

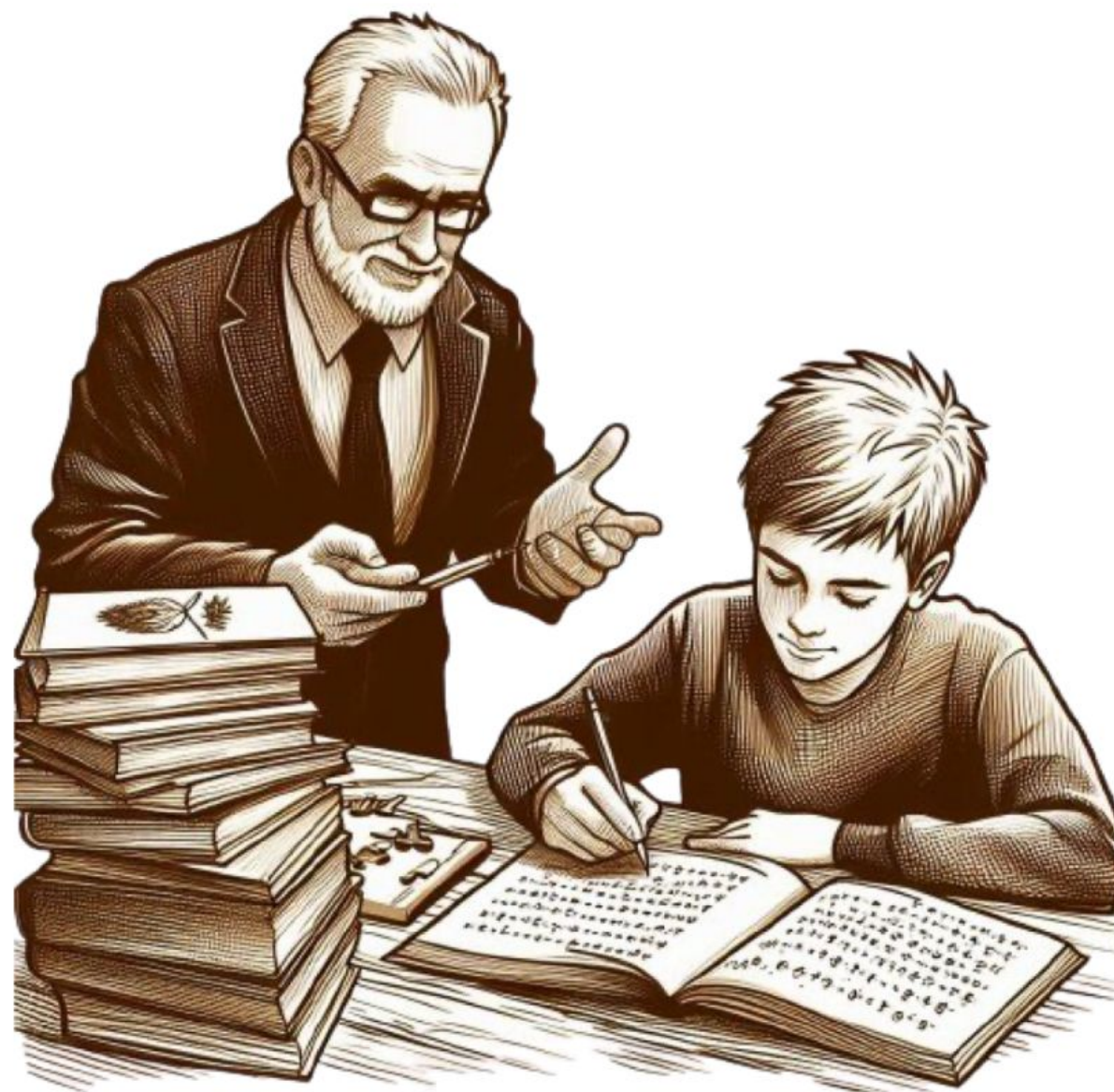
Bsc Data Science for Responsible Business

