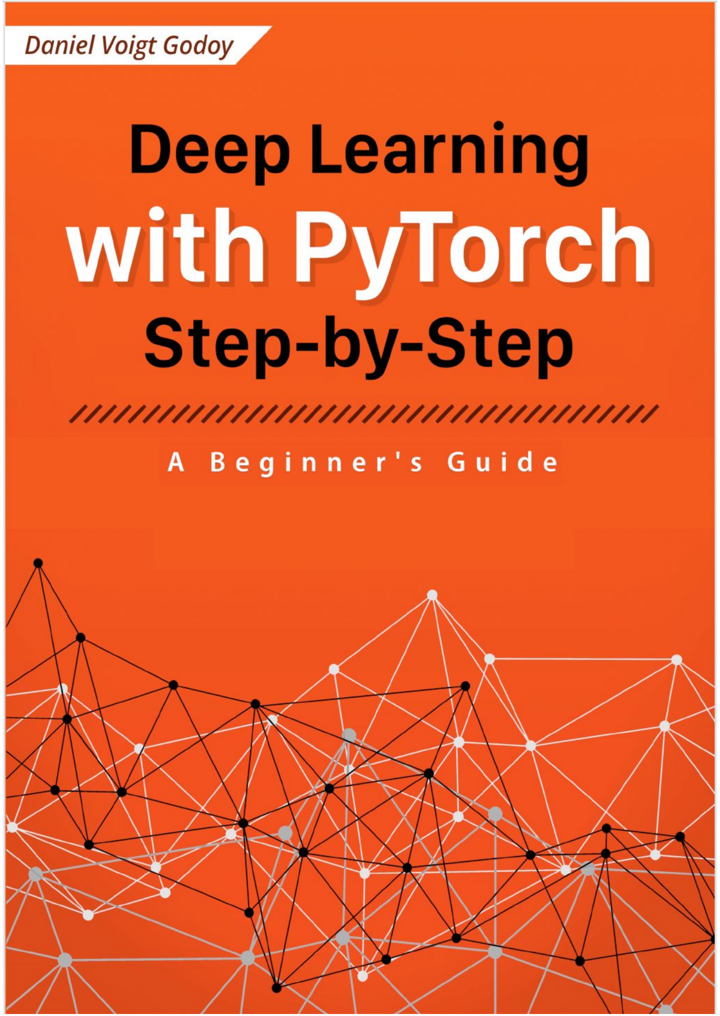
PPGEEC2318

Machine Learning

Rock, Paper, Scissors Cont.

Ivanovitch Silva
ivanovitch.silva@ufrn.br

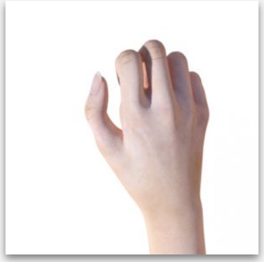


Chapter 6: Rock, Paper, Scissors

agenda

1. ***Standardize*** *an image dataset*
2. ***train*** *a model to predict* ***rock, paper, scissors*** *poses from hand images*
3. *use* ***dropout*** *layers to* ***regularize*** *the model*
4. *learn how to* ***find a learning rate*** *to train the model*
5. *understand how the* ***Adam optimizer*** *uses adaptive learning rates*
6. ***capture gradients*** *and* ***parameters*** *to visualize their evolution during training*
7. *understand how* ***momentum*** *and* ***Nesterov*** *momentum work*
8. *use* ***schedulers*** *to implement l****earning rate changes*** *during training*

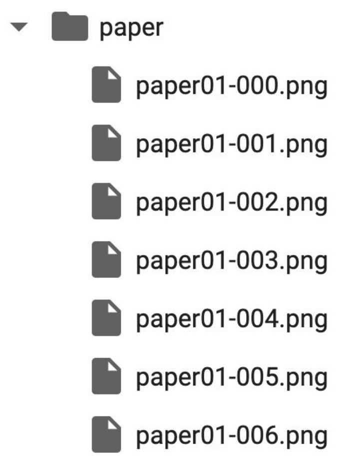
The dataset contains 2,892 images (2,520 train, 372 test) of diverse hands in the typical rock, paper, and scissors poses against a white background. This is a synthetic dataset as well since the images were generated using CGI techniques. Each image is 300x300 pixels in size and has four channels (RGBA).

