S11

Due No Due Date **Points** None **Available** after Jun 7 at 12am

SUPER-CONVERGENCE

[Very Fast Training of Neural Networks using Large Learning Rates](#)

[Abstract]: In this paper, we describe a phenomenon, which we called "super-convergence", where **neural networks can be trained an order of magnitude faster** than with standard training methods. The existence of super-convergence is relevant to understanding why deep networks generalize well.

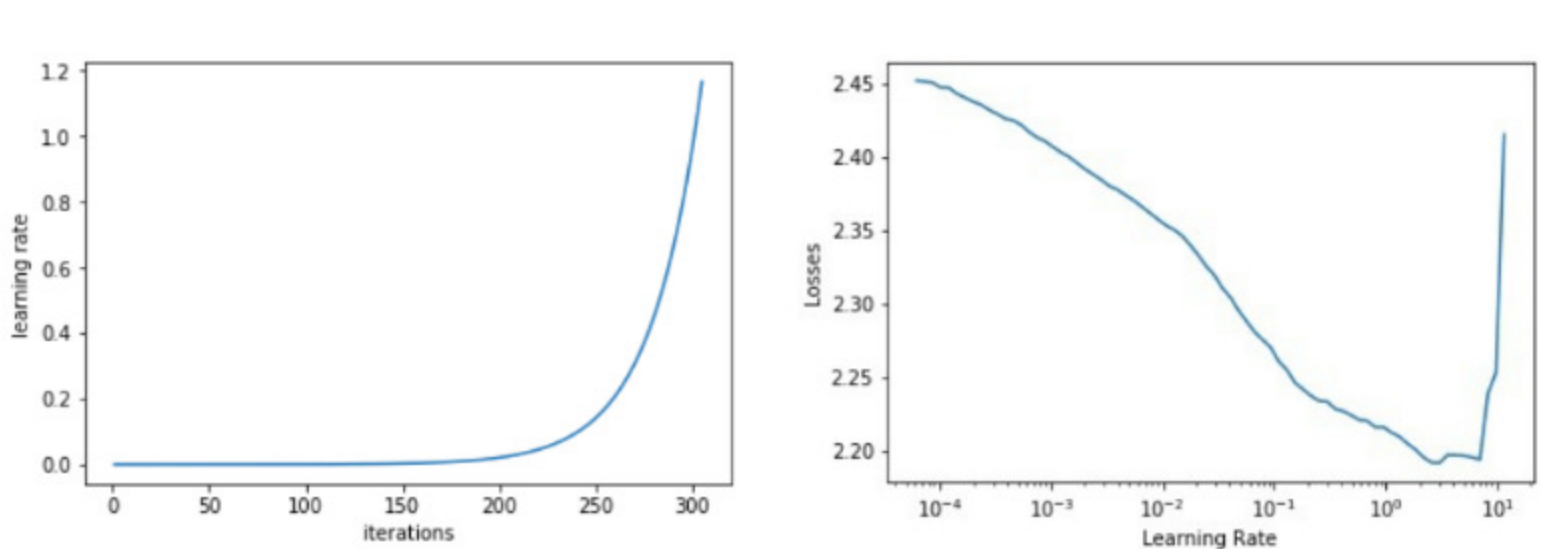
One of the key elements in super-convergence is training with one learning rate cycle and a large maximum learning rate.

A primary insight that allows super-convergence training is that large learning rates **regularize** the training, hence **requiring a reduction of all other forms of regularization** in order to preserve the optimal balance.

BACKGROUND

History and Evolution from Simple learning rates to OCP

In the last session, we worked on finding the right learning rate.



