



# Optimized Deep Learning - Jean Zay

---

## Training and large batches



INSTITUT DU  
DÉVELOPPEMENT ET DES  
RESSOURCES EN  
INFORMATIQUE  
SCIENTIFIQUE



# Loss Landscape

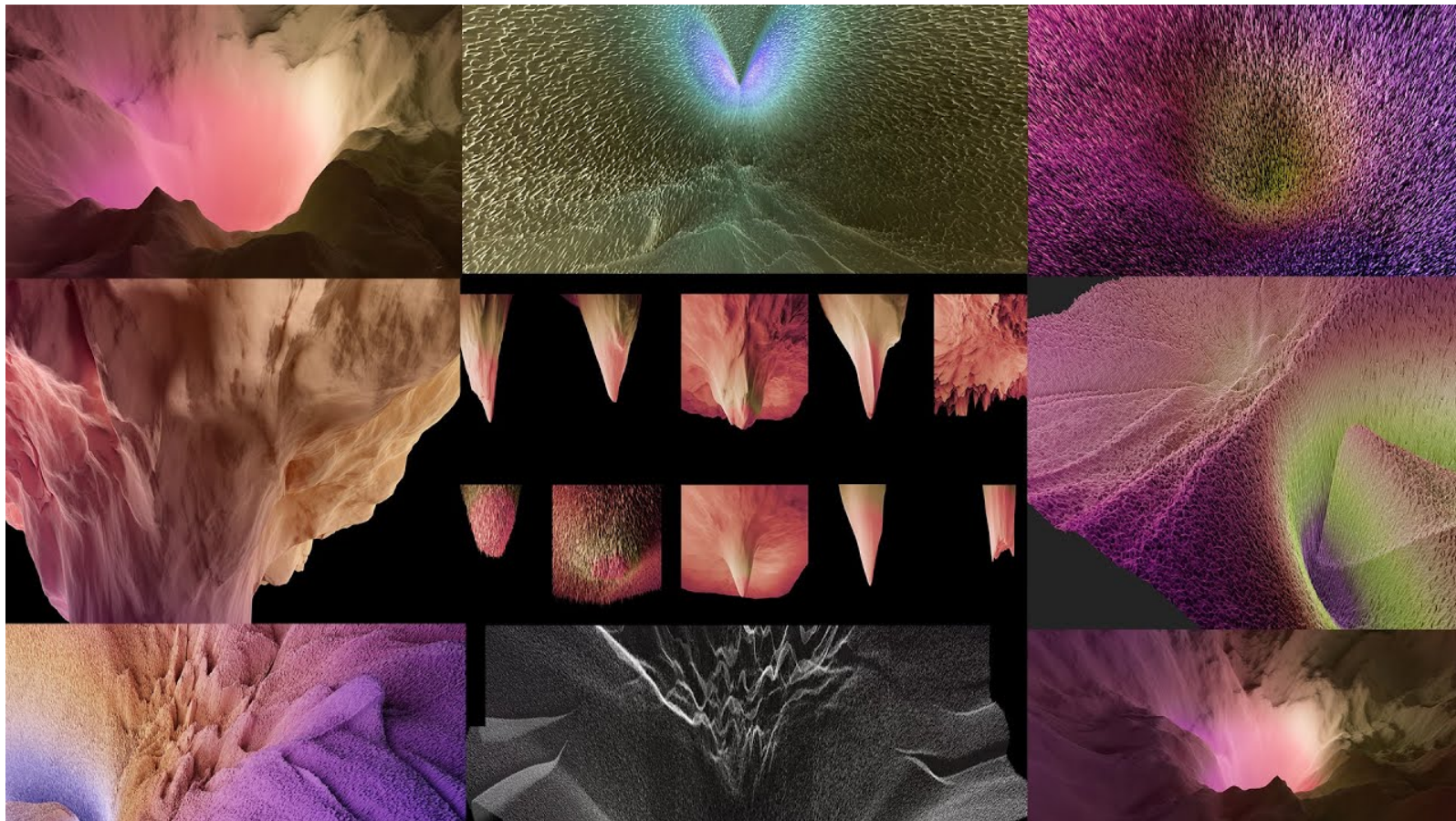
Loss Landscape ◀

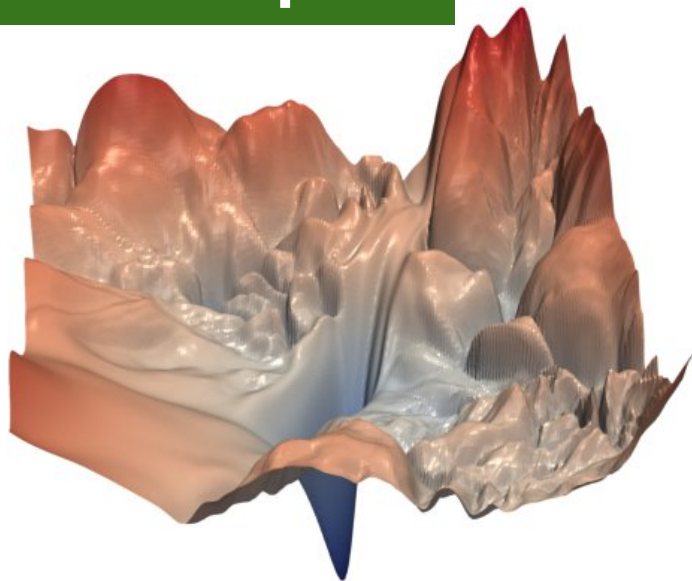
Residual Learning ◀

Initialization ◀

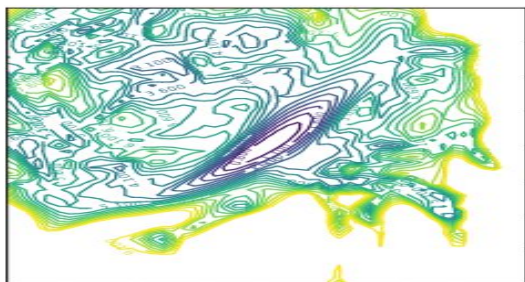
# Loss Landscape

<https://losslandscape.com/>

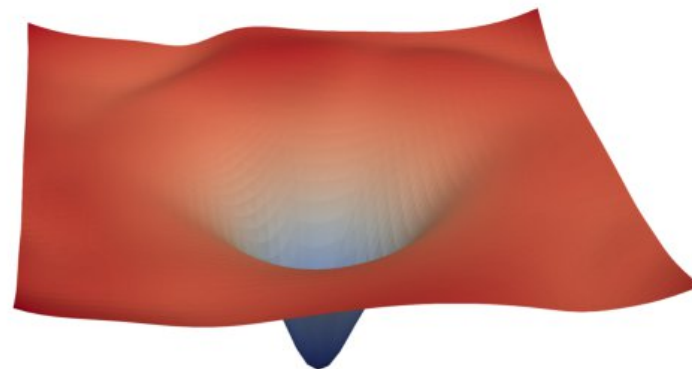




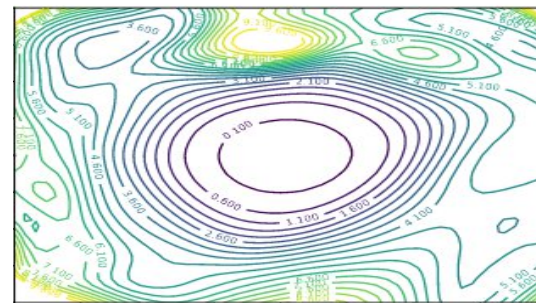
(a) without skip connections



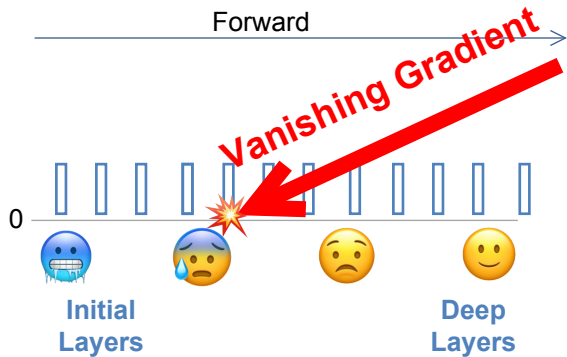
**Residual Learning**  
Since Resnets (2015) ...



(b) with skip connections

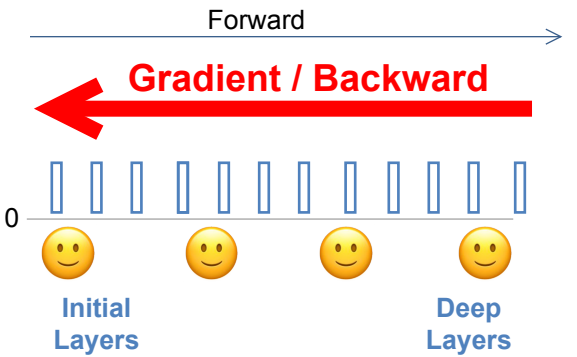


# Residual Learning

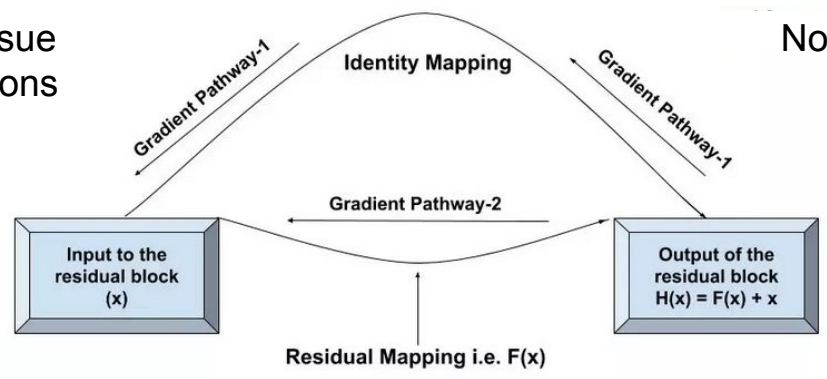


Vanishing Gradient issue **without** skip connections

Residual Block  $F(x)+x$



No Vanishing Gradient issue **with** skip connections

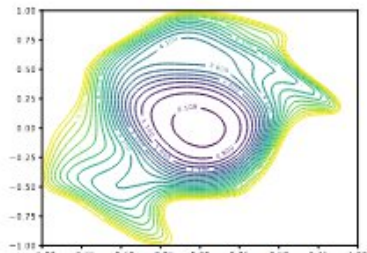


Gradient Pathways in ResNet

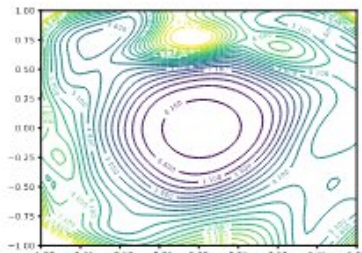
# Residual Learning – depth impact



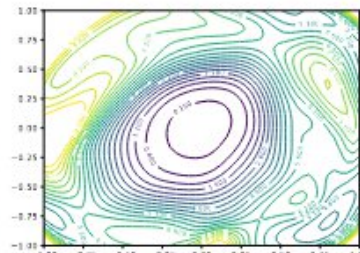
ResNet



(a) ResNet-20, 7.37%

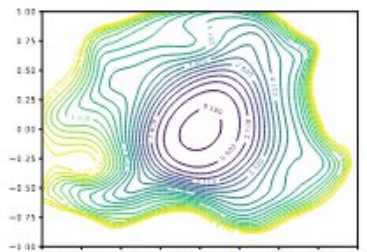


(b) ResNet-56, 5.89%

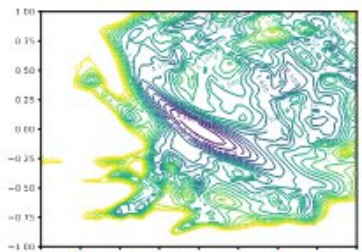


(c) ResNet-110, 5.79%

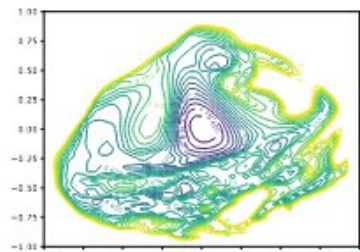
ResNet -  
No Short  
without skip connections