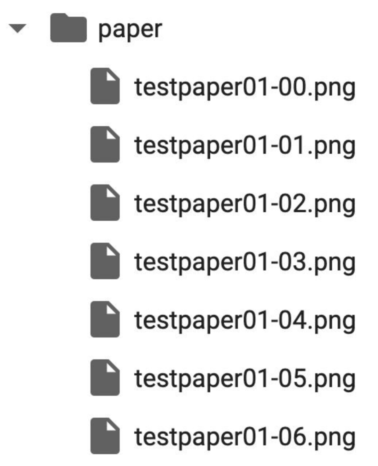


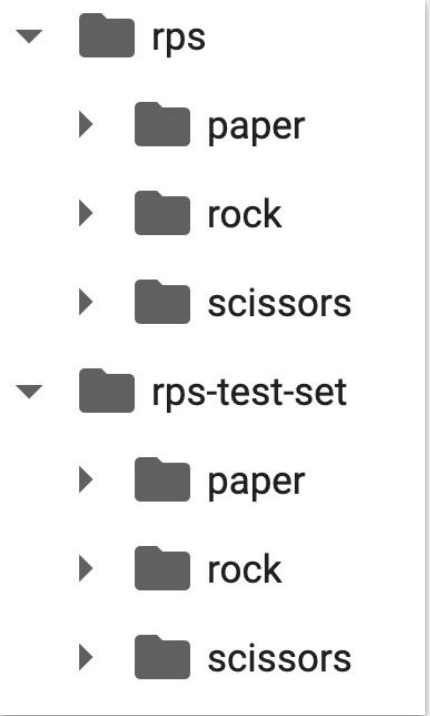
Rock

Paper

Scissors

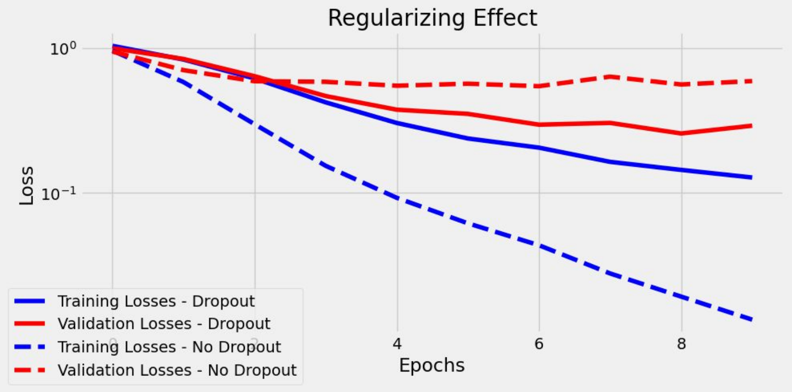


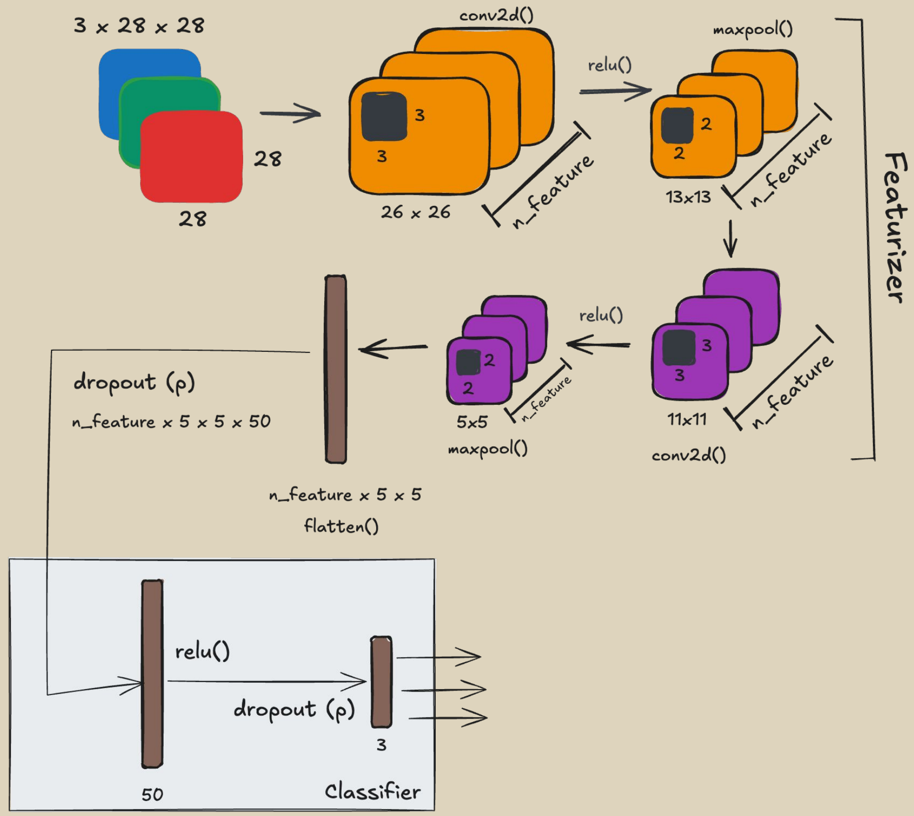


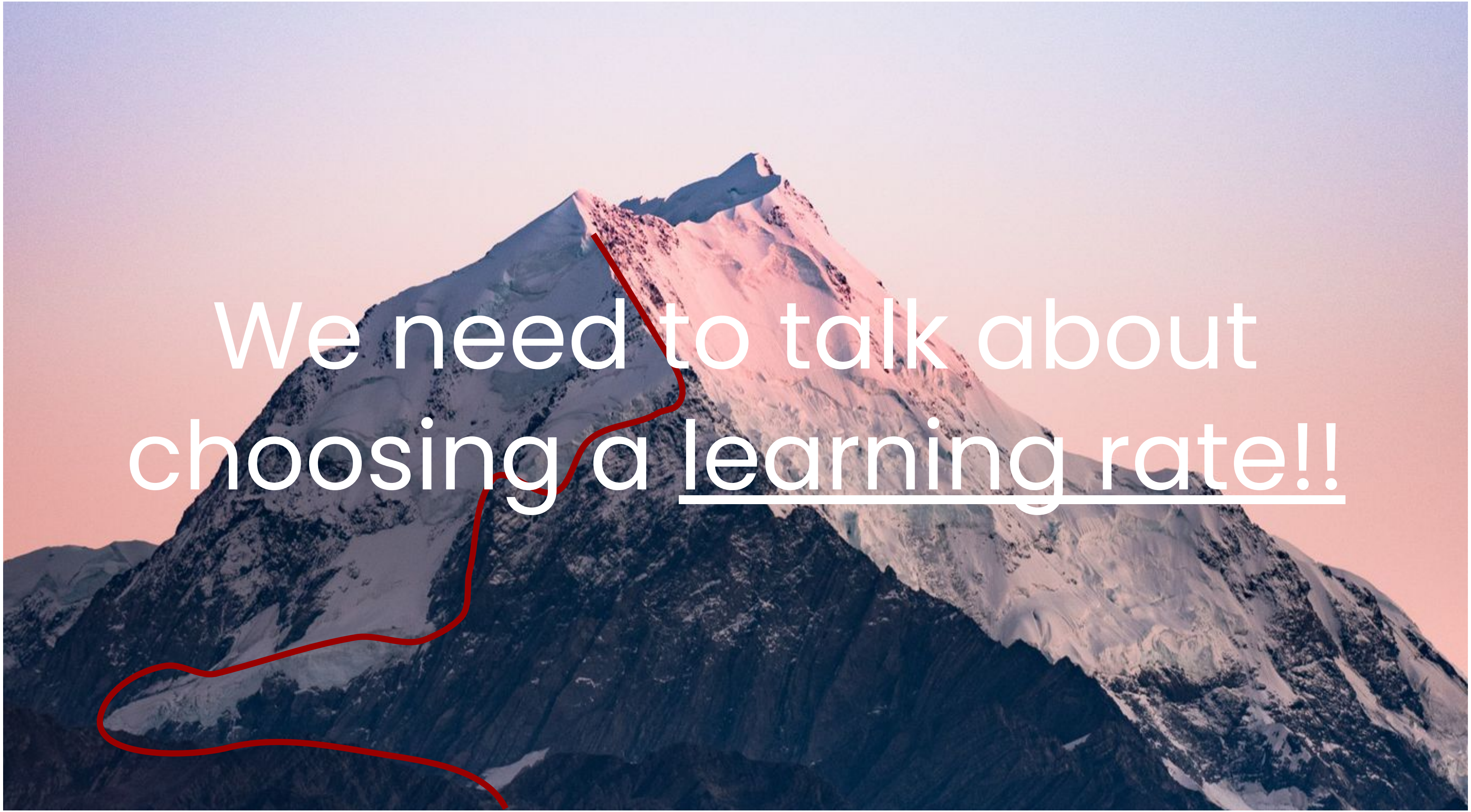


Fancier Model

```
torch.manual_seed(13)

# Model/Architecture
model_cnn2 = CNN2(n_feature=5, p=0.3)

# Loss function
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')

# Optimizer
optimizer_cnn2 = optim.Adam(model_cnn2.parameters(), lr=3e-4)
```





