Package ‘torchopt’

June 6, 2023

Type Package

Title Advanced Optimizers for Torch

Version 0.1.4

Maintainer Gilberto Camara <gilberto.camara.inpe@gmail.com>

Description Optimizers for 'torch' deep learning library. These functions include recent results published in the literature and are not part of the optimizers offered in 'torch'. Prospective users should test these optimizers with their data, since performance depends on the specific problem being solved. The packages includes the following optimizers: (a) 'adabelief' by Zhuang et al (2020), <arXiv:2010.07468>; (b) 'adabound' by Luo et al.(2019), <arXiv:1902.09843>; (c) 'adahessian' by Yao et al.(2021) <arXiv:2006.00719>; (d) 'adamw' by Loshchilov & Hutter (2019), <arXiv:1711.05101>; (e) 'madgrad' by Defazio and Jelassi (2021), <arXiv:2101.11075>; (f) 'nadam' by Dozat (2019), <https://openreview.net/pdf/OM0jvwB8jIp57ZJjtNEZ.pdf>; (g) 'qhadam' by Ma and Yarats(2019), <arXiv:1810.06801>; (h) 'radam' by Liu et al. (2019), <arXiv:1908.03265>; (i) 'swats' by Shekar and Sochee (2018), <arXiv:1712.07628>; (j) 'yogi' by Zaheer et al.(2019), <https://papers.nips.cc/paper/8186-adaptive-methods-for-nonconvex-optimization>.

License Apache License (>= 2)

URL https://github.com/e-sensing/torchopt/

Depends R (>= 4.0.0)

Imports graphics, grDevices, stats, torch

Suggests testthat

ByteCompile true

Encoding UTF-8

Language en-US

RoxygenNote 7.2.0

NeedsCompilation no

Author Gilberto Camara [aut, cre],
Rolf Simoes [aut],
Daniel Falbel [aut],
Felipe Souza [aut]

Repository CRAN

Date/Publication 2023-06-06 13:10:02 UTC

R topics documented:

optim_adabelief *Adabelief optimizer*

Description

R implementation of the adabelief optimizer proposed by Zhuang et al (2020). We used the pytorch implementation developed by the authors which is available at https://github.com/jettify/pytorch-optimizer. Thanks to Nikolay Novik of his work on python optimizers.

The original implementation is licensed using the Apache-2.0 software license. This implementation is also licensed using Apache-2.0 license.

From the abstract by the paper by Zhuang et al (2021): We propose Adabelief to simultaneously achieve three goals: fast convergence as in adaptive methods, good generalization as in SGD, and training stability. The intuition for AdaBelief is to adapt the stepsize according to the "belief" in the current gradient direction. Viewing the exponential moving average of the noisy gradient as the prediction of the gradient at the next time step, if the observed gradient greatly deviates from the prediction, we distrust the current observation and take a small step; if the observed gradient is close to the prediction, we trust it and take a large step.

Usage

```
optim_adabelief(
  params,
  lr = 0.001,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  weight_decay = 1e-06,
  weight_decouple = TRUE,
  fixed_decay = FALSE,
  rectify = TRUE
)
```

Arguments

`params` List of parameters to optimize.

`lr` Learning rate (default: 1e-3)

`betas` Coefficients for computing running averages of gradient and its square (default: (0.9, 0.999))

`eps` Term added to the denominator to improve numerical stability (default: 1e-16)

`weight_decay` Weight decay (L2 penalty) (default: 0)

`weight_decouple` Use decoupled weight decay as is done in AdamW?

`fixed_decay` This is used when weight_decouple is set as True. When fixed_decay == True, weight decay is W_new = W_old - W_old * decay. When fixed_decay == False, the weight decay is W_new = W_old - W_old * decay * learning_rate. In this case, weight decay decreases with learning rate.

`rectify` Perform the rectified update similar to RAdam?

Value

A torch optimizer object implementing the `step` method.

