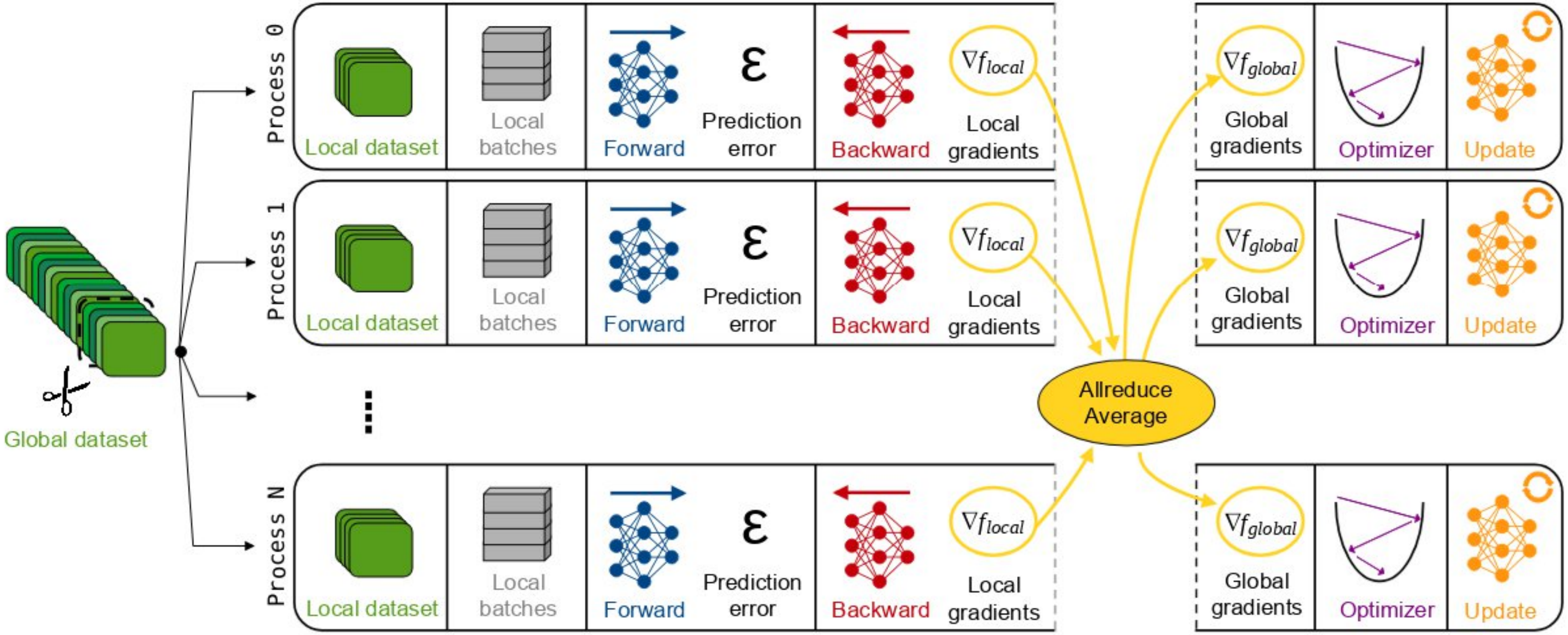
Reducing Optimizer Communication Costs

AllReduce Bottleneck◂

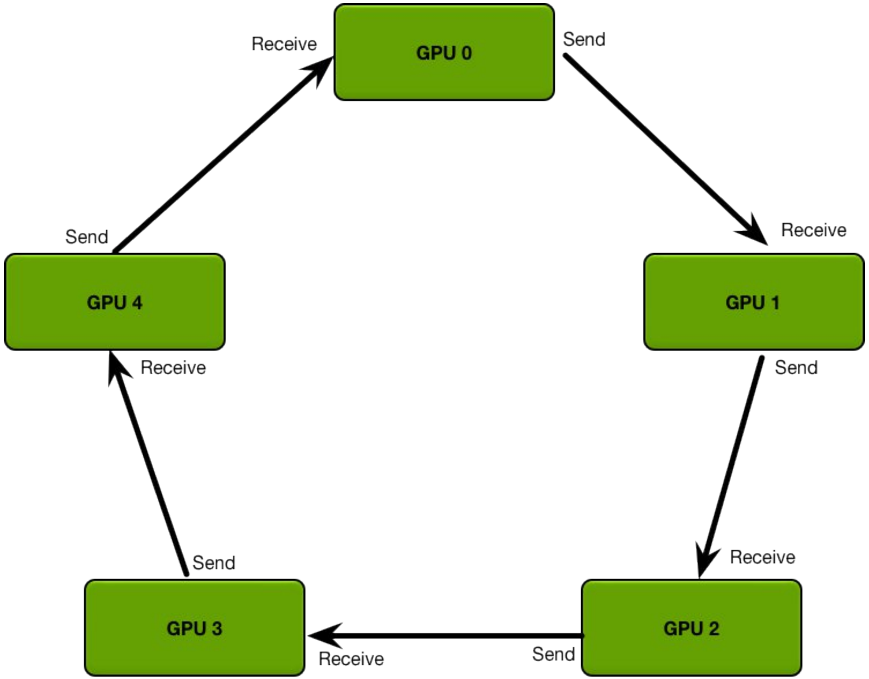
PowerSGD ◂

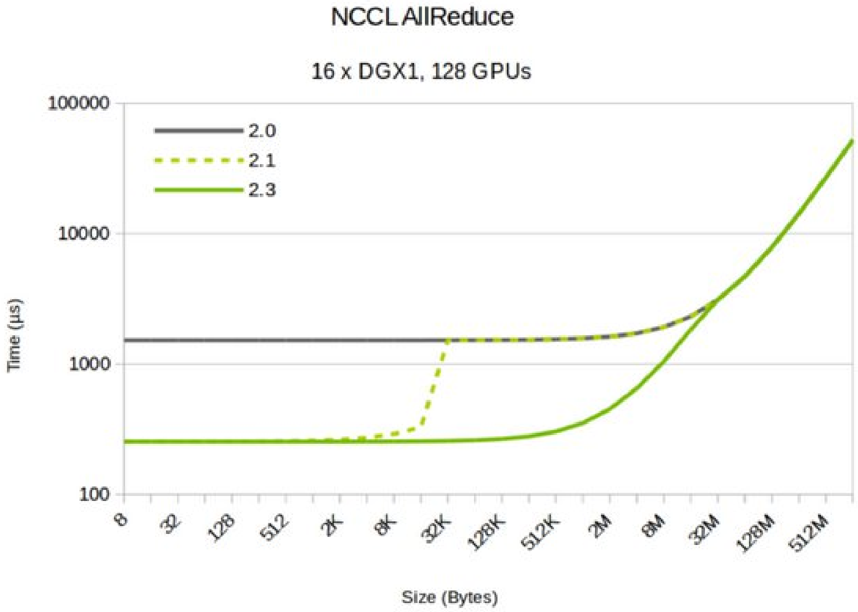
DiLoCo ◂

AllReduce bottleneck



DDP Communication Costs





Gradient compression - PowerSGD

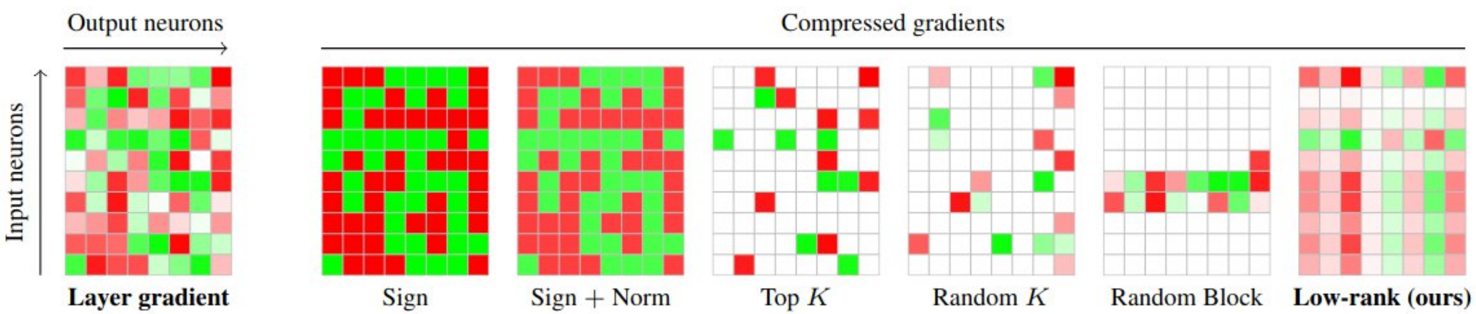


Figure 1: Compression schemes compared in this paper. Left: Interpretation of a layer's gradient as a matrix. Coordinate values are color coded (positive, negative). Right: The output of various compression schemeson the same input. Implementation details arein Appendix G.

Table 3: POWERSGD with varying rank. With sufficient rank, POWERSGD accelerates training of a RESNET18 and an LSTM by reducing communication, achieving test quality on par with regular SGD in the same number of iterations. The time per batch includes the forward/backward pass (constant). See Section 5 for the experimental setup.

Image classification — RESNET18 on CIFAR10

Algorithm	Test accuracy	Data sent per epoch		Time per batch	
SGD	94.3%	1023 MB	(1×)	312 ms	+0%
Rank 1	93.6%	4 MB	(243×)	229 ms	−26%
Rank 2	94.4%	8 MB	(136×)	239 ms	−23%
Rank 4	94.5%	14 MB	(72×)	260 ms	−16%

Language modeling — LSTM on WIKITEXT-2

Algorithm	Test perplexity	Data sent per epoch		Time per batch	
SGD	91	7730 MB	(1×)	300 ms	+0%
Rank 1	102	25 MB	(310×)	131 ms	−56%
Rank 2	93	38 MB	(203×)	141 ms	−53%
Rank 4	91	64 MB	(120×)	134 ms	−55%