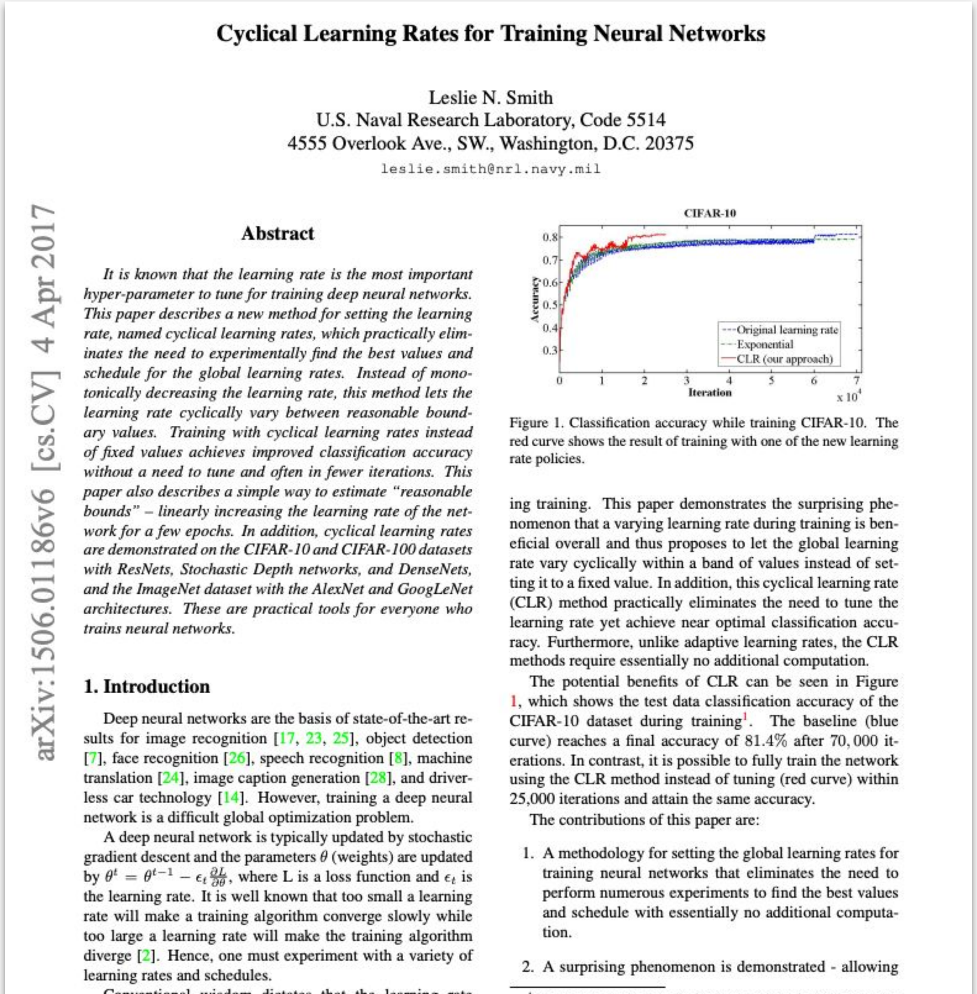

We need to talk about
choosing a learning rate!!

All you need is a grid-search?

Trying multiple learning rates over a few epochs each and the evolution of the losses

```
# it is common to reduce the LR by a factor
# of 3 or a factor of 10

# using factor of 3
[0.1, 0.03, 0.01, 3e-3, 1e-3, 3e-4, 1e-4]

# using a factor of 10
[0.1, 0.01, 1e-3, 1e-4, 1e-5]
```

1) If the learning rate is too low
   a) The model doesn't learn much and the loss remains high.
2) If the learning rate is too high
   a) The model doesn't converge to a solution and the loss gets higher.



Cyclical Learning Rates for Training Neural Networks

Leslie N. Smith
U.S. Naval Research Laboratory, Code 5514
4555 Overlook Ave., SW., Washington, D.C. 20375
leslie.smith@nrl.navy.mil

arXiv:1506.01186v6 [cs.CV] 4 Apr 2017

Abstract

It is known that the learning rate is the most important hyper-parameter to tune for training deep neural networks. This paper describes a new method for setting the learning rate, named cyclical learning rates, which practically eliminates the need to experimentally find the best values and schedule for the global learning rates. Instead of monotonically decreasing the learning rate, this method lets the learning rate cyclically vary between reasonable boundary values. Training with cyclical learning rates instead of fixed values achieves improved classification accuracy without a need to tune and often in fewer iterations. This paper also describes a simple way to estimate "reasonable bounds" – linearly increasing the learning rate of the network for a few epochs. In addition, cyclical learning rates are demonstrated on the CIFAR-10 and CIFAR-100 datasets with ResNets, Stochastic Depth networks, and DenseNets, and the ImageNet dataset with the AlexNet and GoogLeNet architectures. These are practical tools for everyone who trains neural networks.

1. Introduction

Deep neural networks are the basis of state-of-the-art results for image recognition [17, 23, 25], object detection [7], face recognition [26], speech recognition [8], machine translation [24], image caption generation [28], and driverless car technology [14]. However, training a deep neural network is a difficult global optimization problem.

A deep neural network is typically updated by stochastic gradient descent and the parameters θ (weights) are updated by $\theta^t = \theta^{t-1} - \epsilon_t \frac{\partial L}{\partial \theta}$, where L is a loss function and ϵ_t is the learning rate. It is well known that too small a learning rate will make a training algorithm converge slowly while too large a learning rate will make the training algorithm diverge [2]. Hence, one must experiment with a variety of learning rates and schedules.

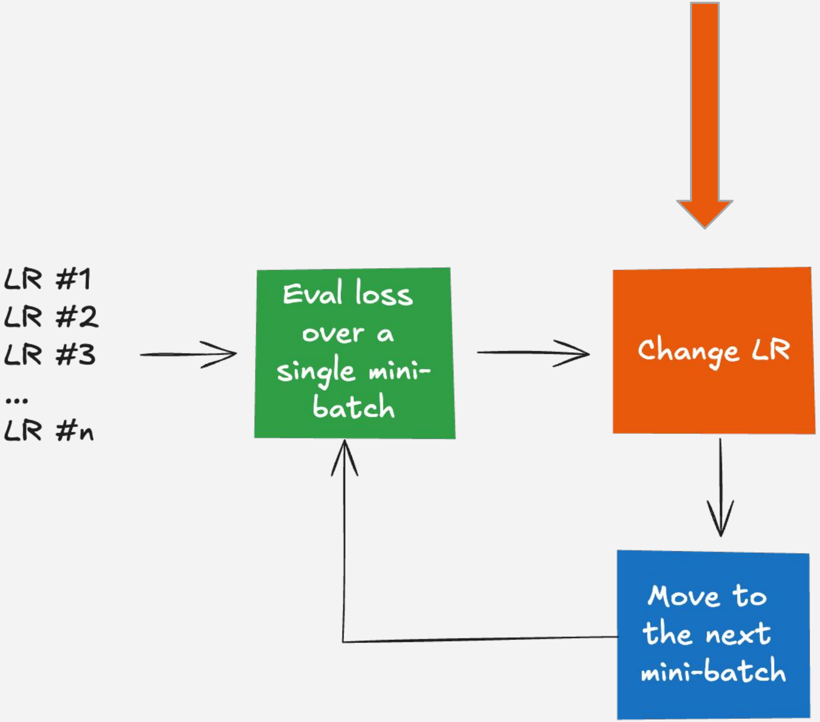
Figure 1. Classification accuracy while training CIFAR-10. The red curve shows the result of training with one of the new learning rate policies.

ing training. This paper demonstrates the surprising phenomenon that a varying learning rate during training is beneficial overall and thus proposes to let the global learning rate vary cyclically within a band of values instead of setting it to a fixed value. In addition, this cyclical learning rate (CLR) method practically eliminates the need to tune the learning rate yet achieve near optimal classification accuracy. Furthermore, unlike adaptive learning rates, the CLR methods require essentially no additional computation.