(d) ResNet-20-NS, 8.18%



(e) ResNet-56-NS, 13.31%



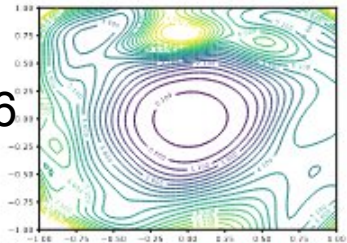
(f) ResNet-110-NS, 16.44%



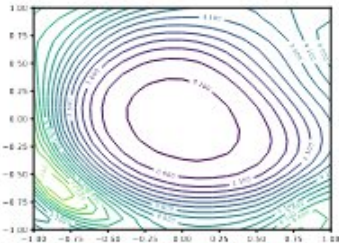
# Residual Learning – width impact



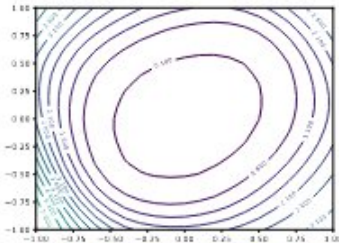
Wide-ResNet-56



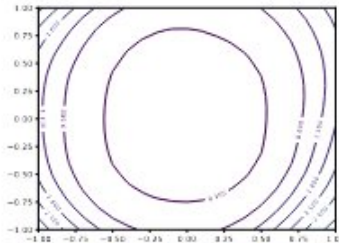
(a)  $k = 1$ , 5.89%



(b)  $k = 2$ , 5.07%

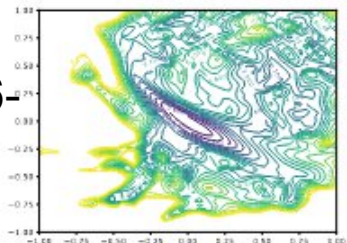


(c)  $k = 4$ , 4.34%

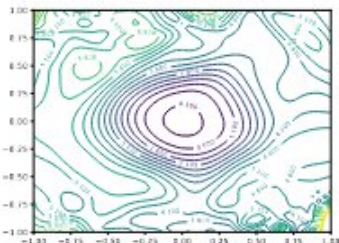


(d)  $k = 8$ , 3.93%

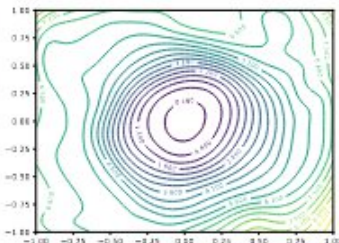
Wide-ResNet-56-  
No Short  
without skip connections



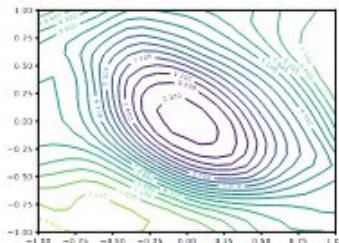
(e)  $k = 1$ , 13.31%



(f)  $k = 2$ , 10.26%



(g)  $k = 4$ , 9.69%



(h)  $k = 8$ , 8.70%



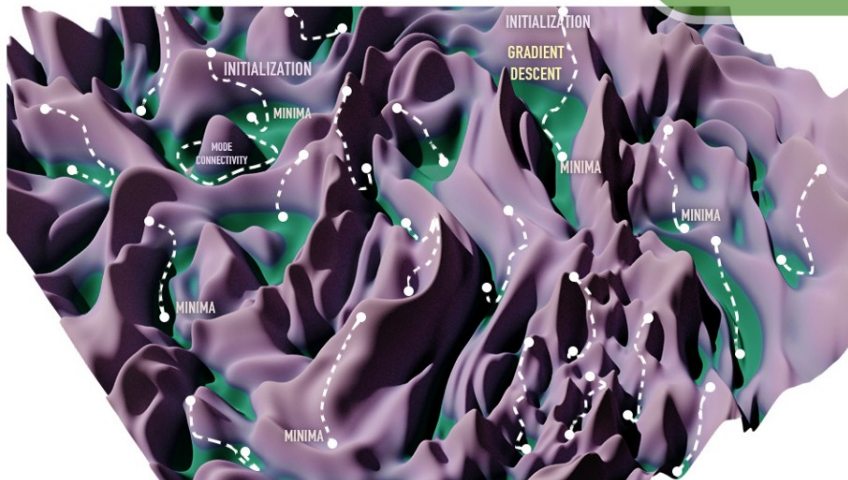
$k$  = channel width coefficient compared to ResNet

# Model Parameters Initialization

The Blessing of Dimensionality :

*Local*

NEARBY PATHS  
TO CONVERGENCE



FINDING A MINIMA BECOMES A "LOCAL" CHALLENGE



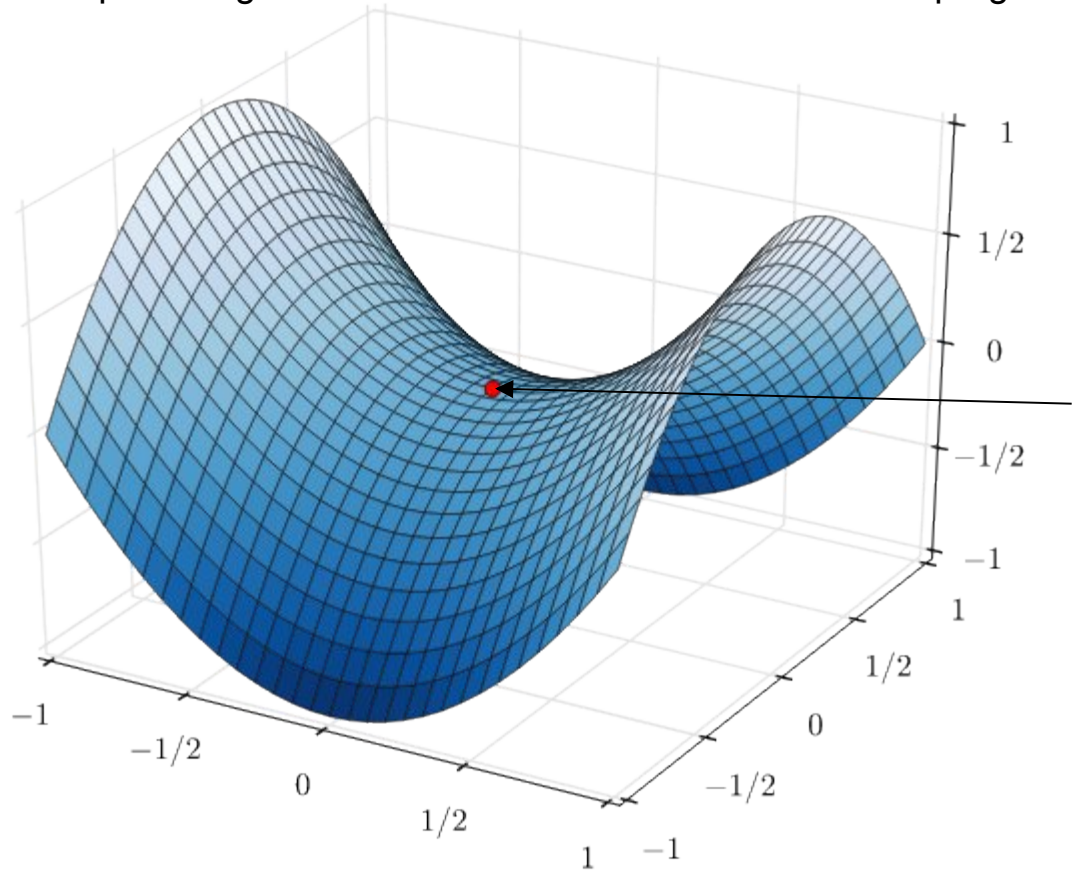
- Xavier Initialization
  - uniform
  - normal
- Kaiming Initialization
  - uniform
  - normal

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.

# 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?

# Learning Rate Scheduler

Learning Rate scheduler ◀

Cyclic scheduler ◀

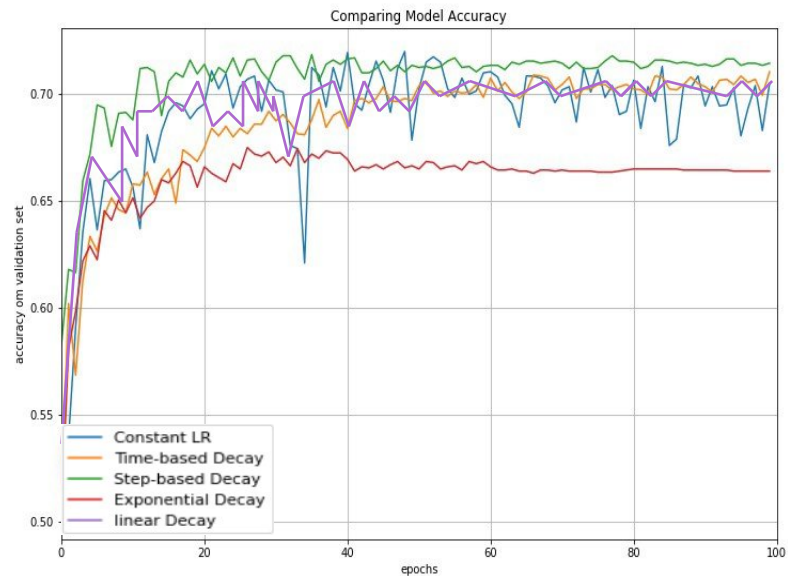
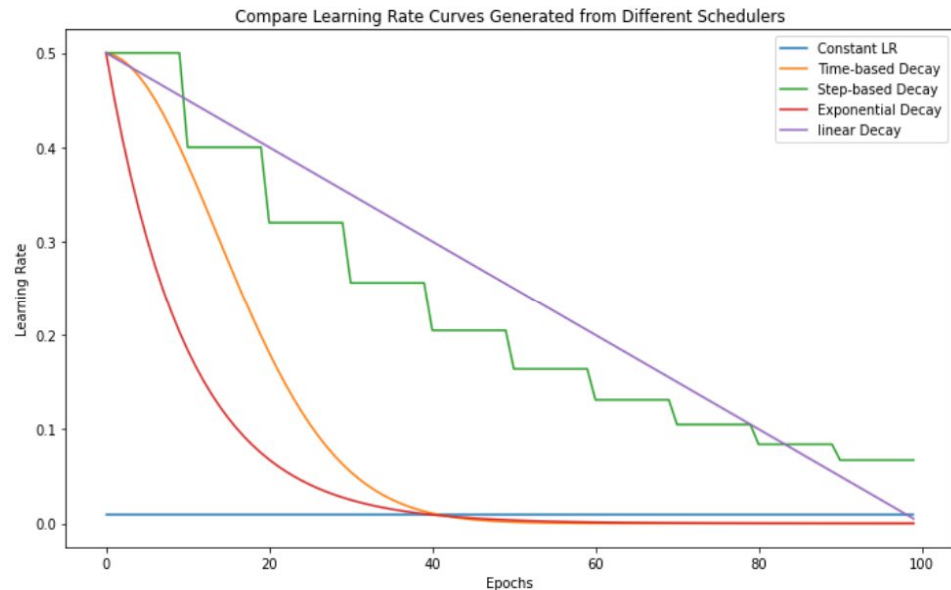
One Cycle scheduler ◀