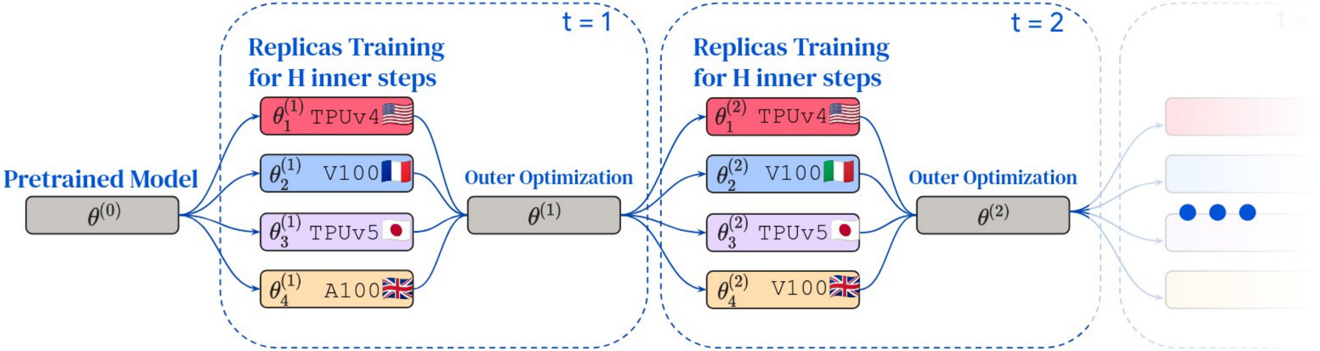
https://arxiv.org/pdf/1905.13727

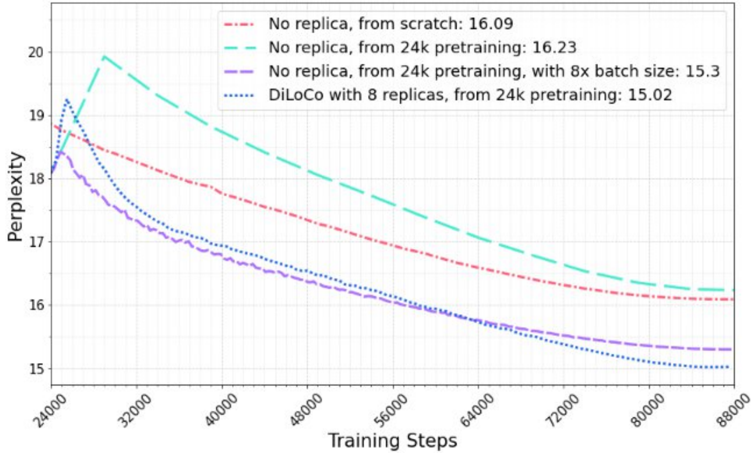
Asynchronous Optimization - DiLoCo

Algorithm 1 DiLoCo Algorithm

Require: Initial model $\theta^{(0)}$
Require: k workers
Require: Data shards $\{\mathcal{D}_1, \dots, \mathcal{D}_k\}$
Require: Optimizers `InnerOpt` and `OuterOpt`

for `outer step` $t = 1 \dots T$ **do**
  for `worker` $i = 1 \dots k$ **do**
    $\theta_i^{(t)} \leftarrow \theta^{(t-1)}$
    for `inner step` $h = 1 \dots H$ **do**
      $x \sim \mathcal{D}_i$
      $\mathcal{L} \leftarrow f(x, \theta_i^{(t)})$
      ▹ Inner optimization:
      $\theta_i^{(t)} \leftarrow \texttt{InnerOpt}(\theta_i^{(t)}, \nabla_{\mathcal{L}})$
    end for
  end for
  ▹ Averaging outer gradients:
  $\Delta^{(t)} \leftarrow \frac{1}{k} \sum_{i=1}^{k} (\theta^{(t-1)} - \theta_i^{(t)})$
  ▹ Outer optimization:
  $\theta^{(t)} \leftarrow \texttt{OuterOpt}(\theta^{(t-1)}, \Delta^{(t)})$
end for