The potential benefits of CLR can be seen in Figure 1, which shows the test data classification accuracy of the CIFAR-10 dataset during training[1]. The baseline (blue curve) reaches a final accuracy of 81.4% after 70,000 iterations. In contrast, it is possible to fully train the network using the CLR method instead of tuning (red curve) within 25,000 iterations and attain the same accuracy.

The contributions of this paper are:

1. A methodology for setting the global learning rates for training neural networks that eliminates the need to perform numerous experiments to find the best values and schedule with essentially no additional computation.
2. A surprising phenomenon is demonstrated - allowing



LR Range Test

Higher-Order Learning Rate Function Builder

```
def make_lr_fn(start_lr, end_lr, num_iter, step_mode='exp'):
    if step_mode == 'linear':
        factor = (end_lr / start_lr - 1) / num_iter
        def lr_fn(iteration):
            return 1 + iteration * factor
    else:
        factor = (np.log(end_lr) - np.log(start_lr)) / num_iter
        def lr_fn(iteration):
            return np.exp(factor)**iteration
    return lr_fn
```

```
start_lr = 0.01
end_lr = 0.1
num_iter = 10
lr_fn = make_lr_fn(start_lr, end_lr,
                   num_iter, step_mode='exp')

lr_fn(np.arange(num_iter + 1))
array([ 1.        ,  1.25892541,  1.58489319,  1.99526231,
 2.51188643,3.16227766,  3.98107171,  5.01187234,
 6.30957344,  7.94328235,10.])

start_lr * lr_fn(np.arange(num_iter + 1))
array([0.01      , 0.01258925, 0.01584893, 0.01995262,
0.02511886, 0.03162278, 0.03981072, 0.05011872, 0.06309573,
0.07943282, 0.1])
```

```
start_lr = 0.01
end_lr = 0.1
num_iter = 10
lr_fn = make_lr_fn(start_lr, end_lr,
                   num_iter, step_mode='linear')

lr_fn(np.arange(num_iter + 1))
array([ 1. ,  1.9,  2.8,  3.7,  4.6,  5.5,  6.4,  7.3,  8.2,  9.1, 10.
])

start_lr * lr_fn(np.arange(num_iter + 1))
array([0.01 , 0.019, 0.028, 0.037, 0.046, 0.055, 0.064, 0.073, 0.082,
       0.091, 0.1  ])
```

How do I change the learning rate of an optimizer?

```
start_lr = 0.01
end_lr = 0.1
num_iter = 10
lr_fn = make_lr_fn(start_lr, end_lr,
                   num_iter, step_mode='exp')

lr_fn(np.arange(num_iter + 1))
array([ 1.        ,  1.25892541,  1.58489319,  1.99526231,
 2.51188643,3.16227766,  3.98107171,  5.01187234,
 6.30957344,  7.94328235,10.])

start_lr * lr_fn(np.arange(num_iter + 1))
array([0.01      , 0.01258925, 0.01584893, 0.01995262,
0.02511886, 0.03162278, 0.03981072, 0.05011872, 0.06309573,
0.07943282, 0.1])
```

```
dummy_model = CNN2(n_feature=5, p=0.3)
dummy_optimizer = optim.Adam(dummy_model.parameters(), lr=start_lr)
```

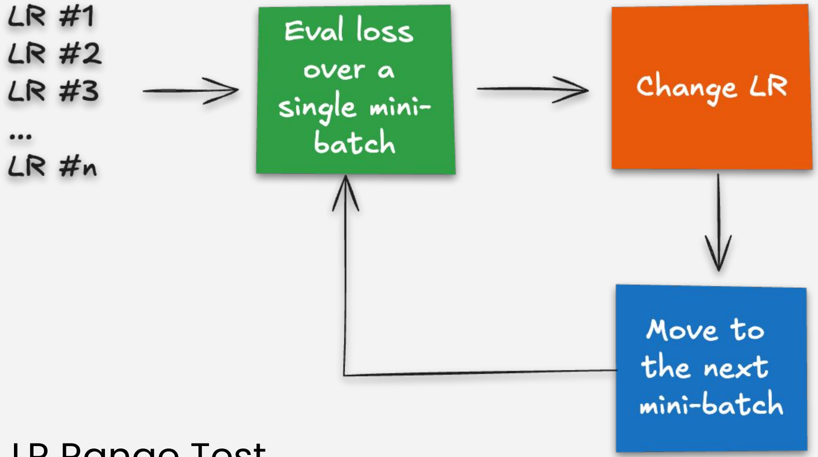
All you need is a scheduler!!!

The LambdaLR scheduler takes an **optimizer** and a **custom function** as arguments and modifies the learning rate of that optimizer accordingly

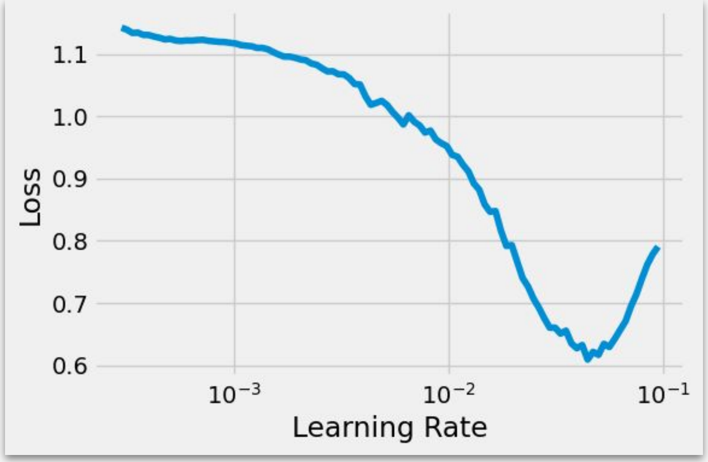
```
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau, MultiStepLR, CyclicLR, LambdaLR

dummy_model = CNN2(n_feature=5, p=0.3)
dummy_optimizer = optim.Adam(dummy_model.parameters(), lr=start_lr)
dummy_scheduler = LambdaLR(dummy_optimizer, lr_lambda=lr_fn)
```

```
dummy_optimizer.step()
dummy_scheduler.step()

dummy_scheduler.get_last_lr()[0]
0.012589254117941673
```

```
start_lr * lr_fn(np.arange(num_iter + 1))
array([0.01      , 0.01258925, 0.01584893, 0.01995262, 0.02511886,
       0.03162278, 0.03981072, 0.05011872, 0.06309573, 0.07943282,
       0.1       ])
```