Deep Learning Course

**Methodology : How to train Neural
Networks efficiently ?**

Emmanuel Dellandrea - emmanuel.dellandrea@ec-lyon.fr

Generalization

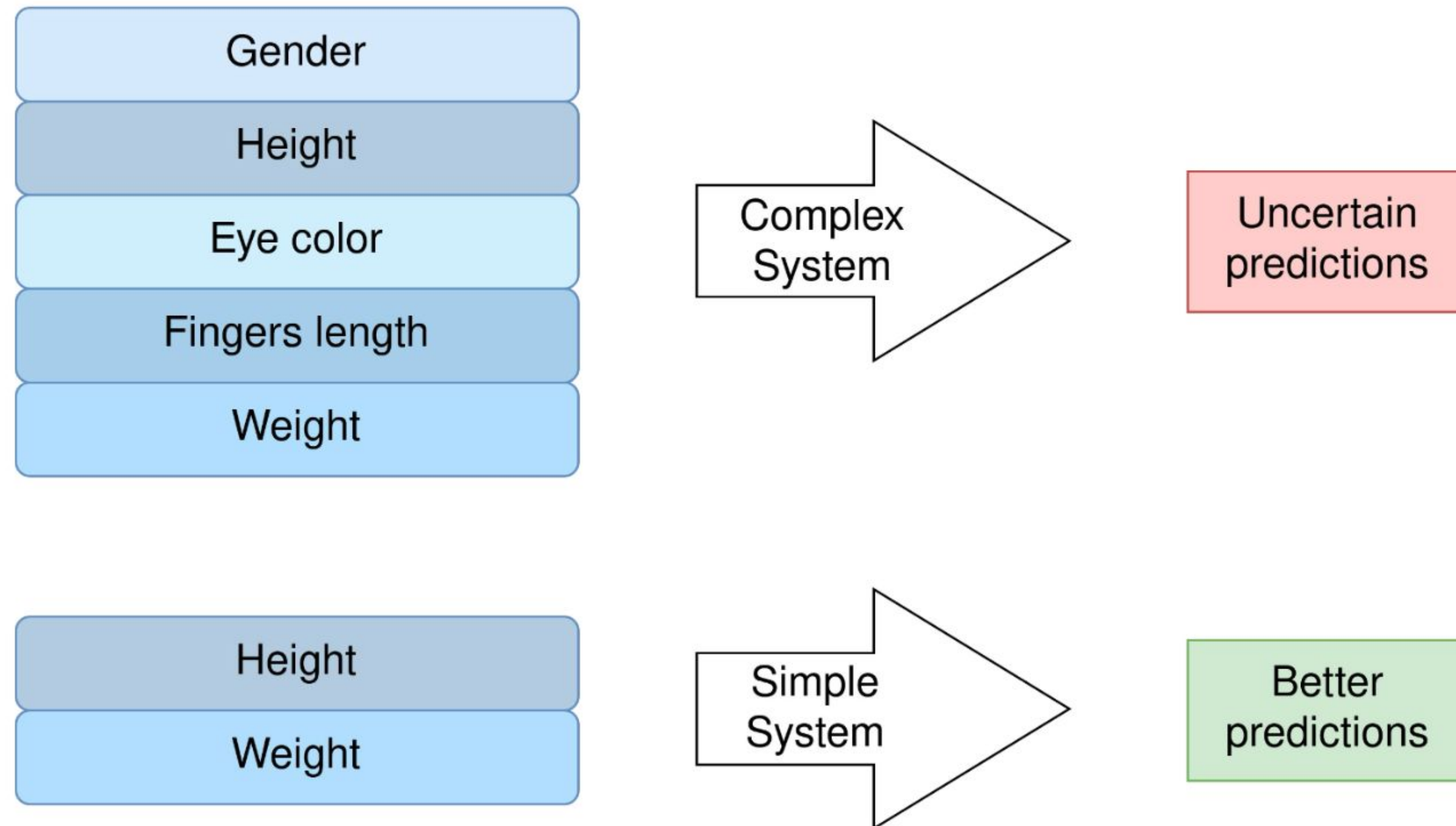


Learning from exercise with a teacher to guide us



Applying what we learn to the real world

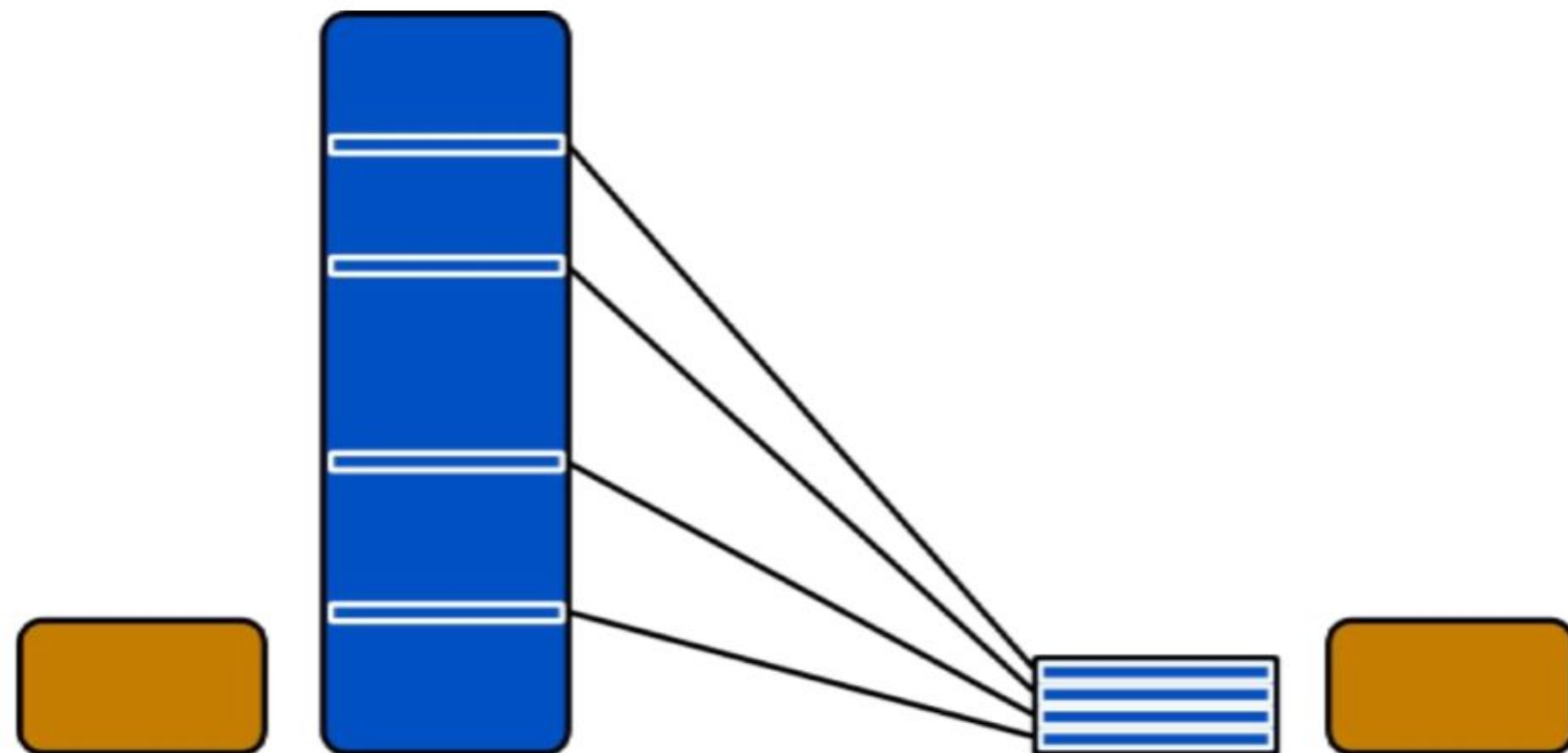
Features



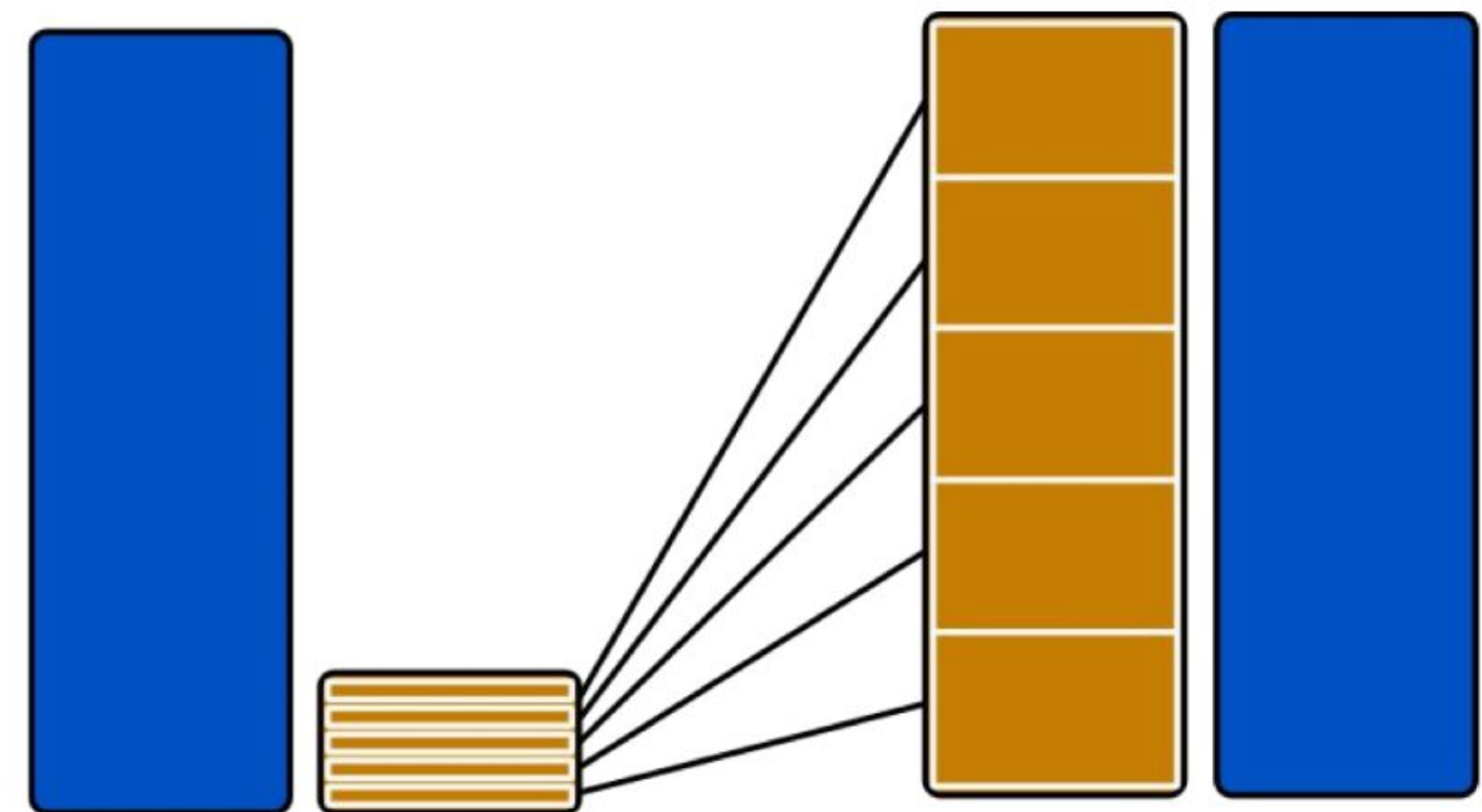
**Diabetes risk
prediction system**

Balance the classes

Undersampling

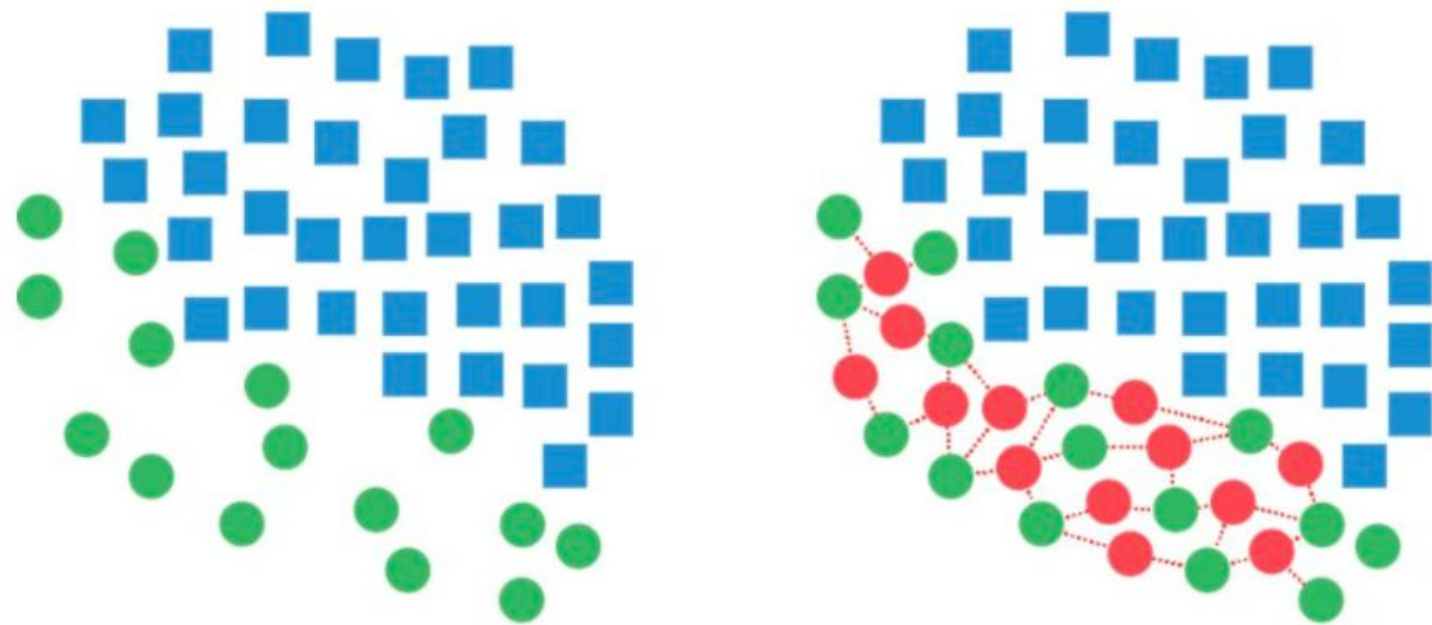


Oversampling



Data augmentation

Synthetic Minority Oversampling Technique



Data Augmentation



Original Image

De-texturized

De-colored

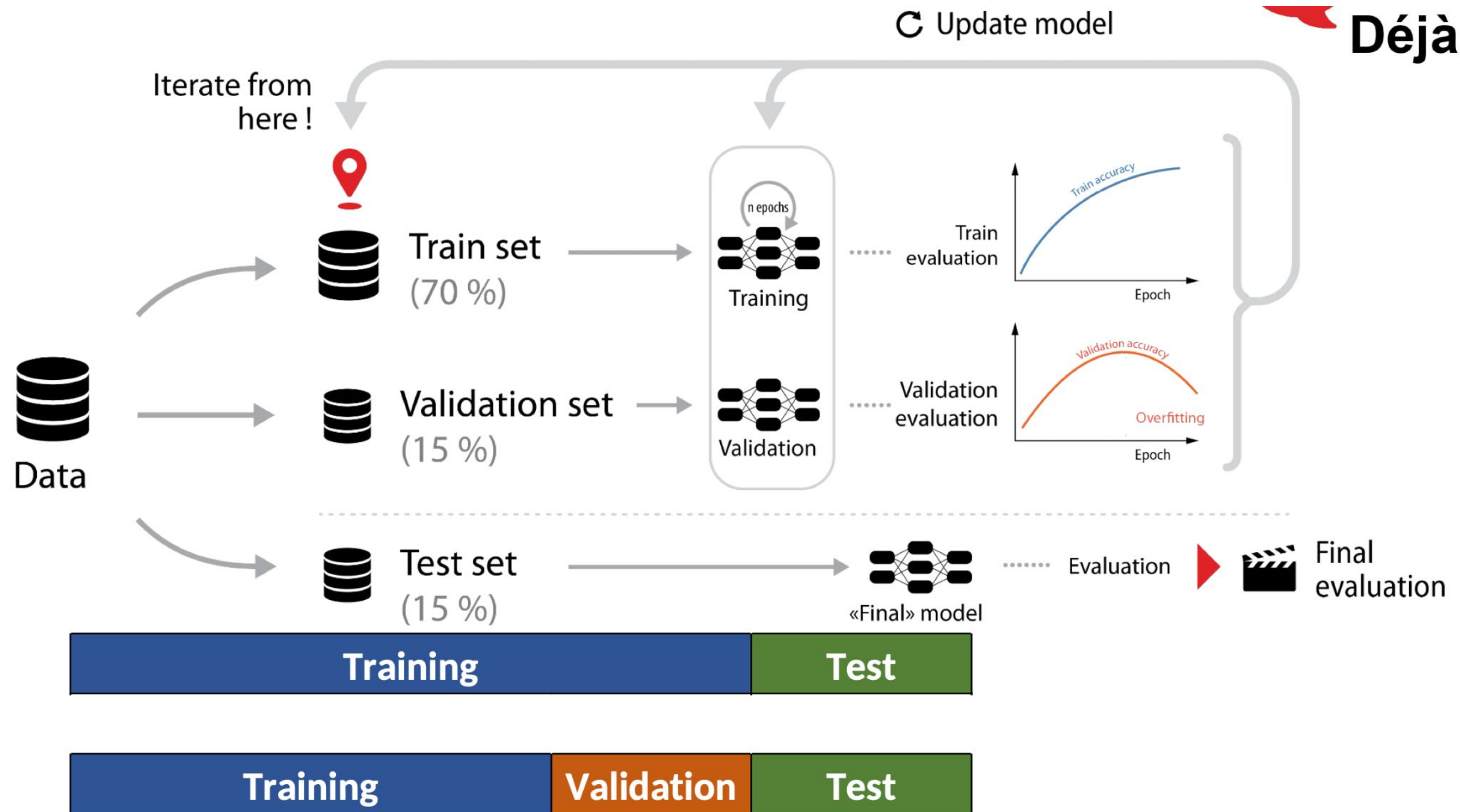
Edge Enhanced

Salient Edge Map

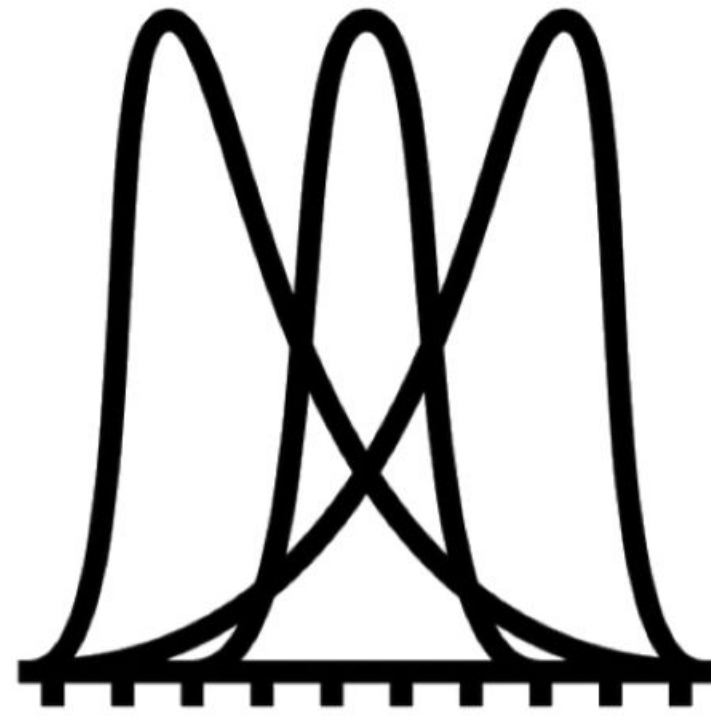
Flip/Rotate

```
# Define the transformations
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.RandomResizedCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```


Data splitting

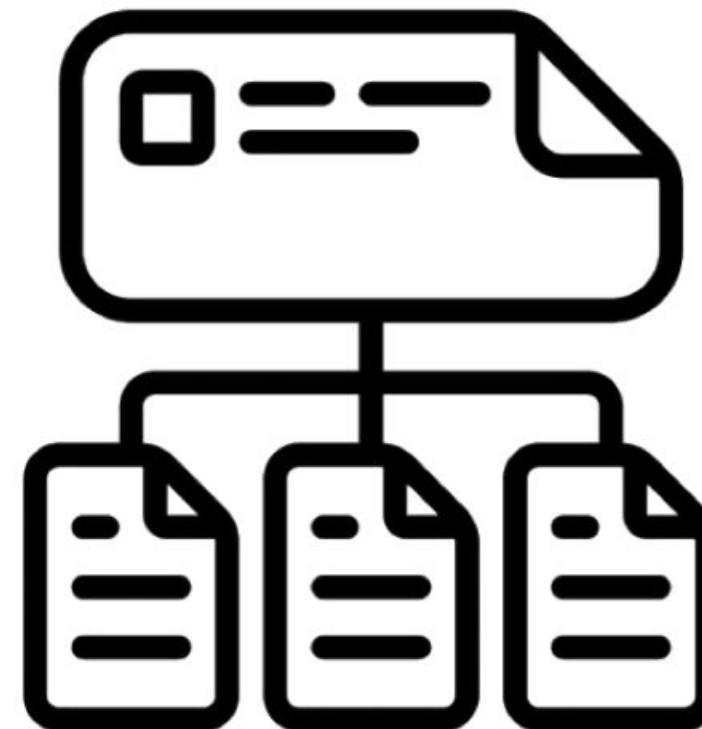
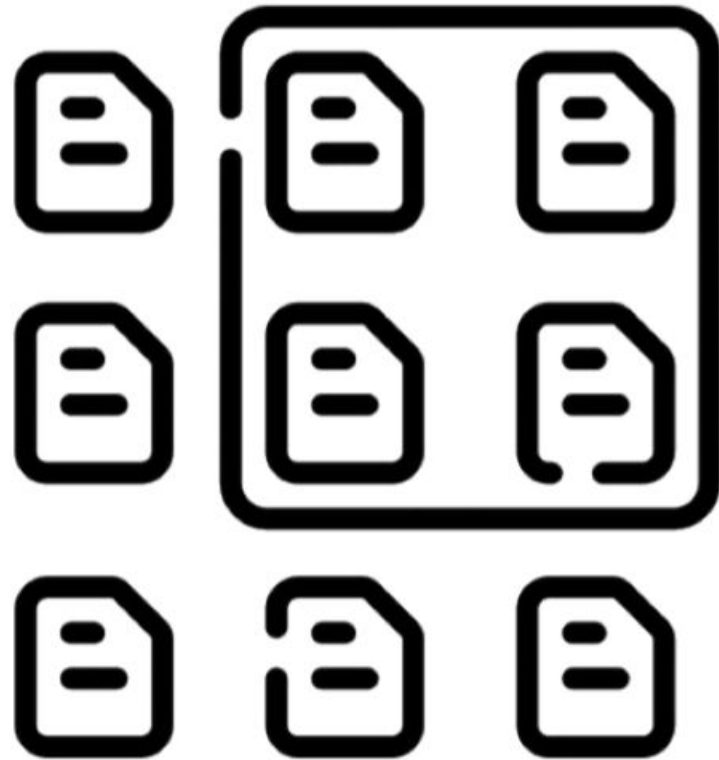


Dataset improvement

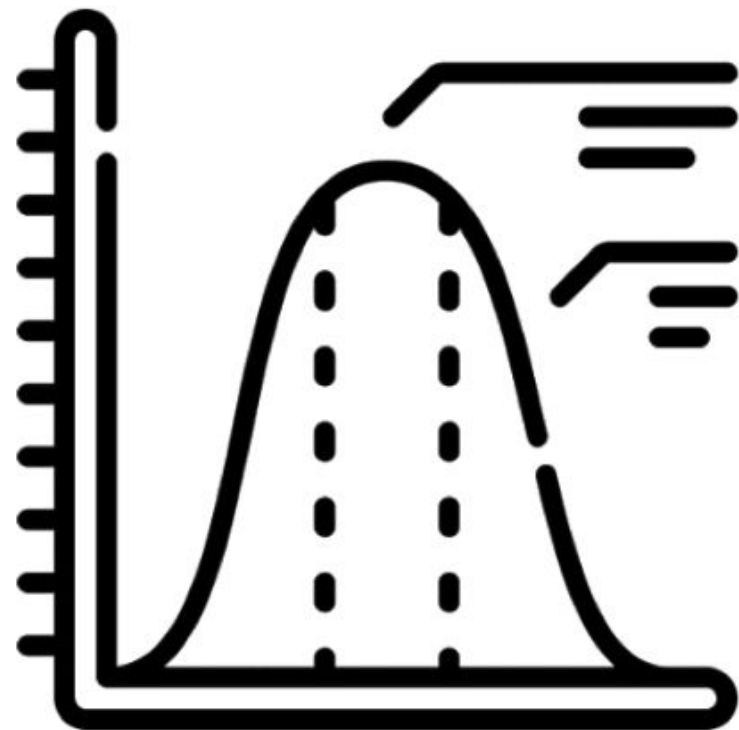


On a **dataset scale** :

- Split your dataset (train, test, val)
- Manage your misproportions and irregularities
- Use data augmentation to have more diversity



Data improvement



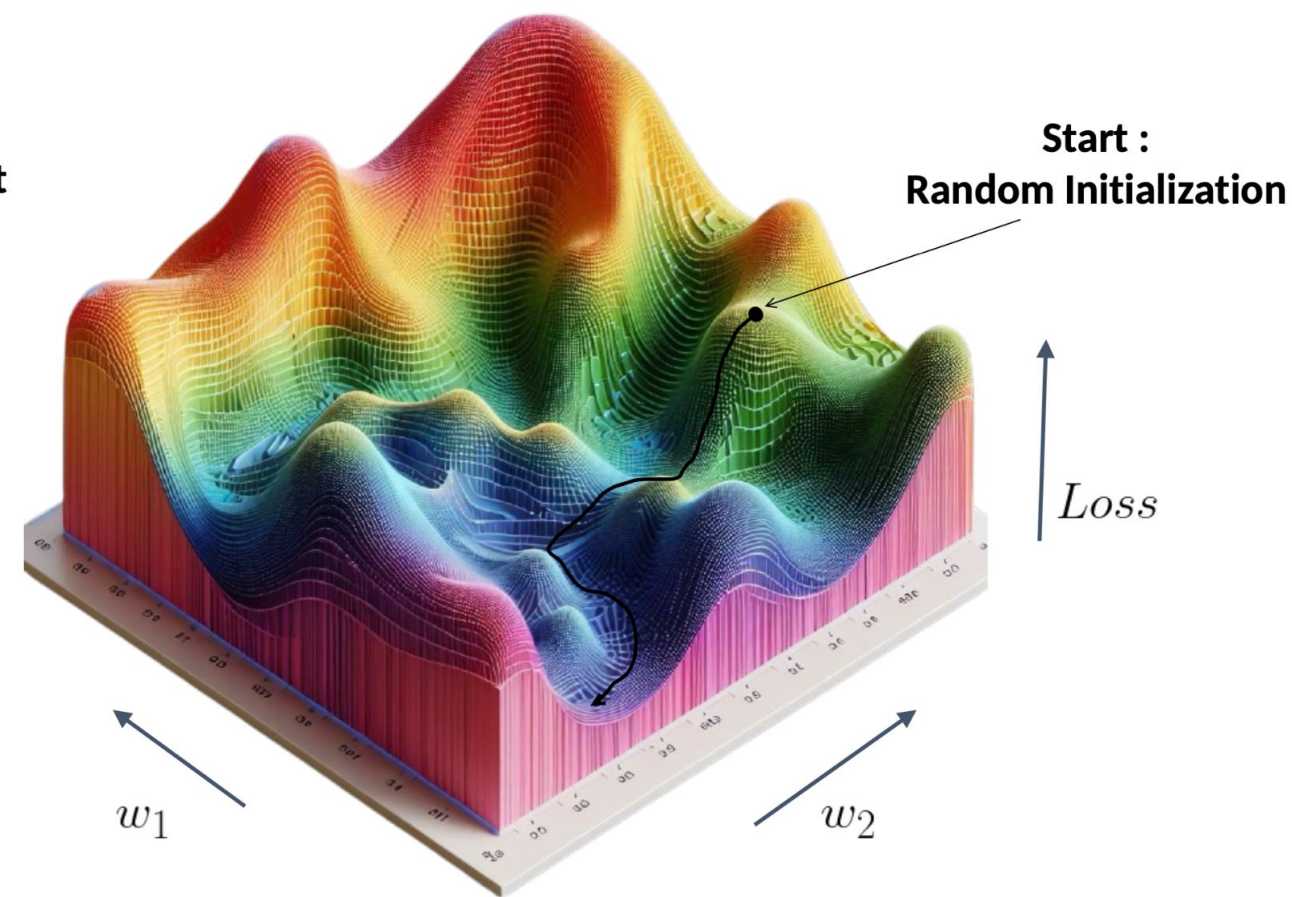
On a per **data scale** :

- Scale features
- Clean bad data
- Select features

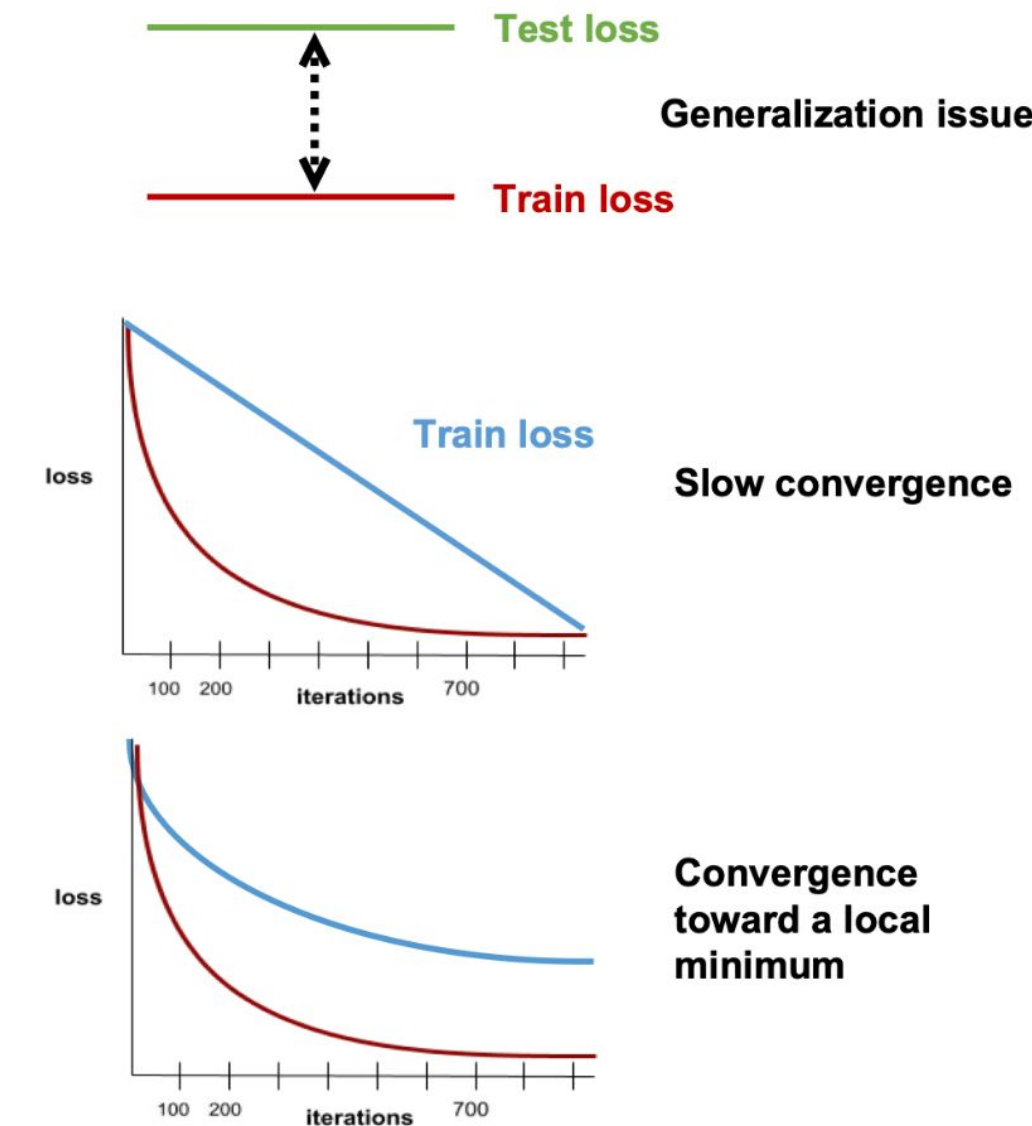
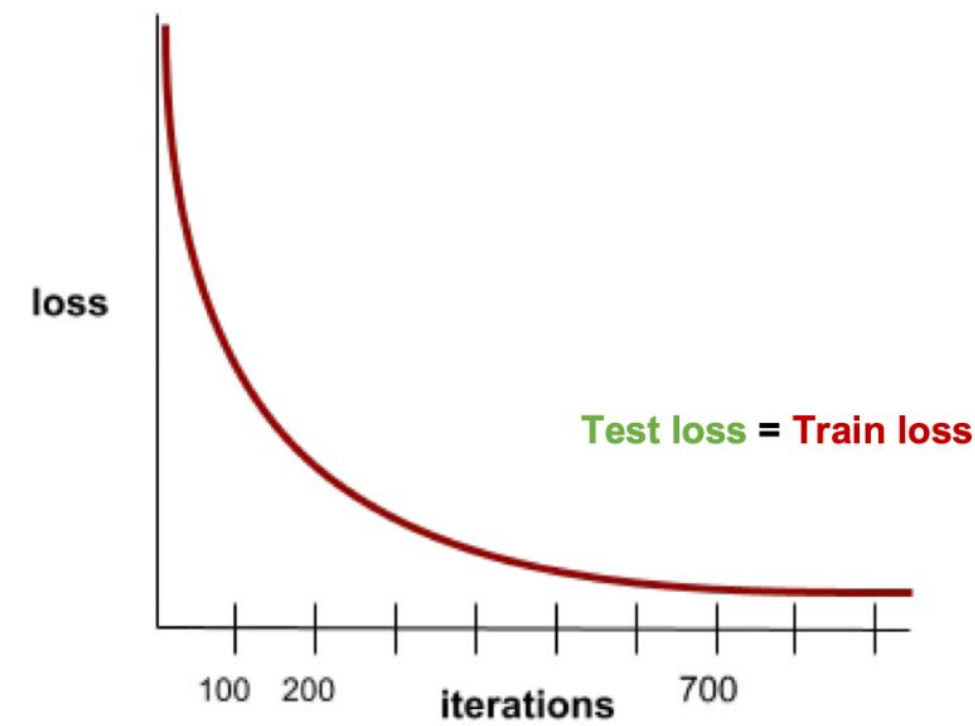
Difficulties to converge

Reality

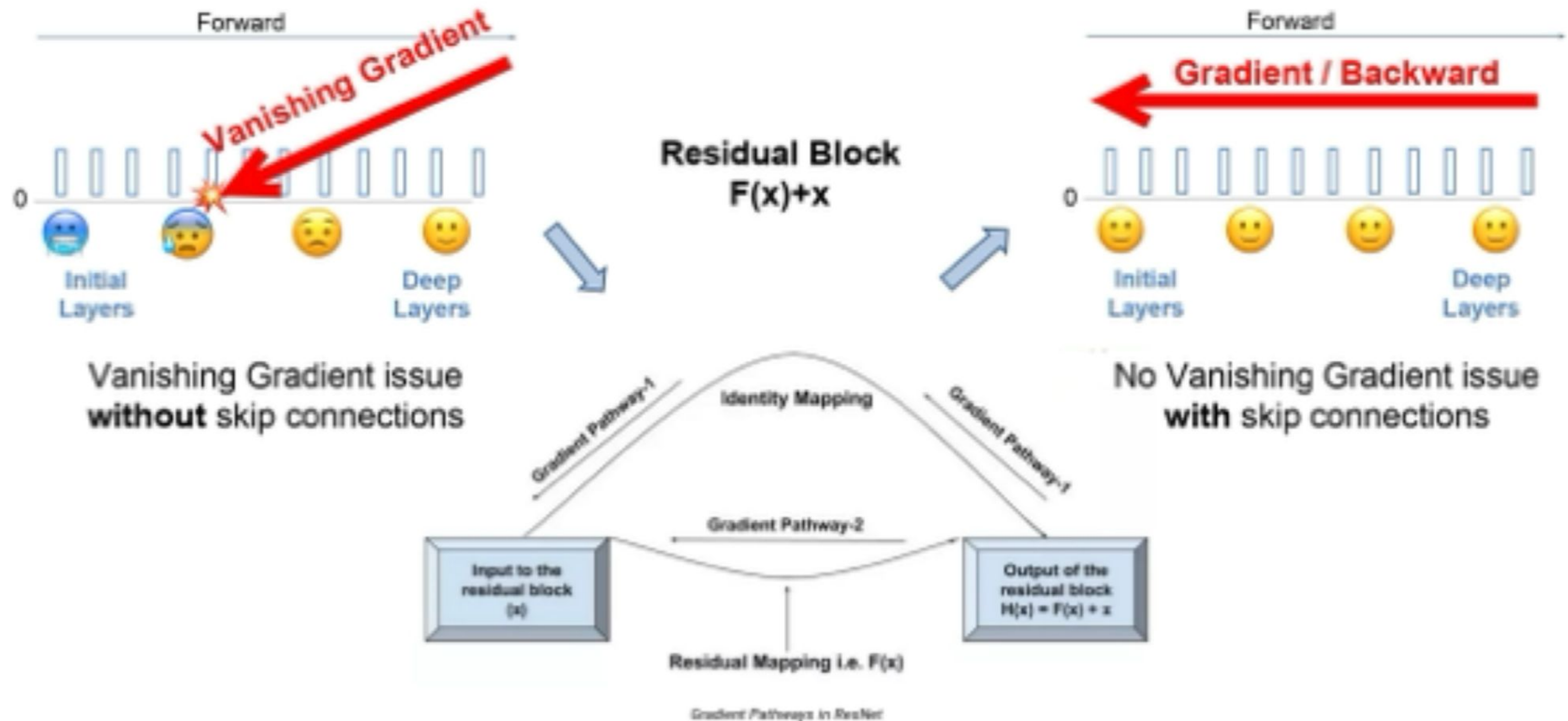
Gradient Descent



Expectation



Residual connections

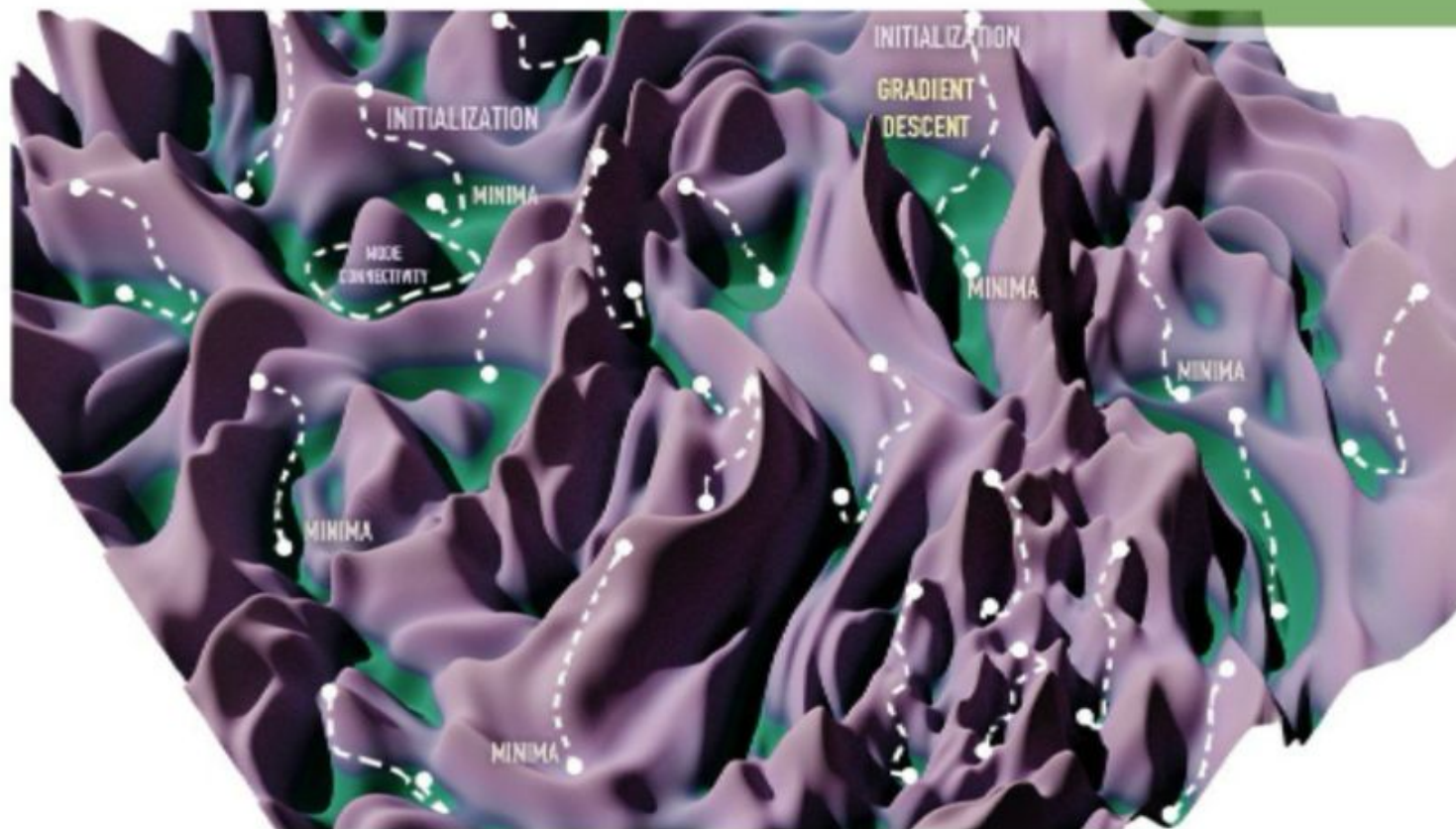


Model parameters initialization

The Blessing of Dimensionality :

Local

NEARBY PATHS