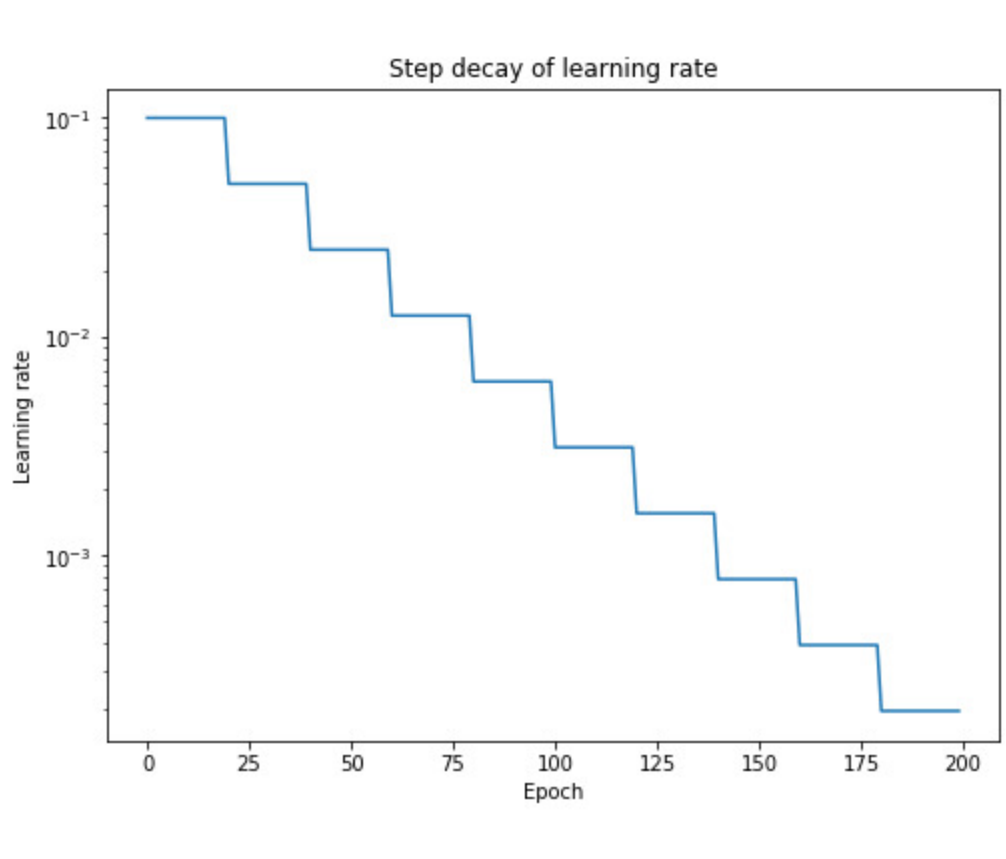
FIRST CHANGE - LEARNING RATE ANNEALING

Selecting a good starting learning rate is merely the first step. In order to efficiently train a robust model, we will need to gradually decrease the learning rate during training. If the learning rate remains unchanged during the course of training, it might be too large to converge and cause the loss function to fluctuate around the local minimum. The approach is to use a higher learning rate to quickly reach the regions of (local) minima during the initial training stage, and set a smaller learning rate as training progresses in order to explore **"deeper and more thoroughly"** in the region to find the minimum.

There is an array of methods for learning rate annealing: step-wise annealing, exponential decay, cosine annealing(strongly suggest by Jeremy Howard), etc. More details on an annealing learning rate at Stanford's CS231 [course website](#)

We have discussed this in the last session.

PIECEWISE CONSTANT



Benefits:

As we have discussed through our variable value example, say a variable needs to be at 0.3456789.

But if our updates are governed by $-\gamma \times \left(\frac{\partial L}{\partial w_i}\right)$, and it can only go as low as some multiple of 0.001,

then we can only achieve **0.345**. For us to achieve the remaining 0.000**6789**, we need to reduce

the magnitude order of updates by 1 decimal unit every time.

In Piecewise constant we reduce the learning rate gradually, keep it constant for a few epochs.

Normally, you'd see magnitude drop by 10 every 25-50 epoch.