ScheduleFree ◀

LR Finder◀

# Learning Rate Scheduler

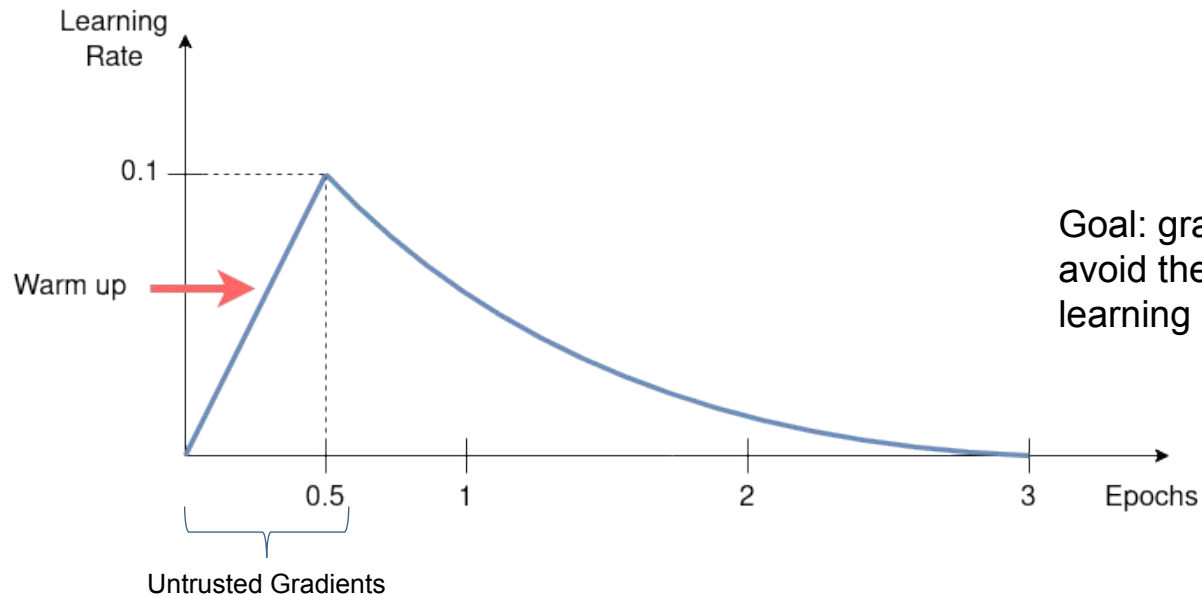
## Learning rate decay



# Learning Rate Scheduler

## WARMUP for *large batches*

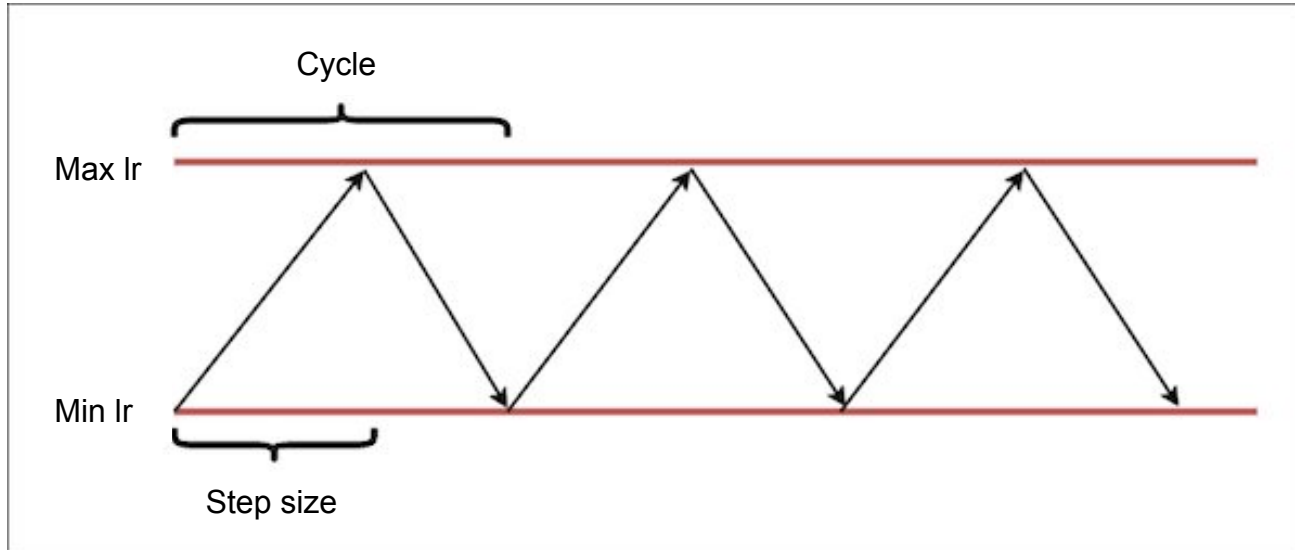
Problems: The first iterations have too much effect on the model (significant losses, high gradients, bias, etc.), a high learning rate can cause strong instability or divergence



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

# Cyclic Learning Rate Scheduler

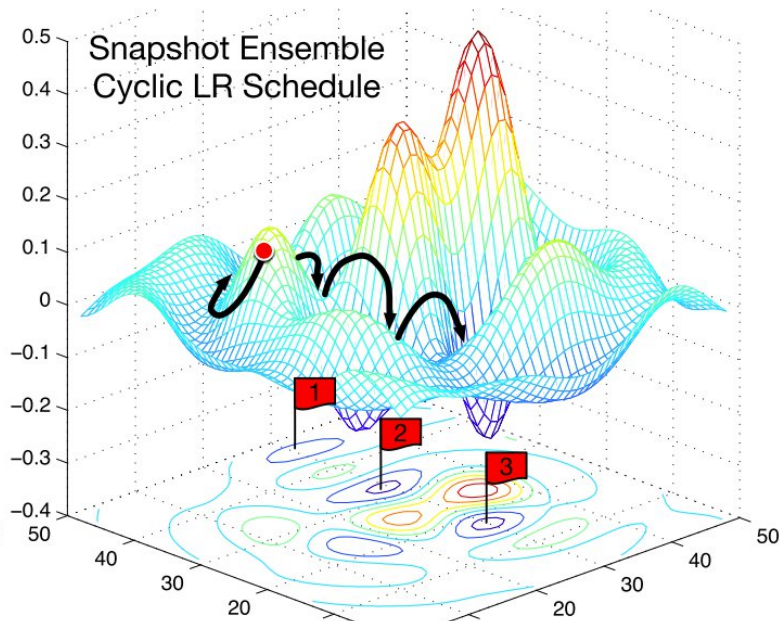
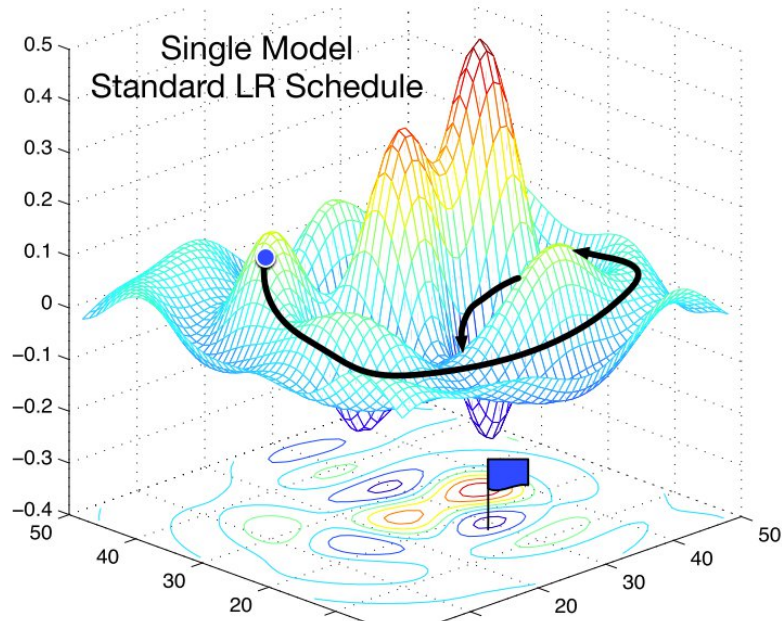
[Cyclical Learning Rates for Training Neural Networks - Leslie N. Smith 2017](#)



- Paramètres :
- $\text{Step\_size} = x * \text{epoch}$  ( $2 \leq x \leq 10$ )
  - $\text{Base\_lr}$  -> min convergence value
  - $\text{max\_lr}$  -> max convergence value

Succession of warmups and learning rate decays

# Cyclic Learning Rate Scheduler

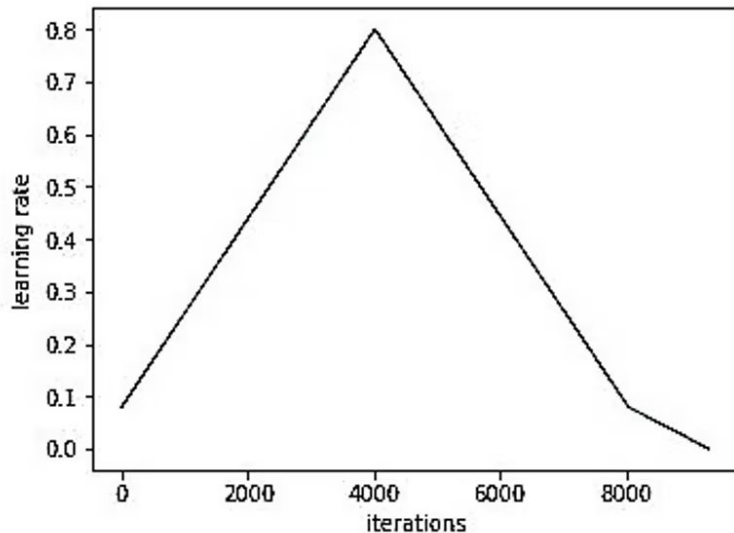


SNAPSHOT ENSEMBLES: TRAIN 1, GET M FOR FREE  
*Gao Huang, Yixuan Li, Geoff Pleiss*

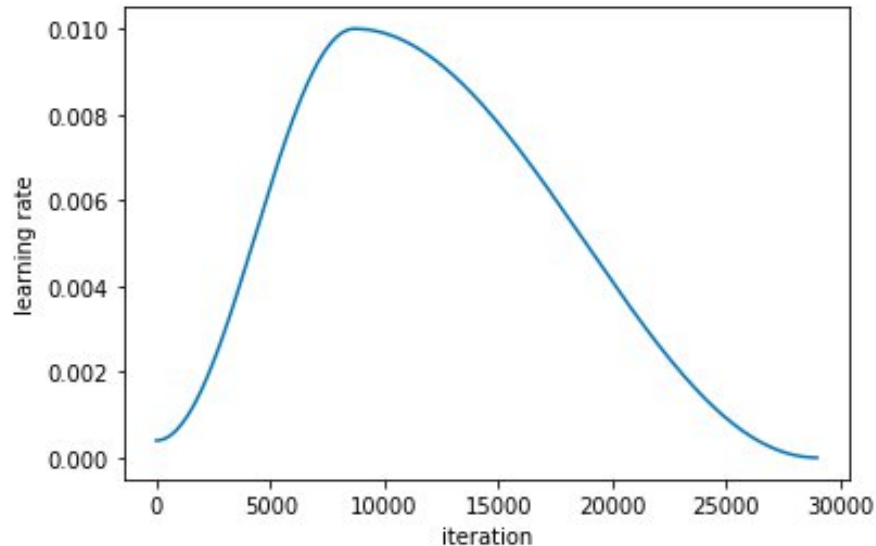
# One Cycle Learning Rate

One cycle is enough! [A disciplined approach to neural network hyper-parameters - Leslie N. Smith 2018](#)