TO CONVERGENCE



FINDING A MINIMA BECOMES A "LOCAL" CHALLENGE



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

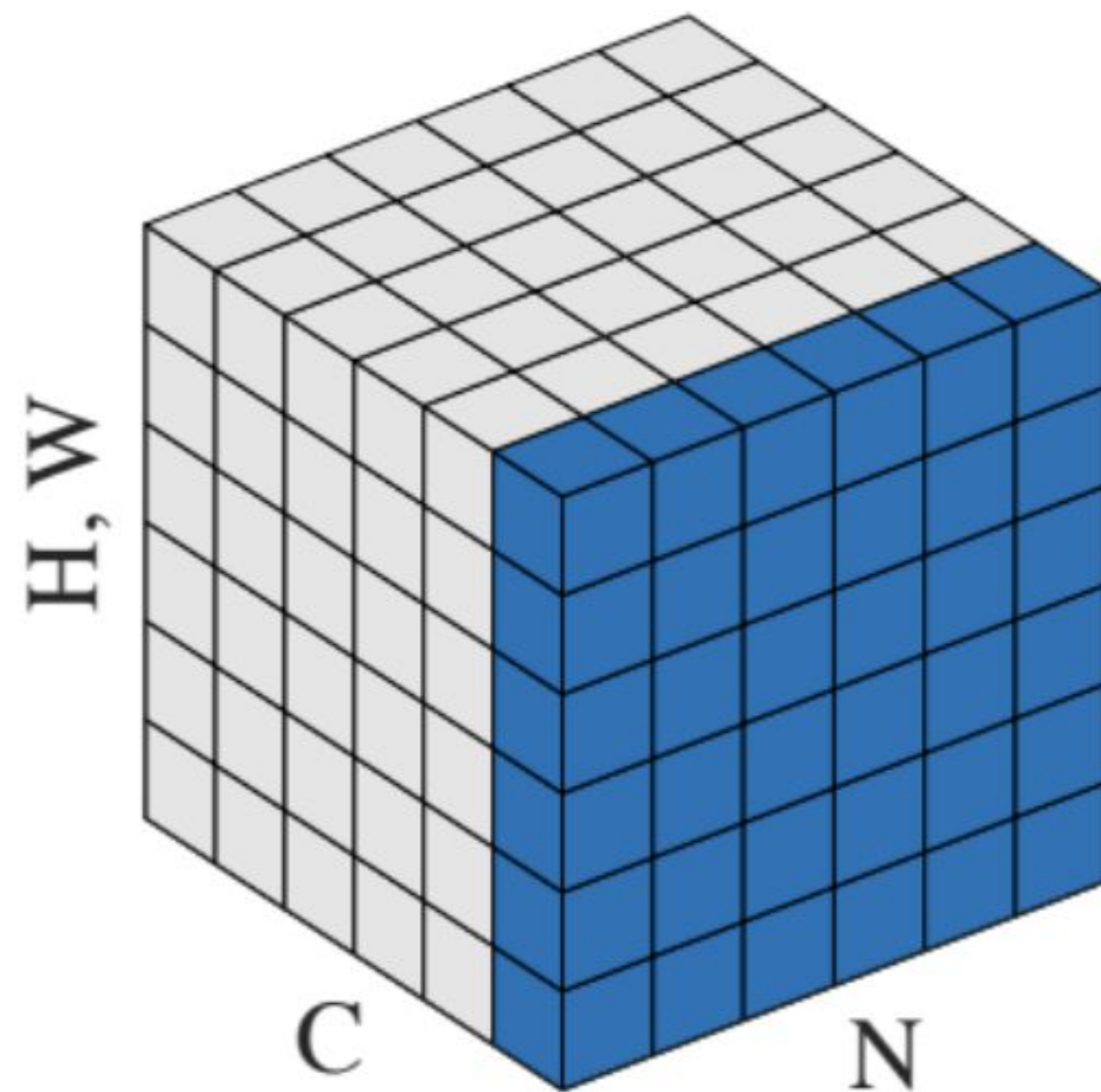
By default in PyTorch:

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

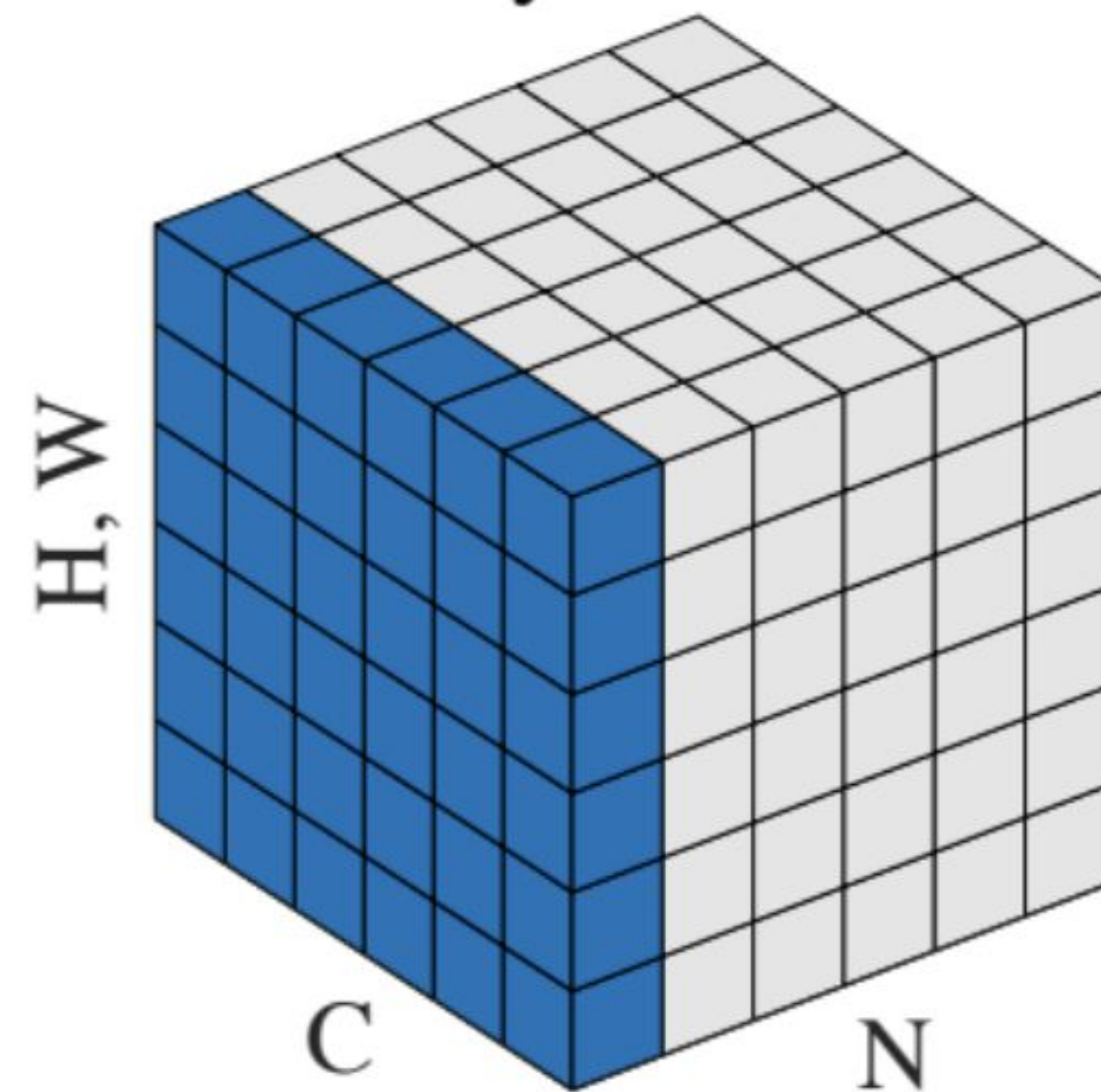
Normalization

Normalization improves both the speed and stability of deep learning models, making training more efficient and robust

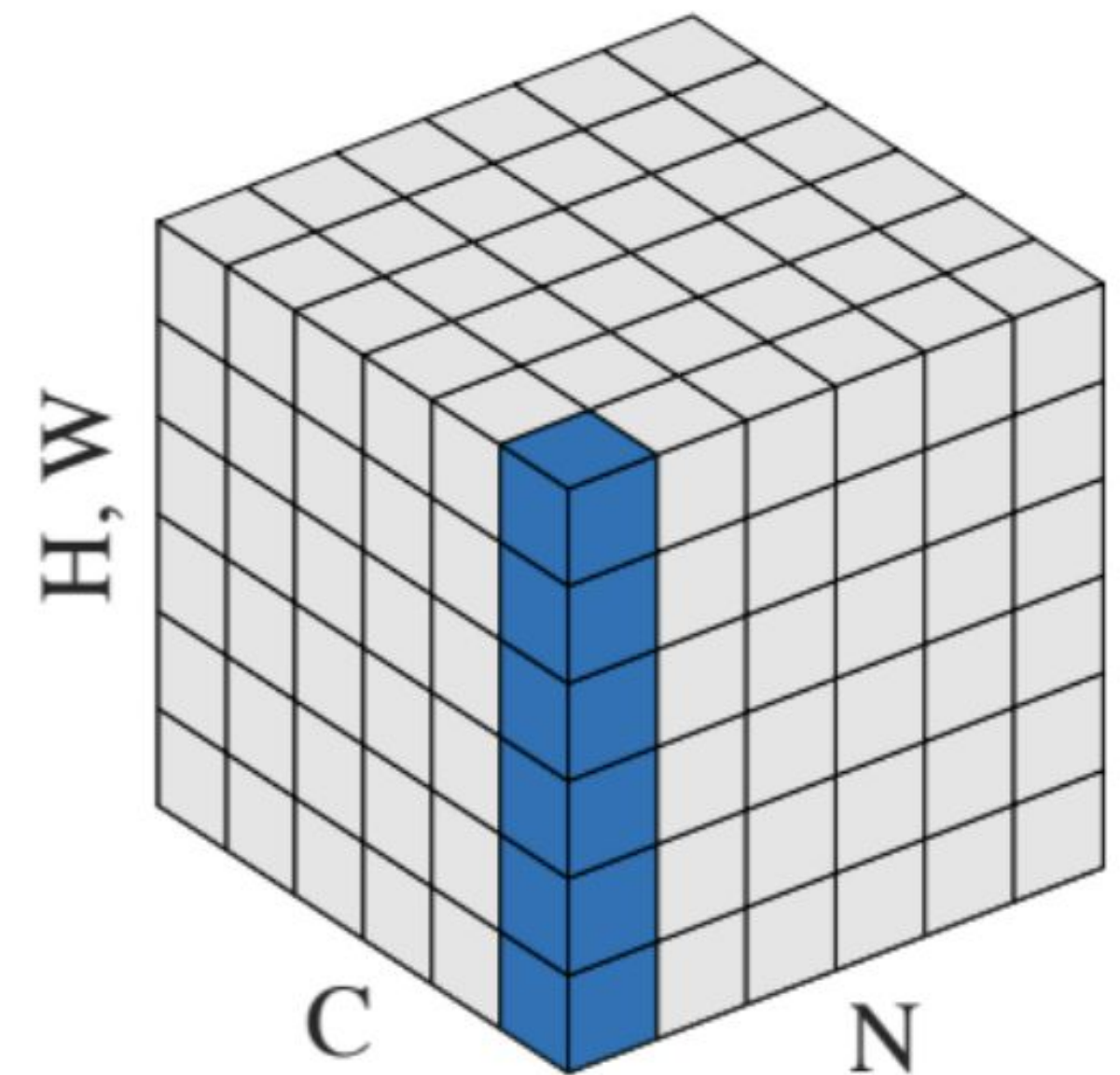
Batch Norm



Layer Norm

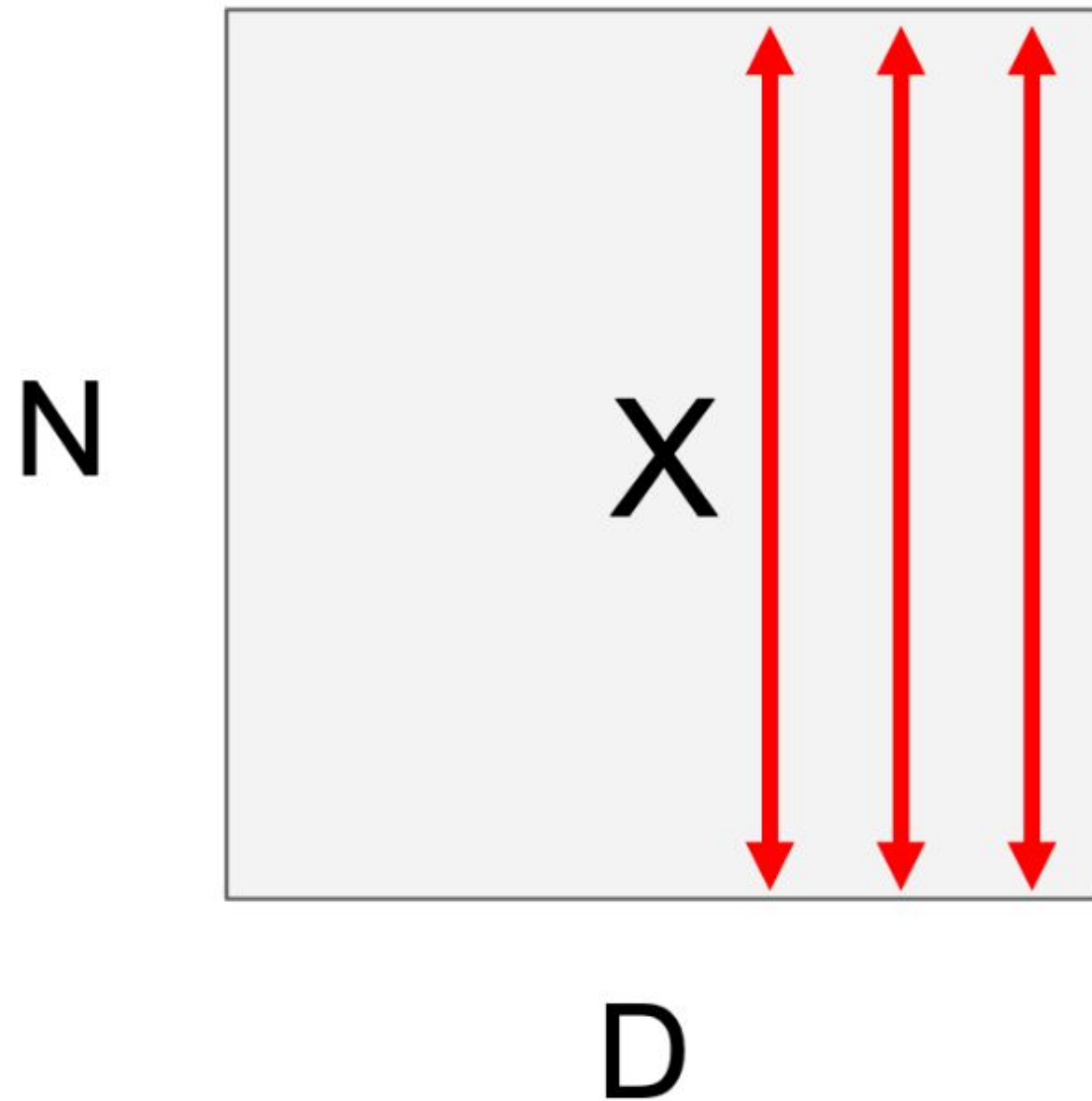


Instance Norm



Batch Normalization

Input: $x : N \times D$



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,
shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel var,
shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x,
Shape is N x D

Batch Normalization

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,
shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel var,
shape is D

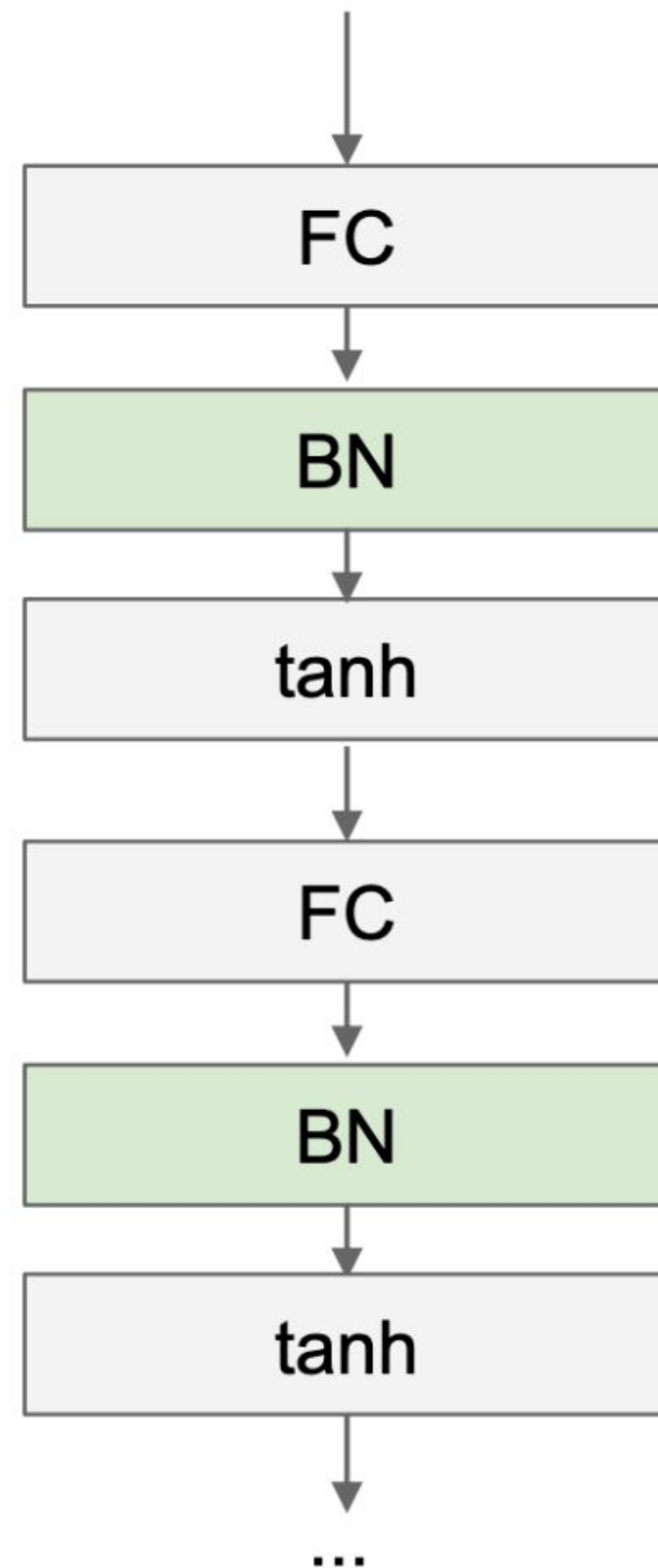
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x,
Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output,
Shape is N x D

Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Batch Normalization for CNNs

Batch Normalization for
fully-connected networks

$$\mathbf{x} : \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{1} \times \mathbf{D}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$$\mathbf{x} : \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1}$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Layer Normalization

Batch Normalization for
fully-connected networks

$$\mathbf{x} : \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{1} \times \mathbf{D}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Layer Normalization for fully-