Author(s)

Gilberto Camara, <`gilberto.camara@inpe.br`>

Rolf Simoes, <`rolf.simoes@inpe.br`>

Felipe Souza, <`lipecaso@gmail.com`>

Alber Sanchez, <`alber.ipia@inpe.br`>

References

Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, James S. Duncan. "Adabelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients", 34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada. https://arxiv.org/abs/2010.07468

Examples

```
if (torch::torch_is_installed()) {
# function to demonstrate optimization
beale <- function(x, y) {
    log((1.5 - x + x * y)^2 + (2.25 - x - x * y^2)^2 + (2.625 - x + x * y^3)^2)
 }
# define optimizer
optim <- torchopt::optim_adabelief
# define hyperparams
opt_hparams <- list(lr = 0.01)

# starting point
x0 <- 3
y0 <- 3
# create tensor
x <- torch::torch_tensor(x0, requires_grad = TRUE)
y <- torch::torch_tensor(y0, requires_grad = TRUE)
# instantiate optimizer
optim <- do.call(optim, c(list(params = list(x, y)), opt_hparams))
# run optimizer
steps <- 400
x_steps <- numeric(steps)
y_steps <- numeric(steps)
for (i in seq_len(steps)) {
    x_steps[i] <- as.numeric(x)
    y_steps[i] <- as.numeric(y)
    optim$zero_grad()
    z <- beale(x, y)
    z$backward()
    optim$step()
}
print(paste0(”starting value = ”, beale(x0, y0)))
print(paste0(”final value = ”, beale(x_steps[steps], y_steps[steps])))
}
```

optim_adabound *Adabound optimizer*

Description

R implementation of the AdaBound optimizer proposed by Luo et al.(2019). We used the implementation available at https://github.com/jettify/pytorch-optimizer/blob/master/torch_optimizer/yogi.py. Thanks to Nikolay Novik for providing the pytorch code.

The original implementation is licensed using the Apache-2.0 software license. This implementation is also licensed using Apache-2.0 license.

AdaBound is a variant of the Adam stochastic optimizer which is designed to be more robust to extreme learning rates. Dynamic bounds are employed on learning rates, where the lower and upper bound are initialized as zero and infinity respectively, and they both smoothly converge to a constant final step size. AdaBound can be regarded as an adaptive method at the beginning of training, and thereafter it gradually and smoothly transforms to SGD (or with momentum) as the time step increases.

Usage

```
optim_adabound(
  params,
  lr = 0.001,
  betas = c(0.9, 0.999),
  final_lr = 0.1,
  gamma = 0.001,
  eps = 1e-08,
  weight_decay = 0
)
```

Arguments

`params`	List of parameters to optimize.
`lr`	Learning rate (default: 1e-3)
`betas`	Coefficients computing running averages of gradient and its square (default: (0.9, 0.999))
`final_lr`	Final (SGD) learning rate (default: 0.1)
`gamma`	Convergence speed of the bound functions (default: 1e-3)
`eps`	Term added to the denominator to improve numerical stability (default: 1e-8)
`weight_decay`	Weight decay (L2 penalty) (default: 0)

Value

A torch optimizer object implementing the `step` method.

Author(s)

Rolf Simoes, <`rolf.simoes@inpe.br`>

Felipe Souza, <`lipecaso@gmail.com`>

Alber Sanchez, <`alber.ipia@inpe.br`>

Gilberto Camara, <`gilberto.camara@inpe.br`>

References

Liangchen Luo, Yuanhao Xiong, Yan Liu, Xu Sun, "Adaptive Gradient Methods with Dynamic Bound of Learning Rate", International Conference on Learning Representations (ICLR), 2019. https://arxiv.org/abs/1902.09843

Examples

```
if (torch::torch_is_installed()) {
# function to demonstrate optimization
beale <- function(x, y) {
    log((1.5 - x + x * y)^2 + (2.25 - x - x * y^2)^2 + (2.625 - x + x * y^3)^2)
 }
# define optimizer
optim <- torchopt::optim_adabound
# define hyperparams
opt_hparams <- list(lr = 0.01)

# starting point
x0 <- 3
y0 <- 3
# create tensor
x <- torch::torch_tensor(x0, requires_grad = TRUE)
y <- torch::torch_tensor(y0, requires_grad = TRUE)
# instantiate optimizer
optim <- do.call(optim, c(list(params = list(x, y)), opt_hparams))
# run optimizer
steps <- 400
x_steps <- numeric(steps)
y_steps <- numeric(steps)
for (i in seq_len(steps)) {
    x_steps[i] <- as.numeric(x)
    y_steps[i] <- as.numeric(y)
    optim$zero_grad()
    z <- beale(x, y)
    z$backward()
    optim$step()
}
print(paste0("starting value = ", beale(x0, y0)))
print(paste0("final value = ", beale(x_steps[steps], y_steps[steps])))
}
```

---

optim_adahessian *Adahessian optimizer*

Description

R implementation of the Adahessian optimizer proposed by Yao et al.(2020). The original implementation is available at https://github.com/amirgholami/adahessian.

