# Optimizers in deep learning - Practical study

CPE 727 - Deep Learning

**Ana Clara Loureiro Cruz** **Bruno Coelho Martins** **Emre Aslan** **Felipe Barreto Andrade** (anaclaralcruz@poli.ufrj.br
bruno.martins@smt.ufrj.br
emre@gta.ufrj.br

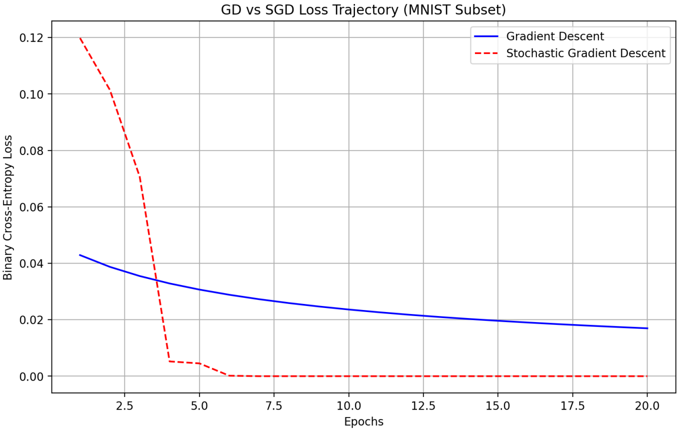
felipebarretoandrade@poli.ufrj.br )

November 10, 2025

# Table of Contents

# Gradient Descent (GD) Optimization

## 1 Gradient Descent




GitHub Link

# Gradient Descent

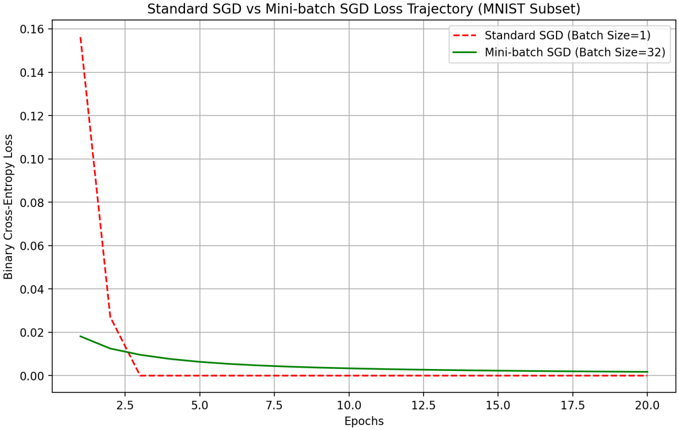


Table: Comparison of Hyperparameters and Key Differences between Gradient Descent (GD) and Stochastic Gradient Descent (SGD)

| Aspect | Gradient Descent (GD) | Stochastic Gradient Descent (SGD) |
|---|---|---|
| **Learning Rate** | 0.1 (fixed) | 0.1 (fixed) |
| **Number of Epochs** | 20 | 20 |
| **Batch Size** | 500 (full dataset) | 1 (single sample) |
| **Gradient Computation** | Full dataset (500 samples) | Single sample per iteration |
| **Updates per Epoch** | 1 update | 500 updates |
| **Computational Cost per Update** | High (processes all samples) | Low (processes one sample) |
| **Convergence Behavior** | Smoother, more stable | Noisier, faster per iteration |
| **Implementation Detail** | Uses entire dataset for gradient | Randomly samples one data point |

# Standard Sgd vs Mini-batch Sgd





# Standard Sgd vs Mini-batch Sgd



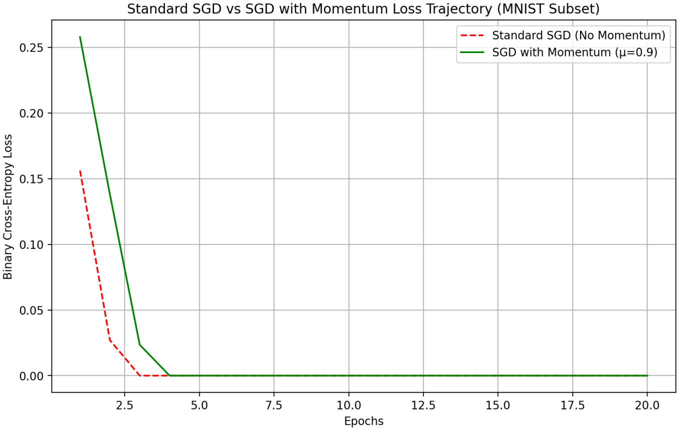
GitHub Link

| Aspect | Standard SGD | Mini-batch SGD |
|---|---|---|
| **Learning Rate** | 0.1 (fixed) | 0.1 (fixed) |
| **Number of Epochs** | 20 | 20 |
| **Batch Size** | 1 (single sample) | 32 (multiple samples) |
| **Gradient Computation** | Single sample | 32 samples per iteration |
| **Updates per Epoch** | 500 updates | 16 updates (500/32) |
| **Computational Cost** | Low (one sample per update) | Moderate (32 samples per update) |
| **Convergence Behavior** | Noisier, high variance | Smoother, reduced variance |
| **Implementation Detail** | Randomly samples one data point | Randomly samples 32 data points |

Table: Comparison of Hyperparameters and Key Differences between Standard SGD and Mini-batch SGD

# SGD with Momentum

## 1 Gradient Descent




GitHub Link

# SGD with Momentum



| Aspect | Standard SGD | SGD with Momentum |
|---|---|---|
| **Learning Rate** | 0.1 (fixed) | 0.1 (fixed) |
| **Number of Epochs** | 20 | 20 |
| **Batch Size** | 1 (single sample) | 1 (single sample) |
| **Momentum Coefficient** | None | 0.9 |
| **Gradient Update** | Direct gradient: $\theta \leftarrow \theta - \eta \nabla J$ | Velocity-based: $v \leftarrow \mu v - \eta \nabla J$, $\theta \leftarrow \theta + v$ |
| **Updates per Epoch** | 500 updates | 500 updates |
| **Convergence Behavior** | Noisier, high variance | Smoother, faster due to momentum |
| **Implementation Detail** | Updates with raw gradient | Uses velocity to accelerate gradients |