connected networks
Same behavior at train and test!
Can be used in recurrent networks

$$\mathbf{x} : \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{N} \times \mathbf{1}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Instance Normalization

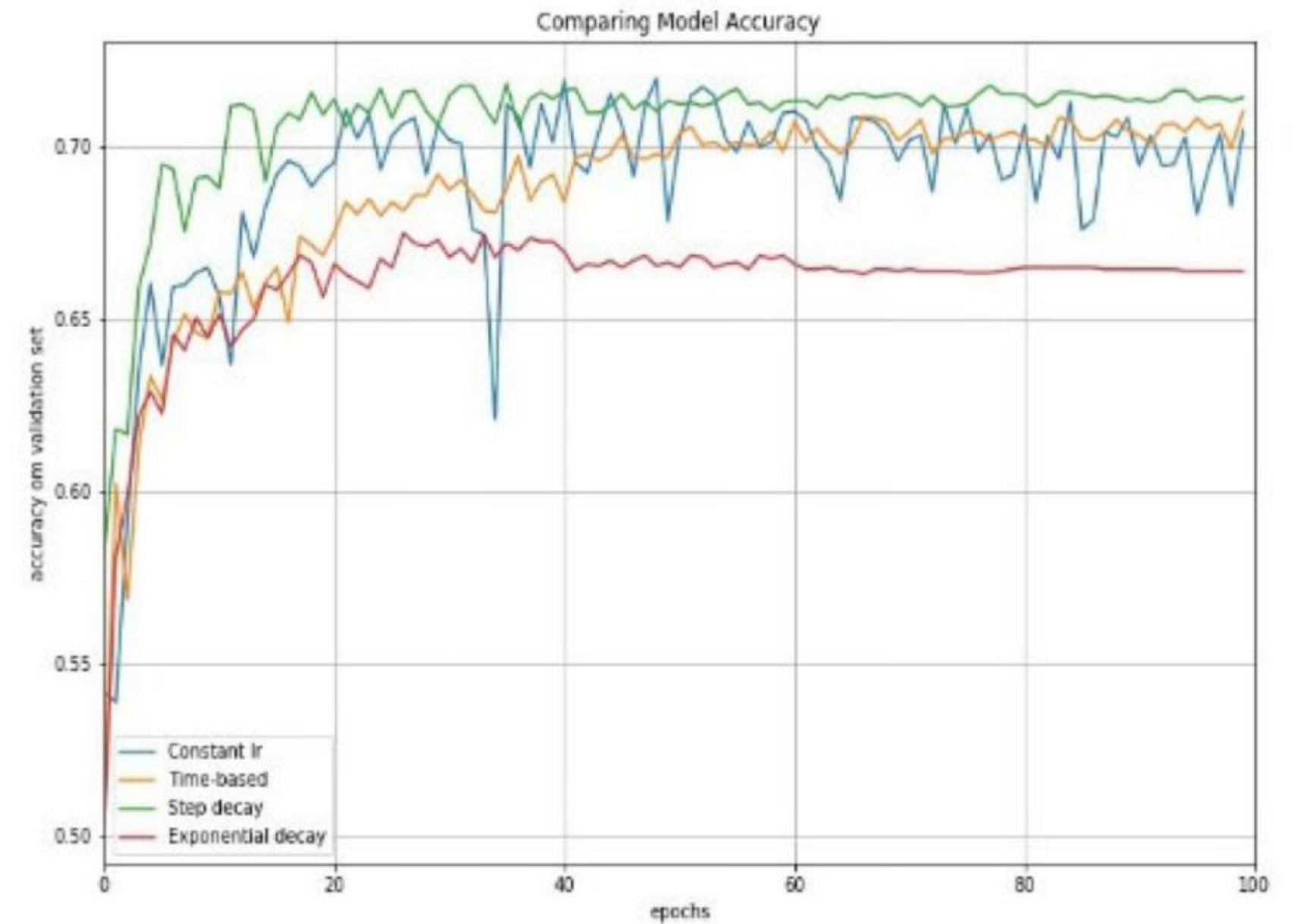
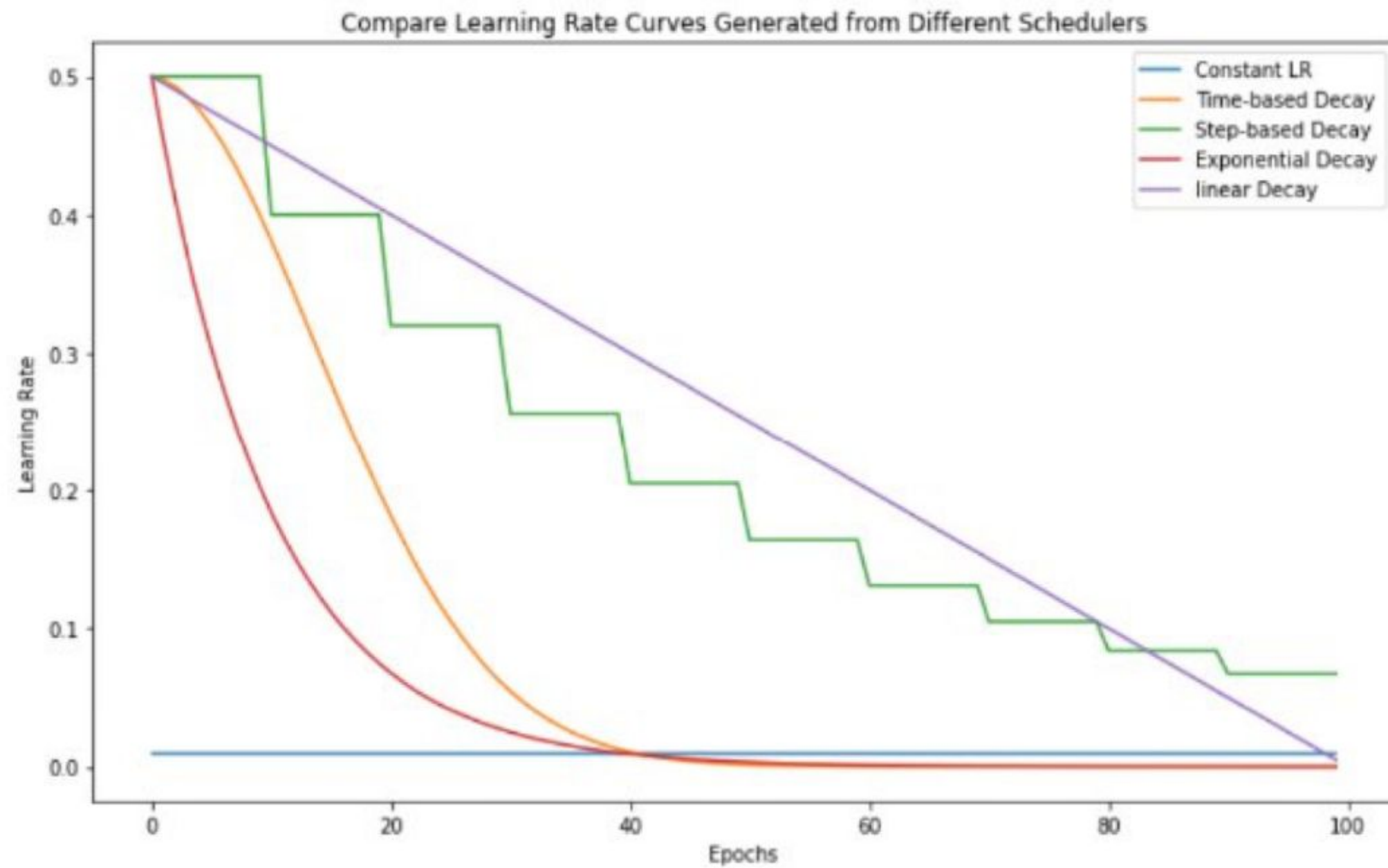
Batch Normalization for
convolutional networks

$$\begin{array}{l} \mathbf{x} : \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} \quad \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

Instance Normalization for
convolutional networks
Same behavior at train / test!

$$\begin{array}{l} \mathbf{x} : \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} \quad \quad \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{N} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

Learning rate decay

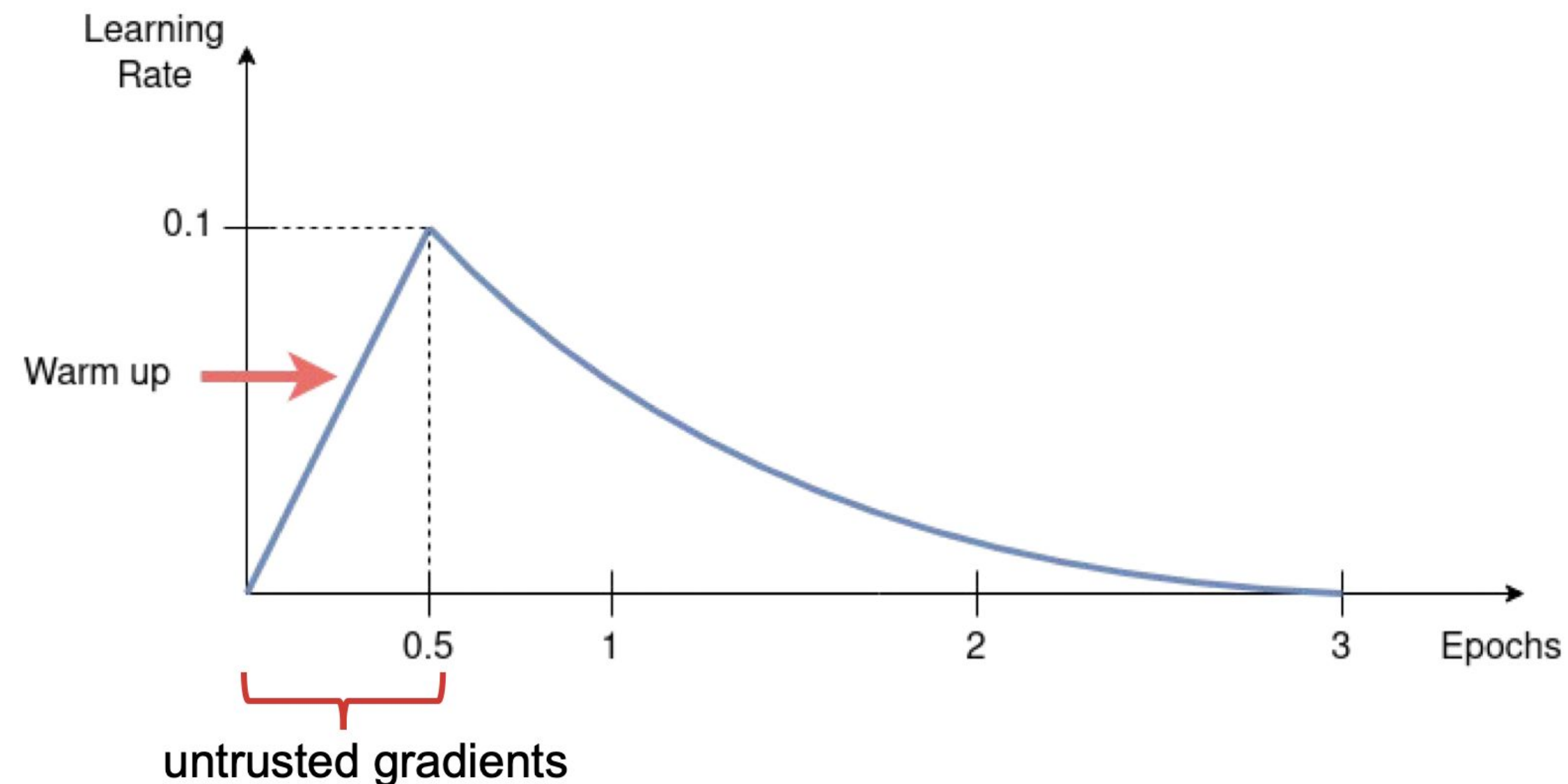


Learning rate scheduler

Problem: **Early iterations** have too much effect on the model (large loss, high gradients, bias, ...),

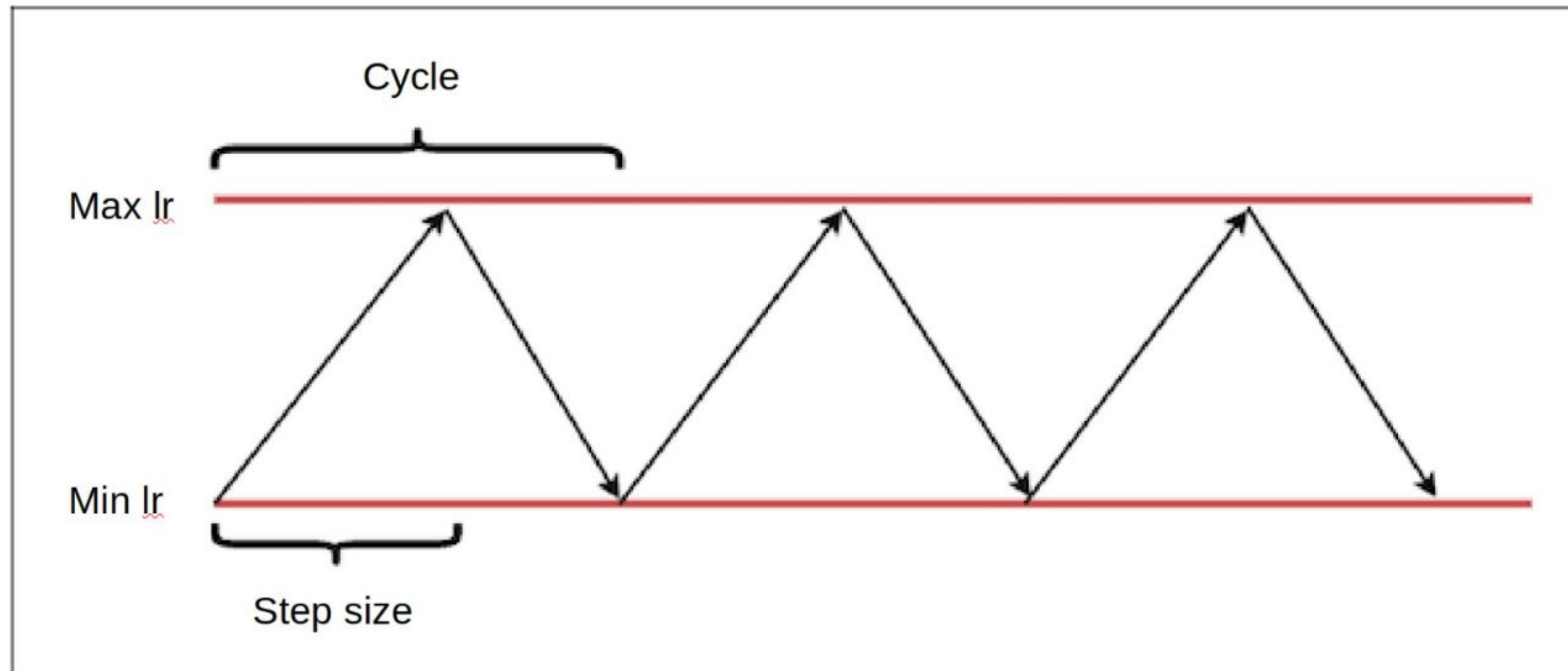
high learning rate can cause high instability or divergence

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



Learning rate

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

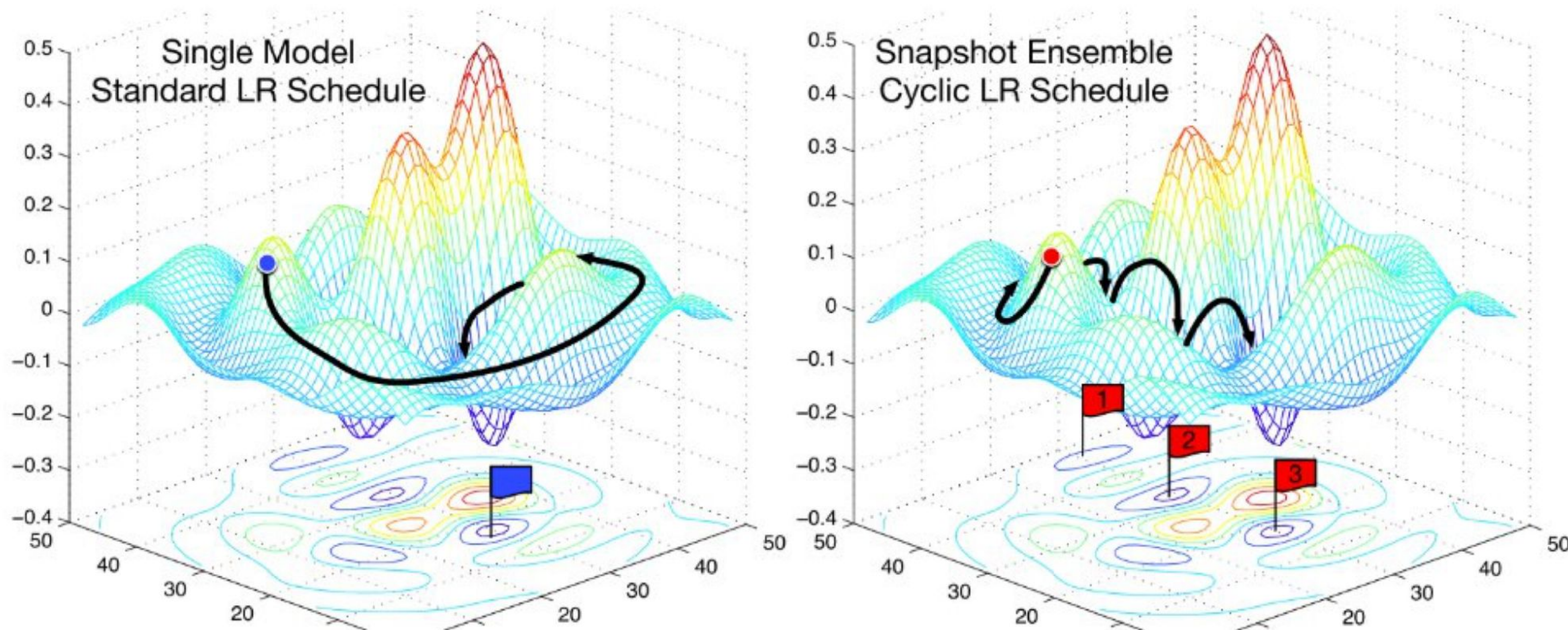


Parameters :

- Step_size : $x \cdot \text{epoch}$ ($2 \leq x \leq 10$)
- Base_lr = > min convergence value
- max_lr => max value before divergence

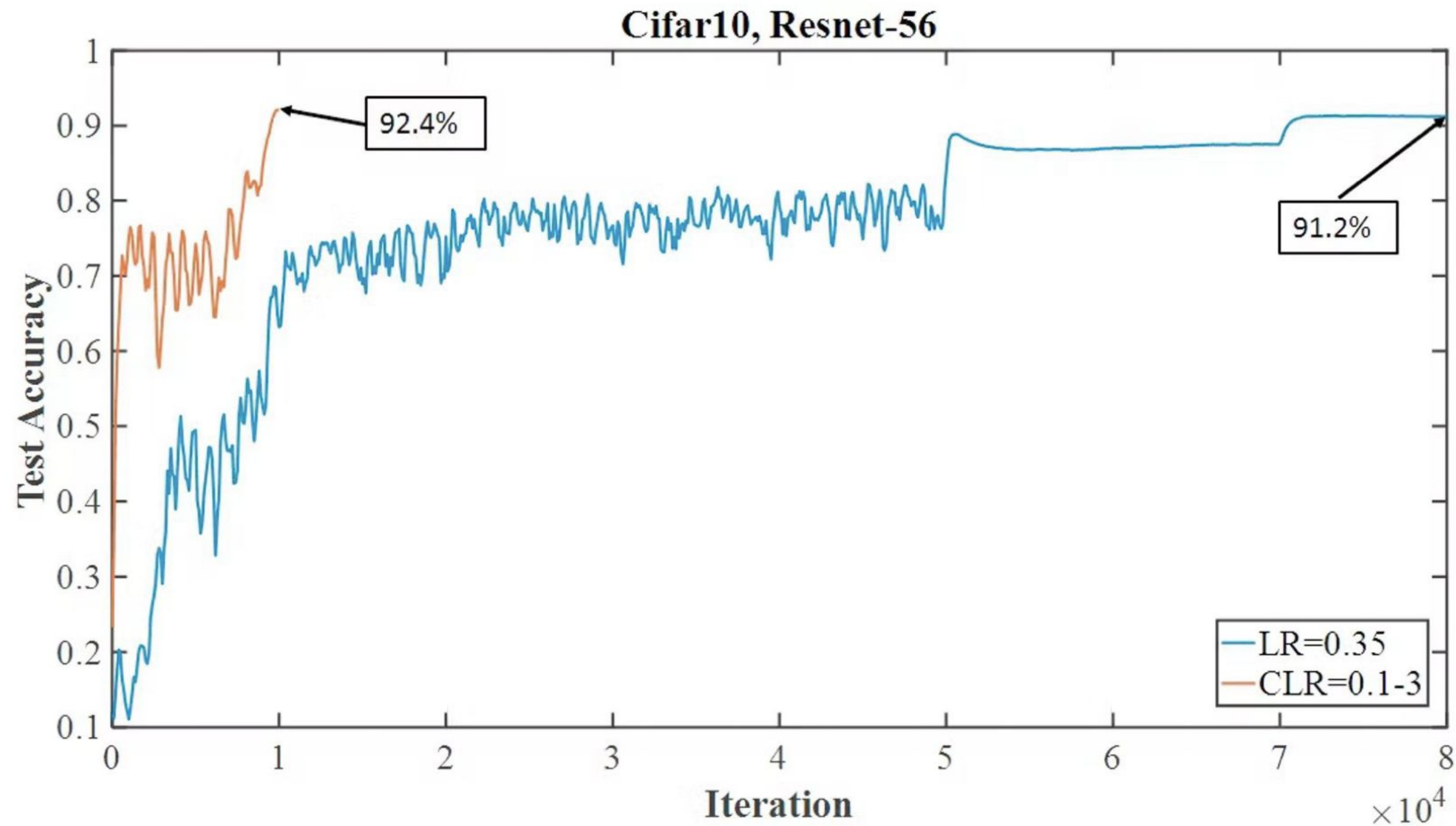
Successions of warmups
and learning rate decays

Learning rate

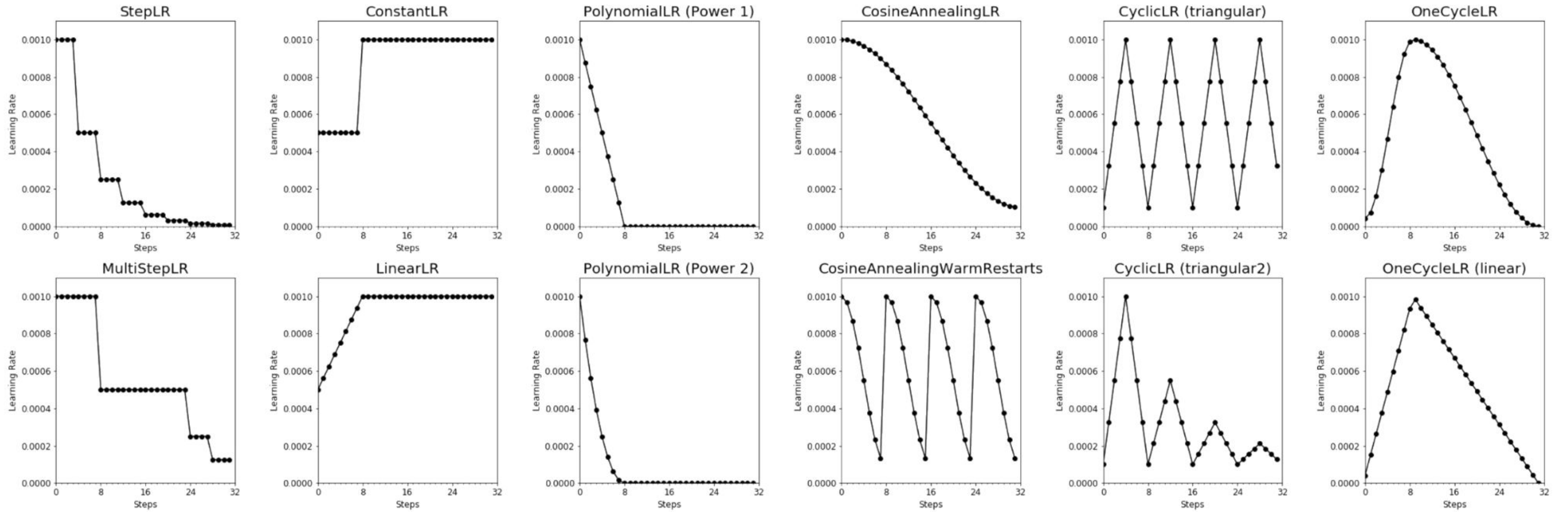


SNAPSHOT ENSEMBLES: TRAIN 1, GET M FOR FREE
Gao Huang, Yixuan Li, Geoff Pleiss

One cycle learning rate



Learning rate



Learning rate