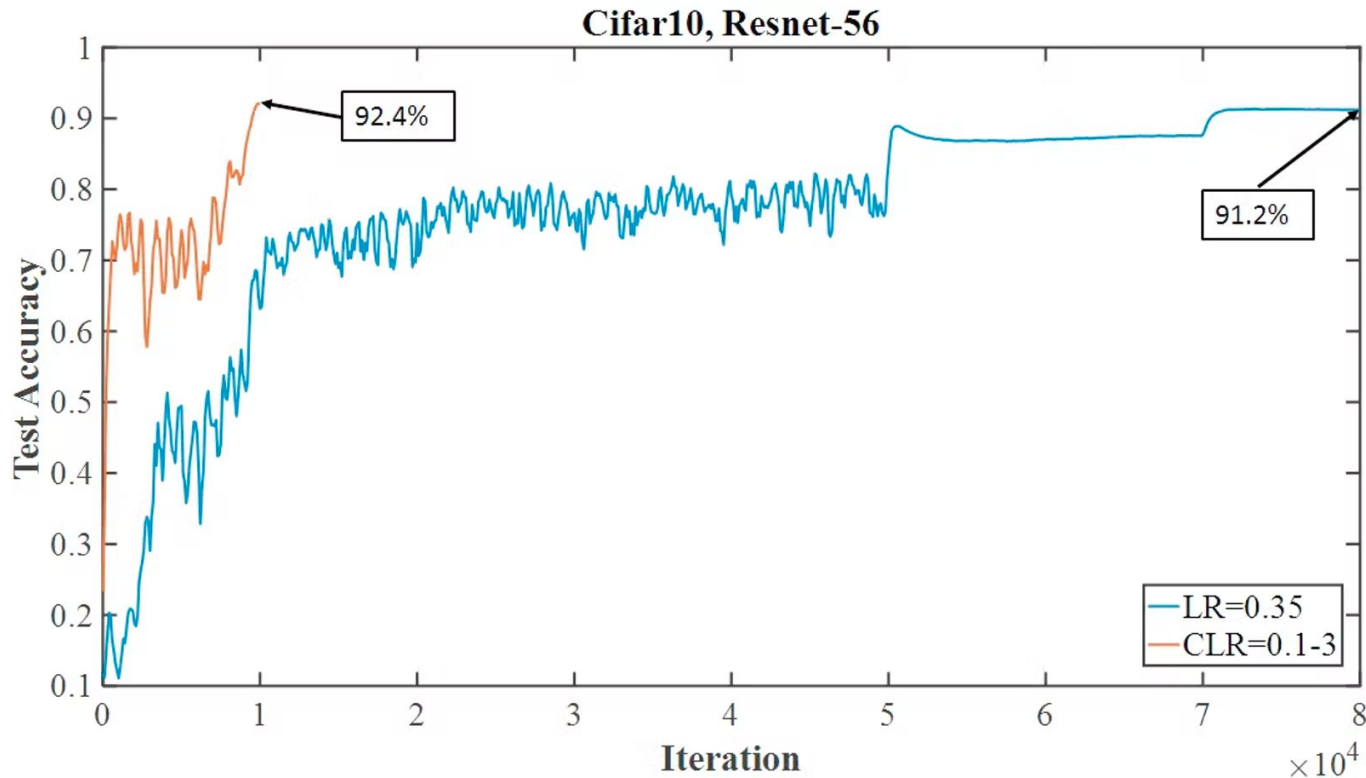
Proposition initiale



cosine annealing : Recommandation par FastAI



# One Cycle Learning Rate - *Super convergence*



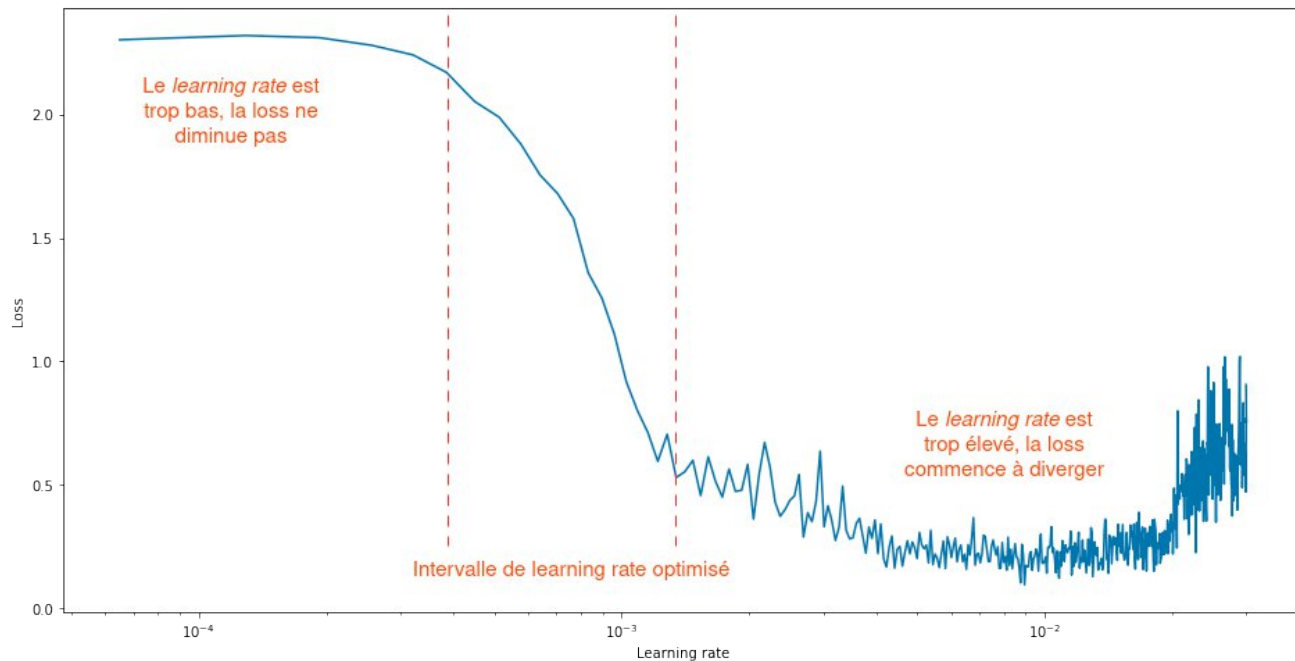
Faster convergence for equivalent final precision

# 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



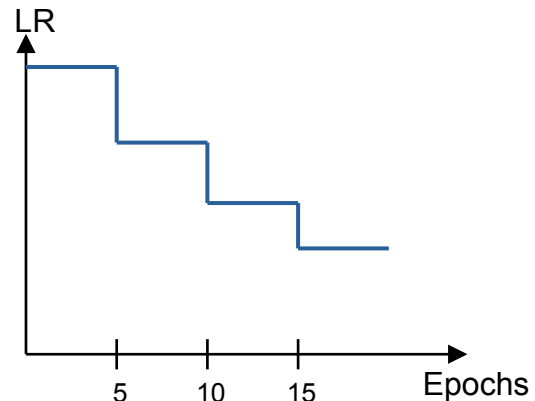
# Learning Rate Scheduler

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

```
import torch.optim as opt
```

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

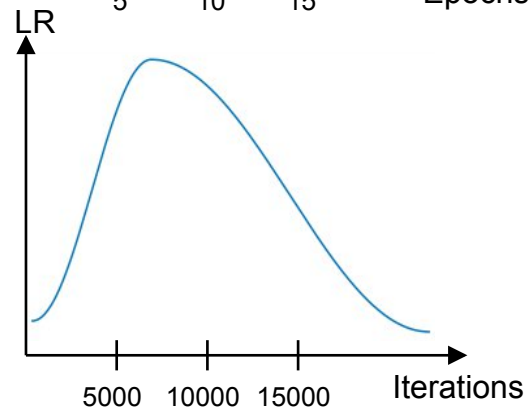
```
for epoch in range(100):  
    train(...)  
    validate(...)  
    scheduler.step()
```



```
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()
```

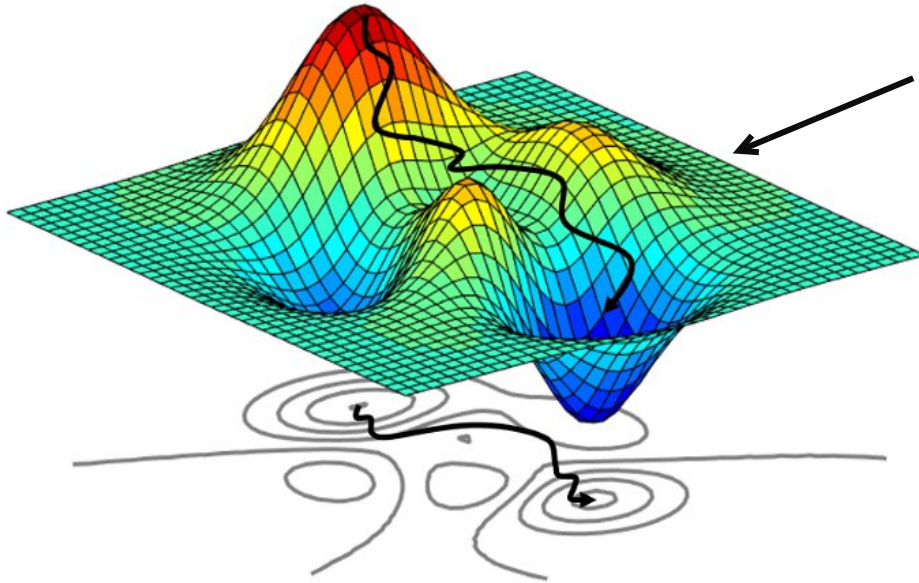


# Gradient Descent Optimizer

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 = Stochastic Gradient Descent**  
*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

$$m_0 = 0$$

Momentum coefficient

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

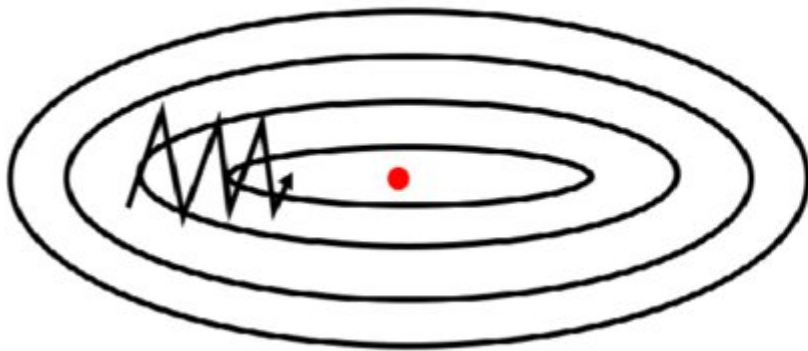
$$\theta_i = \theta_{i-1} - \alpha * m_i$$

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

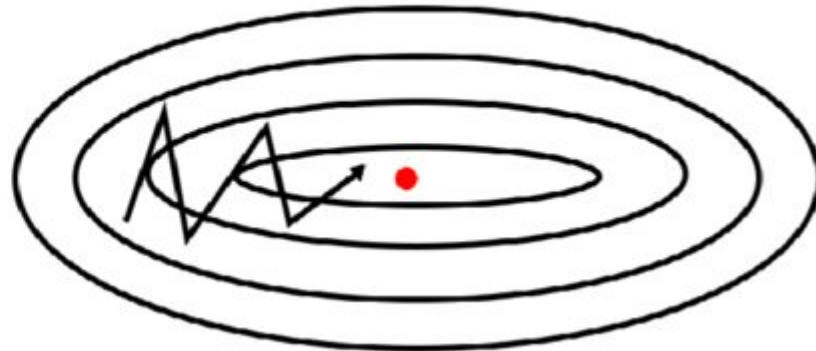
Recommended initial value: 0.9

$$0.85 < \beta < 0.95$$

SGD without momentum



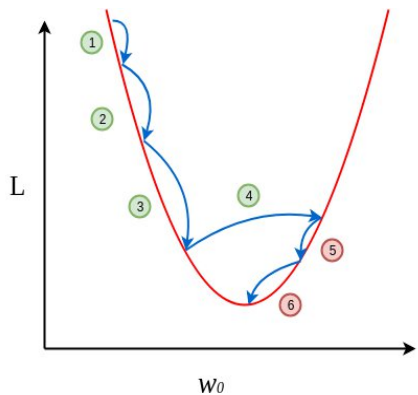
SGD with momentum



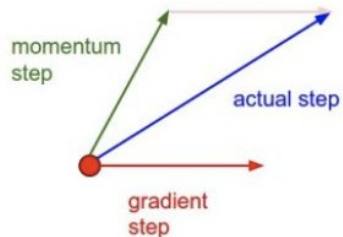
- + Allows you to **converge more quickly**
- **No guarantee** that momentum will take us in the right direction

# Momentum type

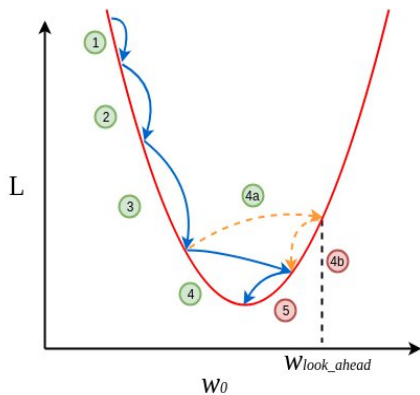
### Momentum



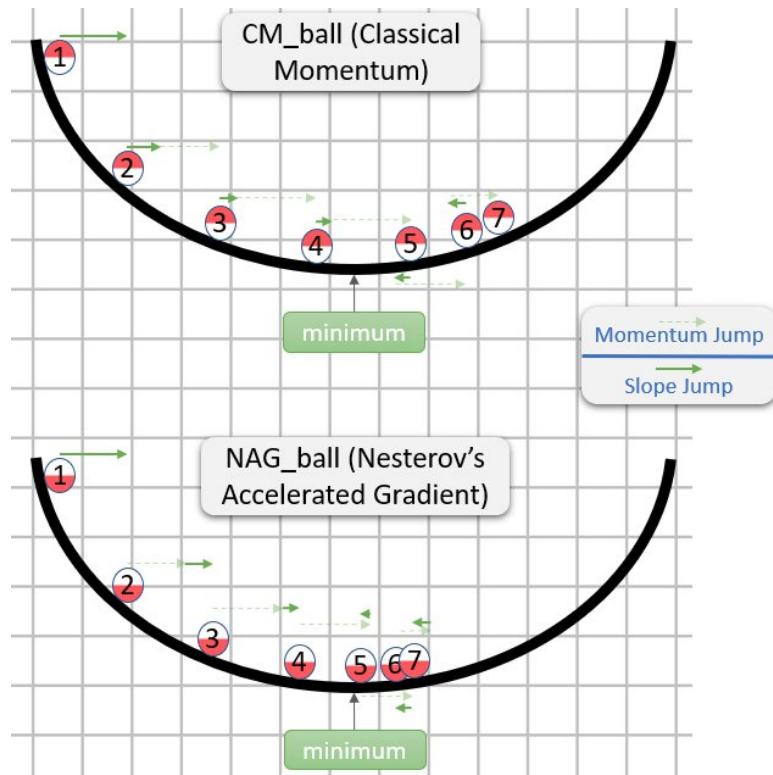
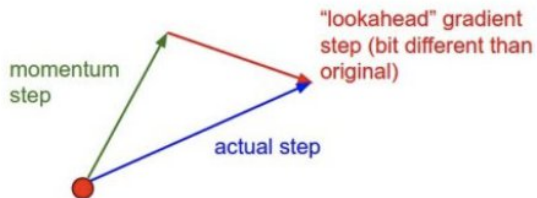
Momentum update