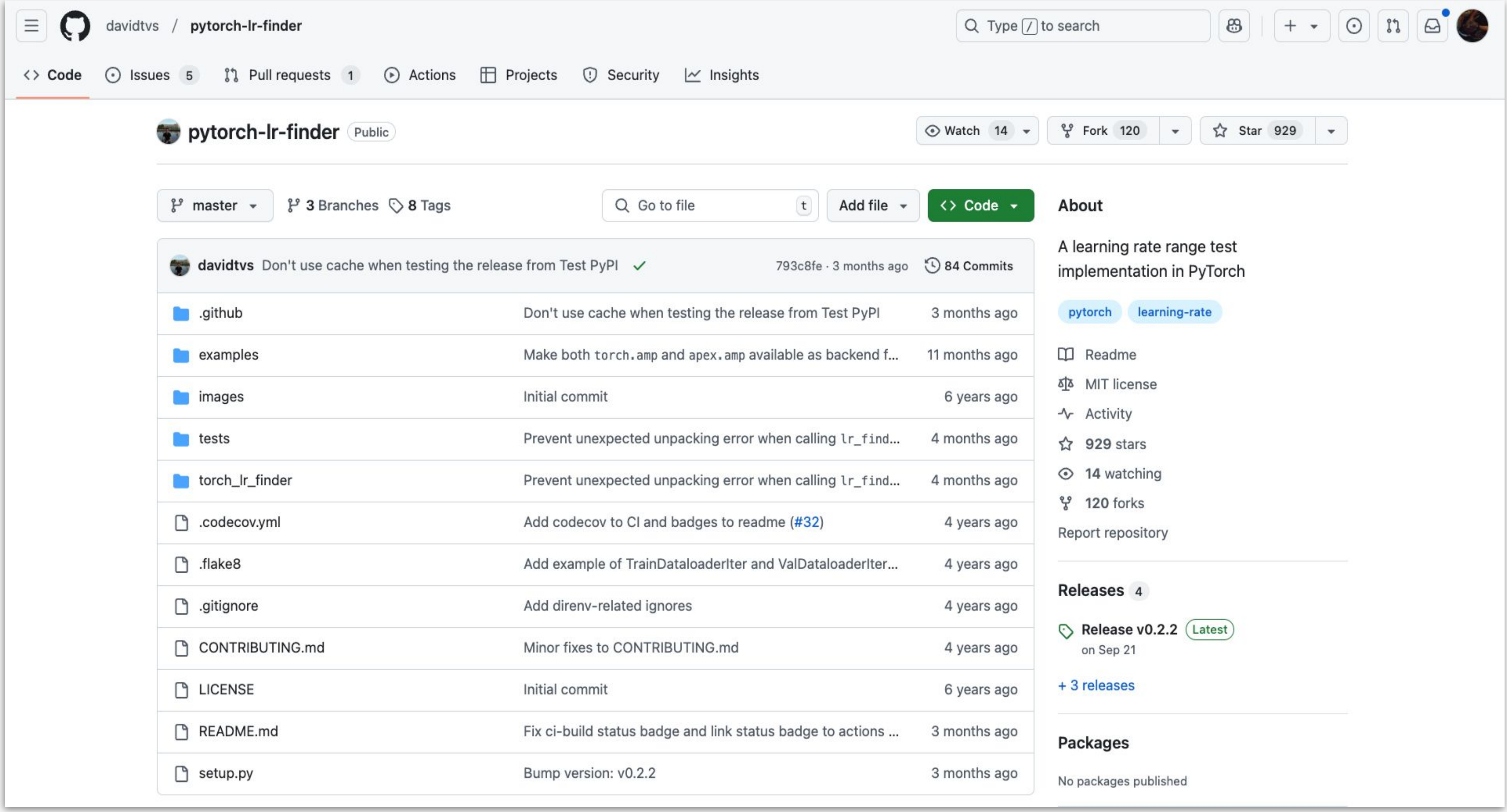
LR Range Test



(0.04434013138652989,0.6091022460474141)

```python
# model
new_model = CNN2(n_feature=5, p=0.3)

# loss function
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')

# optimizer
new_optimizer = optim.Adam(new_model.parameters(), lr=3e-4)

# architecture
arch_new = Architecture(new_model, multi_loss_fn, new_optimizer)

# lr range test
tracking, fig = arch_new.lr_range_test(train_loader, end_lr=1e-1, num_iter=100)
```

"U-Shape curve"



davidtvs / pytorch-lr-finder

Type / to search

Code | Issues 5 | Pull requests 1 | Actions | Projects | Security | Insights

pytorch-lr-finder Public

Watch 14 | Fork 120 | Star 929

master | 3 Branches | 8 Tags | Go to file t | Add file | Code

davidtvs Don't use cache when testing the release from Test PyPI ✓ 793c8fe · 3 months ago 84 Commits

.github	Don't use cache when testing the release from Test PyPI	3 months ago
examples	Make both `torch.amp` and `apex.amp` available as backend f...	11 months ago
images	Initial commit	6 years ago
tests	Prevent unexpected unpacking error when calling `lr_find...`	4 months ago
torch_lr_finder	Prevent unexpected unpacking error when calling `lr_find...`	4 months ago
.codecov.yml	Add codecov to CI and badges to readme (#32)	4 years ago
.flake8	Add example of TrainDataloaderIter and ValDataloaderIter...	4 years ago
.gitignore	Add direnv-related ignores	4 years ago
CONTRIBUTING.md	Minor fixes to CONTRIBUTING.md	4 years ago
LICENSE	Initial commit	6 years ago
README.md	Fix ci-build status badge and link status badge to actions ...	3 months ago
setup.py	Bump version: v0.2.2	3 months ago