```
import torch.optim as opt

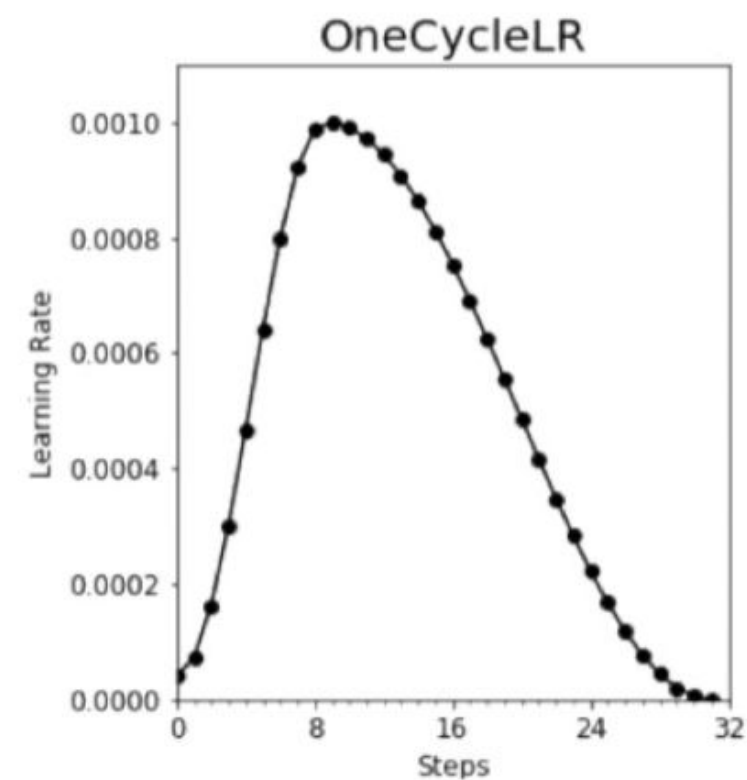
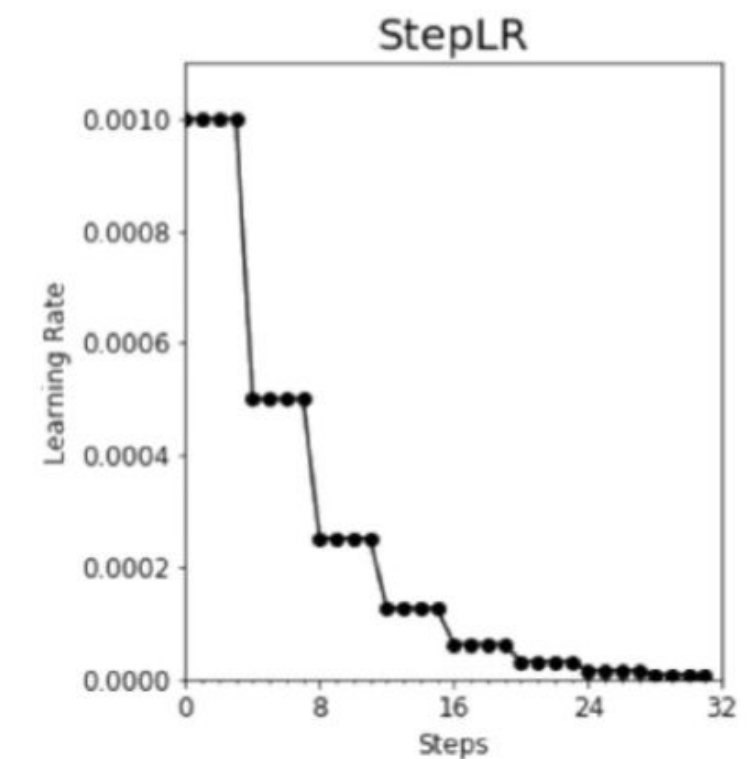
scheduler = opt.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