Usage

```
optim_adahessian(
  params,
  lr = 0.15,
  betas = c(0.9, 0.999),
  eps = 1e-04,
  weight_decay = 0,
  hessian_power = 0.5
)
```

Arguments

`params`	Iterable of parameters to optimize.
`lr`	Learning rate (default: 0.15).
`betas`	Coefficients for computing running averages of gradient and is square(default: (0.9, 0.999)).
`eps`	Term added to the denominator to improve numerical stability (default: 1e-4).
`weight_decay`	L2 penalty (default: 0).
`hessian_power`	Hessian power (default: 1.0).

Value

An optimizer object implementing the `step` and `zero_grad` methods.

Author(s)

Rolf Simoes, <`rolf.simoes@inpe.br`>

Felipe Souza, <`lipecaso@gmail.com`>

Alber Sanchez, <`alber.ipia@inpe.br`>

Gilberto Camara, <`gilberto.camara@inpe.br`>

References

Yao, Z., Gholami, A., Shen, S., Mustafa, M., Keutzer, K., & Mahoney, M. (2021). ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning. Proceedings of the AAAI Conference on Artificial Intelligence, 35(12), 10665-10673. https://arxiv.org/abs/2006.00719

`optim_adamw` *AdamW optimizer*

Description

R implementation of the AdamW optimizer proposed by Loshchilov & Hutter (2019). We used the pytorch implementation developed by Collin Donahue-Oponski available at: https://gist.github.com/colllin/0b146b154c4351f

From the abstract by the paper by Loshchilov & Hutter (2019): L2 regularization and weight decay regularization are equivalent for standard stochastic gradient descent (when rescaled by the learning rate), but as we demonstrate this is not the case for adaptive gradient algorithms, such as Adam. While common implementations of these algorithms employ L2 regularization (often calling it “weight decay” in what may be misleading due to the inequivalence we expose), we propose a simple modification to recover the original formulation of weight decay regularization by decoupling the weight decay from the optimization steps taken w.r.t. the loss function

Usage

```
optim_adamw(
  params,
  lr = 0.01,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  weight_decay = 1e-06
)
```

Arguments

params	List of parameters to optimize.
lr	Learning rate (default: 1e-3)
betas	Coefficients computing running averages of gradient and its square (default: (0.9, 0.999))
eps	Term added to the denominator to improve numerical stability (default: 1e-8)
weight_decay	Weight decay (L2 penalty) (default: 1e-6)

Value

A torch optimizer object implementing the step method.

Author(s)

Gilberto Camara, <gilberto.camara@inpe.br>

Rolf Simoes, <rolf.simoes@inpe.br>

Felipe Souza, <lipecaso@gmail.com>

Alber Sanchez, <alber.ipia@inpe.br>

References

Ilya Loshchilov, Frank Hutter, "Decoupled Weight Decay Regularization", International Conference on Learning Representations (ICLR) 2019. https://arxiv.org/abs/1711.05101

Examples

```
if (torch::torch_is_installed()) {
# function to demonstrate optimization
beale <- function(x, y) {
    log((1.5 - x + x * y)^2 + (2.25 - x - x * y^2)^2 + (2.625 - x + x * y^3)^2)
 }
# define optimizer
optim <- torchopt::optim_adamw
# define hyperparams
opt_hparams <- list(lr = 0.01)

# starting point
x0 <- 3
y0 <- 3
# create tensor
x <- torch::torch_tensor(x0, requires_grad = TRUE)
y <- torch::torch_tensor(y0, requires_grad = TRUE)
# instantiate optimizer
optim <- do.call(optim, c(list(params = list(x, y)), opt_hparams))
# run optimizer
steps <- 400
x_steps <- numeric(steps)
y_steps <- numeric(steps)
for (i in seq_len(steps)) {
    x_steps[i] <- as.numeric(x)
    y_steps[i] <- as.numeric(y)
    optim$zero_grad()
    z <- beale(x, y)
    z$backward()
    optim$step()
}
print(paste0("starting value = ", beale(x0, y0)))
print(paste0("final value = ", beale(x_steps[steps], y_steps[steps])))
}
```