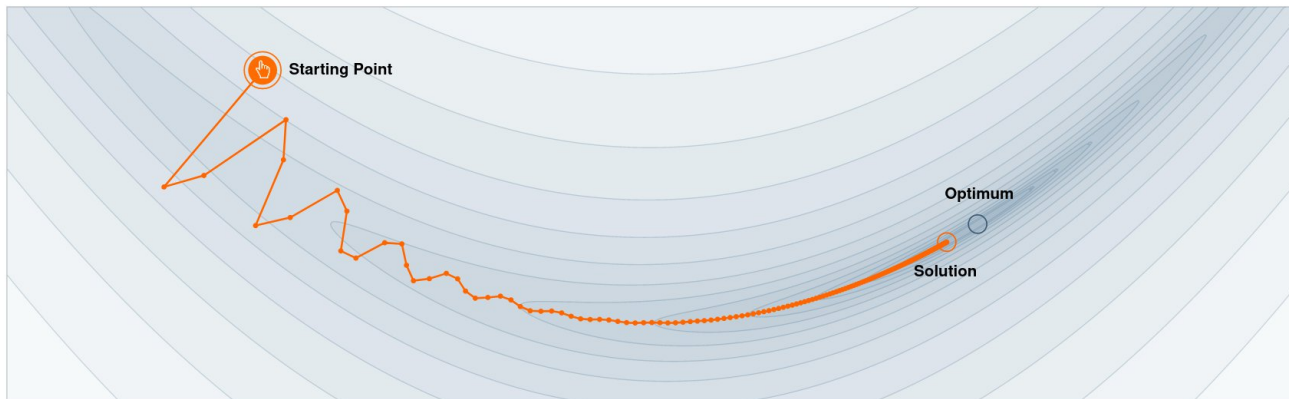
### Nesterov momentum



Nesterov momentum update



# Why Momentum Works ?



Step-size  $\alpha = 0.02$



Momentum  $\beta = 0.99$



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 **gradient<sup>2</sup>**, or the **norm of the weights** of the layer!!!

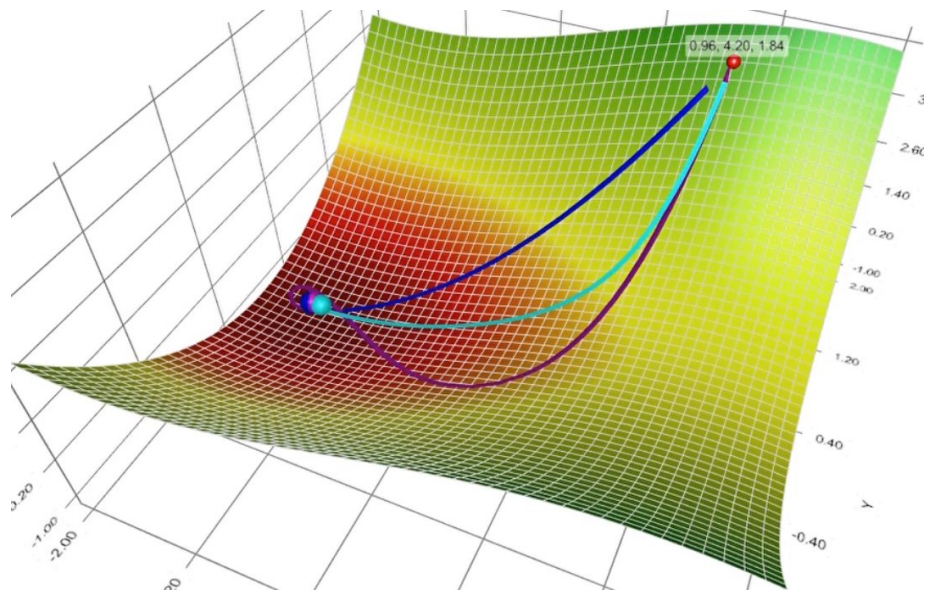
Examples :

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

Specialized for larges batches :

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

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



# Adam

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

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

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

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

Correction of biases of the first iterations

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

Parameters:

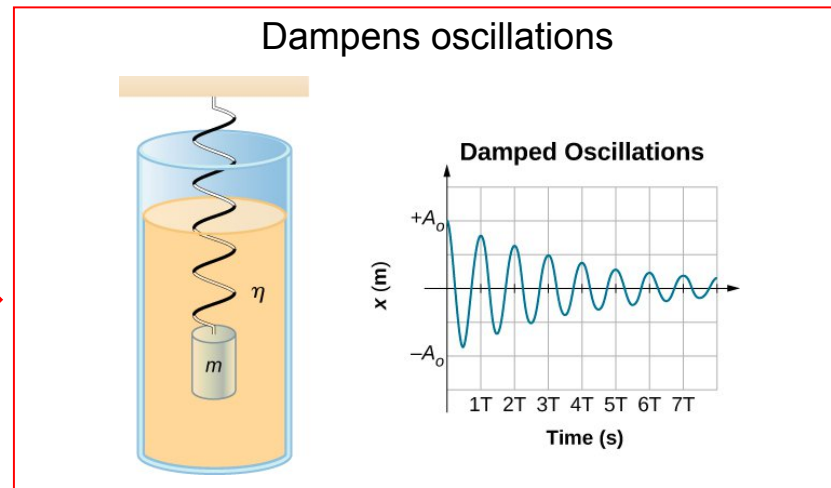
$\beta_1$  &  $\beta_2$  = Regression rate ( $\beta_1 = 0.9$  &  $\beta_2 = 0.999$ )

$\epsilon$  = Very small value to avoid division by zero

Adam : Adaptive moment estimation

First moment : sliding mean

Second moment : sliding non-centered variance

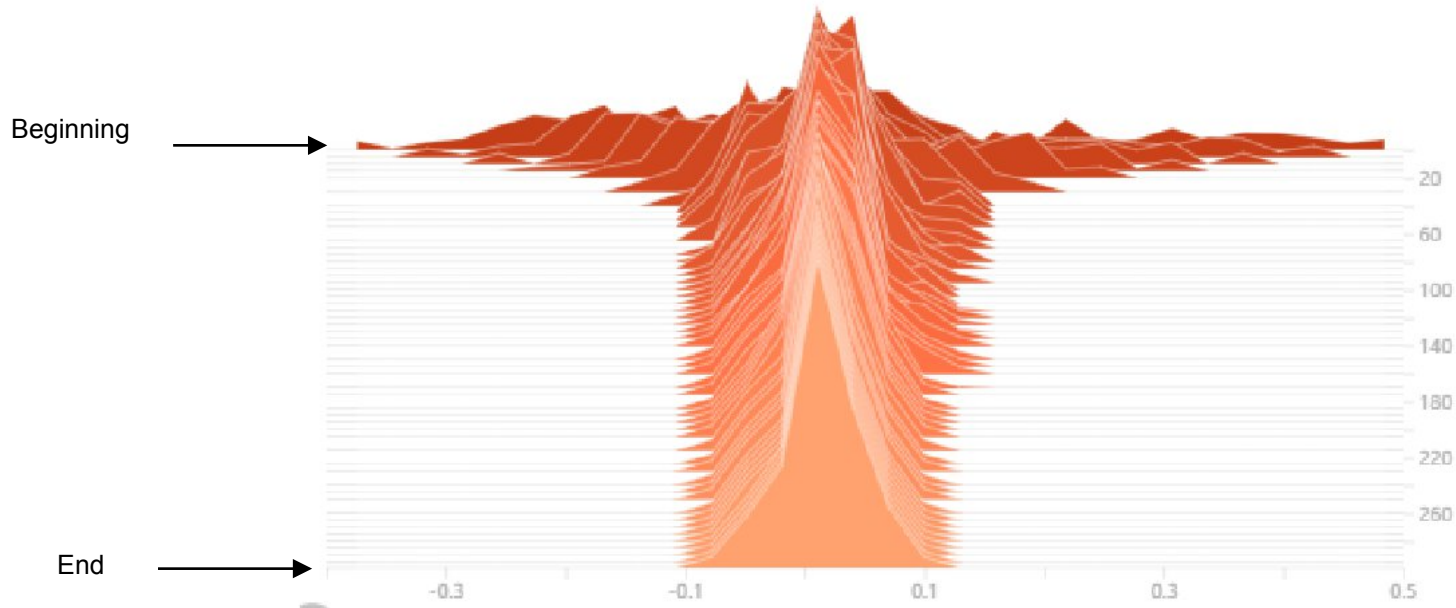


Goal: **Adapt the importance of weight** updates based on previous gradients and gradient variability.

# Weight decay

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

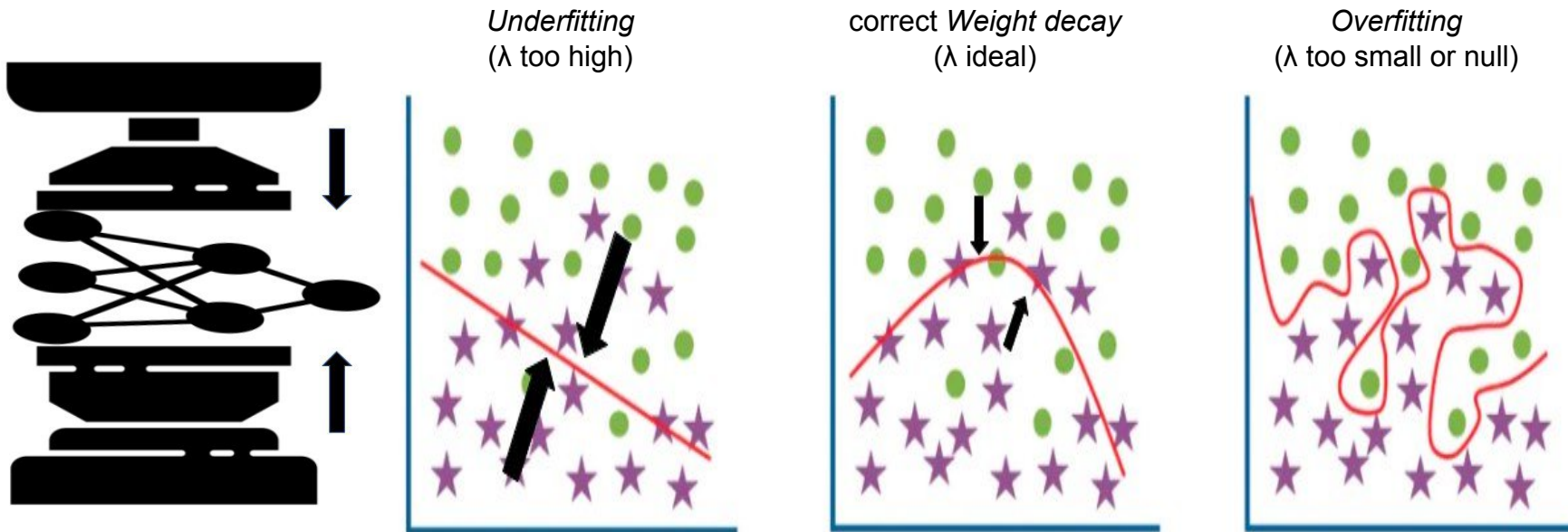
Distribution of weights during learning:



# Weight decay

Preferable to standard L2 regularization defined in loss function

$\lambda$  : 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

ADAMW

For  $i = 1$  to ...

$$g_i = \nabla_{\theta} f_i(\theta_{i-1}) + \lambda \theta_{i-1}$$

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

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

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

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

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

Return  $\theta_i$

Weight decay

For  $i = 1$  to ...

$$g_i = \nabla_{\theta} f_i(\theta_{i-1})$$

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

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

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

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

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

Return  $\theta_i$

Decoupled weight decay

Evolution of weight decay:

*Decoupled weight decay* (decoupled from momentum!!)