Table: Comparison of Hyperparameters and Key Differences between Standard SGD and SGD with Momentum

# Table of Contents

# Experiment Setup: Rosenbrock Function



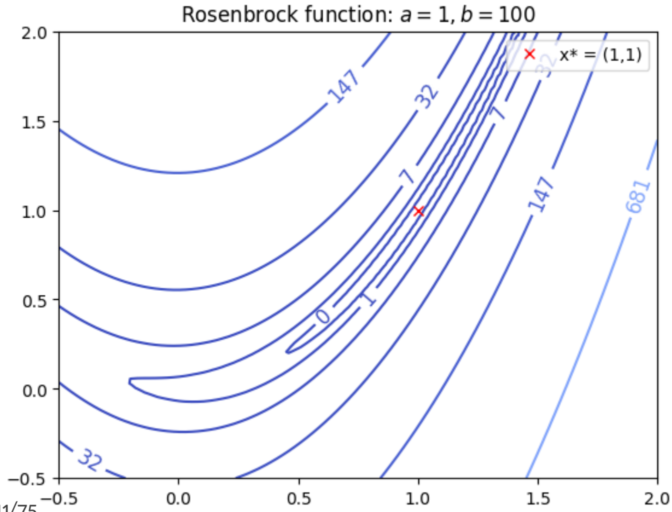
- All experiments are conducted on the **Rosenbrock function**, a classic benchmark for testing optimizers.

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

- This function forms a narrow, curved valley that makes optimization difficult:
  - Gradients are small along the valley but steep across it.
  - Optimizers must balance stability and adaptivity to reach the minimum efficiently.
- It is ideal to visualize how each algorithm handles curvature, adaptivity, and noise.

# Effect of Hyperparameters in Adam





# Adam hyperparameters: effect of `beta2`

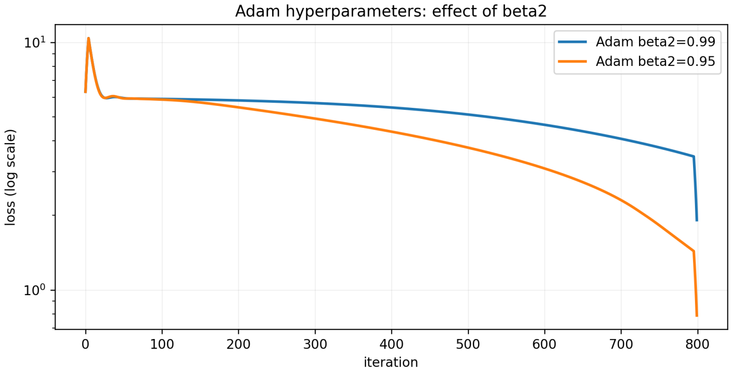
```
sweep_specs = [
    ("Adam beta2=0.99", {"betas": (0.9, 0.99)}),
    ("Adam beta2=0.95", {"betas": (0.9, 0.95)}),
]
for label, kw in sweep_specs:
    _, losses, _ = run_optimizer(torch.optim.Adam, steps=800, lr=3e-3, **kw)
    plt.semilogy(losses, label=label)
plt.title("Adam hyperparameters: effect of beta2")
```

# Effect of Hyperparameters in Adam



- The parameter $\beta_2$ controls the smoothing of the variance (second moment):
  - Higher values ($0.99$) produce smoother but slower updates.
  - Lower values ($0.95$) react faster to gradient changes.
- Small changes in $\beta_2$ can strongly influence training speed and stability.

# Effect of Hyperparameters in Adam



# Adam vs AdamW (decoupled weight decay)



```
_, loss_adam, _ = run_optimizer(torch.optim.Adam,  lr=3e-3, weight_decay=1e-2)
_, loss_adamw, _ = run_optimizer(torch.optim.AdamW, lr=3e-3, weight_decay=1e-2)

plt.semilogy(loss_adam, label="Adam (wd=0.01)")
plt.semilogy(loss_adamw, label="AdamW (wd=0.01)")
plt.title("Coupled L2 (Adam) vs Decoupled (AdamW)")
```

# Conceptual Difference: Adam vs AdamW

- Original **Adam** couples L2 regularization with adaptive updates:

$$\theta_{t+1} = \theta_t - \eta\left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} + \lambda\theta_t\right)$$

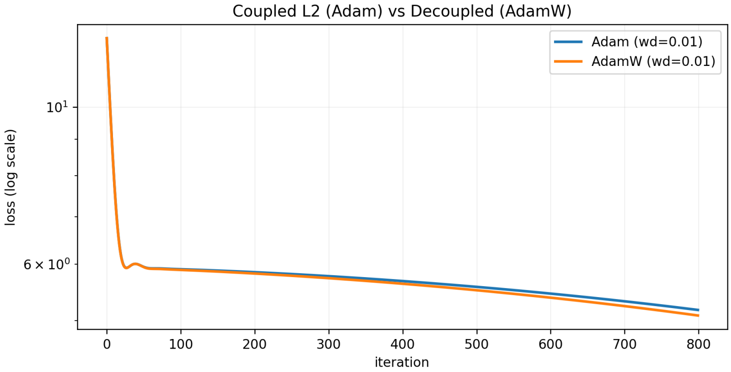
- **AdamW** decouples weight decay:

$$\theta_{t+1} = (1 - \eta\lambda)\theta_t - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

- This avoids the interference between regularization and learning rates.
- Effect: cleaner weight decay, better generalization (especially in Transformers).