https://arxiv.org/pdf/2311.08105

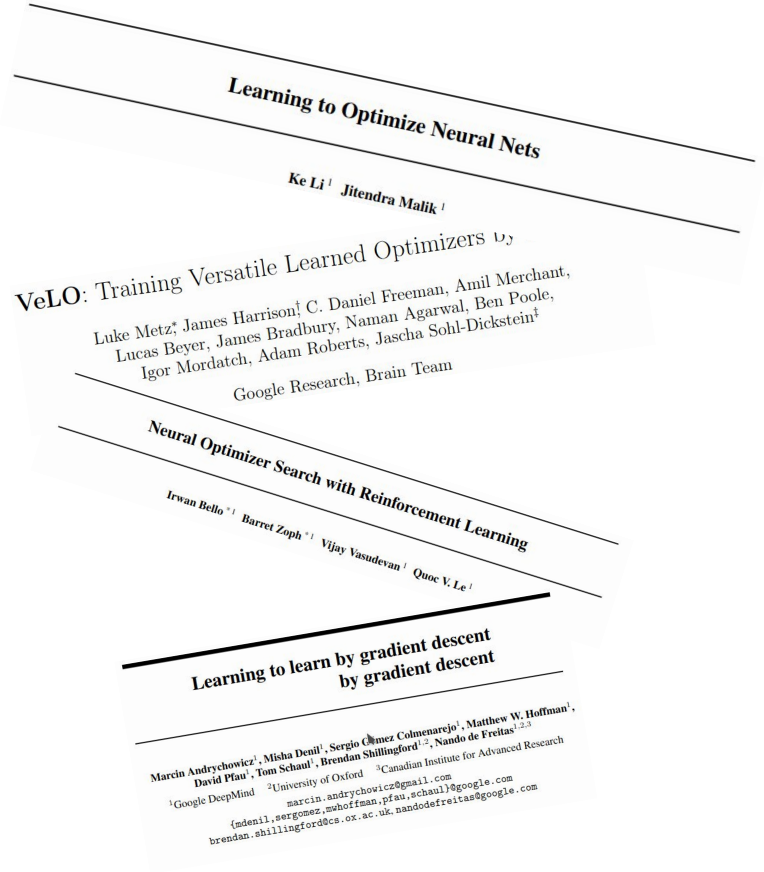
New optimizers

New trend : optimizers learning ◂

New optimizers Abyss ◂

LION : example of a new approach ◂

New trend : optimizers learning


Learning to Optimize Neural Nets

Ke Li [1] Jitendra Malik [1]

VeLO: Training Versatile Learned Optimizers by

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

Irwan Bello *1 Barret Zoph *1 Vijay Vasudevan [1] Quoc V. Le [1]

Learning to learn by gradient descent by gradient descent

Marcin Andrychowicz[1], Misha Denil[1], Sergio Gómez Colmenarejo[1], Matthew W. Hoffman[1], David Pfau[1], Tom Schaul[1], Brendan Shillingford[1,2], Nando de Freitas[1,2,3]

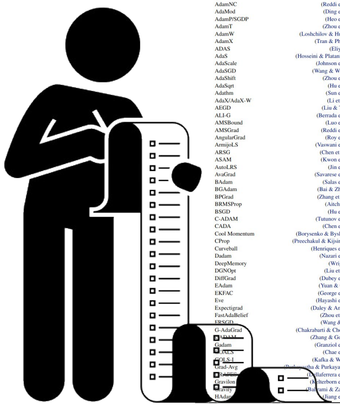
[1]Google DeepMind [2]University of Oxford [3]Canadian Institute for Advanced Research

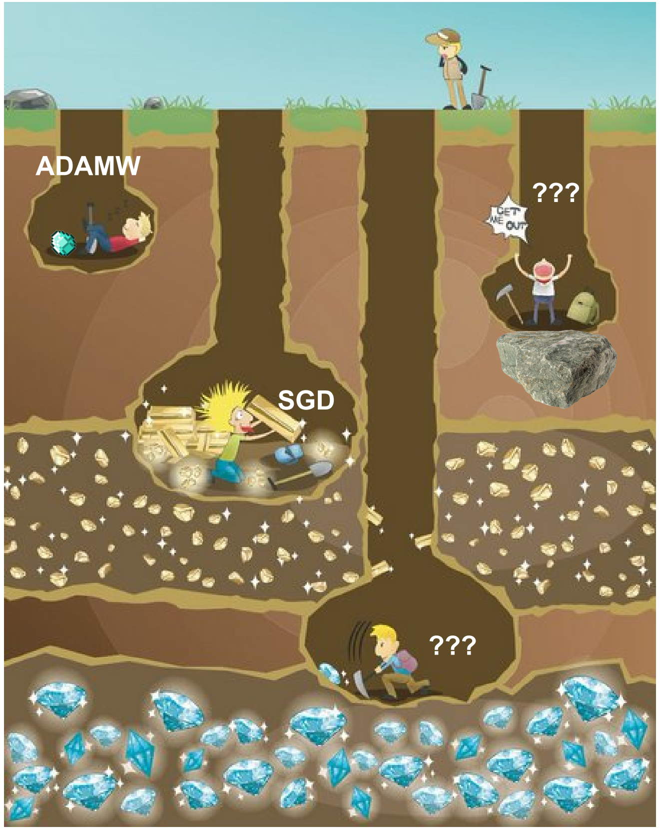
marcin.andrychowicz@gmail.com
{mdenil,sergomez,mwhoffman,pfau,schaul}@google.com
brendan.shillingford@cs.ox.ac.uk, nandodefreitas@google.com

New optimizers *Abyss*

Table 2: List of optimizers considered for our benchmark. This is only a subset of all existing methods for deep learning.