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

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

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

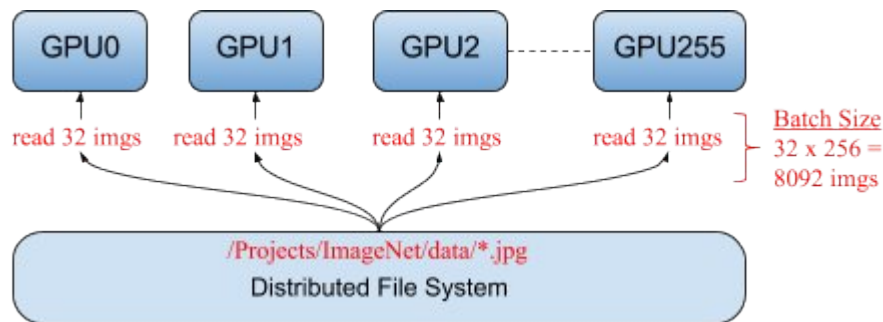
# Optimization of large batches

Large batches issues ◀

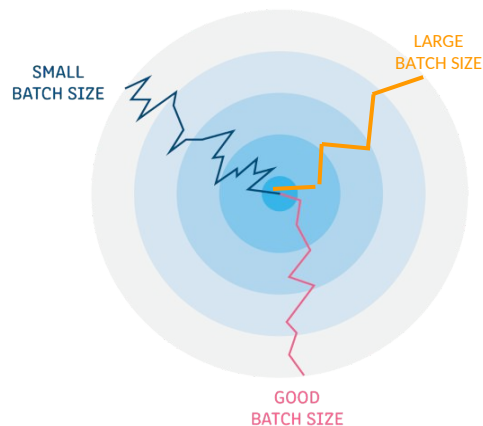
Learning Rate Scaling & Batch Schedulers ◀

Large batches optimizers ◀

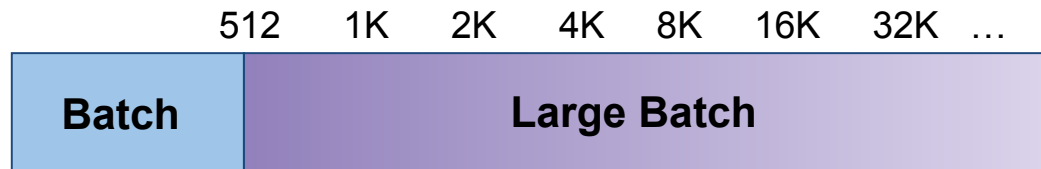
# Large Batches with Data Parallelism



**Data Parallelism:** This parallelism generates very large batches



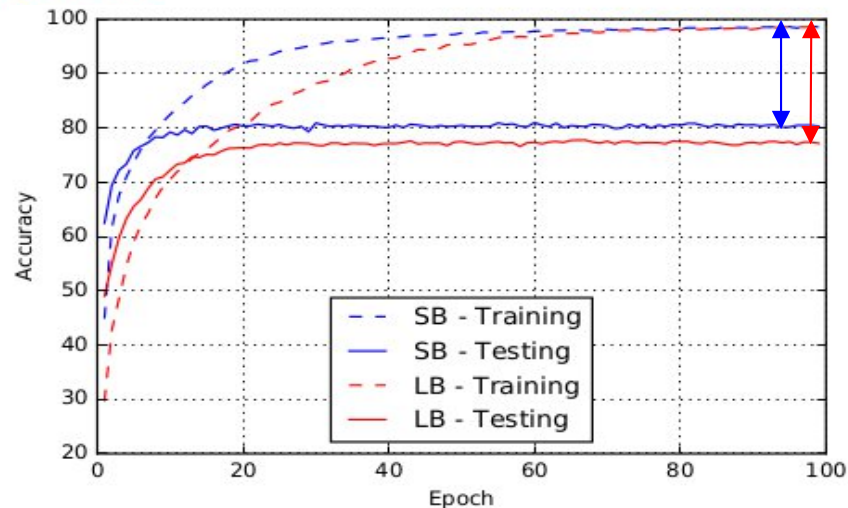
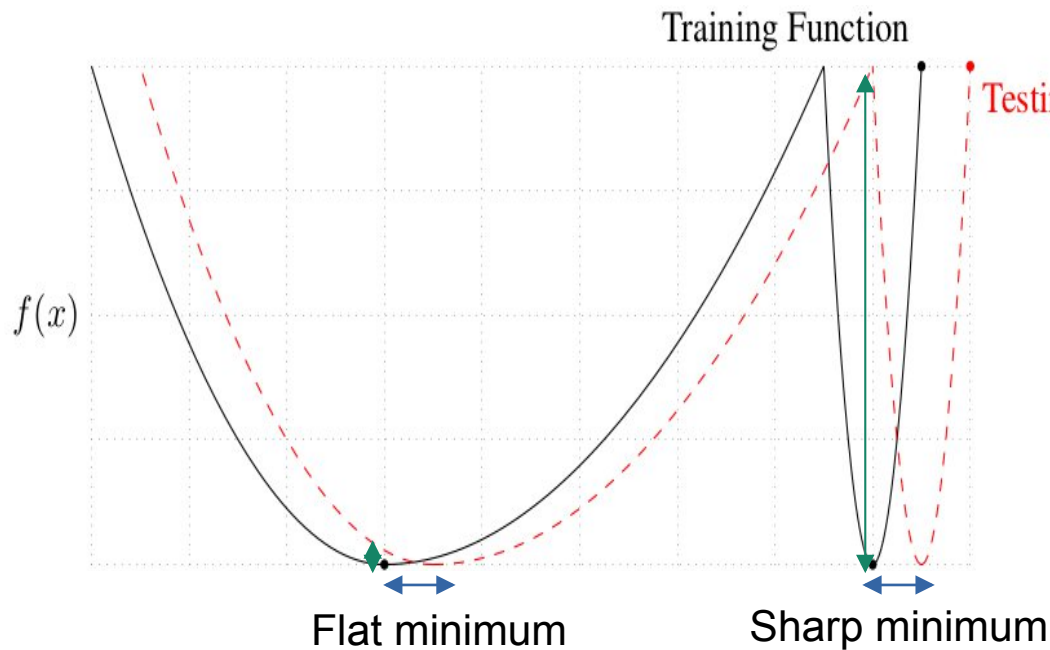
Problem: *Batch* that are too large ( $> 512$ ) tend to result in **poorer performance**



# 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

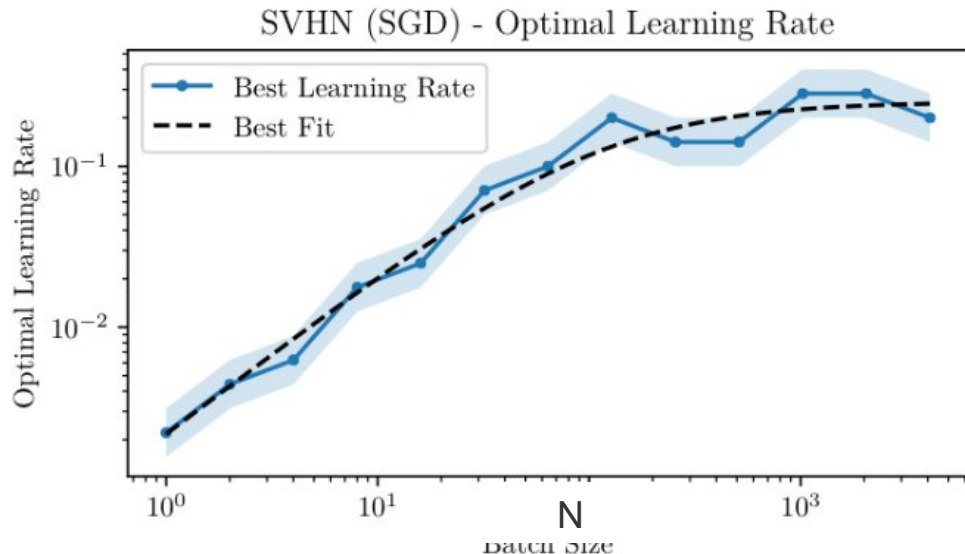
**Linear** growth of *learning rate*:

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

**Square root** growth of *learning rate*:

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

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



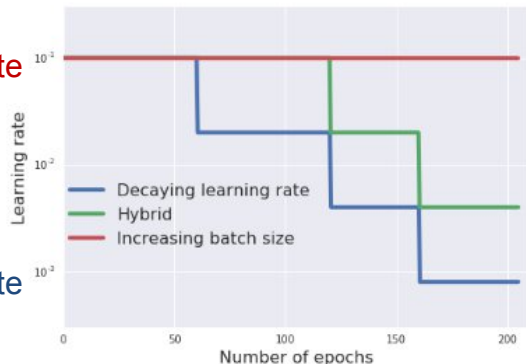
*An Empirical Model of Large-Batch Training*  
Sam McCandlish, Jared Kaplan, Dario Amodei

# Batch Size Scheduler

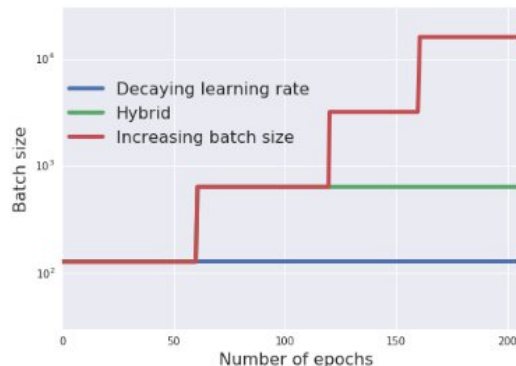
=> Alternative to Learning Rate Scheduler

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

High Learning Rate

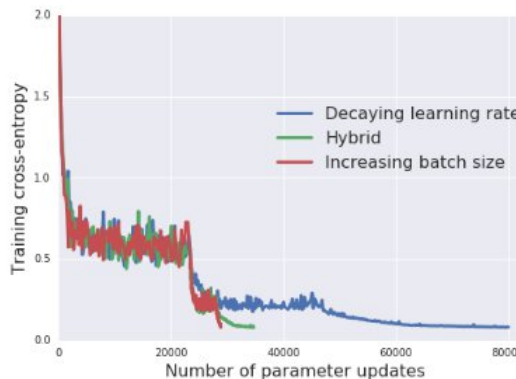
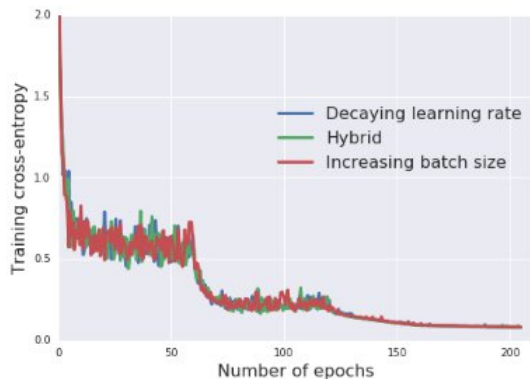


Low Learning Rate



Large Batch

Small Batch



# Large Batches

Trends :