# Adam vs AdamW: Convergence Comparison





# Loss vs iterations for Adam-family optimizers



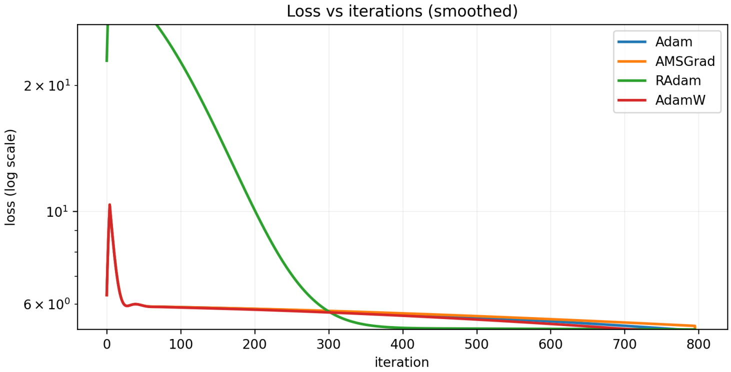
```
variants = [
    ("Adam",    torch.optim.Adam,  {"amsgrad": False}),
    ("AMSGrad", torch.optim.Adam,  {"amsgrad": True}),
    ("RAdam",   torch.optim.RAdam, {}),
    ("AdamW",   torch.optim.AdamW, {}),
]
for name, ctor, kw in variants:
    _, losses, _ = run_optimizer(ctor, lr=3e-3, steps=800, **kw)
    plt.semilogy(losses, label=name)
plt.title("Loss vs iterations (log scale)")
```

# Comparison Across Adam-family Variants

- The plot shows training loss per iteration (log scale).
- **Adam**: fast early convergence, but can oscillate.
- **AMSGrad**: uses $\max(v_t)$ to ensure monotonic variance estimates.
- **RAdam**: introduces variance rectification to fix the warmup problem.
- **AdamW**: stable convergence and cleaner weight decay.

# Comparison Across Adam-family Variants





# Step size evolution

```
for name in results:
    y = results[name]["dtheta"]
    plt.plot(y, label=name)
plt.yscale("symlog", linthresh=1e-5)
plt.title("Step size over time (symlog)")
plt.legend()
```

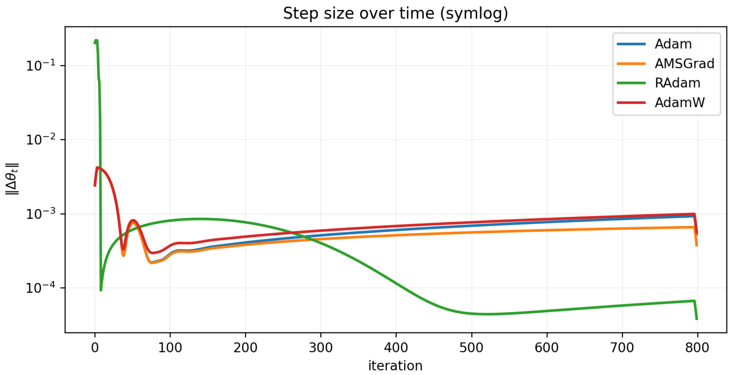
# Evolution of Step Size



- $\|\Delta\theta_t\|$ measures how much parameters change at each iteration.
- Large spikes at the beginning reflect **adaptive warmup**.
- **RAdam** shows the highest variance initially.
- **AdamW** and **AMSGrad** stabilize faster and keep smaller steps.
- The **symlog** scale allows both small and large step magnitudes to be visible.

# Evolution of Step Size

## 2 Adam and Its Variants



# Parameter-space trajectories



```
X, Y, Z = rosenbrock_grid()
for name in results:
    T = results[name]["traj"]
    plt.plot(T[:,0], T[:,1], label=name)
plt.contour(X, Y, Z, levels=np.logspace(-1,3,24))
plt.title("Rosenbrock: parameter-space trajectories")
```

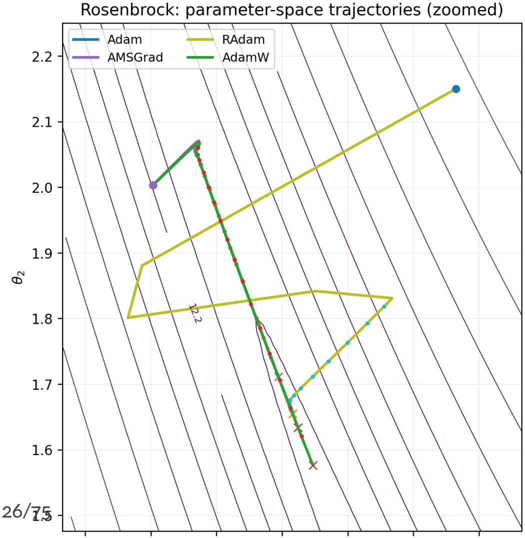
# Parameter-space Trajectories



- The Rosenbrock function forms a curved valley leading to the global minimum.
- **Adam** (blue): follows a relatively smooth trajectory, with minor lateral oscillations.
- **AMSGrad** (purple): exhibits shorter, more consistent steps by using the maximum historical variance to stabilize updates.
- **RAdam** (yellow): shows wider initial swings since the rectification warmup is still adapting.
- **AdamW** (green): maintains a direct and stable path, benefiting from its decoupled weight decay.

# Parameter-space Trajectories

## 2 Adam and Its Variants



# Table of Contents

# Scheduler class

3 Learning-rate (LR) Schedulers

```
class Scheduler:
    """
    Scheduler: create PyTorch LR schedulers from a name + params.

    Supported schedulers:
    LambdaLR, MultiplicativeLR, StepLR, MultiStepLR, ConstantLR, LinearLR,
    ExponentialLR, PolynomialLR (builtin if available, else Lambda fallback),
    CosineAnnealingLR, ChainedScheduler, SequentialLR, ReduceLROnPlateau,
    CyclicLR, OneCycleLR.

    The class returns the scheduler object already constructed and ready to be
    stepped in the training loop.
    """
```

# Experiment Overview



This experiment trains a simple MLP model on the Breast Cancer dataset with preprocessing and supports flexible learning rate schedulers (similar to Breast Cancer MLP Experiment).

