

## elra - gradient descent solver

Gradient descent solver/optimizer with internal good-natured hyperparameter control.

### Overview

This package provides an gradient descent solver/optimizer compatible to `torch.optim.Optimizer` (PyTorch Optimizer, e.g. Adam or SGD). Even the original paper describes two variants (C2M and P2M), only P2M is continued.

**Reference:** Alexander Fauck and Alexander Kleinsorge, \* “ELRA: Exponential learning rate adaption gradient descent optimization method” (2023) arxiv 309.06274

- “A Novel Exponential Continuous Learning Rate Adaption Gradient Descent Optimization Method” (2025) Wildauer Konferenz für Künstliche Intelligenz 2025 (WiKKI25)

### Main principles

- Adaptive alpha: learning rate (LR) control based on actual gradient pair (L2-norms and angle between, easy computable via dot-product). Advantage of huge steps has been shown by Grimmer-2023.
- Retrace steps: revert failed steps, at seldom loss explosion events (threshold not trivial). Huge steps are sometimes risky (even deadly for the run). Internally `step_retrace(self, loss: float)` completely ignores the gradient, which allows for skipping `backward()` calculation.
- Dynamic batch size: averaging gradients of dynamic (integer) collector multiplier, depending on loss progress statistics. Advantage of growing batch size for final results has been shown by Smith-2017.
- Dynamic weight decay (WD): to reduce over-fitting, usually WD decays weights, here controlled depending on comparison of train-loss vs. validation-loss. This requires calling `optim.set_valid_loss(loss: float)` and `optim.set_train_loss(loss: float)` every epoch (or periodically).
- Boosting (SWA): SWA-boosting allows an early estimation of later/final results, but only for print/log reasons (unused by algorithm). `optim.get_boost_model(enable=True, model)` prepares such a model, if needed.
- external Scheduler: ELRA is self adapting its hyperparameters, external Schedulers will be ignored, like: `torch.optim.lr_scheduler.LRScheduler(optimizer)`

**Boosting background** Literature describes different weight avering methods (e.g. SWA, EMA, Polyak-avg.). This is only helpfull after intial warmup phase (single epochs), as it pulls into results of the past. Huge learning rates (as used by ELRA) cause strong path oscillations. SWA-boosting helps an early estimation

of later/final results, but only for print/log reasons (not used for algorithm internal). ELRA-Boosting is a cyclic equal averaging of model weights (usually represented by `x Tensor`). This is similar to Stochastic Weight Averaging (SWA) or sometimes referred to as Polyak-Ruppert averaging, or averaged SGD (but there as exponential moving average EMA). This is helpful after warmup (first few epochs) and before reaching final convergence (normal loss and boosted loss usually flow together at the end). Gradient descent with huge learning rates (ELRA  $>1$ , in contrast to tiny values for Adam/SGD) shows significant advantage for loss and accuracy, which allows an early visible indication of later/final results reached. Important: this has no effect to path of ELRA algorithm, but is a helpful print/log in between. ‘Acceleration of stochastic approximation by averaging’, (Polyak-1992). Similar to torch/SWALR, but for now separated. `torch.optim.swa_utils.SWALR(opt, anneal_strategy="linear", anneal_epochs=1, )` <https://pytorch.org/blog/pytorch-1-6-now-includes-stochastic-weight-averaging/>

### External References for Algorithm motivation

- (Grimmer-2023) ‘Accelerated Gradient Descent via Long Steps’ Benjamin Grimmer, et. al.; eprint arXiv (2023)
- (Smith-2017) ‘Don’t Decay the Learning Rate, Increase the Batch Size’ Samuel L Smith, et. al.; arXiv (2017)
- (Polyak-1992) ‘Acceleration of stochastic approximation by averaging’. Boris T Polyak and Anatoli B Juditsky; SIAM Journal on Control and Optimization, 30(4):838–855, (1992)

### Installation

For a quick start, you can just install the package via pip:

```
pip install elra
```

If you want to use a certain version of PyTorch, make sure to install it first. See the PyTorch website for more information.