**Flat Minimum** | **Sharp Minimum**

- Test Loss

+ Test Loss

Slow Descent

Fast Descent

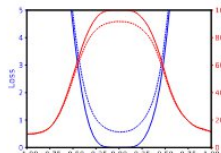
← Small Batch → Large Batch

← SGD → ADAM

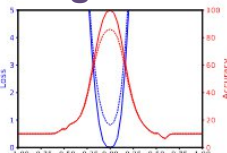
← Weight Decay → w/o W.Decay

## SGD

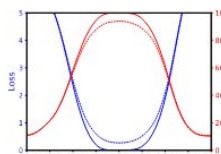
Small Batch Large Batch



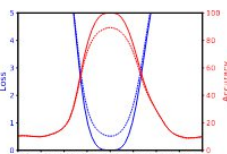
(a) SGD, 128, 8.26%



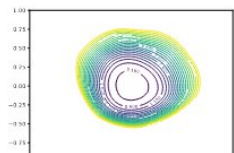
(b) SGD, 4096, 13.93%



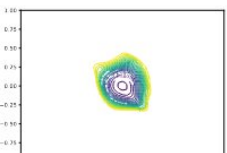
(e) SGD, 128, 5.89%



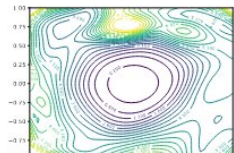
(f) SGD, 4096, 10.59%



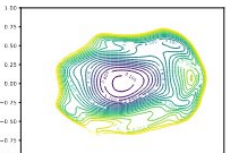
(i) SGD, 128, 8.26%



(j) SGD, 4096, 13.93%



(m) SGD, 128, 5.89%



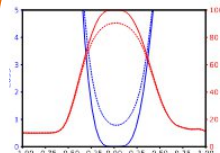
(n) SGD, 4096, 10.59%

## Adam

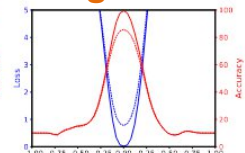
Weight  
Decay

= 0

Small Batch Large Batch

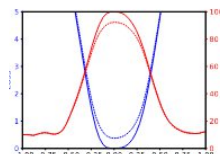


(c) Adam, 128, 9.55%

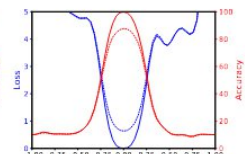


(d) Adam, 4096, 14.30%

=  $5e^{-4}$

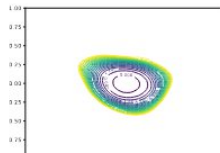


(g) Adam, 128, 7.67%

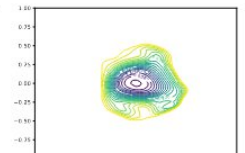


(h) Adam, 4096, 12.36%

= 0

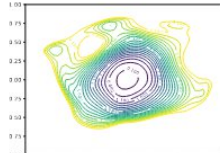


(k) Adam, 128, 9.55%

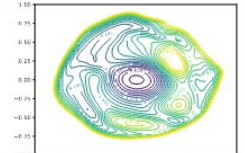


(l) Adam, 4096, 14.30%

=  $5e^{-4}$



(o) Adam, 128, 7.67%



(p) Adam, 4096, 12.36%

# 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

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

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

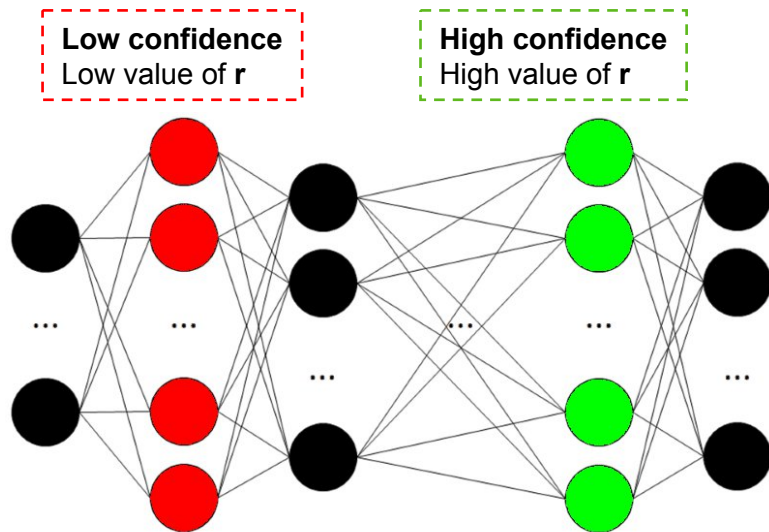
$$r_2 = \left\| m_i^l \right\|_2$$

$$r = r_1 / r_2$$

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

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

Weight decay



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

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

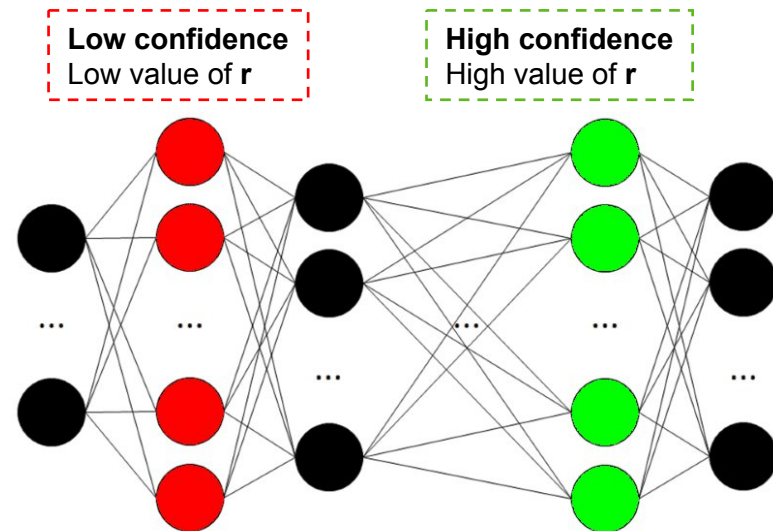
$$r_2 = \left\| \frac{\hat{m}_i^l}{\sqrt{\hat{v}_i^l + \epsilon}} + \lambda \theta_{i-1}^l \right\|_2$$