# Experiment - Default Parameters



- epochs: 100
- batch_size: 32
- learning_rate: 0.05
- hidden_size: 64
- feature_strategy: onehot
- target_strategy: binary
- handle_missing: drop
- device: cpu
- scheduler_name: CosineAnnealingLR
- scheduler_params: {} (JSON string)

# Experiment - Output



- Train loss per epoch plot: `train_loss_per_epoch_<SCHEDULER_NAME>.png`
- Learning rate per epoch plot: `lr_per_epoch_<SCHEDULER_NAME>.png`

# Running the Experiment

Run the experiment using Python module syntax and the CLI script:

```
python -m src.experiments.LRSchedulerExperiment.lr_scheduler_experiment.cli \
    --scheduler <SCHEDULER_NAME> \
    --scheduler-params '<JSON_PARAMS>'
```

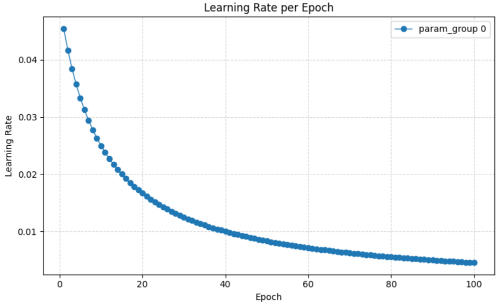
# LambdaLR — Flexible functional schedules

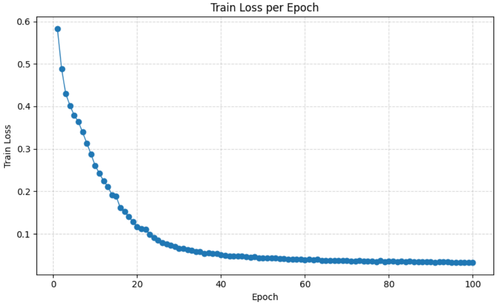


- **lr_lambda (callable)** — function(epoch) that returns a multiplicative factor.
  - Default: `lambda epoch:  1/(1+0.1*epoch)`
  - Effect: completely custom per-epoch scaling (useful for bespoke decays).

# LambdaLR - Curves






- Parameters used: lr_lambda = $1/(1+0.1*epoch)$ (default)
- Final Train Loss: 0.0329
- Final Test Loss: 1.4957
- Final Test Accuracy: 71.43%

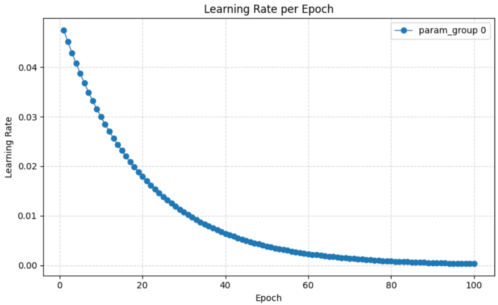
# MultiplicativeLR — Repeated multiplicative updates

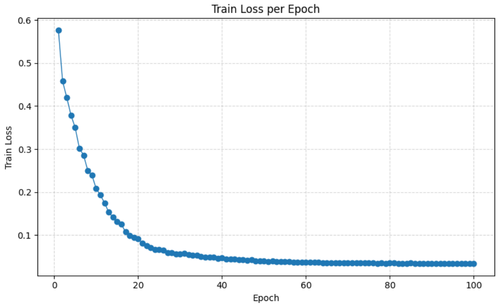


- **factor (float)** — multiplicative factor applied each epoch.
  - Default: `0.95`
  - Effect: $\mathrm{lr}_{t+1} = \mathrm{lr}_t \times$ factor (exponential-like decay).
- **lr_lambda (callable)** — alternative callable (defaults to constant factor).

# MultiplicativeLR - Curves





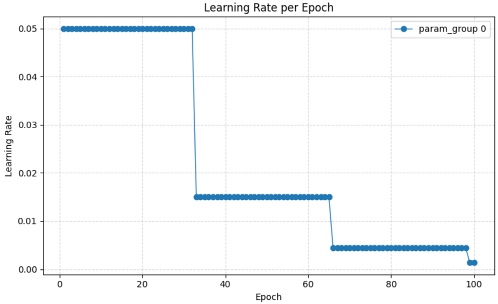

- Parameters used: factor = 0.95 (default)
- Final Train Loss: 0.0337
- Final Test Loss: 1.2416
- Final Test Accuracy: 71.43%

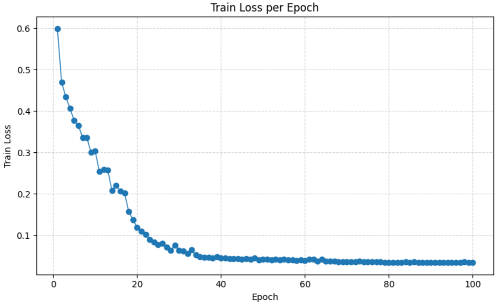
# StepLR — Discrete drops

- **step_size (int)** — epochs between drops.
  - Default: `30`
- **gamma (float)** — multiplicative drop factor.
  - Default: `0.1`

# StepLR — Curves

## 3 Learning-rate (LR) Schedulers



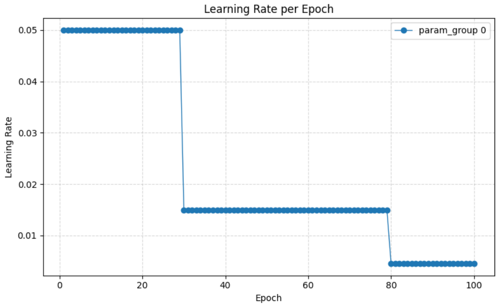

- Parameters used: step_size = 33, gamma = 0.3
- Final Train Loss: 0.0344
- Final Test Loss: 1.7086
- Final Test Accuracy: 67.86%

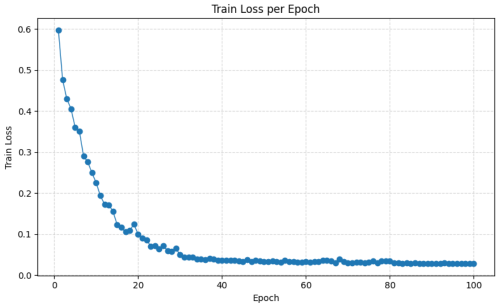
# MultiStepLR — Custom step milestones

- **milestones (list[int])** — epochs where LR is reduced.
  - Default: `[30, 60, 90]`
- **gamma (float)** — multiplicative factor at each milestone.
  - Default: `0.1`

# MultiStepLR — Curves

## 3 Learning-rate (LR) Schedulers




- Parameters used: milestones = [30, 80], gamma = 0.3
- Final Train Loss: 0.0279
- Final Test Loss: 1.1761
- Final Test Accuracy: 73.21%