optim_madgrad *MADGRAD optimizer*

Description

A Momentumized, Adaptive, Dual Averaged Gradient Method for Stochastic Optimization (MADGRAD) is a general purpose optimizer that can be used in place of SGD or Adam may converge faster and generalize better. Currently GPU-only. Typically, the same learning rate schedule that is used for SGD or Adam may be used. The overall learning rate is not comparable to either method and should be determined by a hyper-parameter sweep.

MADGRAD requires less weight decay than other methods, often as little as zero. Momentum values used for SGD or Adam's beta1 should work here also.

On sparse problems both weight_decay and momentum should be set to 0. (not yet supported in the R implementation).

Usage

```
optim_madgrad(params, lr = 0.01, momentum = 0.9, weight_decay = 0, eps = 1e-06)
```

Arguments

`params`	List of parameters to optimize.
`lr`	Learning rate (default: 1e-2).
`momentum`	Momentum value in the range [0,1) (default: 0.9).
`weight_decay`	Weight decay, i.e. a L2 penalty (default: 0).
`eps`	Term added to the denominator outside of the root operation to improve numerical stability (default: 1e-6).

Value

A torch optimizer object implementing the step method.

Author(s)

Daniel Falbel, <dfalbel@gmail.com>

References

Aaron Defazio, Samy Jelassi, "Adaptivity without Compromise: A Momentumized, Adaptive, Dual Averaged Gradient Method for Stochastic Optimization". https://arxiv.org/abs/2101.11075

Examples

```
if (torch::torch_is_installed()) {
# function to demonstrate optimization
beale <- function(x, y) {
    log((1.5 - x + x * y)^2 + (2.25 - x - x * y^2)^2 + (2.625 - x + x * y^3)^2)
 }
# define optimizer
optim <- torchopt::optim_madgrad
# define hyperparams
opt_hparams <- list(lr = 0.01)

# starting point
x0 <- 3
y0 <- 3
# create tensor
x <- torch::torch_tensor(x0, requires_grad = TRUE)
y <- torch::torch_tensor(y0, requires_grad = TRUE)
# instantiate optimizer
optim <- do.call(optim, c(list(params = list(x, y)), opt_hparams))
# run optimizer
steps <- 400
x_steps <- numeric(steps)
y_steps <- numeric(steps)
for (i in seq_len(steps)) {
    x_steps[i] <- as.numeric(x)
    y_steps[i] <- as.numeric(y)
    optim$zero_grad()
    z <- beale(x, y)
    z$backward()
    optim$step()
}
print(paste0("starting value = ", beale(x0, y0)))
print(paste0("final value = ", beale(x_steps[steps], y_steps[steps])))
}
```

optim_nadam *Nadam optimizer*

Description

R implementation of the Nadam optimizer proposed by Dazat (2016).

From the abstract by the paper by Dozat (2016): This work aims to improve upon the recently proposed and rapidly popularized optimization algorithm Adam (Kingma & Ba, 2014). Adam has two main components—a momentum component and an adaptive learning rate component. However, regular momentum can be shown conceptually and empirically to be inferior to a similar algorithm known as Nesterov's accelerated gradient (NAG).

Usage

```
optim_nadam(
  params,
  lr = 0.002,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  weight_decay = 0,
  momentum_decay = 0.004
)
```

Arguments

`params`	List of parameters to optimize.
`lr`	Learning rate (default: 1e-3)
`betas`	Coefficients computing running averages of gradient and its square (default: (0.9, 0.999)).
`eps`	Term added to the denominator to improve numerical stability (default: 1e-8).
`weight_decay`	Weight decay (L2 penalty) (default: 0).
`momentum_decay`	Momentum_decay (default: 4e-3).

Value

A torch optimizer object implementing the `step` method.