Name	Ref.	Name	Ref.
AcceleGrad	(Levy et al., 2018)	HyperAdam	(Wang et al., 2019b)
ACClip	(Zhang et al., 2020)	K-BFGS/K-BFGS(L)	(Goldfarb et al., 2020)
AdaAlter	(Xie et al., 2019)	KF-QN-CNN	(Ren & Goldfarb, 2021)
AdaBatch	(Devarakonda et al., 2017)	KFAC	(Martens & Grosse, 2015)
AdaBayes/AdaBayes-SS	(Aitchison, 2020)	KFLR/KFRA	(Botev et al., 2017)
AdaBelief	(Zhuang et al., 2020)	L4Adam/L4Momentum	(Rolínek & Martius, 2018)
AdaBlock	(Yun et al., 2019)	LAMB	(You et al., 2020)
AdaBound	(Luo et al., 2019)	LaProp	(Ziyin et al., 2020)
AdaComp	(Chen et al., 2018)	LARS	(You et al., 2017)
Adadelta	(Zeiler, 2012)	LHOPT	(Almeida et al., 2021)
Adafactor	(Shazeer & Stern, 2018)	LookAhead	(Zhang et al., 2019)
AdaFix	(Bae et al., 2019)	M-SVAG	(Balles & Hennig, 2018)
AdaFom	(Chen et al., 2019a)	MADGRAD	(Defazio & Jelassi, 2021)
AdaFTRL	(Orabona & Pál, 2015)	MAS	(Landro et al., 2020)
Adagrad	(Duchi et al., 2011)	MEKA	(Chen et al., 2020b)
ADAHESSIAN	(Yao et al., 2020)	MTAdam	(Malkiel & Wolf, 2020)
Adai	(Xie et al., 2020)	MVRC-1/MVRC-2	(Chen & Zhou, 2020)
AdaLoss	(Teixeira et al., 2019)	Nadam	(Dozat, 2016)
Adam	(Kingma & Ba, 2015)	NAMSB/NAMSG	(Chen et al., 2019b)
$Adam^{+}$	(Liu et al., 2020b)	ND-Adam	(Zhang et al., 2017a)
AdamAL	(Tao et al., 2019)	Nero	(Liu et al., 2021b)
AdaMax	(Kingma & Ba, 2015)	Nesterov	(Nesterov, 1983)
AdamBS	(Liu et al., 2020c)	Noisy Adam/Noisy K-FAC	(Zhang et al., 2018)
AdamNC	(Reddi et al., 2018)	NosAdam	(Huang et al., 2019)
AdaMod	(Ding et al., 2019)	Novograd	(Ginsburg et al., 2019)
AdamP/SGDP	(Heo et al., 2021)	NT-SGD	(Zhou et al., 2021b)
AdamT	(Zhou et al., 2020)	Padam	(Chen et al., 2020a)
AdamW	(Loshchilov & Hutter, 2019)	PAGE	(Li et al., 2020b)
AdamX	(Tran & Phong, 2019)	PAL	(Mutschler & Zell, 2020)
ADAS	(Eliyahu, 2020)	PolyAdam	(Orvieto et al., 2019)
AdaS	(Hosseini & Plataniotis, 2020)	Polyak	(Polyak, 1964)
AdaScale	(Johnson et al., 2020)	PowerSGD/PowerSGDM	(Vogels et al., 2019)
AdaSGD	(Wang & Wiens, 2020)	Probabilistic Polyak	(de Roos et al., 2021)
AdaShift	(Zhou et al., 2019)	ProbLS	(Mahsereci & Hennig, 2017)
AdaSqrt	(Hu et al., 2019)	PStorm	(Xu, 2020)
Adathm	(Sun et al., 2019)	QHAdam/QHM	(Ma & Yarats, 2019)
AdaX/AdaX-W	(Li et al., 2020a)	RAdam	(Liu et al., 2020a)
AEGD	(Liu & Tian, 2020)	Ranger	(Wright, 2020b)
ALI-G	(Berrada et al., 2020)	RangerLars	(Grankin, 2020)