[Problem](#):

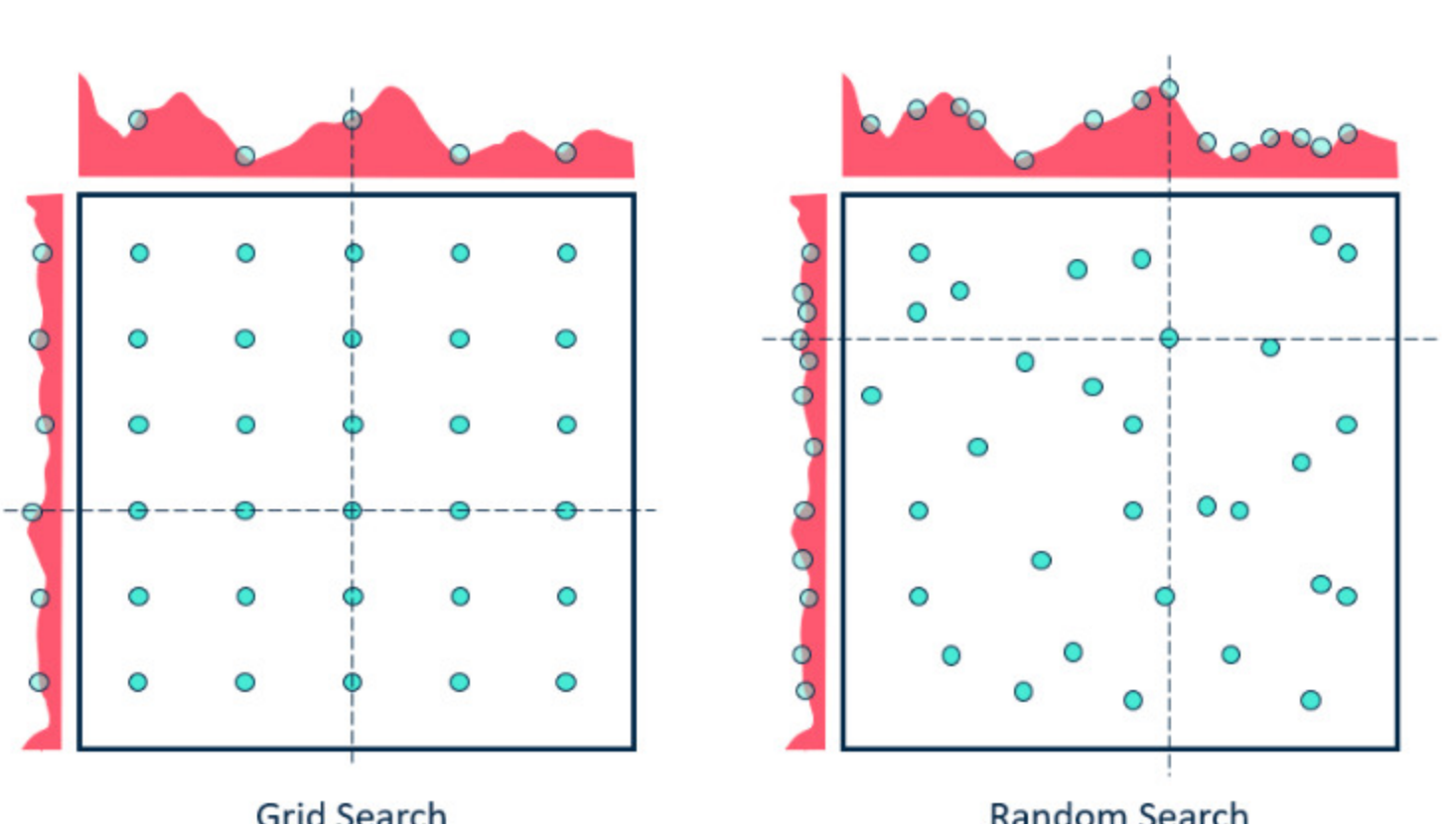
When should actually we change the LR. Process is as follows:

Assume that you know when to change.