Author(s)

Gilberto Camara, <gilberto.camara@inpe.br>

Rolf Simoes, <rolf.simoes@inpe.br>

Felipe Souza, <lipecaso@gmail.com>

Alber Sanchez, <alber.ipia@inpe.br>

References

Timothy Dozat, "Incorporating Nesterov Momentum into Adam", International Conference on Learning Representations (ICLR) 2016. https://openreview.net/pdf/OM0jvwB8jIp57ZJjtNEZ.pdf

Examples

```
if (torch::torch_is_installed()) {
# function to demonstrate optimization
beale <- function(x, y) {
    log((1.5 - x + x * y)^2 + (2.25 - x - x * y^2)^2 + (2.625 - x + x * y^3)^2)
 }
# define optimizer
optim <- torchopt::optim_nadam
# define hyperparams
opt_hparams <- list(lr = 0.01)

# starting point
x0 <- 3
y0 <- 3
# create tensor
x <- torch::torch_tensor(x0, requires_grad = TRUE)
y <- torch::torch_tensor(y0, requires_grad = TRUE)
# instantiate optimizer
optim <- do.call(optim, c(list(params = list(x, y)), opt_hparams))
# run optimizer
steps <- 400
x_steps <- numeric(steps)
y_steps <- numeric(steps)
for (i in seq_len(steps)) {
    x_steps[i] <- as.numeric(x)
    y_steps[i] <- as.numeric(y)
    optim$zero_grad()
    z <- beale(x, y)
    z$backward()
    optim$step()
}
print(paste0("starting value = ", beale(x0, y0)))
print(paste0("final value = ", beale(x_steps[steps], y_steps[steps])))
}
```

optim_qhadam *QHAdam optimization algorithm*

Description

R implementation of the QHAdam optimizer proposed by Ma and Yarats(2019). We used the implementation available at https://github.com/jettify/pytorch-optimizer/blob/master/torch_optimizer/qhadam.py. Thanks to Nikolay Novik for providing the pytorch code.

The original implementation has been developed by Facebook AI and is licensed using the MIT license.

From the the paper by Ma and Yarats(2019): QHAdam is a QH augmented version of Adam, where we replace both of Adam's moment estimators with quasi-hyperbolic terms. QHAdam decouples the momentum term from the current gradient when updating the weights, and decouples the mean squared gradients term from the current squared gradient when updating the weights.

Usage

```
optim_qhadam(
  params,
  lr = 0.01,
  betas = c(0.9, 0.999),
  eps = 0.001,
  nus = c(1, 1),
  weight_decay = 0,
  decouple_weight_decay = FALSE
)
```

Arguments

params	List of parameters to optimize.
lr	Learning rate (default: 1e-3)
betas	Coefficients computing running averages of gradient and its square (default: (0.9, 0.999))
eps	Term added to the denominator to improve numerical stability (default: 1e-8)
nus	Immediate discount factors used to estimate the gradient and its square (default: (1.0, 1.0))
weight_decay	Weight decay (L2 penalty) (default: 0)
decouple_weight_decay	Whether to decouple the weight decay from the gradient-based optimization step.

Value

A torch optimizer object implementing the `step` method.

Author(s)

Gilberto Camara, <gilberto.camara@inpe.br>

Daniel Falbel, <daniel.falble@gmail.com>

Rolf Simoes, <rolf.simoes@inpe.br>

Felipe Souza, <lipecaso@gmail.com>

Alber Sanchez, <alber.ipia@inpe.br>

References

Jerry Ma, Denis Yarats, "Quasi-hyperbolic momentum and Adam for deep learning". https://arxiv.org/abs/1810.06801

Examples

```
if (torch::torch_is_installed()) {
# function to demonstrate optimization
beale <- function(x, y) {
    log((1.5 - x + x * y)^2 + (2.25 - x - x * y^2)^2 + (2.625 - x + x * y^3)^2)
 }
# define optimizer
optim <- torchopt::optim_qhadam
# define hyperparams
opt_hparams <- list(lr = 0.01)

# starting point
x0 <- 3
y0 <- 3
# create tensor
x <- torch::torch_tensor(x0, requires_grad = TRUE)
y <- torch::torch_tensor(y0, requires_grad = TRUE)
# instantiate optimizer
optim <- do.call(optim, c(list(params = list(x, y)), opt_hparams))
# run optimizer
steps <- 400
x_steps <- numeric(steps)
y_steps <- numeric(steps)
for (i in seq_len(steps)) {
    x_steps[i] <- as.numeric(x)
    y_steps[i] <- as.numeric(y)
    optim$zero_grad()
    z <- beale(x, y)
    z$backward()
    optim$step()
}
print(paste0("starting value = ", beale(x0, y0)))
print(paste0("final value = ", beale(x_steps[steps], y_steps[steps])))
}
```

optim_radam *AdamW optimizer*

Description

R implementation of the RAdam optimizer proposed by Liu et al. (2019). We used the implementation in PyTorch as a basis for our implementation.