AMSBound	(Luo et al., 2019)	RMSProp	(Tieleman & Hinton, 2012)
AMSGrad	(Reddi et al., 2018)	RMSterov	(Choi et al., 2019)
AngularGrad	(Roy et al., 2021)	S-SGD	(Sung et al., 2020)
ArmijoLS	(Vaswani et al., 2019)	SAdam	(Wang et al., 2020b)
ARSG	(Chen et al., 2019b)	Sadam/SAMSGrad	(Tong et al., 2019)
ASAM	(Kwon et al., 2021)	SALR	(Yue et al., 2020)
AutoLRS	(Jin et al., 2021)	SAM	(Foret et al., 2021)
AvaGrad	(Savarese et al., 2019)	SC-Adagrad/SC-RMSProp	(Mukkamala & Hein, 2017)
BAdam	(Salas et al., 2018)	SDProp	(Ida et al., 2017)
BGAdam	(Bai & Zhang, 2019)	SGD	(Robbins & Monro, 1951)
BPGrad	(Zhang et al., 2017b)	SGD-BB	(Tan et al., 2016)
BRMSProp	(Aitchison, 2020)	SGD-G2	(Ayadi & Turinici, 2020)
BSGD	(Hu et al., 2020)	SGDEM	(Ramezani-Kebrya et al., 2021)
C-ADAM	(Tutunov et al., 2020)	SGDHess	(Tran & Cutkosky, 2021)
CADA	(Chen et al., 2021)	SGDM	(Liu & Luo, 2020)
Cool Momentum	(Borysenko & Byshkin, 2020)	SGDR	(Loshchilov & Hutter, 2017)
CProp	(Preechakul & Kijsirikul, 2019)	SHAdagrad	(Huang et al., 2020)
Curveball	(Henriques et al., 2019)	Shampoo	(Anil et al., 2020; Gupta et al., 2018)
Dadam	(Nazari et al., 2019)	SignAdam++	(Wang et al., 2019a)
DeepMemory	(Wright, 2020a)	SignSGD	(Bernstein et al., 2018)
DGNOpt	(Liu et al., 2021a)	SKQN/S4QN	(Yang et al., 2020)
DiffGrad	(Dubey et al., 2020)	SM3	(Anil et al., 2019)
EAdam	(Yuan & Gao, 2020)	SMG	(Tran et al., 2020)
EKFAC	(George et al., 2018)	SNGM	(Zhao et al., 2020)
Eve	(Hayashi et al., 2018)	SoftAdam	(Fetterman et al., 2019)
Expectigrad	(Daley & Amato, 2020)	SRSGD	(Wang et al., 2020a)
FastAdaBelief	(Zhou et al., 2021a)	Step-Tuned SGD	(Castera et al., 2021)
FRSGD	(Wang & Ye, 2020)	SWATS	(Keskar & Socher, 2017)
G-AdaGrad	(Chakrabarti & Chopra, 2021)	SWNTS	(Chen et al., 2019c)
[illegible]	(Zhang & Gouza, 2018)	TAdam	(Ilboudo et al., 2020)
Gadam	(Granziol et al., 2020)	TEKFAC	(Gao et al., 2020)
[illegible]	(Chae et al., 2021)	VAdam	(Khan et al., 2018)
GOLS-I	(Kafka & Wilke, 2019)	VR-SGD	(Shang et al., 2020)
Grad-Avg	([illegible]tha & Purkayastha, 2020)	vSGD-b/vSGD-g/vSGD-l	(Schaul et al., 2013)
[illegible]	([illegible]llaferrera et al., 2021)	vSGD-fd	(Schaul & LeCun, 2013)
Gravilon	([illegible]elterborn et al., 2020)	WNGrad	(Wu et al., 2018)
[illegible]ravity	([illegible]rami & Zadeh, 2021)	YellowFin	(Zhang & Mitliagkas, 2019)
HAda[illegible]	(Jiang et al., 2019)	Yogi	(Zaheer et al., 2018)