About

A learning rate range test implementation in PyTorch

pytorch learning-rate

Readme
MIT license
Activity
929 stars
14 watching
120 forks
Report repository

Releases 4

Release v0.2.2 Latest
on Sep 21

+ 3 releases

Packages

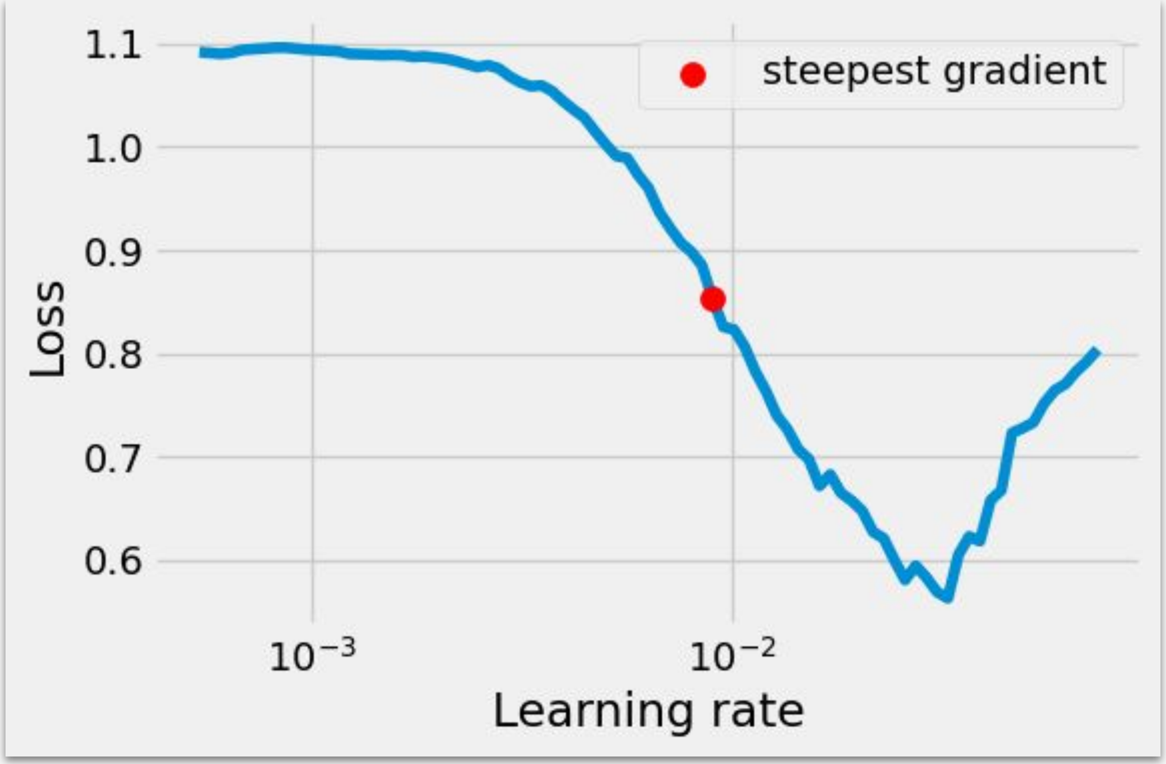
No packages published

```
!pip install --quiet torch-lr-finder
from torch_lr_finder import LRFinder
```

```
fig, ax = plt.subplots(1, 1, figsize=(6, 4))

# model, loss function, optimizer and device
new_model = CNN2(n_feature=5, p=0.3)
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
new_optimizer = optim.Adam(new_model.parameters(), lr=3e-4)
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# instantiate LR Finder object
lr_finder = LRFinder(new_model, new_optimizer, multi_loss_fn, device=device)

# LR range test
lr_finder.range_test(train_loader, end_lr=1e-1, num_iter=100)

# plot
lr_finder.plot(ax=ax, log_lr=True)
fig.tight_layout()

# reset model
lr_finder.reset()
```

The concept of the "steepest gradient" refers to the point where the loss decreases most rapidly.

The idea is to find the highest learning rate that still results in a decrease in loss before the loss starts to increase due to an excessively high learning rate. This zone is typically at the steepest part of the loss curve.



LR suggestion: **steepest gradient**
Suggested LR: **9.02E-03**

"OK, if I manage to choose a good learning rate from the start, am I done with it?"

```
optimizer = optim.Adam(model_cnn2.parameters(),
                       lr=0.0125,
                       betas=(0.9, 0.999), eps=1e-8)
optimizer.step()
```

$$\mathbf{W} = \mathbf{W} - \eta \times \nabla(\mathbf{W})$$
$$\mathbf{b} = \mathbf{b} - \eta \times \nabla(\mathbf{b})$$


PETROLEO BRASILEIRO SA PETROBR · 1M · NYSE
O14.41 H14.46 L13.51 C13.59 −0.68 (−4.77%)
13.59 0.00 13.59
SMA · 1M 7 close 15.23
SMA · 1M 60 close
TradingView
2009
2011
2013
2015
2017
2019
2021
2023
2025
Números da Covid-19 - Gravatá-PE
Dados epidemiológicos da COVID-19 e de distanciamento social
Casos Confirmados por dia
Média Móvel de 14 dias
(a)
Óbitos confirmados por dia
Média Móvel de 14 dias
(b)
Número de Reprodução R(t)
(c)
Distanciamento Social - Média Móvel de 14 dias
Gravatá
Pernambuco
(d)
Maio
Junho
Julho
Agosto
2020
Dados distribuídos entre 16 de abril e 03 de agosto
Source: http://www.prefeituradegravata.pe.gov.br/
Indicador de Distanciamento Social foi desenvolvido pela In Loco

Moving Average (MA)

$$MA_t = \frac{1}{N}\sum_{i=0}^{N-1} x_{t-i}$$

A Moving Average (MA) is the average of the most recent values over a fixed period N.

```python
def moving_average(data, N):
    return [sum(data[i:i+N])/N for i in range(len(data)-N+1)]

data = [2, 4, 6, 8, 10, 12, 14, 16]
N = 3

ma_result = moving_average(data, N)
print(ma_result)

[4.0, 6.0, 8.0, 10.0, 12.0, 14.0]
```

Average Age for MA

$$x = [2, 4, 6, 8, 10, 12, 14, 16]$$

$$MA_3 = \frac{1}{3}(2 + 4 + 6) = 4$$

age (1)

age (2)

age (3)

$$\text{average age}_{\text{MA}} = \frac{1 + 2 + 3 + \cdots + N}{N}$$

$$= \frac{\frac{N(N+1)}{2}}{N} = \frac{N+1}{2}$$

$$\text{average age}_{\text{MA}} = \frac{1 + 2 + 3}{3} = 2$$

For a period of 3, it means that, on average, the data points contributing to the MA value are 2 time steps old.

Exponential Weighted Moving Average (EWMA)