From the abstract by the paper by Liu et al. (2019): The learning rate warmup heuristic achieves remarkable success in stabilizing training, accelerating convergence and improving generalization for adaptive stochastic optimization algorithms like RMSprop and Adam. Here, we study its mechanism in details. Pursuing the theory behind warmup, we identify a problem of the adaptive learning rate (i.e., it has problematically large variance in the early stage), suggest warmup works as a variance reduction technique, and provide both empirical and theoretical evidence to verify our hypothesis. We further propose RAdam, a new variant of Adam, by introducing a term to rectify the variance of the adaptive learning rate. Extensive experimental results on image classification, language modeling, and neural machine translation verify our intuition and demonstrate the effectiveness and robustness of our proposed method.

Usage

```
optim_radam(
  params,
  lr = 0.01,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  weight_decay = 0
)
```

Arguments

`params`	List of parameters to optimize.
`lr`	Learning rate (default: 1e-3)
`betas`	Coefficients computing running averages of gradient and its square (default: (0.9, 0.999))
`eps`	Term added to the denominator to improve numerical stability (default: 1e-8)
`weight_decay`	Weight decay (L2 penalty) (default: 0)

Value

A torch optimizer object implementing the `step` method.

Author(s)

Gilberto Camara, <gilberto.camara@inpe.br>

Daniel Falbel, <daniel.falble@gmail.com>

Rolf Simoes, <rolf.simoes@inpe.br>

Felipe Souza, <lipecaso@gmail.com>

Alber Sanchez, <alber.ipia@inpe.br>

References

Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, Jiawei Han, "On the Variance of the Adaptive Learning Rate and Beyond", International Conference on Learning Representations (ICLR) 2020. https://arxiv.org/abs/1908.03265

Examples

```
if (torch::torch_is_installed()) {
# function to demonstrate optimization
beale <- function(x, y) {
    log((1.5 - x + x * y)^2 + (2.25 - x - x * y^2)^2 + (2.625 - x + x * y^3)^2)
 }
# define optimizer
optim <- torchopt::optim_radam
# define hyperparams
opt_hparams <- list(lr = 0.01)

# starting point
x0 <- 3
y0 <- 3
# create tensor
x <- torch::torch_tensor(x0, requires_grad = TRUE)
y <- torch::torch_tensor(y0, requires_grad = TRUE)
# instantiate optimizer
optim <- do.call(optim, c(list(params = list(x, y)), opt_hparams))
# run optimizer
steps <- 400
x_steps <- numeric(steps)
y_steps <- numeric(steps)
for (i in seq_len(steps)) {
    x_steps[i] <- as.numeric(x)
    y_steps[i] <- as.numeric(y)
    optim$zero_grad()
    z <- beale(x, y)
    z$backward()
    optim$step()
}
print(paste0("starting value = ", beale(x0, y0)))
print(paste0("final value = ", beale(x_steps[steps], y_steps[steps])))
}
```

---

optim_swats *SWATS optimizer*

Description

R implementation of the SWATS optimizer proposed by Shekar and Sochee (2018). We used the implementation available at https://github.com/jettify/pytorch-optimizer/ Thanks to Nikolay Novik for providing the pytorch code.

From the abstract by the paper by Shekar and Sochee (2018): Adaptive optimization methods such as Adam, Adagrad or RMSprop have been found to generalize poorly compared to Stochastic gradient descent (SGD). These methods tend to perform well i in the initial portion of training but are outperformed by SGD at later stages of training. We investigate a hybrid strategy that begins training with an adaptive method and switches to SGD when a triggering condition is satisfied. The condition we propose relates to the projection of Adam steps on the gradient subspace. By design, the monitoring process for this condition adds very little overhead and does not increase the number of hyperparameters in the optimizer.