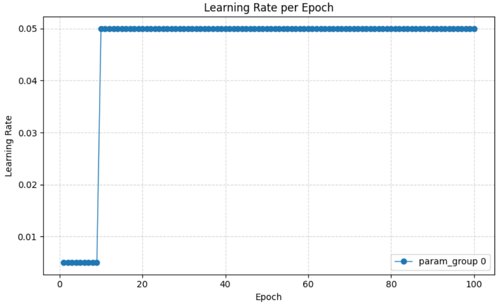
# ConstantLR — Constant schedule (warmup-like)

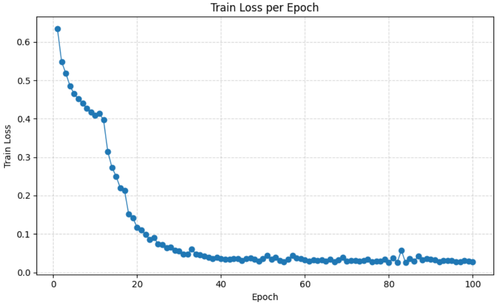


- **factor (float)** — multiplier applied during the constant period.
  - Default: `0.1`
- **total_iters (int)** — number of iterations the factor is kept.
  - Default: `max(1, int(0.05 * total_steps))` ( 5% of total steps if known)

# ConstantLR — Curves

## 3 Learning-rate (LR) Schedulers



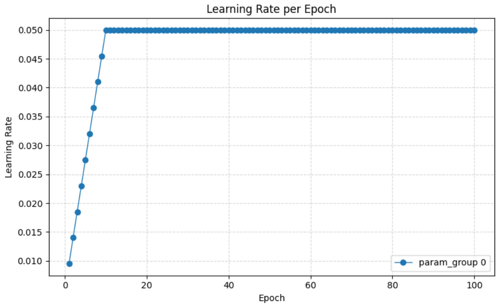

- Parameters used: factor = 0.1, total_iters = 10
- Final Train Loss: 0.0281
- Final Test Loss: 1.7522
- Final Test Accuracy: 69.64%

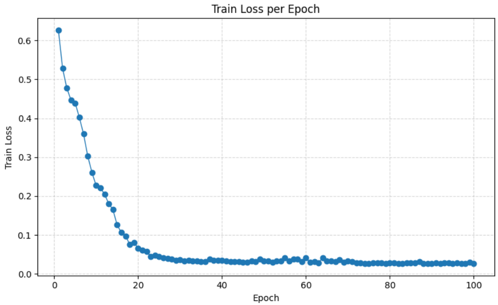
# LinearLR — Linear warmup

- **start_factor (float)** — starting multiplier of base LR.
  - Default: `0.1`
- **total_iters (int)** — warmup duration in iterations.
  - Default: `max(1, int(0.05 * total_steps))`

# LinearLR — Curves

3 Learning-rate (LR) Schedulers



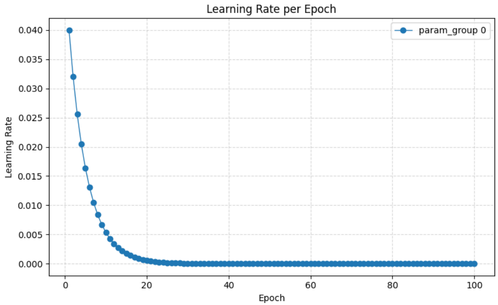

- Parameters used: start_factor = 0.1, total_iters = 10
- Final Train Loss: 0.0264
- Final Test Loss: 1.6642
- Final Test Accuracy: 82.14%

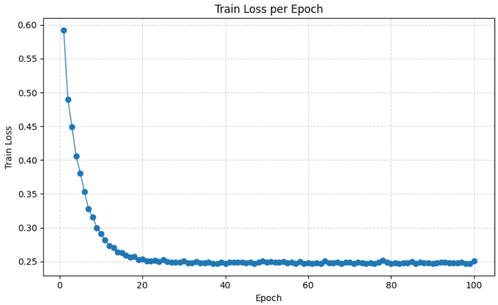
# ExponentialLR — Exponential decay

- **gamma (float)** — multiplicative decay factor per epoch/step.
  - Default: `0.95`

# ExponentialLR — Curves

## 3 Learning-rate (LR) Schedulers



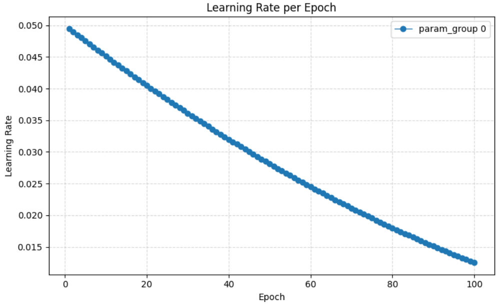

- Parameters used: gamma = 0.8
- Final Train Loss: 0.2508
- Final Test Loss: 0.6361
- Final Test Accuracy: 73.21%

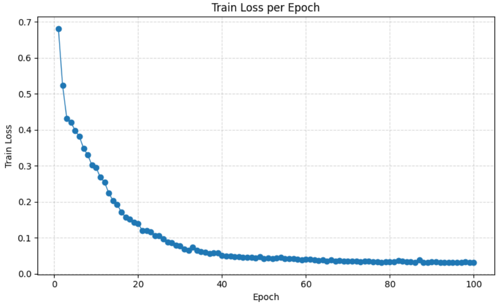
# PolynomialLR — Polynomial decay

- **power (float)** — exponent of the polynomial.
  - Default: `2.0`
- **total_iters (int)** — total number of iterations the decay spans.
  - Default: `total_steps` (if known), else number of epochs
- Effect: $\mathrm{lr}_t = \mathrm{lr}_0 \times (1 - t/\text{max\_iter})^{\text{power}}$

# PolynomialLR — Curves





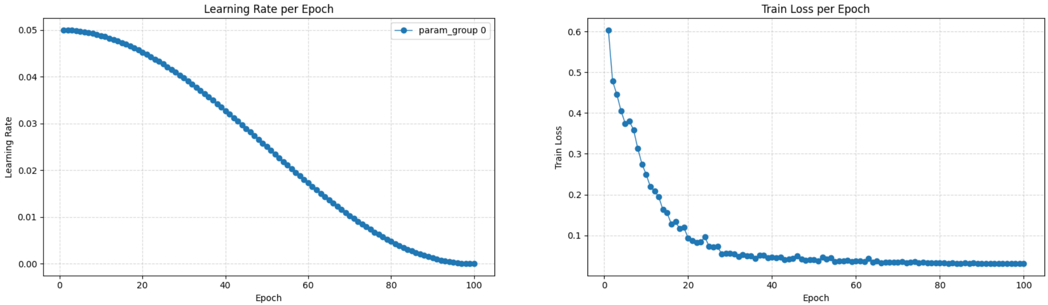

- Parameters used: power = 2.0, max_iter = 200
- Final Train Loss: 0.0312
- Final Test Loss: 1.3706
- Final Test Accuracy: 76.79%