$$r = r_1 / r_2$$

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

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

Decoupled weight decay



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

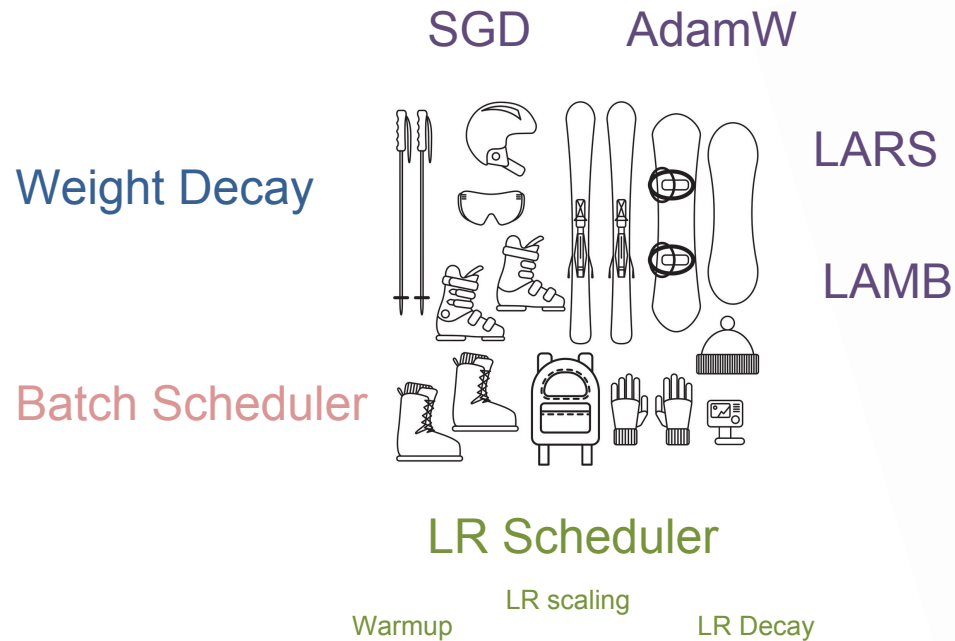
LARS  
optimisation  
from APEX

```
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



# BLOOM example

**BLOOM**  
176B params 59 languages Open-access



95281 steps (116.8 days)

AdamW,

$\beta_1=0.9$ ,  $\beta_2=0.95$ ,  $\text{eps}=1e-8$

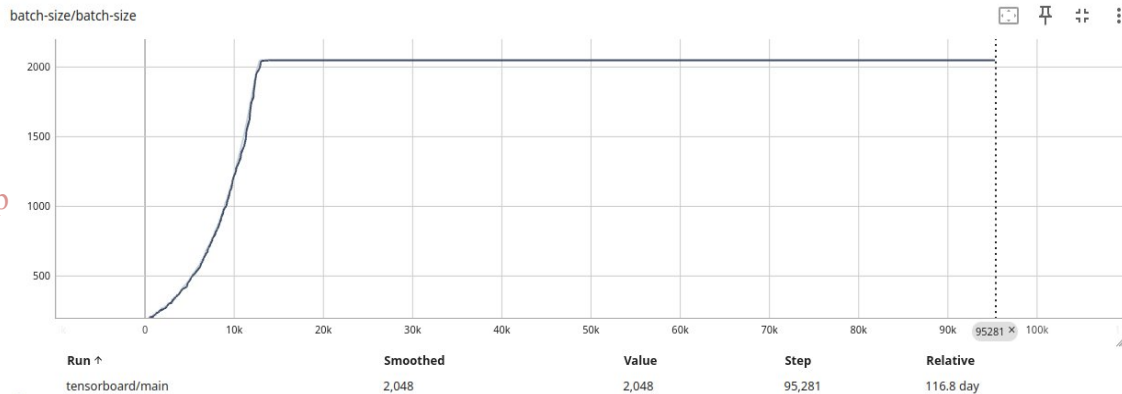
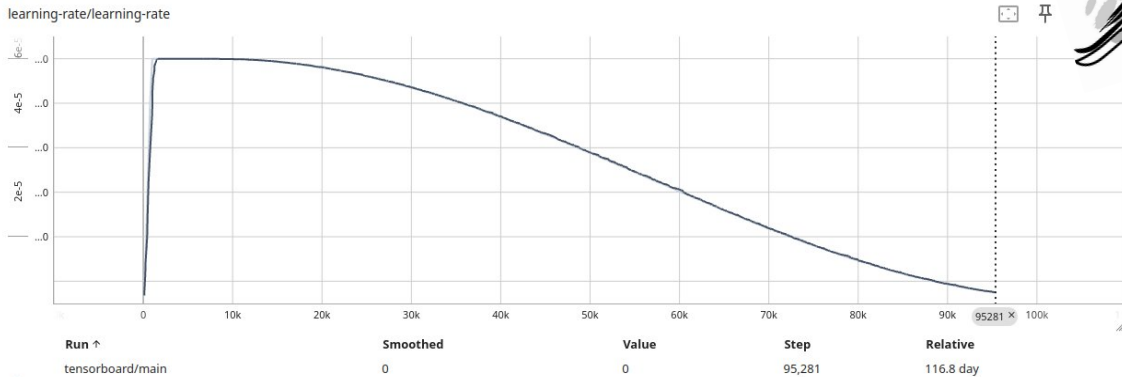
Weight Decay of 0.1

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  $\sim 20B$  tokens
- then continue at 4.2M tokens/step (GBS=2048) for 430B tokens



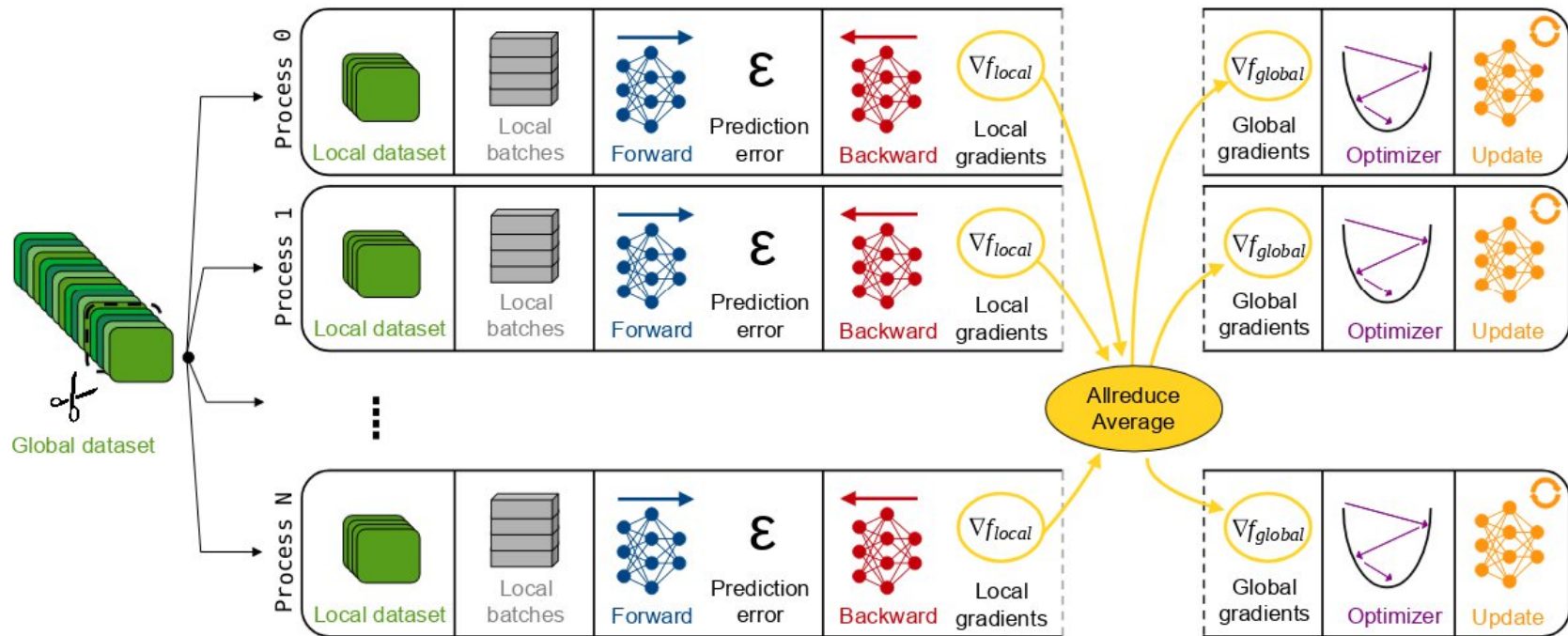
# Reducing Optimizer Communication Costs

AllReduce Bottleneck ◀

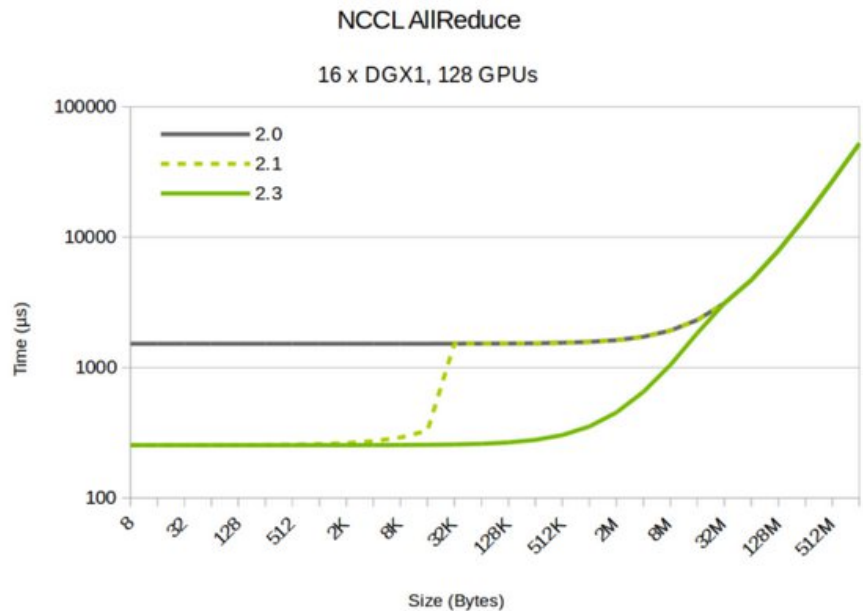
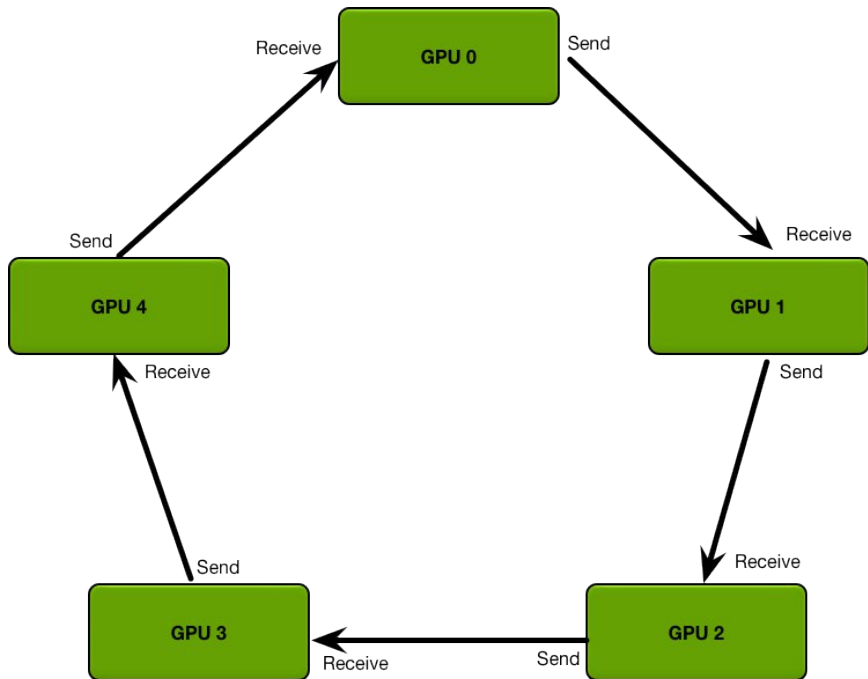
PowerSGD ◀

DiLoCo ◀

# AllReduce bottleneck



# DDP Communication Costs



# Gradient compression - PowerSGD

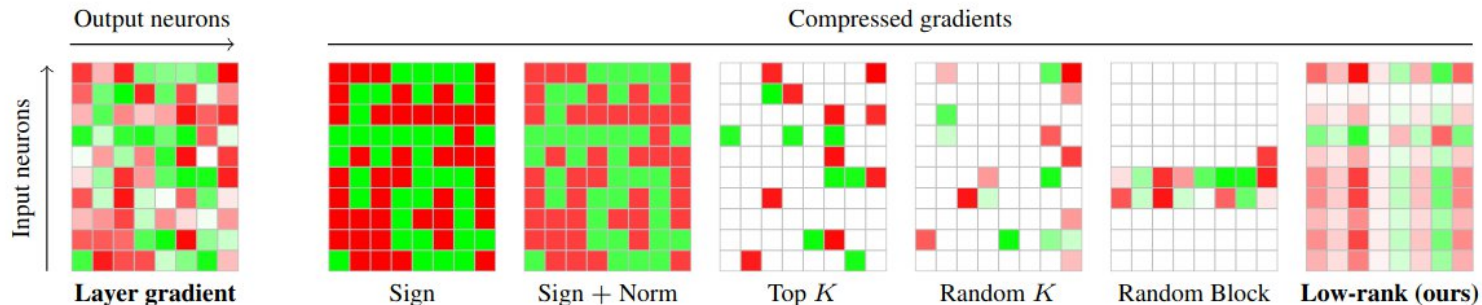


Figure 1: Compression schemes compared in this paper. Left: Interpretation of a layer’s gradient as a matrix. Coordinate values are color coded (**positive**, **negative**). Right: The output of various compression schemes on the same input. Implementation details are in Appendix G.

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% — $\leftrightarrow$	1023 MB (1 $\times$ )	312 ms +0%
Rank 1	93.6% — $\#$	4 MB (243 $\times$ )	229 ms -26%
Rank 2	94.4% — $\#$	8 MB (136 $\times$ )	239 ms -23%
Rank 4	94.5% — $\#$	14 MB (72 $\times$ )	260 ms -16%

Language modeling — LSTM on WIKITEXT-2

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

# Asynchronous Optimization - DiLoCo

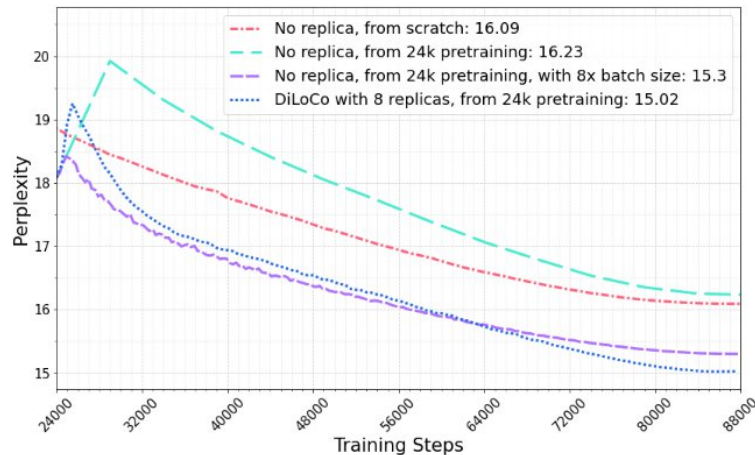
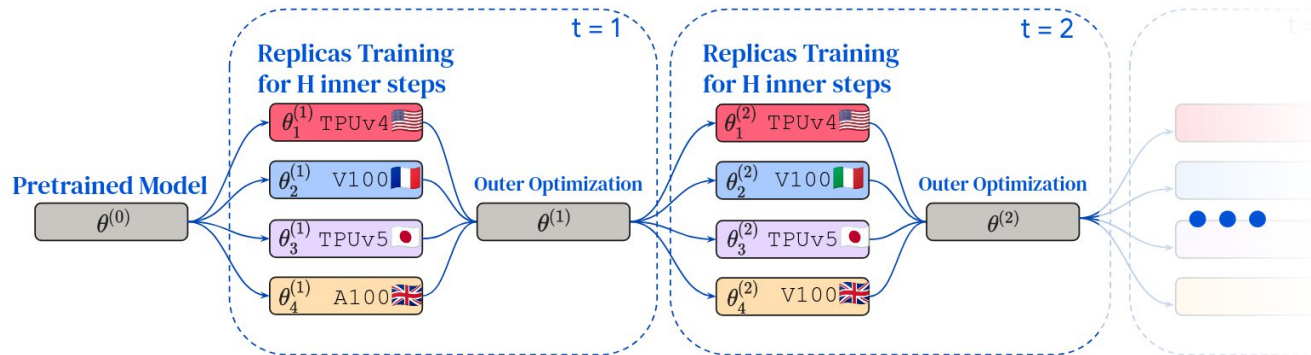
## Algorithm 1 DiLoCo Algorithm

---

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

- 1: **for** outer step  $t = 1 \dots T$  **do**
- 2:   **for** worker  $i = 1 \dots k$  **do**
- 3:      $\theta_i^{(t)} \leftarrow \theta^{(t-1)}$
- 4:     **for** inner step  $h = 1 \dots H$  **do**
- 5:        $x \sim \mathcal{D}_i$
- 6:        $\mathcal{L} \leftarrow f(x, \theta_i^{(t)})$
- 7:        ▸ Inner optimization:
- 8:        $\theta_i^{(t)} \leftarrow \text{InnerOpt}(\theta_i^{(t)}, \nabla \mathcal{L})$
- 9:     **end for**
- 10:   **end for**
- 11:     ▸ Averaging outer gradients:
- 12:      $\Delta^{(t)} \leftarrow \frac{1}{k} \sum_{i=1}^k (\theta^{(t-1)} - \theta_i^{(t)})$
- 13:     ▸ Outer optimization:
- 14:      $\theta^{(t)} \leftarrow \text{OuterOpt}(\theta^{(t-1)}, \Delta^{(t)})$
- 15: **end for**

---



# 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<sup>1</sup> Jitendra Malik<sup>1</sup>

**VeLO: Training Versatile Learned Optimizers**  
Luke Metz<sup>2</sup> James Harrison<sup>1</sup> C. Daniel Freeman, Amil Merchant,  
Lucas Beyer, James Bradbury, Naman Agarwal, Ben Poole,  
Igor Mordatch, Adam Roberts, Jascha Sohl-Dickstein<sup>2</sup>  
Google Research, Brain Team

**Neural Optimizer Search with Reinforcement Learning**  
Irwan Bello<sup>1\*</sup> Barret Zoph<sup>1\*</sup> Vijay Vasudevan<sup>1</sup> Quoc V. Le<sup>1</sup>

**Learning to learn by gradient descent  
by gradient descent**  
Marcin Andrychowicz<sup>2</sup>, Misha Denil<sup>1</sup>, Sergio Gómez Colmenarejo<sup>1</sup>, Matthew W. Hoffman<sup>1</sup>,  
David Pfau<sup>2</sup>, Tom Schaul<sup>1</sup>, Brendan Shillingford<sup>1,2</sup>, Nando de Freitas<sup>1,2,3</sup>  
<sup>1</sup>Google DeepMind <sup>2</sup>University of Oxford <sup>3</sup>Canadian Institute for Advanced Research  
marcin.andrychowicz@gmail.com  
{denil, sergomez, mhoffman, pfau, schaul}@google.com  
brendan.shillingford@cs.ox.ac.uk, nandodefruits@google.com



# 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  $\theta_t$  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  $\theta_t$ 
```

---

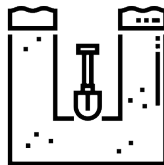
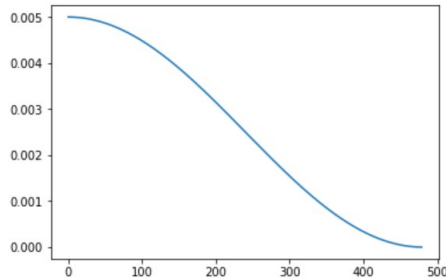
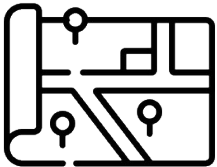
---

## Algorithm 2 Lion Optimizer (ours)

---

```
given  $\beta_1, \beta_2, \lambda, \eta, f$   
initialize  $\theta_0, m_0 \leftarrow 0$   
while  $\theta_t$  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  $\theta_t$ 
```

---

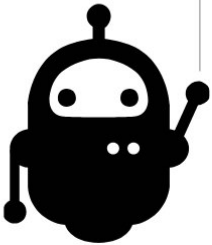


# Pratice : Learning rate + Optimiseurs



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