Usage

```
optim_swats(
  params,
  lr = 0.01,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  weight_decay = 0,
  nesterov = FALSE
)
```

Arguments

`params`	List of parameters to optimize.
`lr`	Learning rate (default: 1e-3)
`betas`	Coefficients computing running averages of gradient and its square (default: (0.9, 0.999)).
`eps`	Term added to the denominator to improve numerical stability (default: 1e-8).
`weight_decay`	Weight decay (L2 penalty) (default: 0).
`nesterov`	Enables Nesterov momentum (default: False).

Value

A torch optimizer object implementing the `step` method.

Author(s)

Gilberto Camara, <`gilberto.camara@inpe.br`>

Daniel Falbel, <`daniel.falble@gmail.com`>

Rolf Simoes, <`rolf.simoes@inpe.br`>

Felipe Souza, <`lipecaso@gmail.com`>

Alber Sanchez, <`alber.ipia@inpe.br`>

References

Nitish Shirish Keskar, Richard Socher "Improving Generalization Performance by Switching from Adam to SGD". International Conference on Learning Representations (ICLR) 2018. https://arxiv.org/abs/1712.07628

Examples

```
if (torch::torch_is_installed()) {
# function to demonstrate optimization
beale <- function(x, y) {
    log((1.5 - x + x * y)^2 + (2.25 - x - x * y^2)^2 + (2.625 - x + x * y^3)^2)
```

```
 }
# define optimizer
optim <- torchopt::optim_swats
# define hyperparams
opt_hparams <- list(lr = 0.01)

# starting point
x0 <- 3
y0 <- 3
# create tensor
x <- torch::torch_tensor(x0, requires_grad = TRUE)
y <- torch::torch_tensor(y0, requires_grad = TRUE)
# instantiate optimizer
optim <- do.call(optim, c(list(params = list(x, y)), opt_hparams))
# run optimizer
steps <- 400
x_steps <- numeric(steps)
y_steps <- numeric(steps)
for (i in seq_len(steps)) {
    x_steps[i] <- as.numeric(x)
    y_steps[i] <- as.numeric(y)
    optim$zero_grad()
    z <- beale(x, y)
    z$backward()
    optim$step()
}
print(paste0(”starting value = ”, beale(x0, y0)))
print(paste0(”final value = ”, beale(x_steps[steps], y_steps[steps])))
}
```

optim_yogi *Yogi optimizer*

Description

R implementation of the Yogi optimizer proposed by Zaheer et al.(2019). We used the implementation available at https://github.com/jettify/pytorch-optimizer/blob/master/torch_optimizer/yogi.py. Thanks to Nikolay Novik for providing the pytorch code.

The original implementation is licensed using the Apache-2.0 software license. This implementation is also licensed using Apache-2.0 license.

From the abstract by the paper by Zaheer et al.(2019): Adaptive gradient methods that rely on scaling gradients down by the square root of exponential moving averages of past squared gradients, such RMSProp, Adam, Adadelta have found wide application in optimizing the nonconvex problems that arise in deep learning. However, it has been recently demonstrated that such methods can fail to converge even in simple convex optimization settings. Yogi is a new adaptive optimization algorithm, which controls the increase in effective learning rate, leading to even better performance with similar theoretical guarantees on convergence. Extensive experiments show that Yogi with very little hyperparameter tuning outperforms methods such as Adam in several challenging machine learning tasks.

Usage

```
optim_yogi(
  params,
  lr = 0.01,
  betas = c(0.9, 0.999),
  eps = 0.001,
  initial_accumulator = 1e-06,
  weight_decay = 0
)
```

Arguments

`params`	List of parameters to optimize.
`lr`	Learning rate (default: 1e-3)
`betas`	Coefficients computing running averages of gradient and its square (default: (0.9, 0.999))
`eps`	Term added to the denominator to improve numerical stability (default: 1e-8)
`initial_accumulator`	Initial values for first and second moments.
`weight_decay`	Weight decay (L2 penalty) (default: 0)

Value

A torch optimizer object implementing the `step` method.