# CosineAnnealingLR — Smooth cosine annealing

- **T_max (int)** — number of epochs or steps in one cycle.
  - Default: `num_epochs`
- **eta_min (float)** — minimum learning rate (floor).
  - Default: `0.0`

# CosineAnnealingLR — Curves

3 Learning-rate (LR) Schedulers




- Parameters used: T_max = 100, eta_min = 1e-6
- Final Train Loss: 0.0317
- Final Test Loss: 1.9866
- Final Test Accuracy: 62.50%

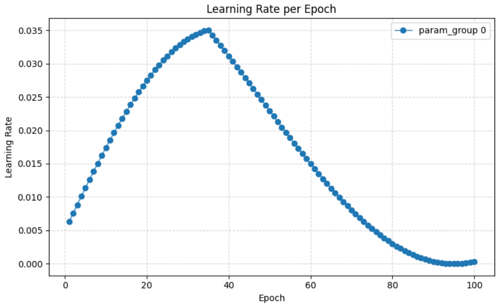
# ChainedScheduler — Combining schedules

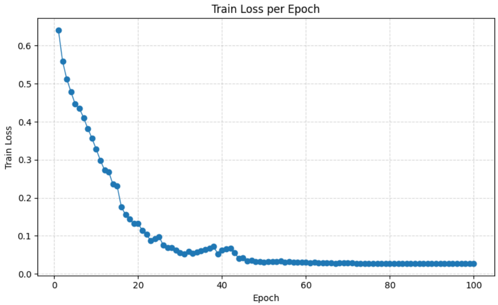


- Chains multiple schedulers sequentially (each scheduler runs for its configured duration).
- default factory: **LinearLR warmup** then **CosineAnnealingLR main**.
- No user params required for the default chain (factory builds sensible warmup length).

# ChainedScheduler — Curves

## 3 Learning-rate (LR) Schedulers




- Parameters used: default (Linear warmup + Cosine main)
- Final Train Loss: 0.0267
- Final Test Loss: 1.7336
- Final Test Accuracy: 71.43%

COPPE 60 anos UFRJ

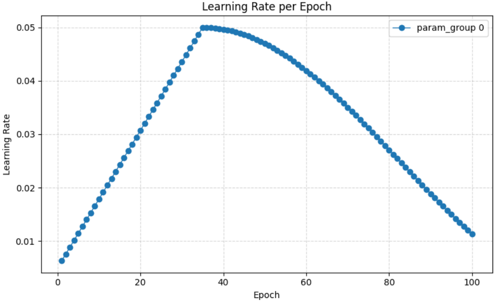
# SequentialLR — Sequential composition

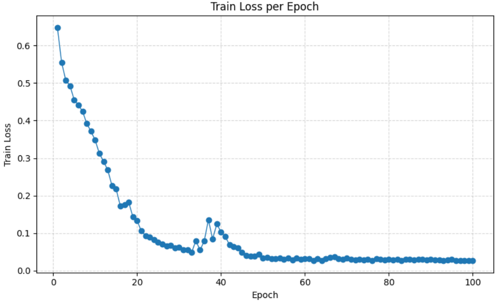


- **schedulers**: list of schedulers to run in sequence.
- **milestones**: list of integers indicating when to switch to the next scheduler.
  - Default: `[warmup_iters]` where warmup_iters 5% of total steps
- Use-case: warmup (LinearLR) → main (CosineAnnealingLR).

# SequentialLR — Curves

## 3 Learning-rate (LR) Schedulers




- Parameters used: default (LinearLR warmup + CosineAnnealingLR)
- Final Train Loss: 0.0276
- Final Test Loss: 1.7895
- Final Test Accuracy: 71.43%

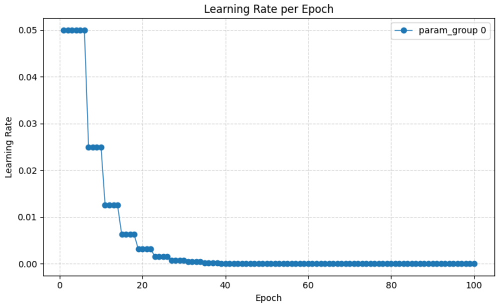
# ReduceLROnPlateau — Metric-driven reductions

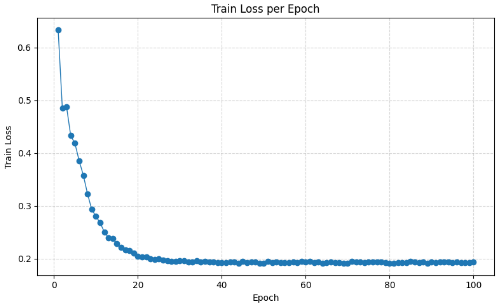


- **mode** — ''min'' or ''max''; which direction is "better".
  - Default: `'min'` (monitor metrics like validation loss)
- **factor (float)** — LR multiplication factor when reducing.
  - Default: `0.1`
- **patience (int)** — epochs without improvement before reducing.
  - Default: `5`
- **threshold (float)** — minimal change to count as improvement.
  - Default: `1e-4`
- **cooldown (int)** — epochs to wait after reduction.
  - Default: `0`
- **min_lr (float)** — lower bound for LR.
  - Default: `0.0`
- **eps (float)** — minimal decay to avoid tiny updates.
  - Default: `1e-8`

# ReduceLROnPlateau — Curves

## 3 Learning-rate (LR) Schedulers




- Parameters used: mode = min, factor = 0.5, patience = 3
- Final Train Loss: 0.1935
- Final Test Loss: 0.7553
- Final Test Accuracy: 66.07%