### Requirements

- Python 3.10+
- PyTorch 2.0+ (for vector math and cuda support)

### Quick Start

#### Example 1

This example is taken from the PyTorch documentation and based on the example PyTorch: optim

```
import torch
import math
```

```

from elra import ElraOptimizer # import the elra optimizer

# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# Prepare the input tensor (x, x^2, x^3).
p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)
loss_fn = torch.nn.MSELoss(reduction='sum')

# Use the optim package to define an Optimizer that will update the weights of
# the model for us. Here we will use elra
learning_rate = 1e-6

optimizer = ElraOptimizer(model.parameters(), model=model, lr=learning_rate)

for t in range(20):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(xx)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    loss_val = loss.item()

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers( i.e, not overwritten) whenever .backward()
    # is called. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    if isinstance(optimizer, ElraOptimizer):

```

```

        optimizer.step(loss_val)  # elra needs loss value for internal control
    else:
        optimizer.step()

linear_layer = model[0]
print(f'Result: y = {linear_layer.bias.item()} + {linear_layer.weight[:, 0].item()} x + {lin

```

## Example 2

This example is the classical mathematical 2D landscape of non-convex Rosenbrock function. Hint: here (no noise, no stochastic), ELRA is using a modified control method. Rosenbrock (1960)

```

import torch
from elra import ElraOptimizer

assert callable(model_class), "model_class not model()"  # as usual

batch_size, num_classes = 32, 10  # int: optional (0=default)
lr, wdecay = 1e-5, 0.99999  # float

# create/init optimizer class
optim = opt_class(model.parameters(), None, batch_size, num_classes, lr=lr, weight_decay=wdecay)
optim.add_param_group(...)  # optional (e.g. YOLO-5), Caution: on WD is considered

loss_limit: float = initial_loss * 1.5  # for retrace decision (failure check)
params = tuple(model.parameters())
optim.get_boost_model(True, model, None)  # turn on optional boosting (see below)

for X, y in data_loader:
    # Training Step
    optim.zero_grad(set_to_none=True)

    loss = loss_func(model(X), y)
    loss_item: float = loss.item()

    if not (loss_item < limitf):  # isnan(loss_item)
        optim.step_retrace(loss_item)  # happens < 1% of steps
        continue  # save backward on mis-step

    # grads = tt.cat( [p.grad.data.flatten() for p in params] )
    loss.backward()  # computes gradient (as usual)
    optim.step(loss_item)  # Caution: ELRA needs loss value (by design)

optim.set_valid_loss(val_loss_item: float)  # WD-control needs alidation loss values (e.g.

```

```

# optimal boosting: kind of trajectory_averaging (Polyak:1992)
cnt, avg_loss, model2 = optim.get_boost_model(True, model, device=None)
calc_FullBatchLoss(
    model2) # early indication of better loss (due to aggressive step-size oscillation ELRA

```

### Example 3

This example is similar to Example 1 (MNIST-10 with trivial 2-layer DNN), but using our ElraGradScaler and half-precision (float16). `class ElraGradScaler` is similar to `torch.amp.GradScaler()`, but shows the most simple and fastest way of ELRA integration. Only single line of code changed `GradScaler => ElraGradScaler` (covering ELRA specifics, like loss-value argument) and fast because double usage of same gradient tensor (`scaler.update( grad.norm(inf))` and `optim.step(grad, loss.item())`).

```

import torch
from timm.data.loader import MultiEpochsDataLoader
from elra import ElraOptimizer, ElraGradScaler

torch.set_default_dtype(torch.float16)
data_loader: DataLoader = MultiEpochsDataLoader(train_data, ..)
model: torch.nn.Module = model_class(*model_hyperparams).to(device)
optim = ElraOptimizer(model.parameters(), lr=1.0e-4, warmup=10)
scaler = ElraGradScaler(init_scale=2.0**16, enabled=True)

for X, y in data_loader:
    optim.zero_grad()
    loss: torch.Tensor = loss_function(model(X), y)

    # loss.backward() # would compute gradient w/o scaling
    scaler.scale(loss).backward() # loss*=scaler.get_scale() inside GradScaler
    scaler.step(optim) # argument step(loss.item()) hidden in ElraGradScaler
    scaler.update()

```

### Creating Custom Tests

Should be comparable to `torch.optim.Optimizer`.

### Documentation

The user manual is included in the package:

```

import elra_optimizer as elra

print(get_user_manual_path()) # Path to userManual.pdf

```

## **License**

GNU General Public License v3 or later (GPLv3+)

## **Authors**

- Alexander Kleinsorge (Technische Hochschule Wildau)
- Alexander Fauck (Technische Hochschule Wildau)

## **Links**

- Repository: [gitlab.com/elra](https://gitlab.com/elra)
- Documentation (PDF)