Author(s)

Gilberto Camara, <gilberto.camara@inpe.br>

Rolf Simoes, <rolf.simoes@inpe.br>

Felipe Souza, <lipecaso@gmail.com>

Alber Sanchez, <alber.ipia@inpe.br>

References

Manzil Zaheer, Sashank Reddi, Devendra Sachan, Satyen Kale, Sanjiv Kumar, "Adaptive Methods for Nonconvex Optimization", Advances in Neural Information Processing Systems 31 (NeurIPS 2018). https://papers.nips.cc/paper/8186-adaptive-methods-for-nonconvex-optimization

Examples

```
if (torch::torch_is_installed()) {
# function to demonstrate optimization
beale <- function(x, y) {
    log((1.5 - x + x * y)^2 + (2.25 - x - x * y^2)^2 + (2.625 - x + x * y^3)^2)
 }
# define optimizer
optim <- torchopt::optim_yogi
# define hyperparams
opt_hparams <- list(lr = 0.01)

# starting point
x0 <- 3
y0 <- 3
# create tensor
x <- torch::torch_tensor(x0, requires_grad = TRUE)
y <- torch::torch_tensor(y0, requires_grad = TRUE)
# instantiate optimizer
optim <- do.call(optim, c(list(params = list(x, y)), opt_hparams))
# run optimizer
steps <- 400
x_steps <- numeric(steps)
y_steps <- numeric(steps)
for (i in seq_len(steps)) {
    x_steps[i] <- as.numeric(x)
    y_steps[i] <- as.numeric(y)
    optim$zero_grad()
    z <- beale(x, y)
    z$backward()
    optim$step()
}
print(paste0("starting value = ", beale(x0, y0)))
print(paste0("final value = ", beale(x_steps[steps], y_steps[steps])))
}
```

test_optim *Test optimization function*

Description

test_optim() function is useful to visualize how optimizers solve the minimization problem by showing the convergence path using a test function. User can choose any test optimization functions provided by torchopt:

"beale", "booth", "bukin_n6", "easom", "goldstein_price", "himmelblau", "levi_n13", "matyas", "rastrigin", "rosenbrock", and "sphere".

Besides these functions, users can pass any function that receives two numerical values and returns a scalar.

Optimization functions are useful to evaluate characteristics of optimization algorithms, such as convergence rate, precision, robustness, and performance. These functions give an idea about the different situations that optimization algorithms can face.

Function test_function() plot the 2D-space of a test optimization function.

Usage

```
test_optim(
  optim,
  ...,
  opt_hparams = list(),
  test_fn = "beale",
  steps = 200,
  pt_start_color = "#5050FF7F",
  pt_end_color = "#FF5050FF",
  ln_color = "#FF0000FF",
  ln_weight = 2,
  bg_xy_breaks = 100,
  bg_z_breaks = 32,
  bg_palette = "viridis",
  ct_levels = 10,
  ct_labels = FALSE,
  ct_color = "#FFFFFF7F",
  plot_each_step = FALSE
)
```

Arguments

`optim`	Torch optimizer function.
`...`	Additional parameters (passed to `image` function).
`opt_hparams`	A list with optimizer initialization parameters (default: `list()`). If missing, for each optimizer its individual defaults will be used.
`test_fn`	A test function (default `"beale"`). You can also pass a list with 2 elements. The first should be a function that will be optimized and the second is a function that returns a named vector with `x0`, `y0` (the starting points) and `xmax`, `xmin`, `ymax` and `ymin` (the domain). An example: `c(x0 = x0, y0 = y0, xmax = 5, xmin = -5, ymax = 5, ymin = -5)`
`steps`	Number of steps to run (default `200`).
`pt_start_color`	Starting point color (default `"#5050FF7F"`)
`pt_end_color`	Ending point color (default `"#FF5050FF"`)
`ln_color`	Line path color (default `"#FF0000FF"`)
`ln_weight`	Line path weight (default `2`)
`bg_xy_breaks`	Background X and Y resolution (default `100`)
`bg_z_breaks`	Background Z resolution (default `32`)
`bg_palette`	Background palette (default `"viridis"`)
`ct_levels`	Contour levels (default `10`)
`ct_labels`	Should show contour labels? (default `FALSE`)
`ct_color`	Contour color (default `"#FFFFFF7F"`)
`plot_each_step`	Should output each step? (default `FALSE`)

Value

No return value, called for producing animated gifs

Author(s)

Rolf Simoes, <rolf.simoes@inpe.br>

Index