Realize that you don't know @#$@#

Then test multiple times and see what gives you great results. This process in academia is called

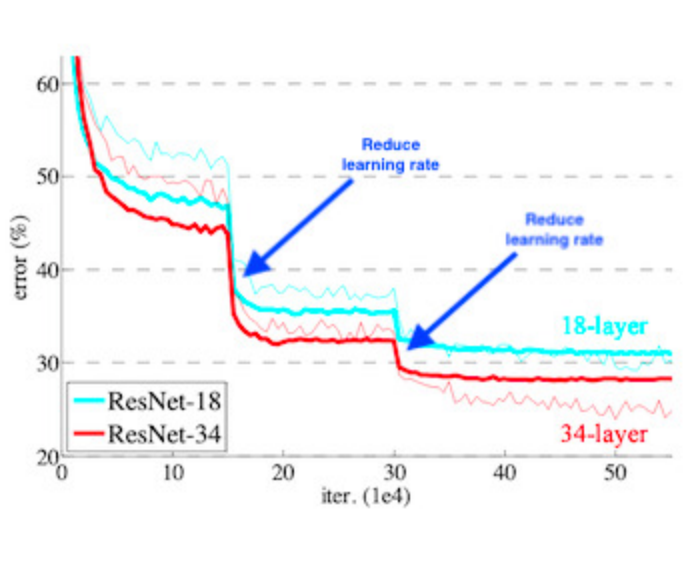
Hyper-parameter Grid Search.



There is a ton of documentation on the different process we can follow, but each has one underlying problem:

you need extraordinary resources to do this, and this isn't practical, except outside academia or large MNCs.

SIMPLER SOLUTION REDUCE_LR_ON_PLATEAU



We move the task of deciding when to change to Pytorch/Keras etc. The params we need to focus now on are:

```
CLASS torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1,
patience=10, verbose=False, threshold=0.0001, threshold_mode='rel', cooldown=0,
min_lr=0, eps=1e-08)
```

optimizer (Optimizer) – Wrapped optimizer.

mode (str) – One of min, max. In min mode, lr will be reduced when the quantity monitored has stopped decreasing; in max mode it will be reduced when the quantity monitored has stopped increasing. Default: 'min'.

factor (python:float) – Factor by which the learning rate will be reduced. new_lr = lr * factor. Default: 0.1.

patience (python:int) – Number of epochs with no improvement after which learning rate will be reduced. For example, if patience = 2, then we will ignore the first 2 epochs with no improvement, and will only decrease the LR after the 3rd epoch if the loss still hasn't improved then. Default: 10.

verbose (bool) – If True, prints a message to stdout for each update. Default: False.

threshold (python:float) – Threshold for measuring the new optimum, to only focus on significant changes. Default: 1e-4.

threshold_mode (str) – One of rel, abs. In rel mode, dynamic_threshold = best * (1 + threshold) in 'max' mode or best * (1 - threshold) in min mode. In abs mode, dynamic_threshold = best + threshold in max mode or best - threshold in min mode. Default: 'rel'.

cooldown (python:int) – Number of epochs to wait before resuming normal operation after lr has been reduced. Default: 0.

min_lr (python:float or list) – A scalar or a list of scalars. A lower bound on the learning rate of all param groups or each group respectively. Default: 0.

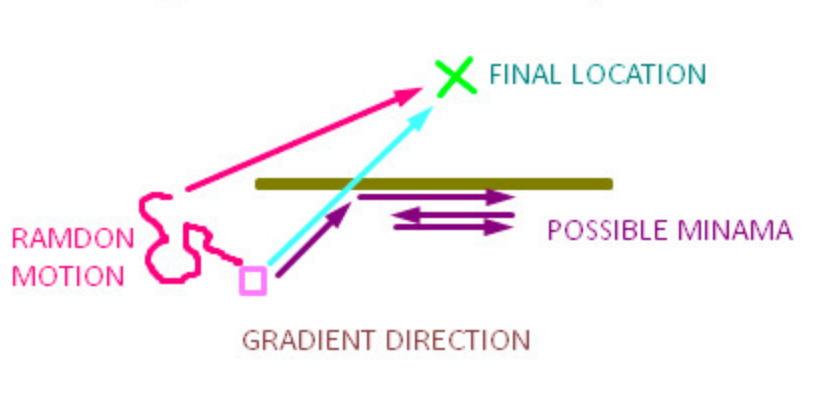
eps (python:float) – Minimal decay applied to lr. If the difference between new and old lr is smaller than eps, the update is ignored. Default: 1e-8.

These are not yet easy to pick by the way. For example, you really need to understand the difference between

rel vs **abs** threshold_mode.