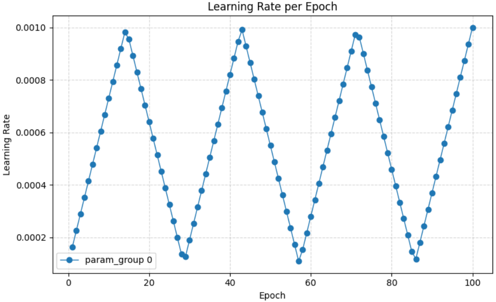
# CyclicLR — Cyclical policies

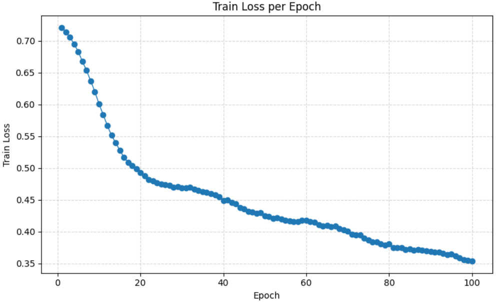


- **base_lr** — lower bound of cycle.
  - Default: `0.1 × initial lr`
- **max_lr** — upper bound of cycle.
  - Default: `10 × initial lr`
- **step_size_up (int)** — iterations to increase from base to max.
  - Default: `max(1, floor(steps_per_epoch/2))`
- **mode** — ‘triangular‘, ‘triangular2‘, ‘exp_range‘.
  - Default: `‘triangular‘`
- **cycle_momentum** — whether to cycle momentum as well.
  - Default: `False`

# CyclicLR — Curves

## 3 Learning-rate (LR) Schedulers



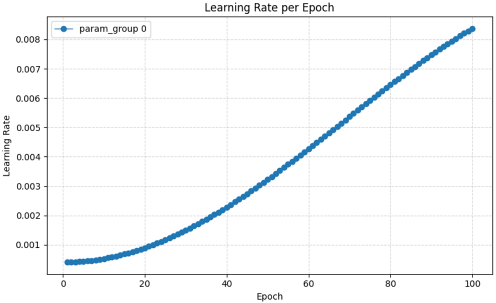

- Parameters used: base_lr = 0.0001, max_lr = 0.001, step_size_up = 100
- Final Train Loss: 0.3535
- Final Test Loss: 0.6226
- Final Test Accuracy: 69.64%

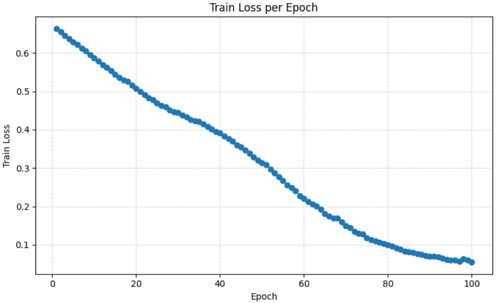
# OneCycleLR — Single-cycle super-convergence

- **max_lr (float)** — peak learning rate.
  - Default: `10 × initial lr`
- **total_steps (int)** — total number of optimizer steps (batches).
  - Default: `num_epochs × steps_per_epoch` (must be correct for batch stepping)
- **pct_start (float)** — fraction of total steps spent increasing to max_lr.
  - Default: `0.3`
- **anneal_strategy** — ‘cos‘ or ‘linear‘.
  - Default: `‘cos‘`
- **div_factor** — initial LR = max_lr / div_factor.
  - Default: `25.0`
- **final_div_factor** — min LR = initial LR / final_div_factor.
  - Default: `1e4`
- Note: **step this scheduler every batch**, not per epoch.

# OneCycleLR — Curves

## 3 Learning-rate (LR) Schedulers




- Parameters used: max_lr = 0.01, total_steps = 3200, batch_size = 32
- Final Train Loss: 0.0544
- Final Test Loss: 0.7502
- Final Test Accuracy: 69.64%

# Table of Contents

# Experiment: *VariateHistorySizeInLBFGSExperiment*



Link to GitHub

- Goal: isolates the effect of m (memory size) in L-BFGS.
- Dataset: MNIST [2]
- Base model: simple MLP (Multi-Layer Perceptron).
- Evaluation:
  - Áverage loss.
  - Accuracy.

# Experiment: *VariateHistorySizeInLBFGSExperiment*

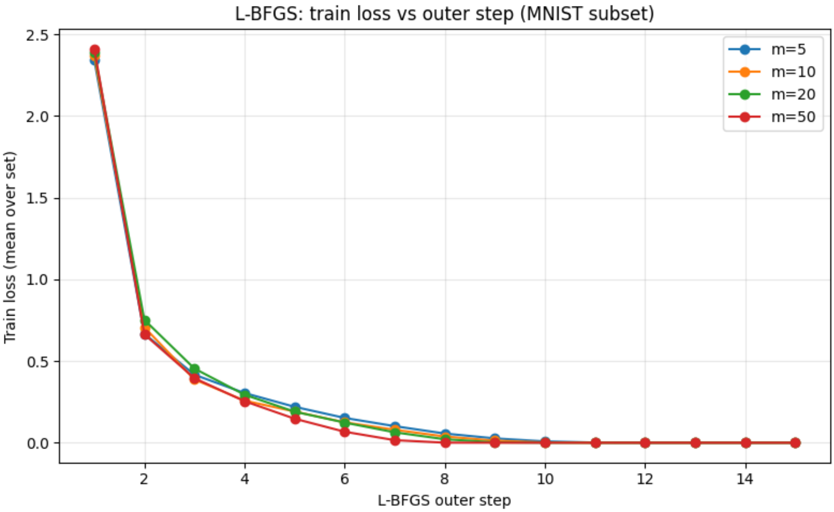
Link to GitHub

The comparison is done as following:

```
for step in range(outer_steps):
    loss = optim.step(closure)
    # Log da perda (como número Python)
    outer_losses.append(float(loss.detach().cpu()))
```

# Results




L-BFGS: train loss vs outer step (MNIST subset)
Train loss (mean over set)
L-BFGS outer step
m=5
m=10
m=20
m=50
2.5
2.0
1.5
1.0
0.5
0.0
2
4
6
8
10
12
14