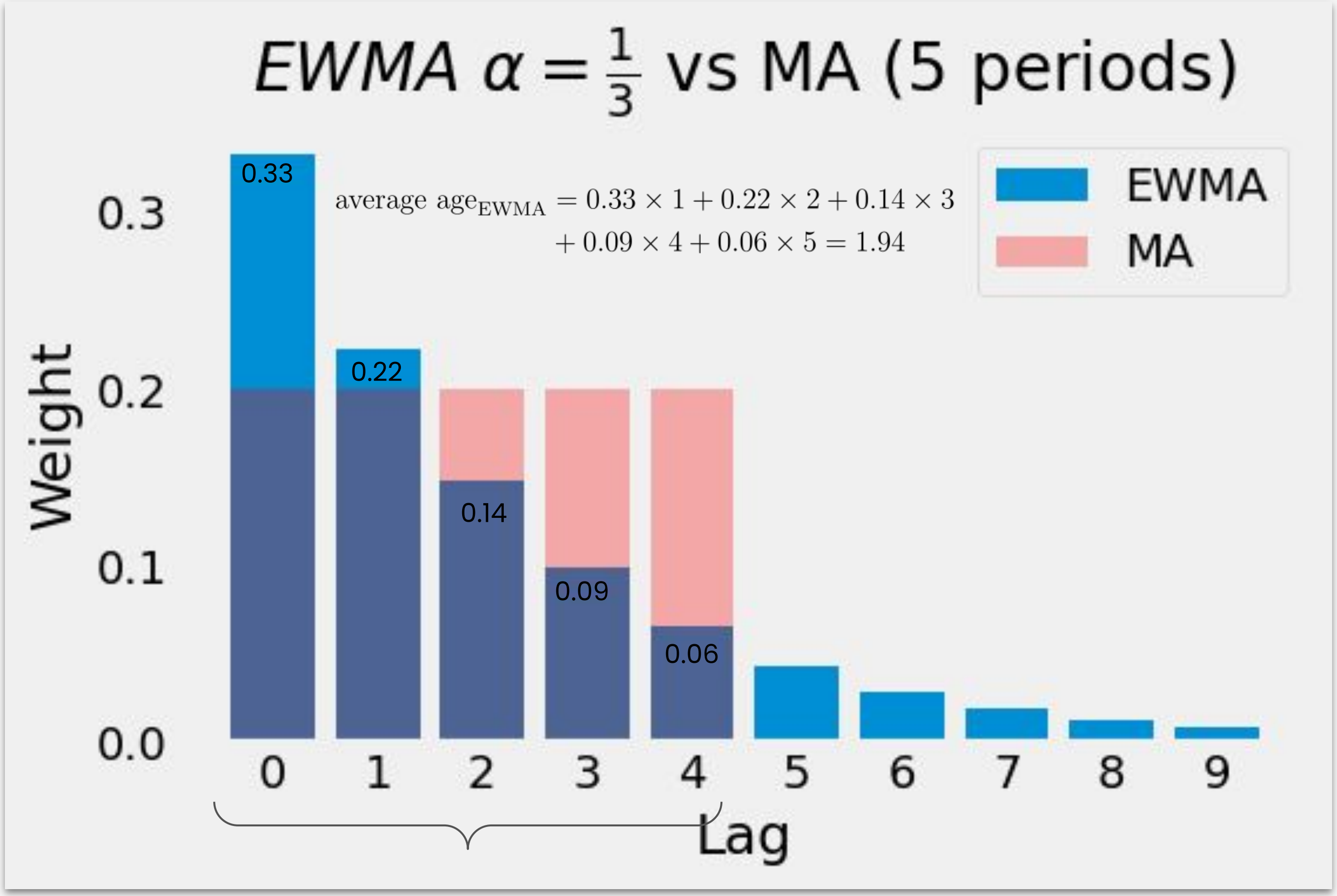
$$
\begin{aligned}
\mathrm{EWMA}_t(\alpha, x) =& \alpha \quad x_t + \quad (1-\alpha)\mathrm{EWMA}_{t-1}(\alpha, x) \\
\mathrm{EWMA}_t(\beta, x) =& (1-\beta) \quad x_t + \quad \beta\mathrm{EWMA}_{t-1}(\beta, x)
\end{aligned}
$$

The first alternative, using alpha as the weight of the current value, is most common in other fields, like finance. But, the beta alternative is the one commonly found when the Adam optimizer is discussed.

Exponential Weighted Moving Average (EWMA)

$$
\begin{aligned}
\mathrm{EWMA}_t(\alpha, x) &= \alpha \quad x_t + \quad (1-\alpha)\mathrm{EWMA}_{t-1}(\alpha, x) \\
&= \alpha x_t \quad +(1-\alpha)\ (\alpha x_{t-1} \quad +(1-\alpha)\ \ \mathrm{EWMA}_{t-2}(\alpha, x)) \\
&= \alpha x_t \quad +(1-\alpha)\ \ \alpha x_{t-1} \quad +(1-\alpha)^2 \alpha x_{t-2} + \ldots \\
&= (1-\alpha)^0 \alpha x_{t-0} \quad +(1-\alpha)^1\ \alpha x_{t-1} \quad +(1-\alpha)^2 \alpha x_{t-2} + \ldots \\
&= \alpha \quad ((1-\alpha)^0\ \ x_{t-0} \quad +(1-\alpha)^1 \quad x_{t-1} \quad +(1-\alpha)^2\ \ x_{t-2} + \ldots)
\end{aligned}
$$

$$
\mathrm{EWMA}_t = \alpha \sum_{\mathrm{lag}=0}^{N-1} \underbrace{(1-\alpha)^{\mathrm{lag}}}_{\text{weight}} x_{t-\mathrm{lag}}
$$



$$\text{average age}_{\text{MA}} = \frac{N+1}{2} = \frac{5+1}{2} = 3$$

Average Age for EWMA

As the total number of observed values (T) grows, the average age approaches the inverse of alpha.

$$\text{average age}_{\text{EWMA}} = \alpha \sum_{\text{lag}=0}^{T-1} (1-\alpha)^{\text{lag}} (\text{lag}+1) \approx \frac{1}{\alpha}$$

```
alpha = 1/3; T = 93
t = np.arange(1, T + 1)
age = alpha * sum((1 - alpha)**(t - 1) * t)
age
3.0
```

Note:

Now that we know how to compute the **average age** of an EWMA given its **alpha**, we can figure out which **moving average** has the same **average age**:

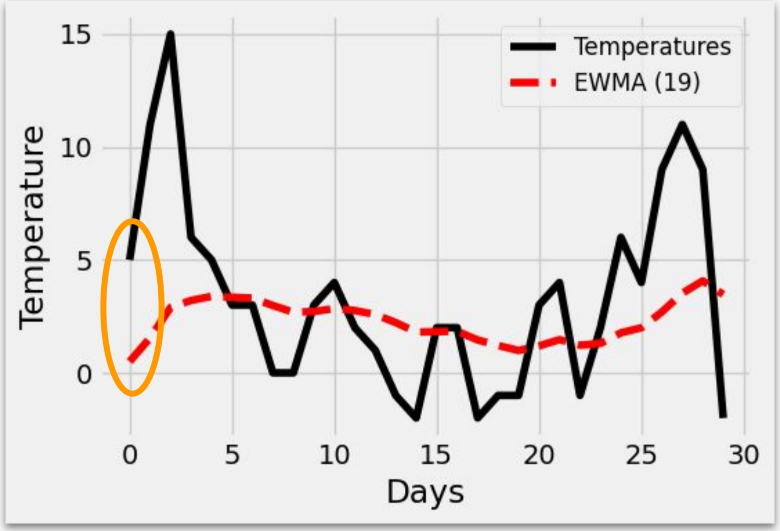
$$\text{average age} = \frac{\mathrm{N}+1}{2} = \frac{1}{\alpha} \implies \alpha = \frac{2}{\mathrm{N}+1};\ \mathrm{N} = \frac{2}{\alpha} - 1$$

α	β	N
1/3	2/3	5
0.1	0.9	19
0.001	0.999	1999

If we'd like to compute the EWMA equivalent to a 19-period moving average, the corresponding alpha would be 0.1.

```python
def EWMA(past_value, current_value, alpha):
    return (1 - alpha) * past_value + alpha * current_value


def calc_ewma(values, period):
    alpha = 2 / (period + 1)
    result = []
    for v in values:
        try:
            prev_value = result[-1]
        except IndexError:
            prev_value = 0

        new_value = EWMA(prev_value, v, alpha)
        result.append(new_value)
    return np.array(result)
```

```python
temperatures = np.array([5, 11, 15, 6, 5, 3, 3, 0, 0, 3, 4, 2, 1,
    -1, -2, 2, 2, -2, -1, -1, 3, 4, -1, 2, 6, 4, 9, 11, 9, -2])

calc_ewma(values, 19)
```



In the try..except block, you can see that, if there is no previous value for the EWMA (as in the very first step), it assumes a previous value of zero.