SECOND CHANGE - WARMUP STRATEGIES

Our variables are initialized randomly, shouldn't we give the network some time to "warm-up" and "align" the variables in the right direction before we actually train them?



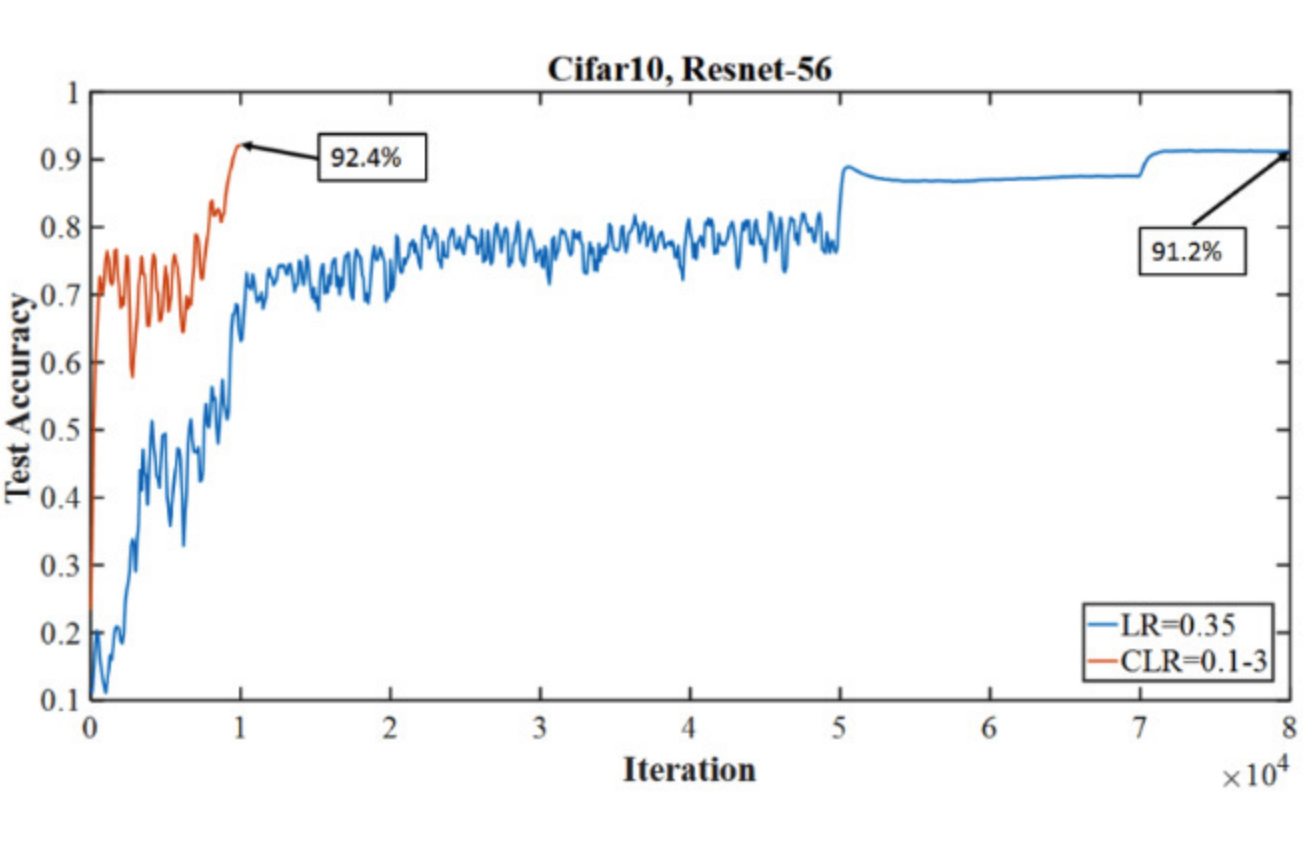
Solution - Warmup!