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

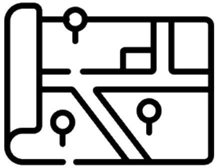
LION : example of a new approach

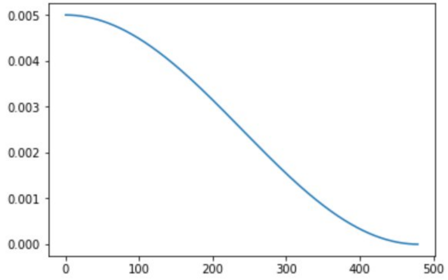
Algorithm 1 AdamW Optimizer

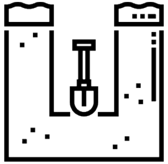
given $\beta_1, \beta_2, \epsilon, \lambda, \eta, f$
initialize $\theta_0, m_0 \leftarrow 0, v_0 \leftarrow 0, t \leftarrow 0$
while θ_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
  update EMA of g_t **and** g_t^2
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  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

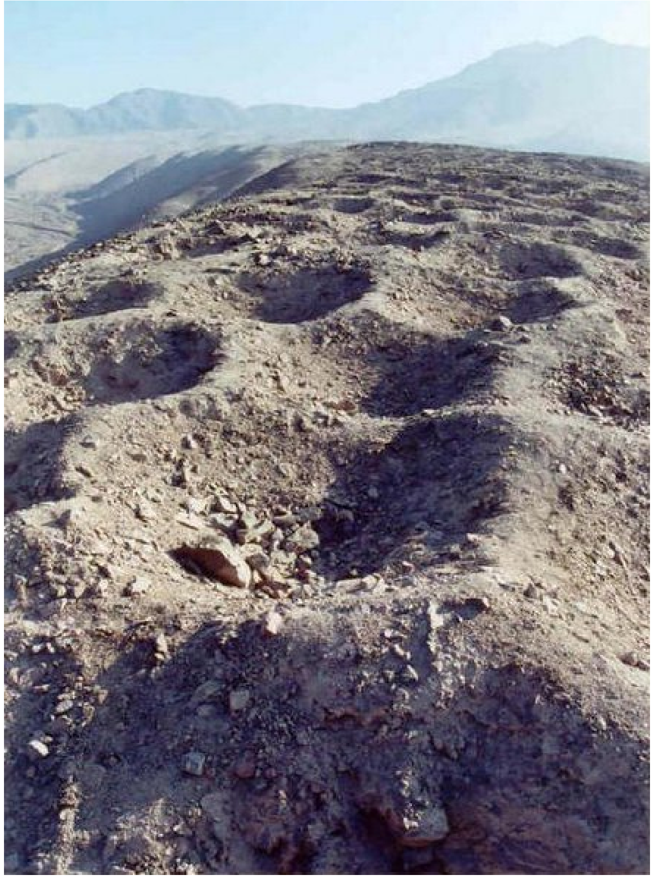
Algorithm 2 Lion Optimizer (ours)

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).

Pratice : Learning rate + Optimiseurs

Goals :

- Edit the **learning rate scheduler**
- Edit the **optimizer**
- Do training with **large batches**

From **JupyterHub**:

- ➢ Launch an interactive instance
- ➢ Go to the tp_optimizers folder
- ➢ Open the DLO-JZ_Optimizers notebook