Note:

$$\mathrm{EWMA}_t(\alpha, x) = \alpha \sum_{\mathrm{lag}=0}^{N-1} (1-\alpha)^{\mathrm{lag}} x_{t-\mathrm{lag}}$$

For the average to be unbiased, the sum of the weights must be 1.

$$\alpha \sum_{\mathrm{lag}=0}^{N-1} (1-\alpha)^{\mathrm{lag}} = 1$$

At the beginning of a time series, when *t* is small, this sum is less than 1, resulting in a biased average!!!!

Note:

$$\mathrm{EWMA}_t(\alpha, x) = \alpha \sum_{\mathrm{lag}=0}^{N-1} (1-\alpha)^{\mathrm{lag}} x_{t-\mathrm{lag}}$$

Using the formula for the sum of a geometric progression:

$$\sum_{lag=0}^{N-1} (1-\alpha)^{lag} = \frac{1-(1-\alpha)^N}{\alpha}$$

Therefore, the sum of the weights is:

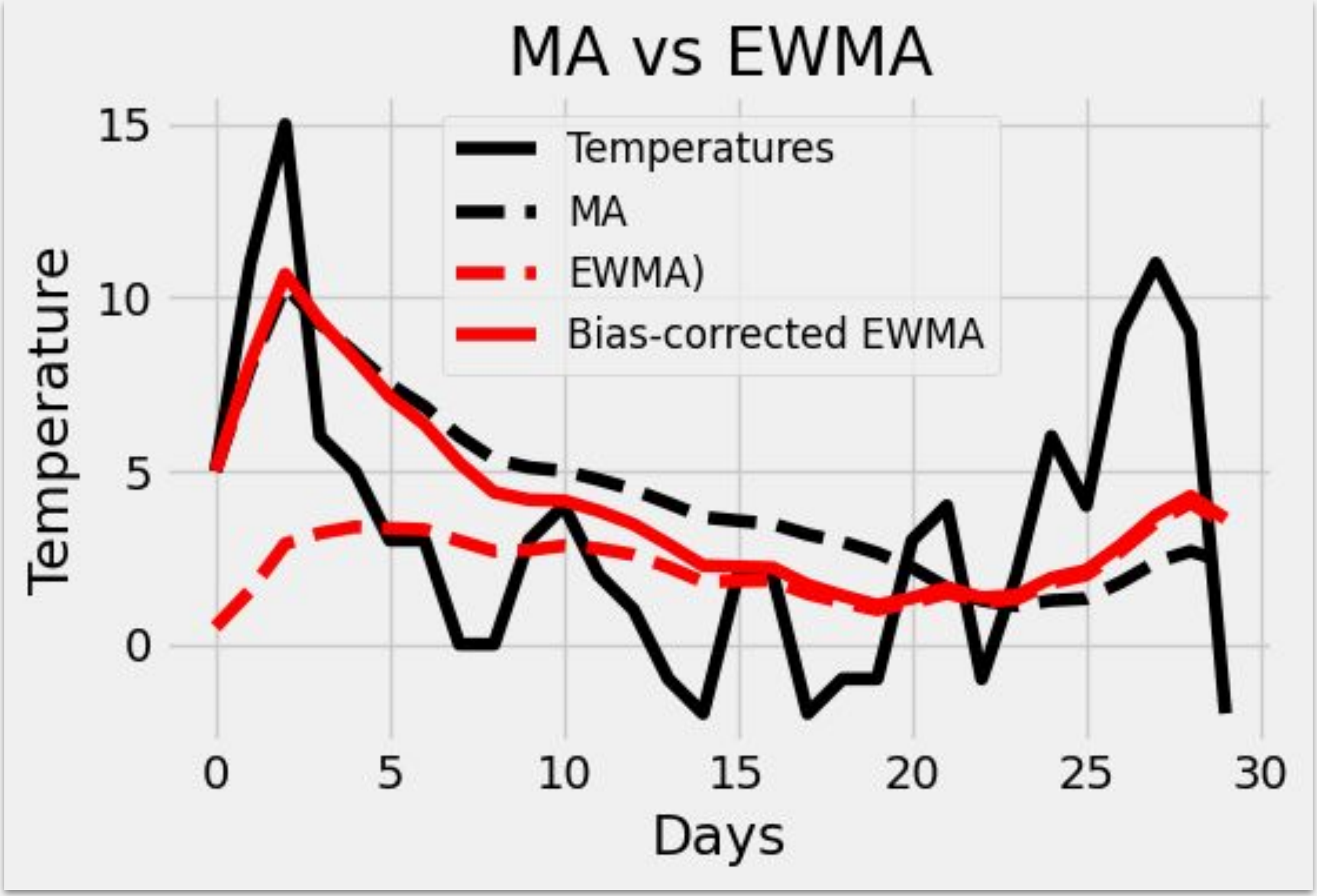
$$\alpha \sum_{\mathrm{lag}=0}^{N-1} (1-\alpha)^{\mathrm{lag}} = \alpha \cdot \frac{1-(1-\alpha)^N}{\alpha} = 1-(1-\alpha)^N$$

Bias Correction

To correct the bias, we divide the EWMA by the factor:

$$\text{Bias Corrected EWMA}_t = \frac{\text{EWMA}_t}{1-(1-\alpha)^t} = \frac{\text{EWMA}_t}{1-\beta^t}$$

```python
def correction(averaged_value, beta, steps):
    return averaged_value / (1 - (beta ** steps))


def calc_corrected_ewma(values, period):
    ewma = calc_ewma(values, period)

    alpha = 2 / (period + 1)
    beta = 1 - alpha

    result = []
    for step, v in enumerate(ewma):
        adj_value = correction(v, beta, step + 1)
        result.append(adj_value)

    return np.array(result)
```


MA vs EWMA
Temperatures
MA
EWMA)
Bias-corrected EWMA
Temperature
Days
15
10
5
0
0
5
10
15
20
25
30

EWMA Meets Gradients

α	β	N
1/3	2/3	5
0.1	0.9	19
0.001	0.999	1999

```
optimizer = optim.Adam(model_cnn2.parameters(),
                       lr=0.0125,
                       betas=(0.9, 0.999), eps=1e-8)
optimizer.step()
```

$$\text{adapted-gradient}_t = \frac{\text{Bias Corrected EWMA}_t(\beta_1, \text{gradients})}{\sqrt{\text{Bias Corrected EWMA}_t(\beta_2, \text{gradients}^2)} + \epsilon}$$

$$\text{SGD}: \text{param}_t = \text{param}_{t-1} - \eta \ \text{gradient}_t$$

$$\text{Adam}: \text{param}_t = \text{param}_{t-1} - \eta \ \text{adapted gradient}_t$$