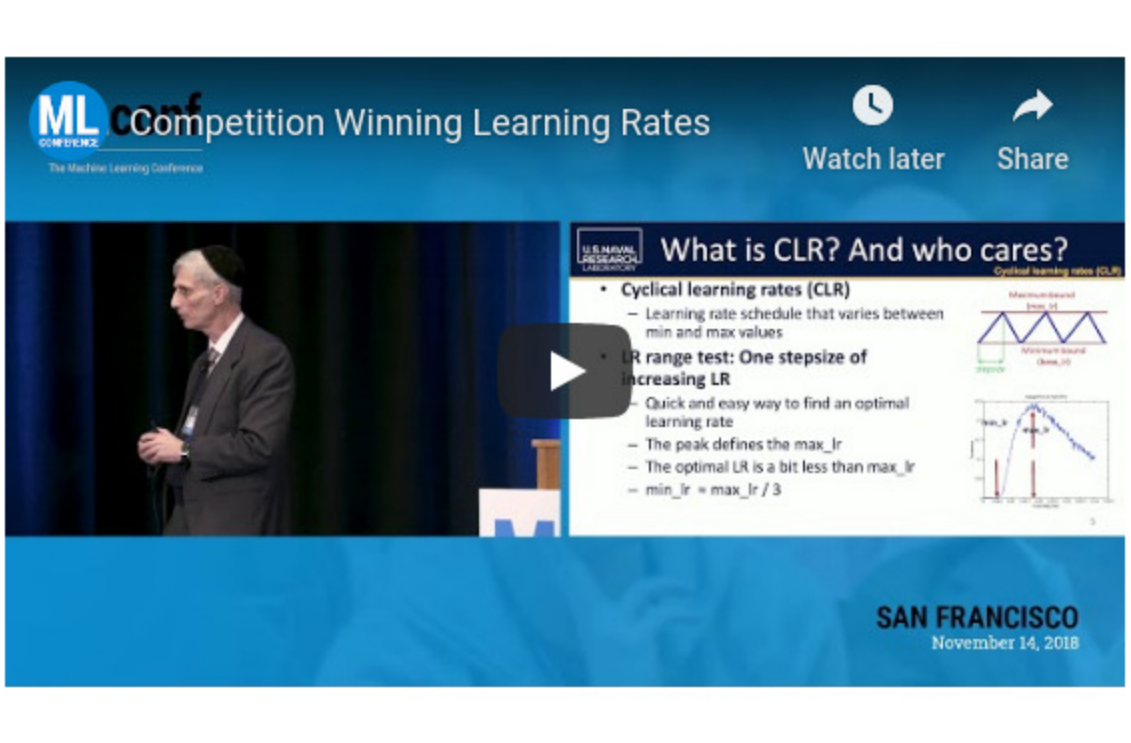
Constant warmup: In constant warm up, you train the model with a small learning rate for few epochs (say 5 epochs) and then increase the learning rate to "k times learning rate". However, this approach causes a spike in the training error when the learning rate is changed.

Gradual warmup: As the name suggests, you start with a small learning rate and then gradually increase it by a constant for each epoch till it reaches "k times learning rate". This approach helps the model to perform better with huge batch sizes (8k in this example) , which is in par with the training error of the model trained with smaller batches.