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

```
import torch.optim as opt

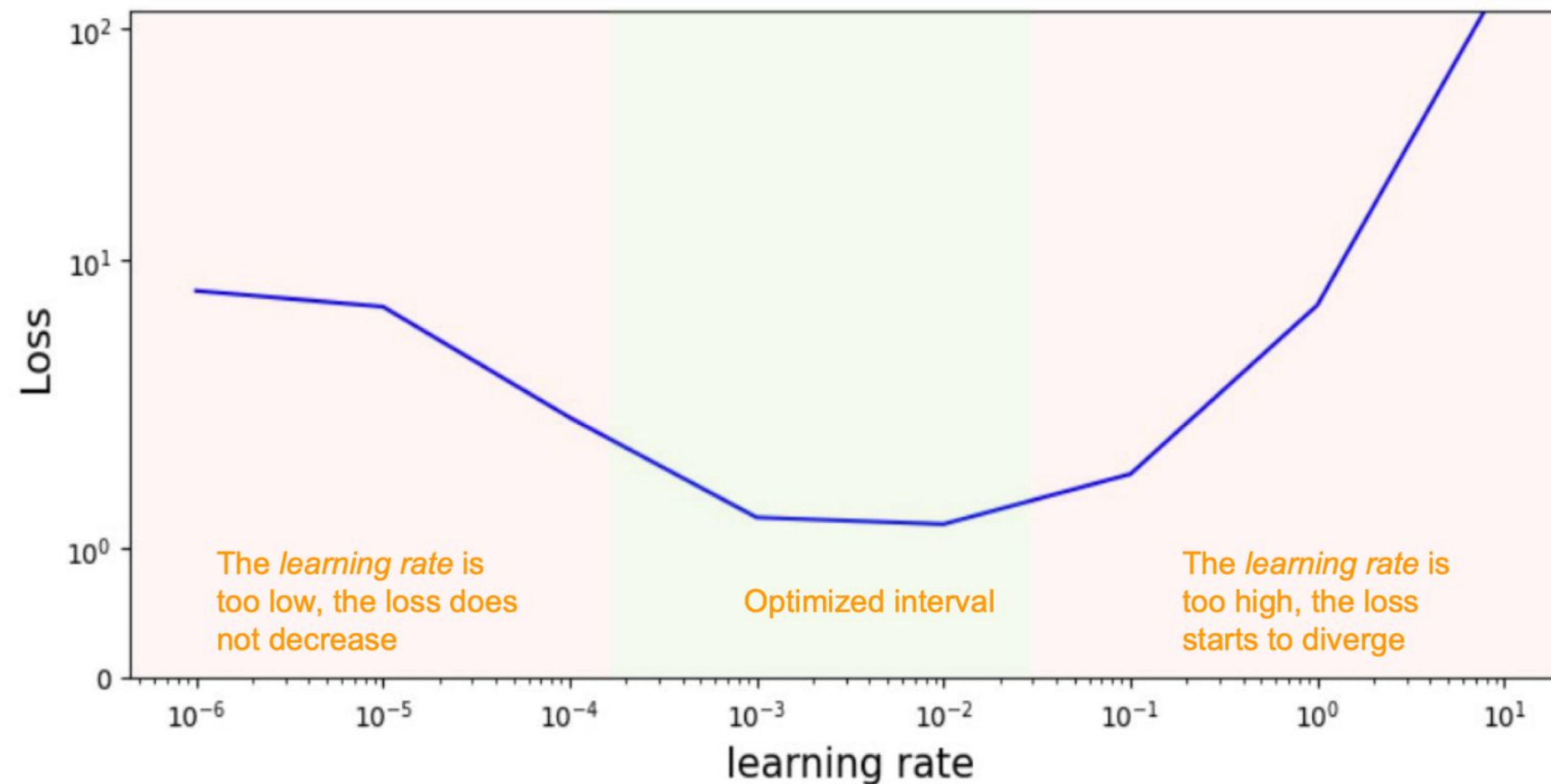
scheduler = opt.lr_scheduler.CyclicLR(optimizer, base_lr=0.01, max_lr=0.1)

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



Learning finder

Goal: Find the optimal learning rate values for his model, especially for the maximum value of a cyclic scheduler



Run your model over a few step/epochs by increasing its learning rate (with/without model reset)

- Start of decline in loss
→ **min learning rate**
- Start of loss variation
→ **max learning rate**

Key Aspects of Regularization:

- Reduces Overfitting – Prevents the model from learning noise and unnecessary details.
- Improves Generalization – Helps the model perform well on new, unseen data.
- Controls Model Complexity – Encourages simpler models that are more robust.

→ **Regularization is crucial for achieving a balance between bias and variance, ensuring the neural network learns meaningful patterns rather than just memorizing the training data**

Regularization: L1 & L2

$$\Theta_{t+1} = \Theta_t - \eta \nabla_{\Theta} [\mathcal{L}(\hat{y}_i, y_i) + \lambda R(\Theta_t)]$$

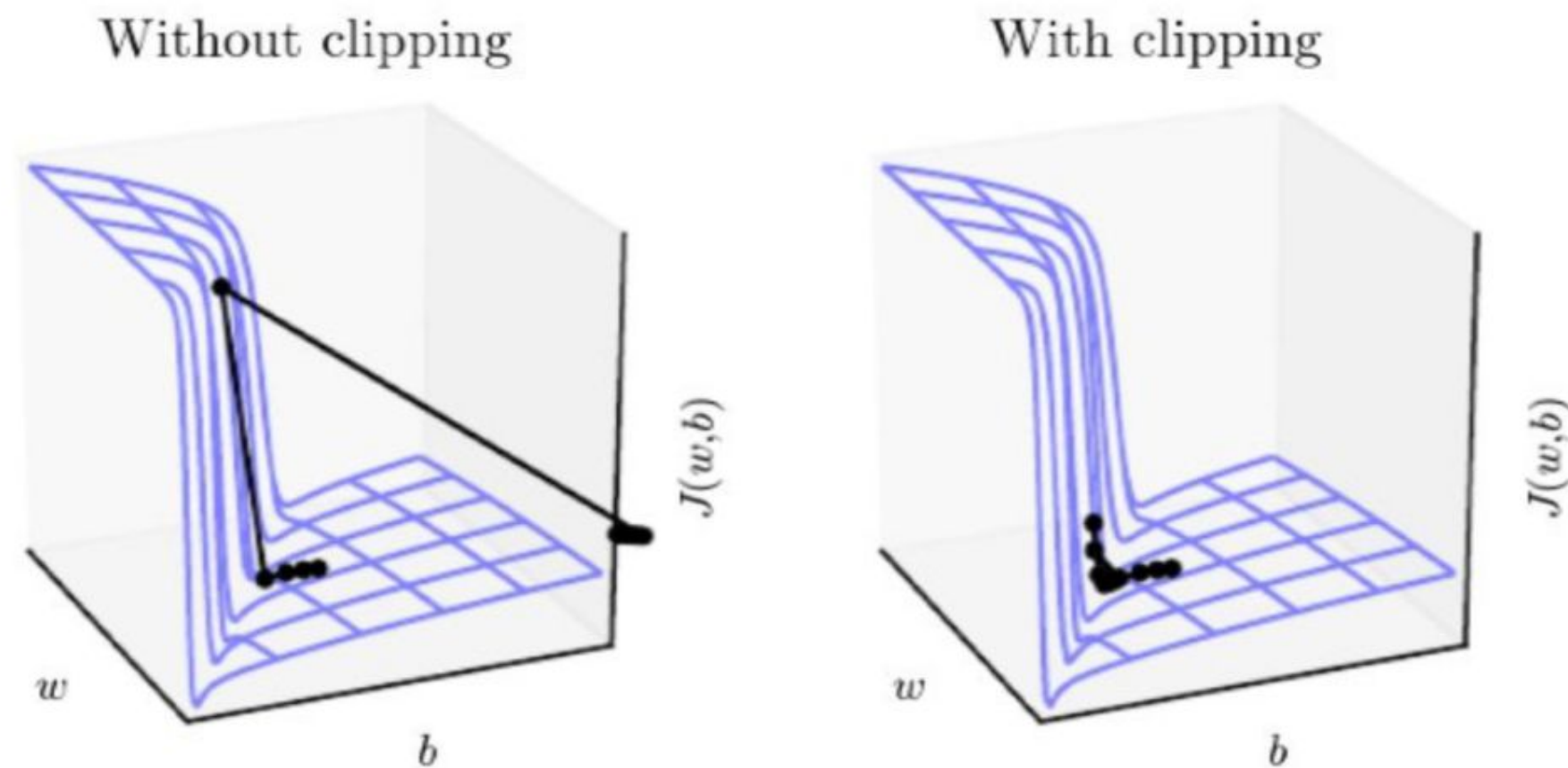
Updated weights = Weights before update - Learning rate * Gradient [Cost function (Prediction, Label) + Regularization rate * Regularization function (Weights before update)]

Weight update equation

L1 : LASSO	L2 : Ridge
$ \Theta $	Θ^2

- L1 Regularization
- L2 Regularization
- Max norm Regularization
- Regularization with the cost function
- Dropout

Regularization: Gradient clipping



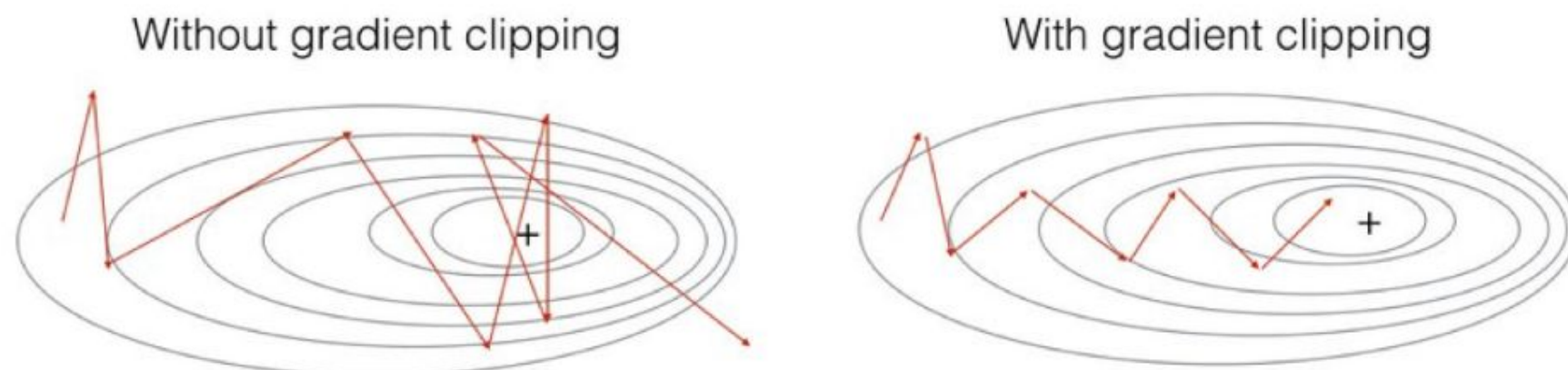
Problem : **Exploding gradients**

- Large increase in the norm of the gradient during training
- Produce unstable network, the gradient descent step impossible to execute

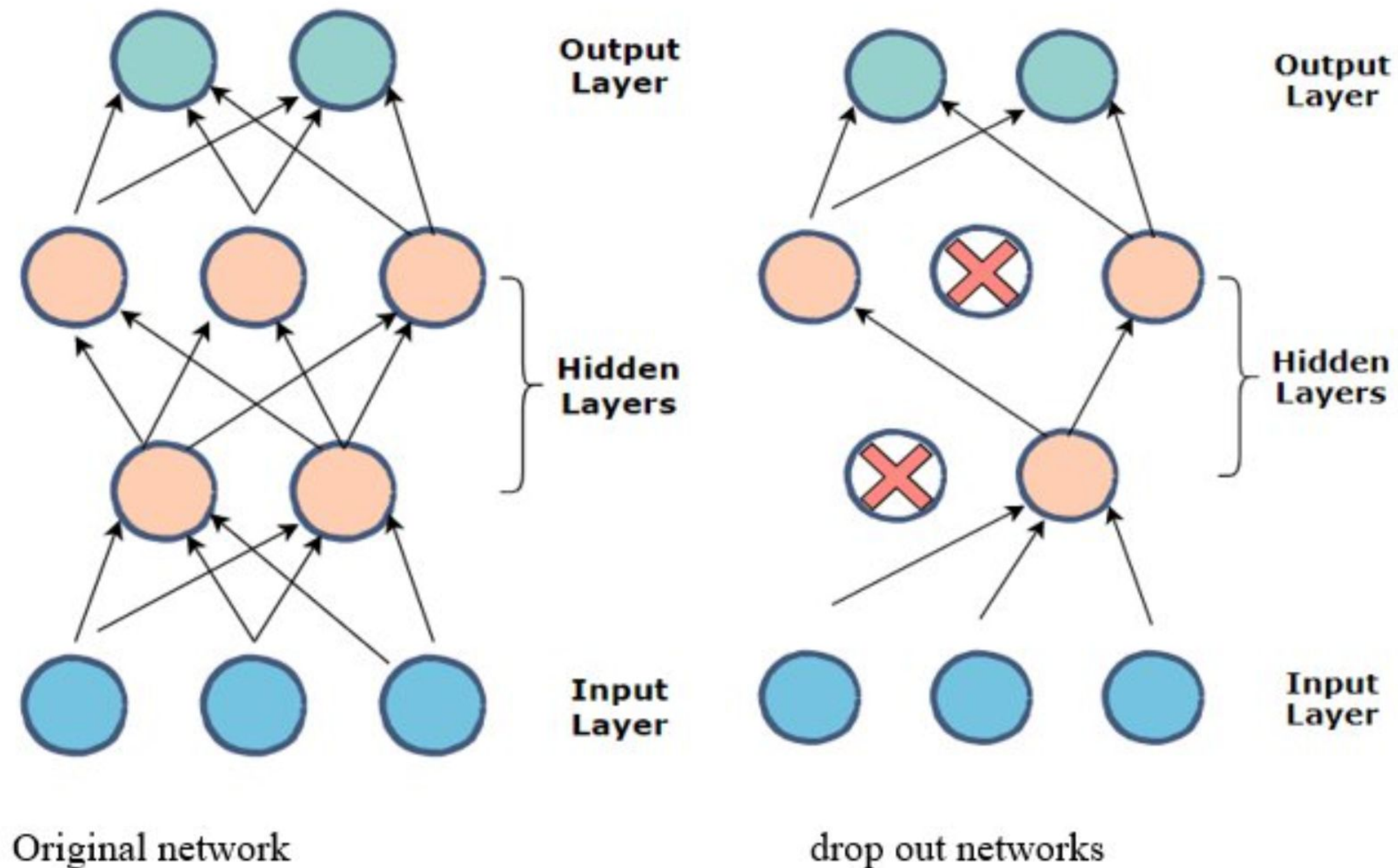
Gradient clipping solution :

The error derivative is changed or clipped to a threshold during backward propagation through the network. The clipped gradients is used to update the weights.