# Table of Contents

# Dataloader: CIFAR-10 [3]



Link to GitHub

```python
class LabelNoiseDataset(torch.utils.data.Dataset):
    """Aplica ruído de rótulo com prob p_noise."""
    def __init__(self, base_ds, p_noise: float, num_classes: int = 10, seed: int = 0):
        self.base = base_ds
        self.p = float(p_noise)
        self.C = int(num_classes)
        rng = np.random.RandomState(seed)
        self.flip_mask = rng.rand(len(self.base)) < self.p
        self.rand_labels = rng.randint(0, self.C, size=len(self.base))
```

```python
class TestNoiseWrapper(torch.utils.data.Dataset):
    """
    Adiciona ruído gaussiano em pixel space, faz clamp [0,1] e depois normaliza com mean/std.
    """
    def __init__(self, base_ds, sigma: float, mean_=mean, std_=std):
        self.base, self.sigma = base_ds, float(sigma)
        self.normalize = T.Normalize(mean_, std_)
```

# Experimet: *OptimizersRobustnessToNoiseExperiment*

Link to GitHub

- Goal: compare robustness to noise and memory use among different optimizers in CIFAR-10.
- Base model: `ResNet-18` (10 classes).
- Types of noise:
  - **Label noise** in training (with $p = 0.2$).
  - **Input noise** in testing ($\sigma \in \{0, 0.05, 0.1, 0.2\}$).
- Evaluation:
  - Área unther the curve (AUC): model's ability to rank positive examples higher than negative ones.
  - Pick of memory usage CUDA.

# Configuration

5 Comparing different types of optimizers

Link to GitHub

```python
def make_optimizer(name, params):
    name = name.lower()
    if name == 'sgd':
        return torch.optim.SGD(params, lr=0.1, momentum=0.0, weight_decay=5e-4)
    if name == 'nesterov':
        return torch.optim.SGD(params, lr=0.1, momentum=0.9, nesterov=True, weight_decay=5e-4)
    if name == 'rmsprop':
        return torch.optim.RMSprop(params, lr=1e-3, alpha=0.99, eps=1e-8, weight_decay=1e-5)
    if name == 'adamw':
        return torch.optim.AdamW(params, lr=3e-4, betas=(0.9,0.999), eps=1e-8, weight_decay=0.01)
    if name == 'radam':
        return torch.optim.RAdam(params, lr=3e-4, betas=(0.9,0.999), eps=1e-8, weight_decay=1e-4)
    if name == 'l-bfgs':
        return torch.optim.LBFGS(params, lr=1.0, history_size=10, line_search_fn=None)
```

# Evaluation

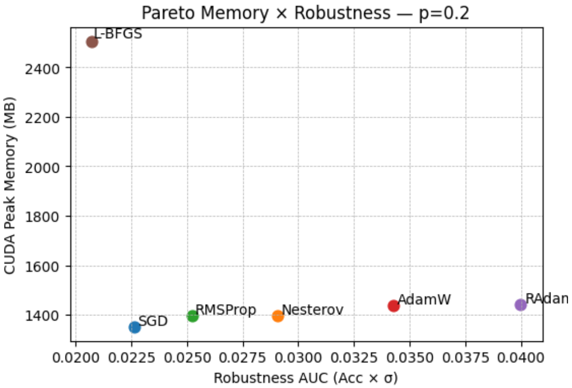
Link to GitHub

```
def auc_robustez(acc_by_sigma: dict):
    xs = sorted(acc_by_sigma.keys())
    ys = [acc_by_sigma[x]['acc'] for x in xs]
    area = 0.0
    for i in range(len(xs)-1):
        dx = xs[i+1] - xs[i]
        area += 0.5 * (ys[i] + ys[i+1]) * dx
    return area
```

# Results

## 5 Comparing different types of optimizers

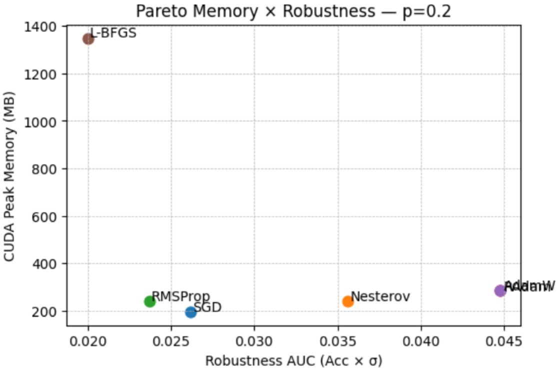
```
Starting experiment on cuda | EPOCHS=5 | BATCH_SIZE=128 | SUBSET_TRAIN=10000
Optimizers: ['SGD', 'Nesterov', 'RMSProp', 'AdamW', 'RAdam', 'L-BFGS']
```



# Results

## 5 Comparing different types of optimizers

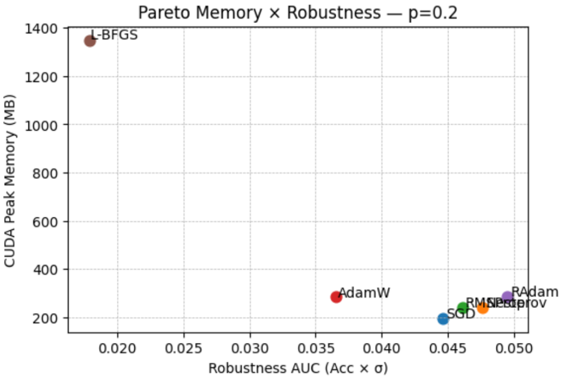
```
Starting experiment on cuda | EPOCHS=10 | BATCH_SIZE=128 | SUBSET_TRAIN=10000
Optimizers: ['SGD', 'Nesterov', 'RMSProp', 'AdamW', 'RAdam', 'L-BFGS']
```



# Results

## 5 Comparing different types of optimizers

```
Starting experiment on cuda | EPOCHS=10 | BATCH_SIZE=128 | SUBSET_TRAIN=20000
Optimizers: ['SGD', 'Nesterov', 'RMSProp', 'AdamW', 'RAdam', 'L-BFGS']
```



# Bibliography



[1] PyTorch Core Team, *LBFGS - PyTorch Documentation*, 2025. Accessed: October 2025.

[2] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[3] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., 2009.

# Optimizers in deep learning - Practical study

*Obrigado pela Atenção!*
*Alguma Pergunta?*