Other solutions : Regularization, weight-decay



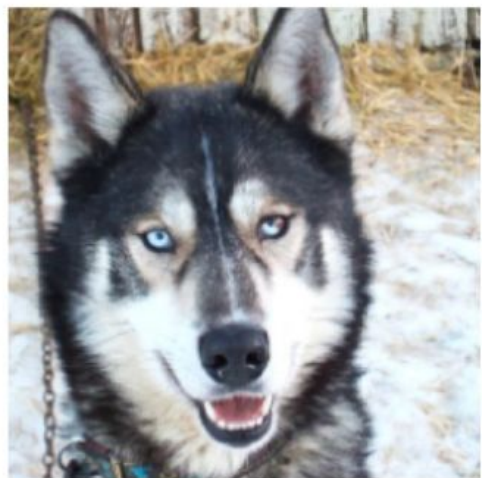
Regularization: Dropout



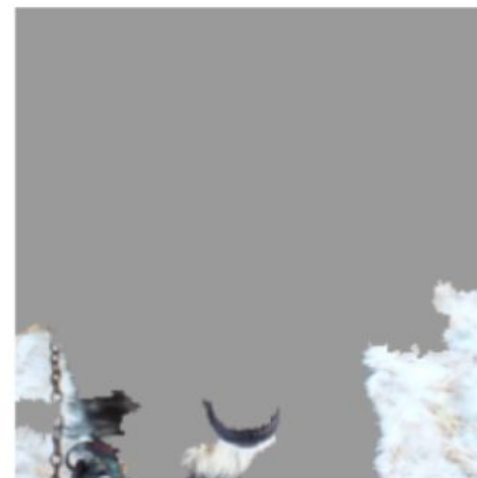
Regularization: Dropout



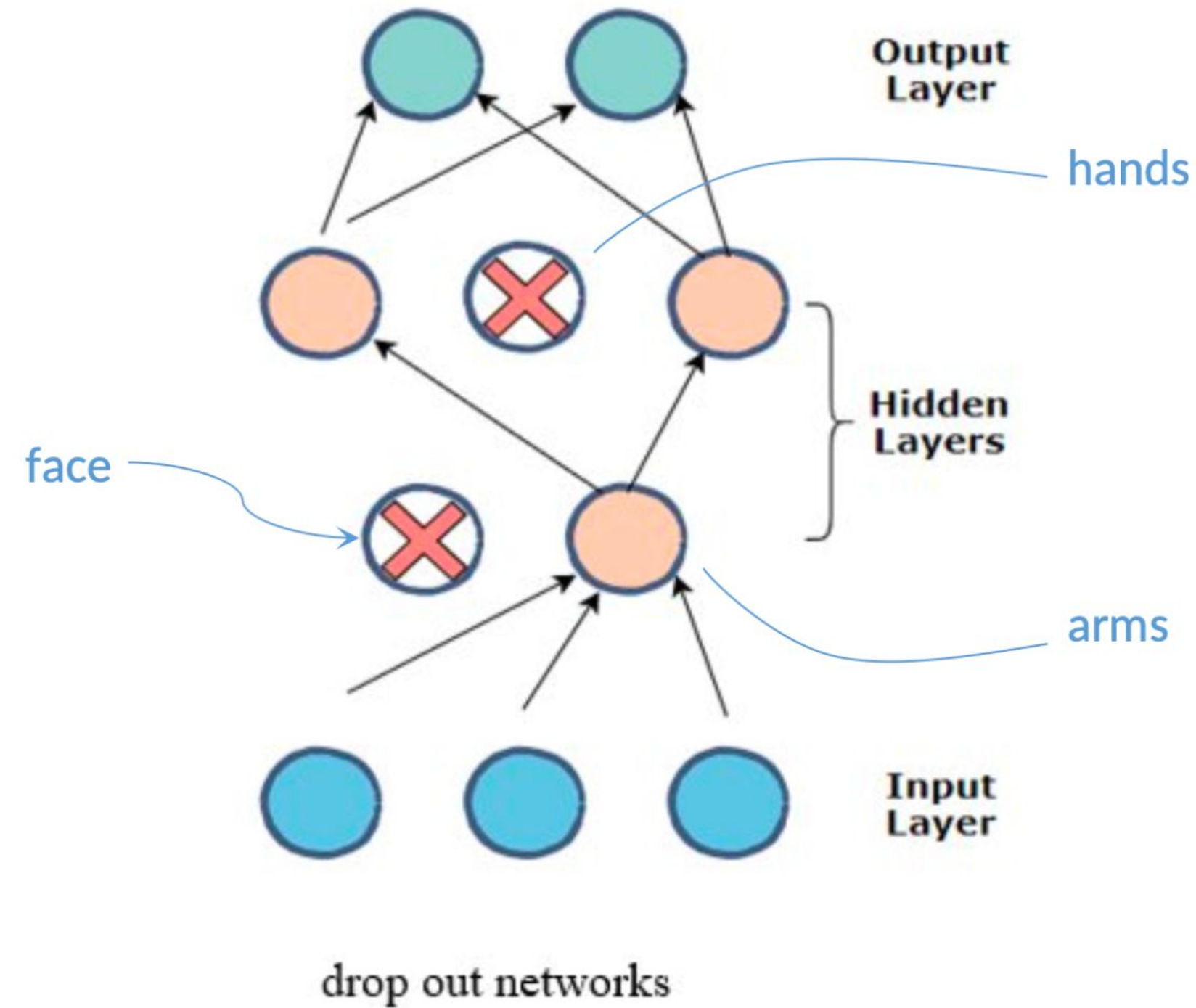
Human = face ? 



(a) Husky classified as wolf

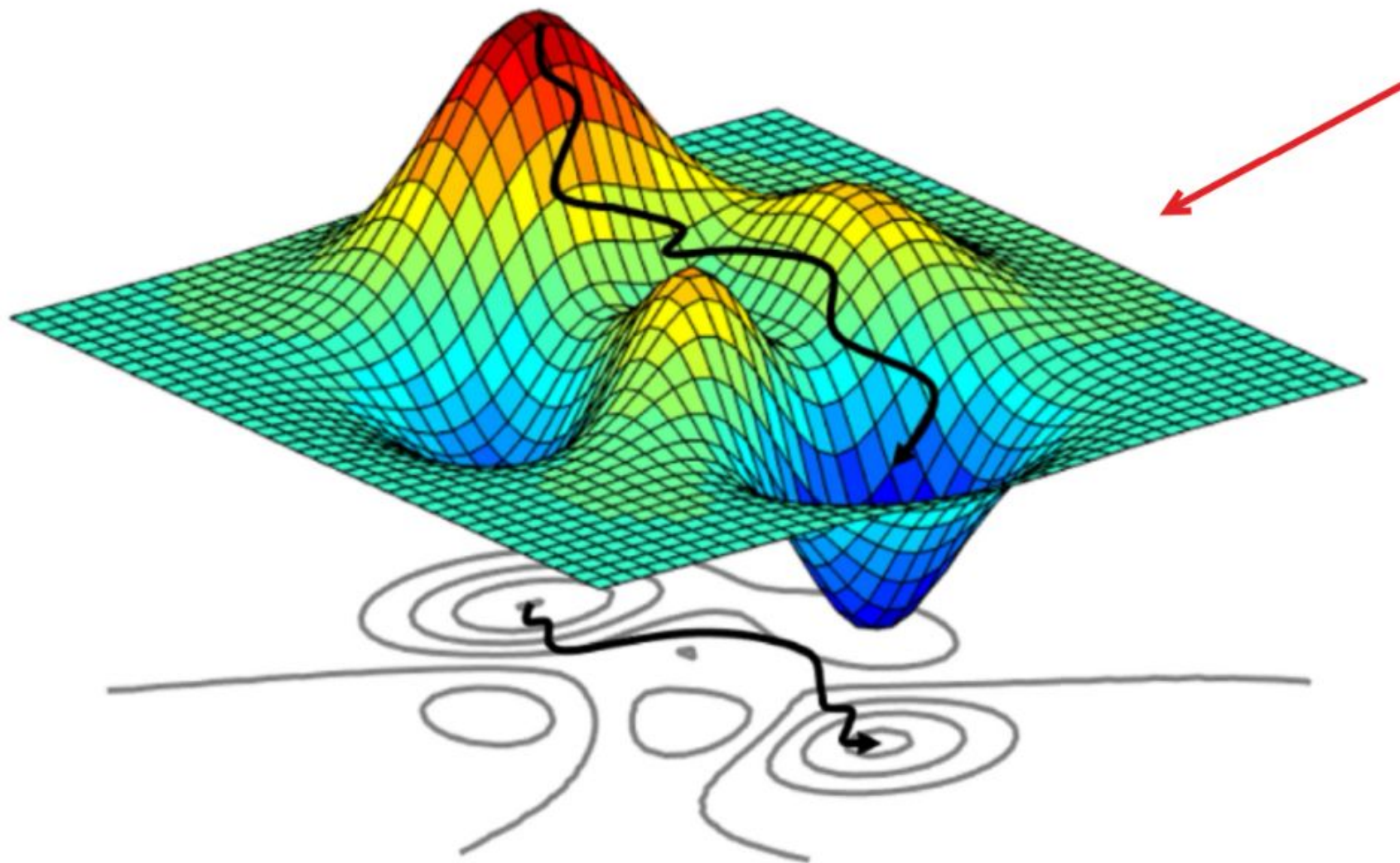


(b) Explanation



Optimizers

SGD



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

Batch size and learning rate adaptable to conflicting needs:

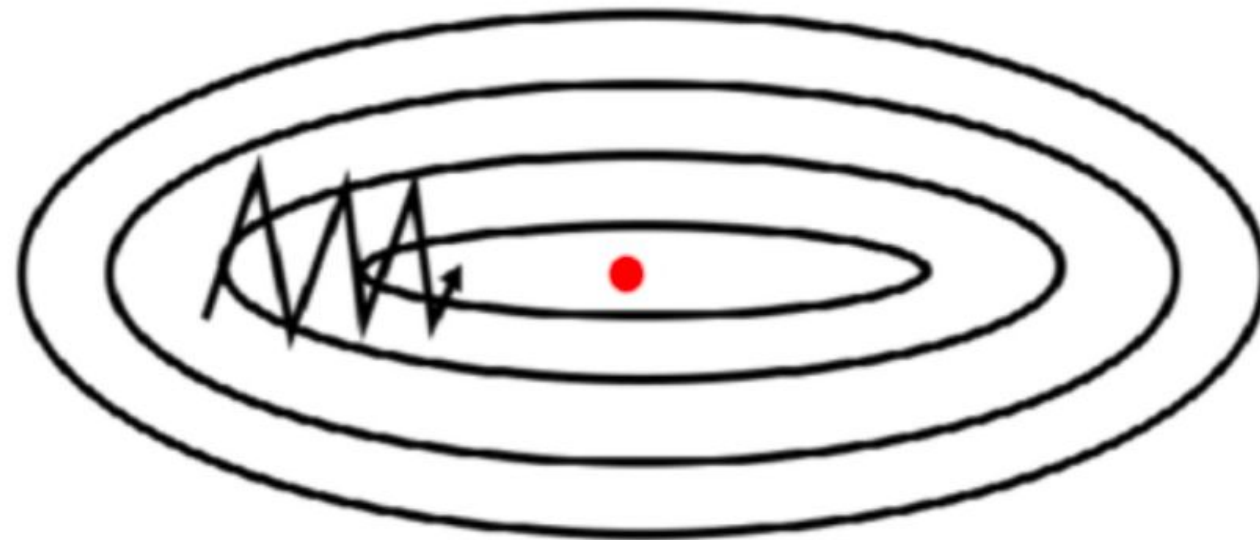
- Exploration to find the best local minimum
- Gradient descent acceleration

SGD = Stochastic Gradient Descent
Gradient calculation and weight update **at each batch**

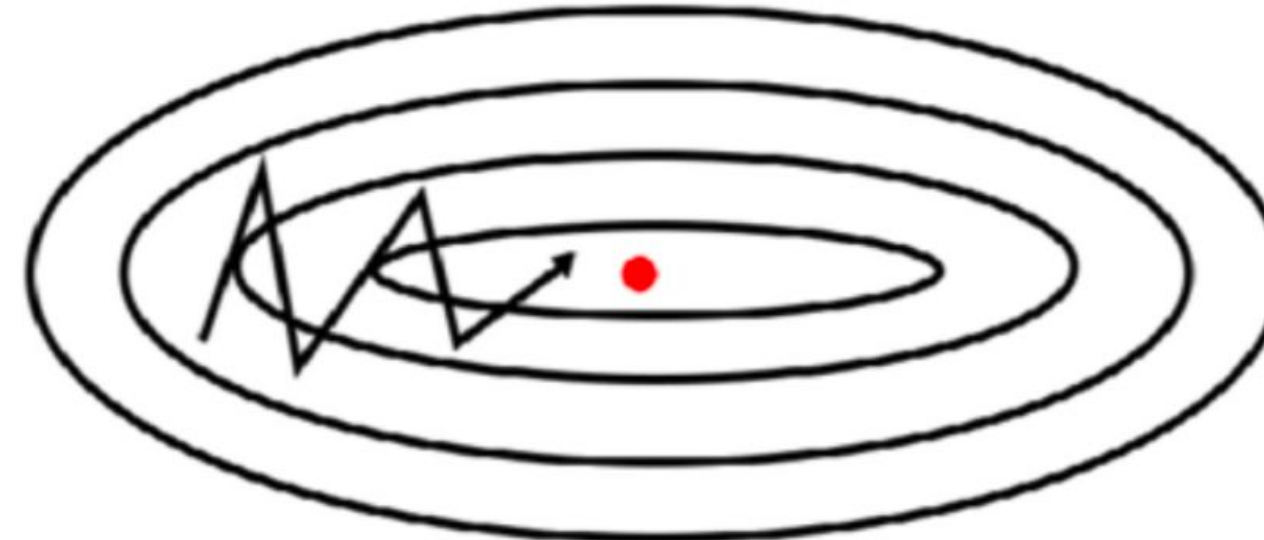
Optimizers

SGD with momentum

SGD without momentum



SGD with momentum



$$m_0 = 0$$

Momentum Factor

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

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

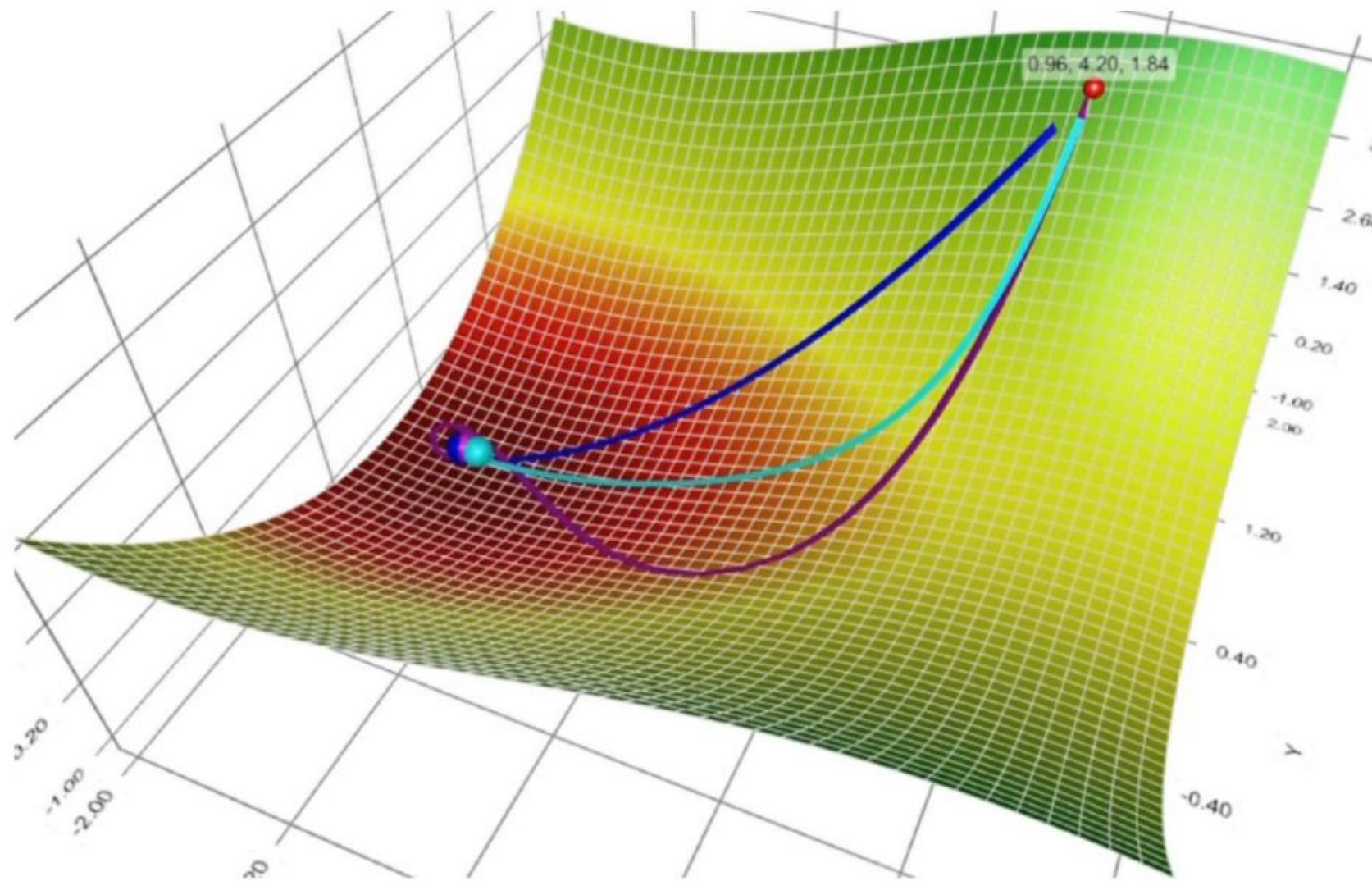
Goal: Consider previous gradients for faster gradient descent.

$$0.85 < \beta < 0.95$$

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

Adaptive Optimizers

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



Rather than driving 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**

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

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

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

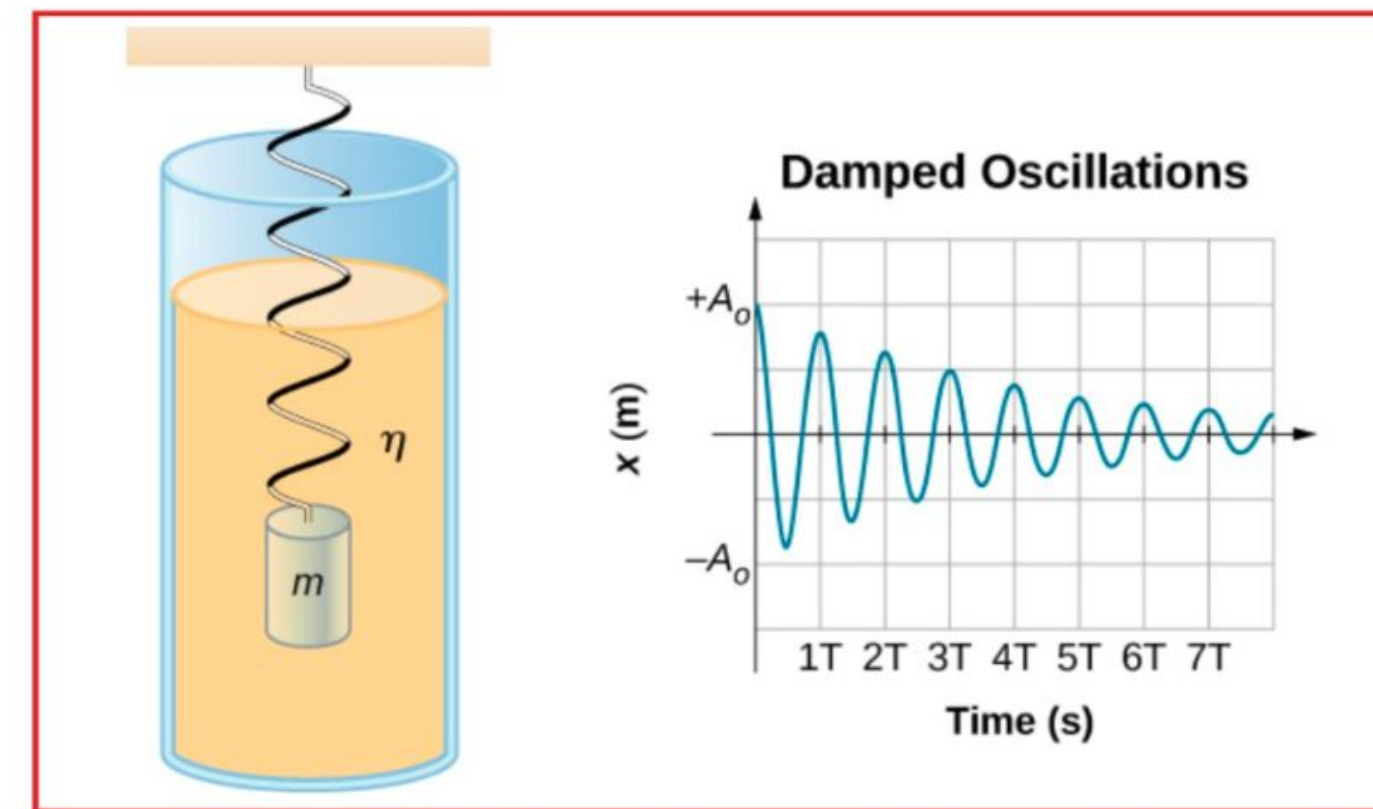
Correction of the biases of the first iterations

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

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

First moment: moving average

Second moment: sliding uncentered variance

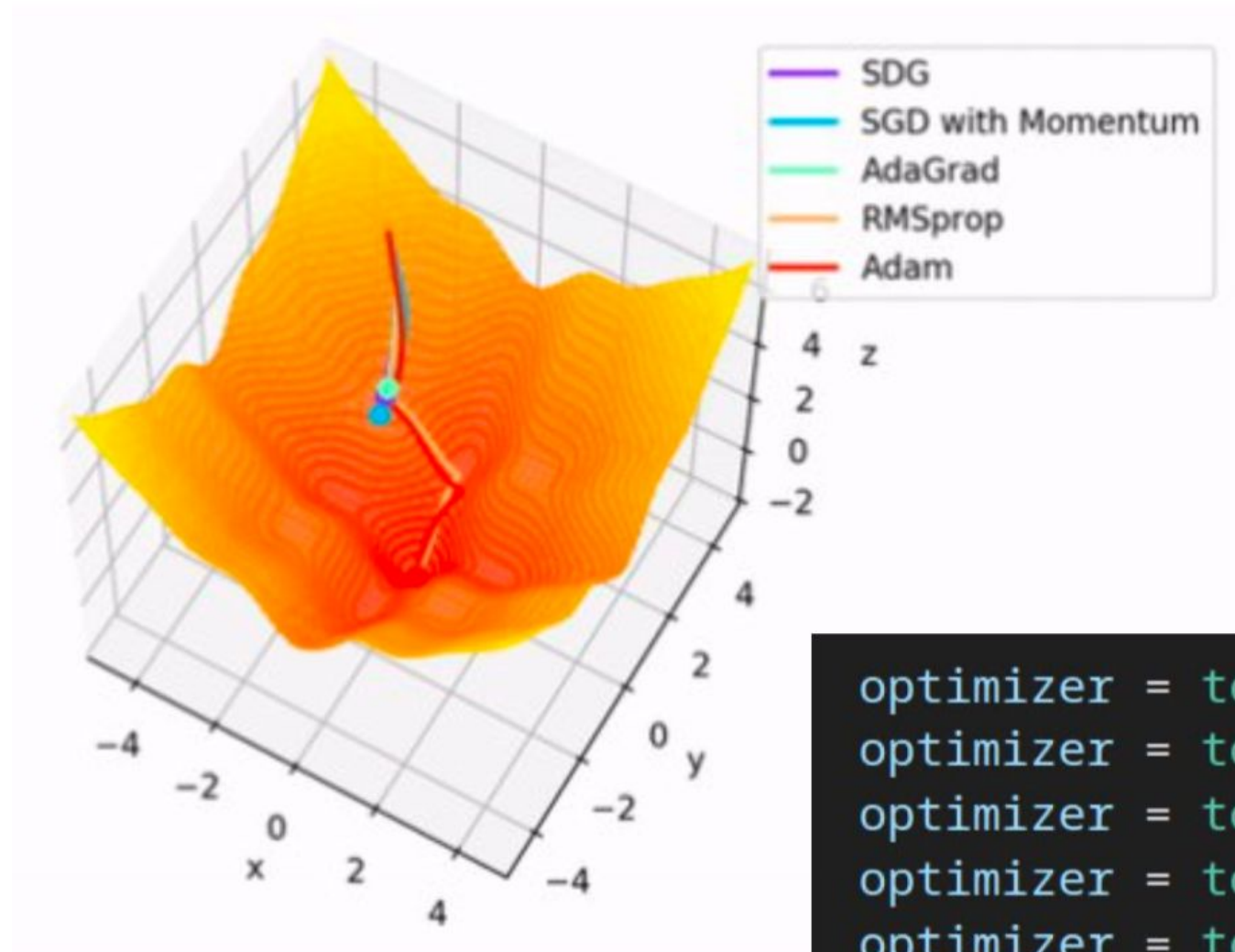


Parameters :

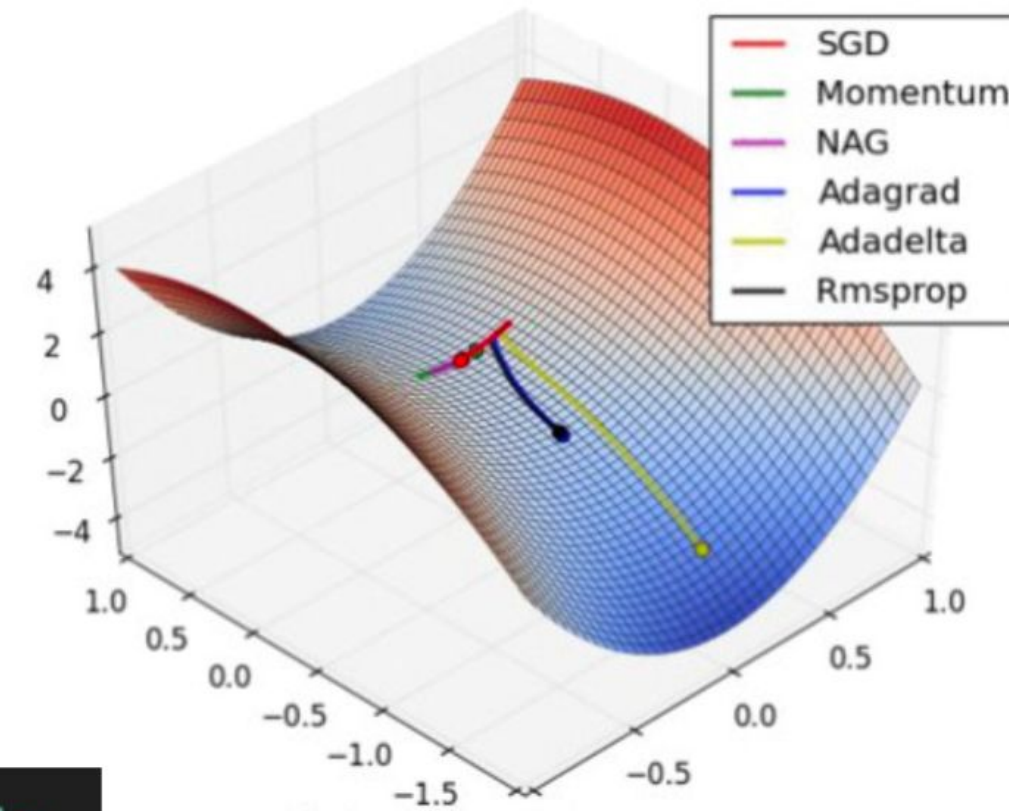
β_1 & β_2 = Regression rate ($\beta_1 = 0.9$ & $\beta_2 = 0.999$)

ϵ — Very small value to avoid division by zero

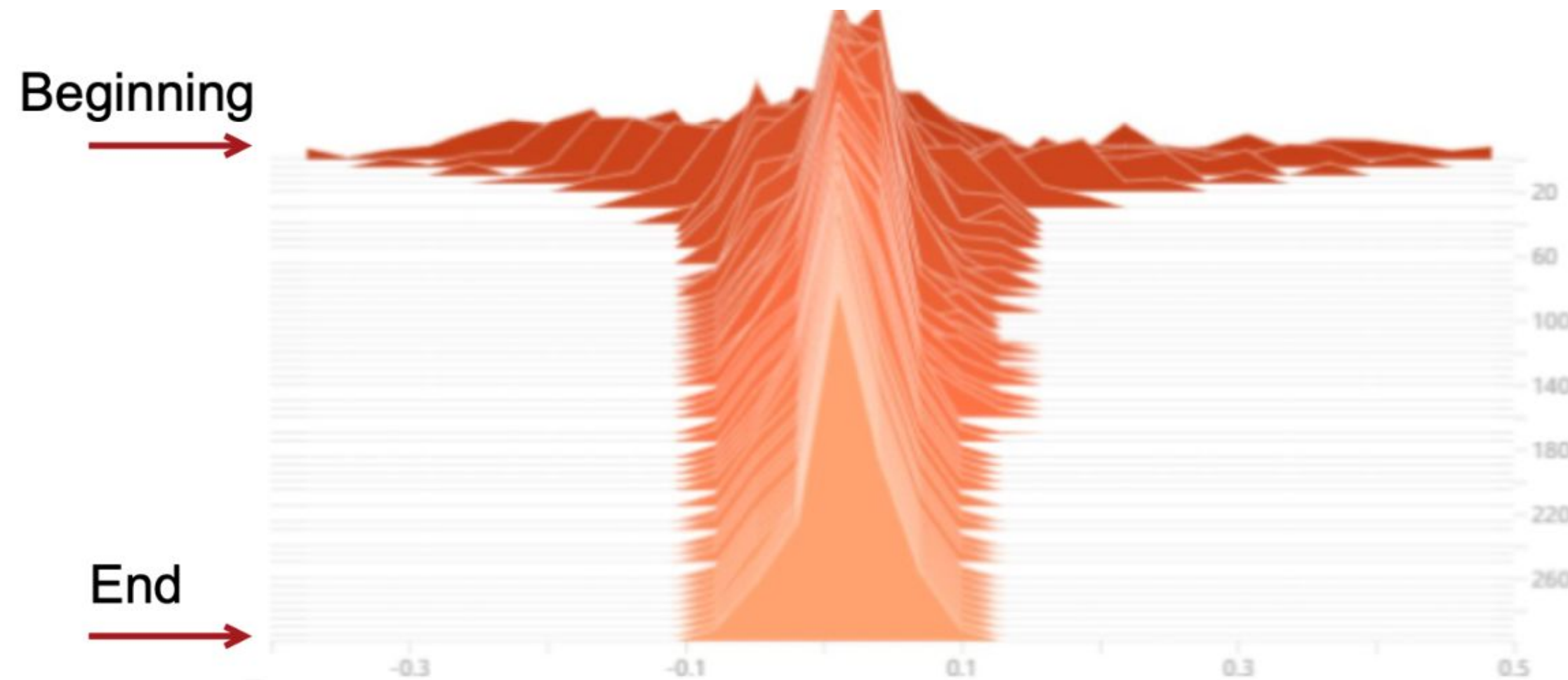
Optimizers



```
optimizer = torch.optim.Adadelta
optimizer = torch.optim.Adagrad
optimizer = torch.optim.Adam
optimizer = torch.optim.AdamW
optimizer = torch.optim.Adamax
optimizer = torch.optim.ASGD
optimizer = torch.optim.LBFGS
optimizer = torch.optim.NAdam
optimizer = torch.optim.RAdam
optimizer = torch.optim.RMSprop
optimizer = torch.optim.Rprop
optimizer = torch.optim.SGD
```



Weight decay



Weight distribution during
learning

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

Weight decay & decoupled weight decay

ADAM

```
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

ADAMW

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

Decoupled weight decay

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

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

SGD and SGDW are roughly equivalent in performance.

However AdamW is noticeably better than Adam!!

Weight decay & decoupled weight decay

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,...)
```

Batch size

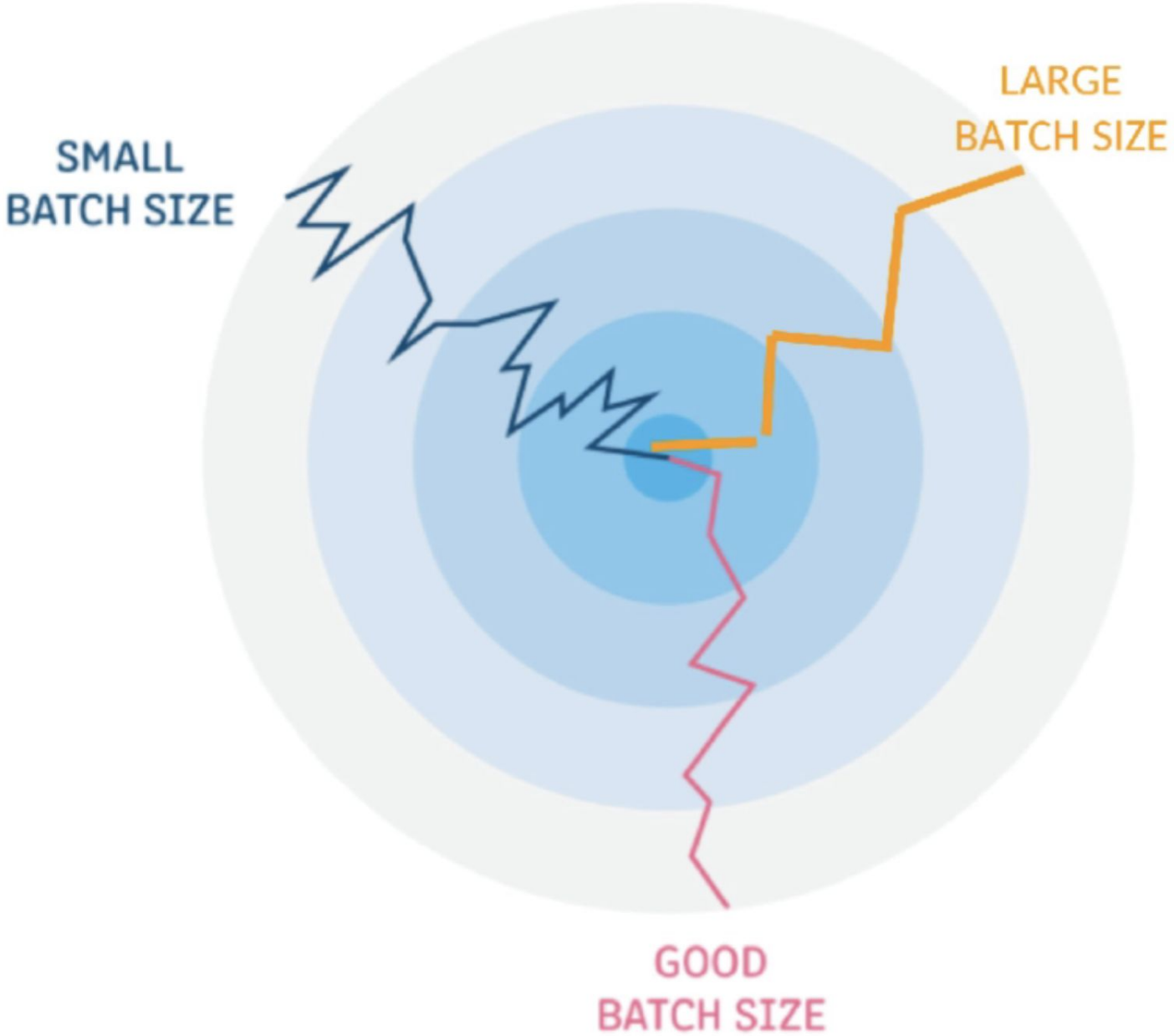
Batch Size	Pros	Cons
Small Batch (<32)	Better generalization, less memory usage	Noisy updates, slower convergence
Medium Batch (32-512)	Balance between stability and generalization	Requires careful tuning
Large Batch (>512)	Faster training, stable updates	Poor generalization, risk of overfitting

- **Small batches** introduce more noise in updates but help avoid local minima.
- **Large batches** stabilize learning but may generalize worse (overfit to training data).

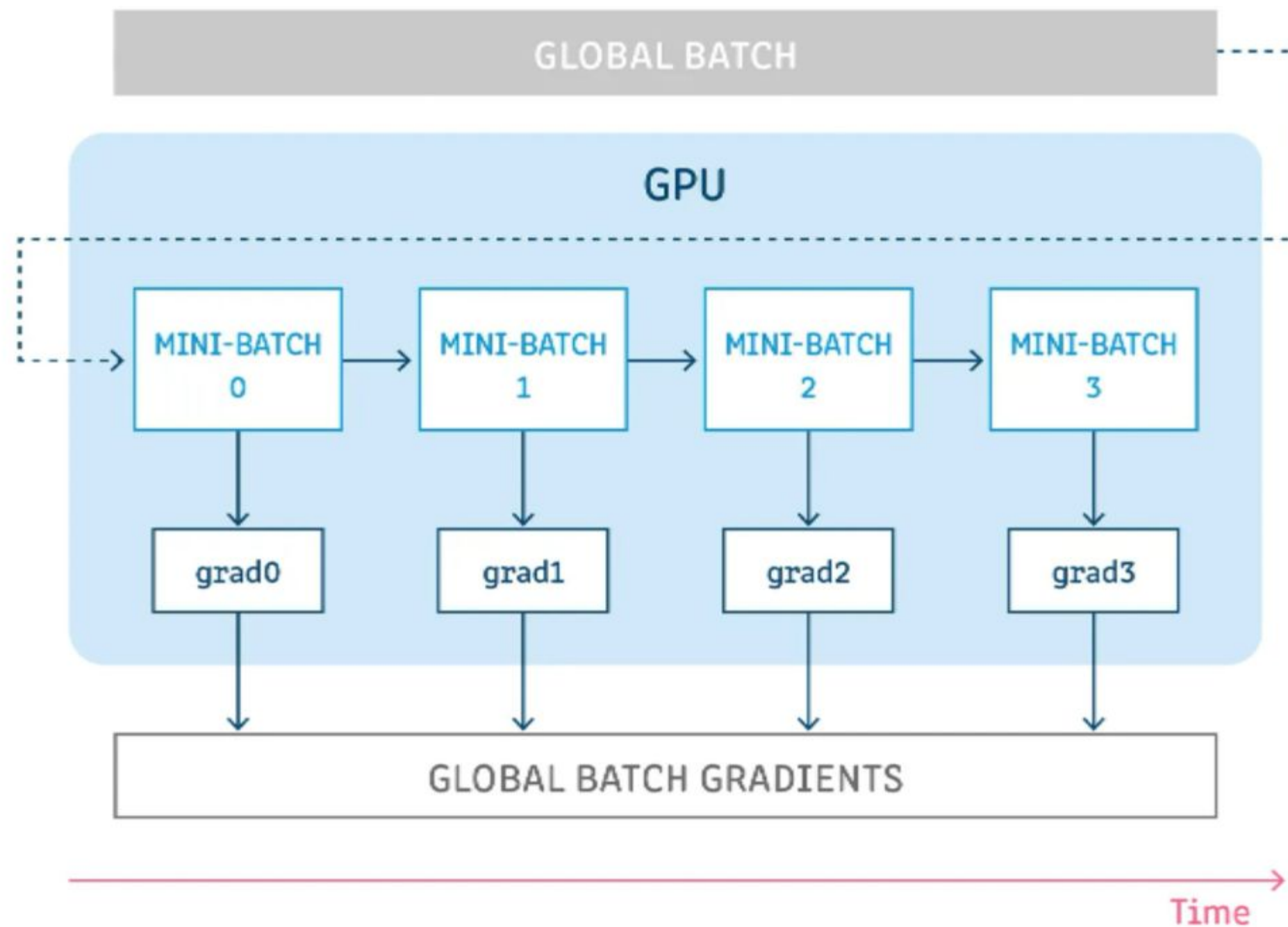
Good Practices:

- Start small and scale up if training is too slow.
- Check generalization – Don't just focus on training accuracy.
- Monitor memory usage – Large batches may crash the GPU.
- Tune learning rate when increasing batch size.

→ Try 32 or 64 as a starting point, and adjust based on memory limits, training speed, and validation performance.



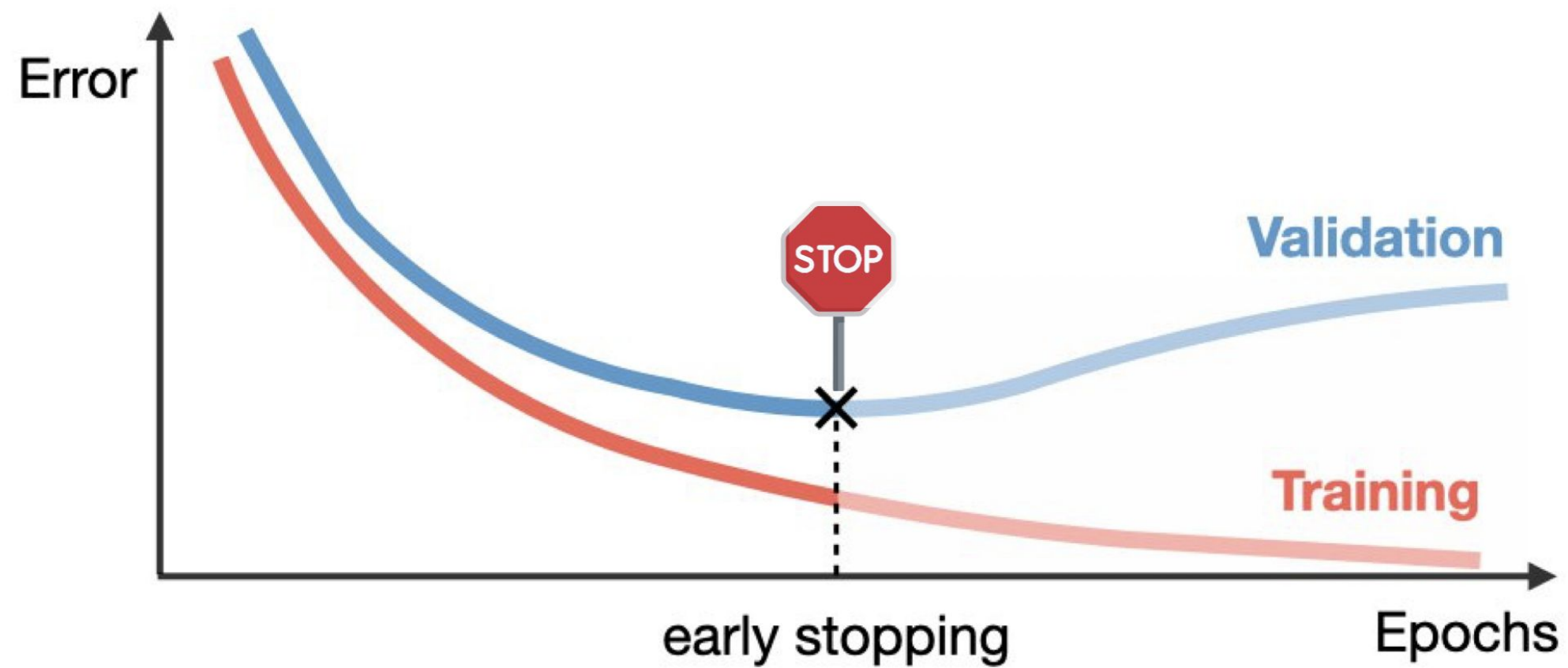
Gradient accumulation



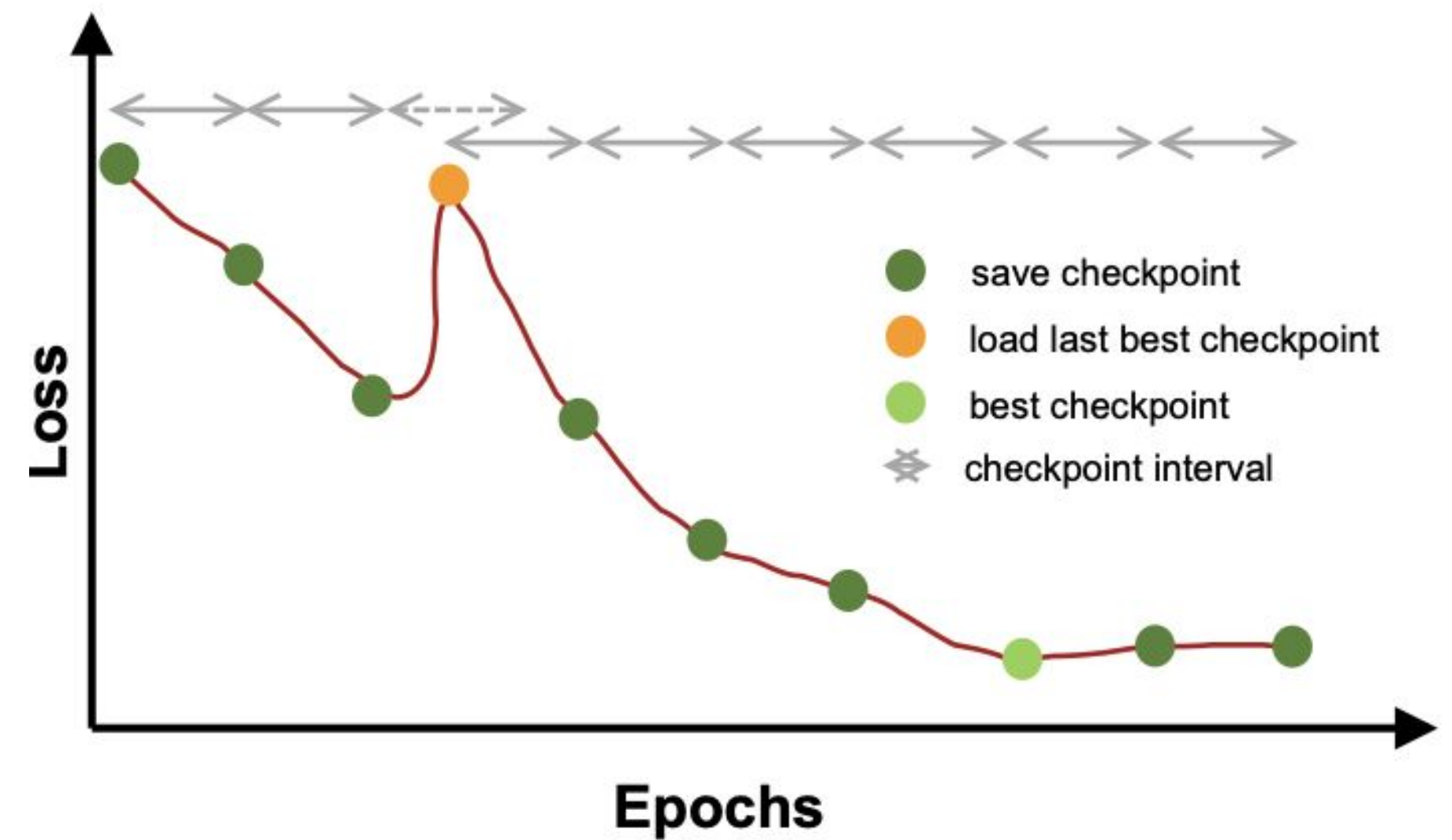
Use Case :

- Not enough memory to have a normal Batch Size
- Reduces the number of communication between host and device (accelerator)

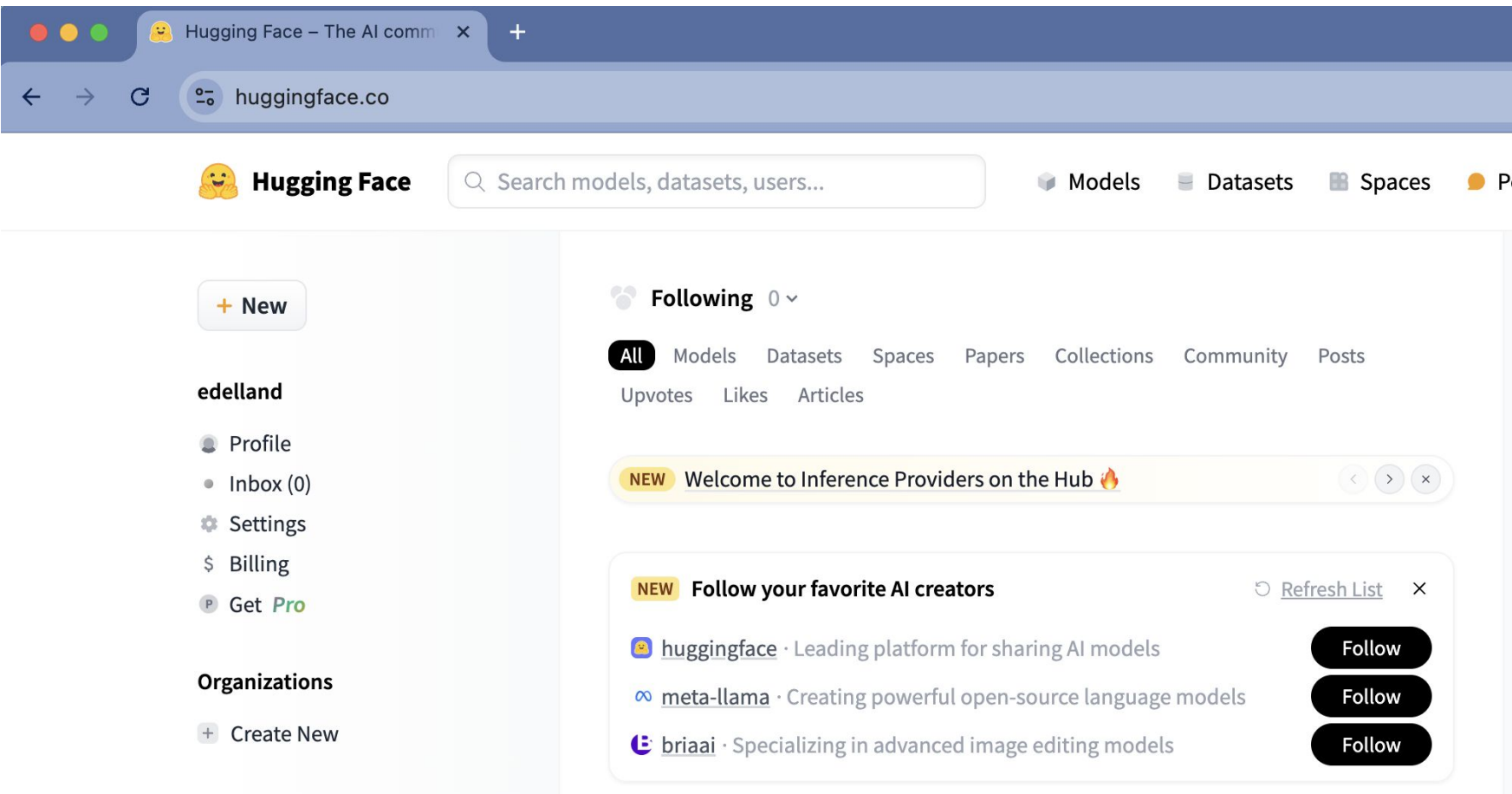
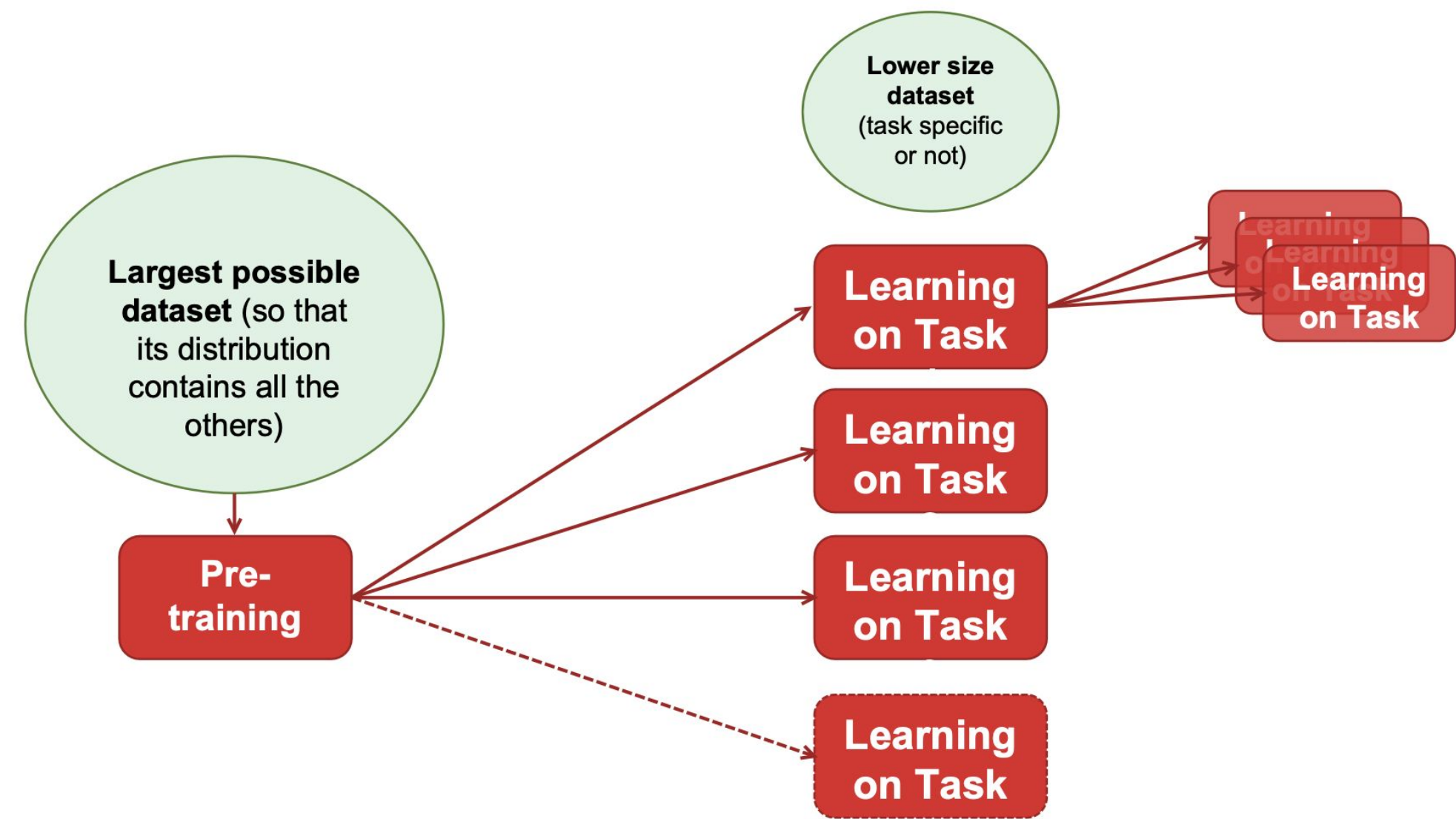
Training tricks



Checkpointing

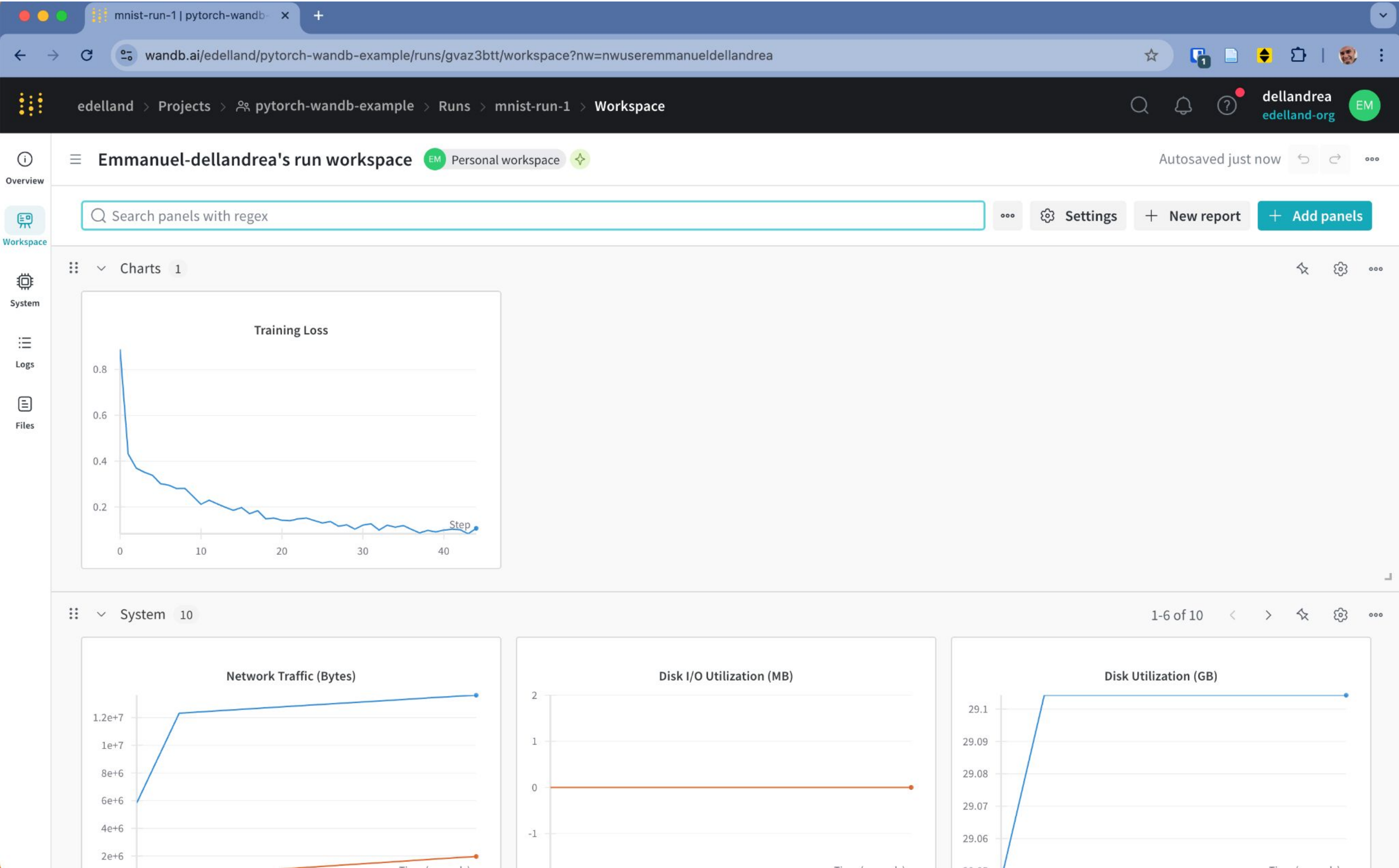


Transfer Learning



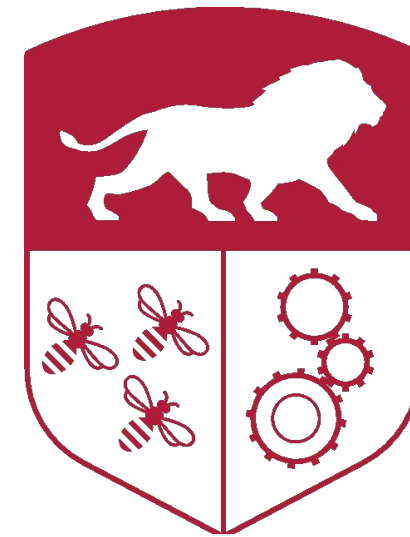
Hugging Face: Models & datasets available

Experiments tracking: Weights & Biases



Some useful references

- Fidle - Deep Learning Introduction (<https://www.fidle.cnrs.fr/w3/>)
- CS231n: Convolutional Neural Networks for Visual Recognition (<http://cs231n.stanford.edu>)
- Neural Networks and Deep Learning (<http://neuralnetworksanddeeplearning.com>)
- Deep Learning (<http://www.deeplearningbook.org>)
- PyTorch (<http://pytorch.org>)
- Weights & Biases (<https://wandb.ai/site/>)
- Hugging Face (<https://huggingface.co/>)



**CENTRALE
LYON**

36, avenue Guy de Collongue 69130 Écully
www.ec-lyon.fr | [@centralelyon](https://twitter.com/centralelyon)