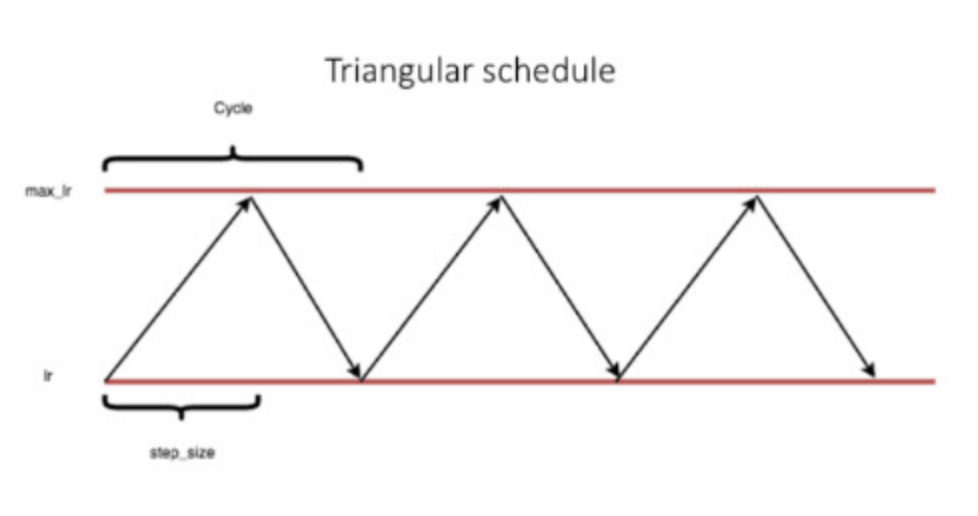
ENTER LESLIE SMITH

CYCLIC LEARNING RATES FOR TRAINING NEURAL NETWORKS





The essence of the learning rate policy comes from the observation that increasing the LR might have a short term negative effect and yet achieve a longer-term beneficial effect. This observation leads to the idea of letting the LR vary within a range of values rather than adopting a stepwise fixed or exponentially decreasing value.



Let's see how do we calculate a general schedule.

Let's define a cycle first:

$$cycle = floor\left(1 + \frac{iterations}{2(stepsize)}\right)$$

floor function makes 2.3 = 2 and so on.
iterations represents the number of batches we have **completed**
stepsize defines one-half of a cycle

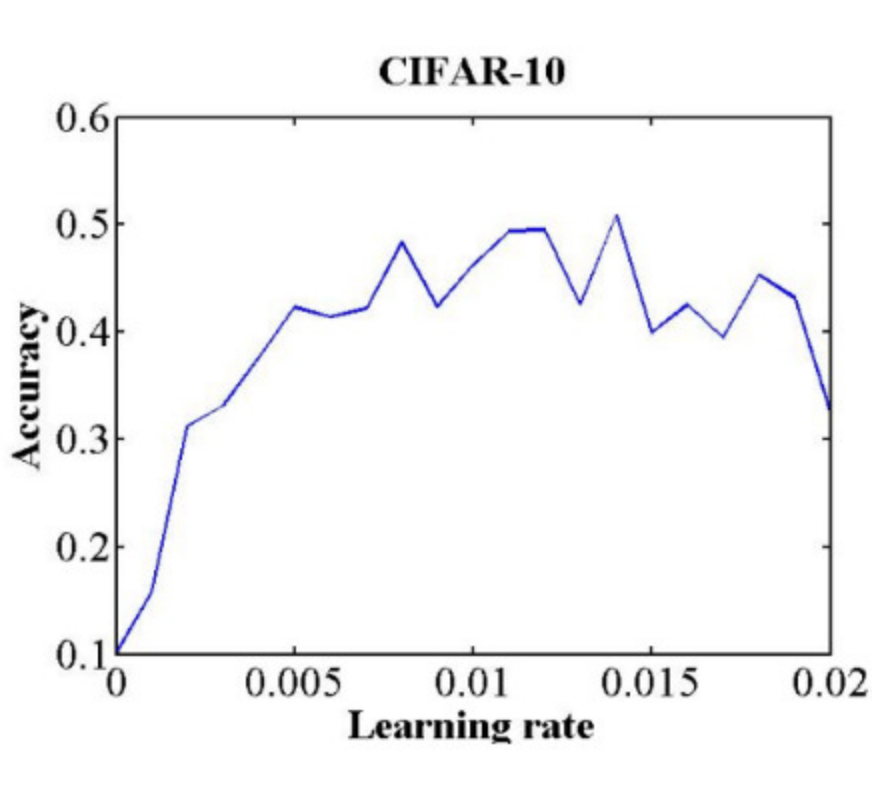
Cycles = floor (1 + 10010/(2*1000)) = floor (~5) = 5.

$$lr_t = lr_{\min} + (lr_{\max} - lr_{\min})(1 - x)$$

$$x = \left|\frac{iterations}{stepsize} - 2(cycle) + 1\right|$$

How can one estimate reasonable minimum and maximum boundary values?

LR Range Test: Run your model for several epochs while letting the learning rate increase linearly between low and high LR values.



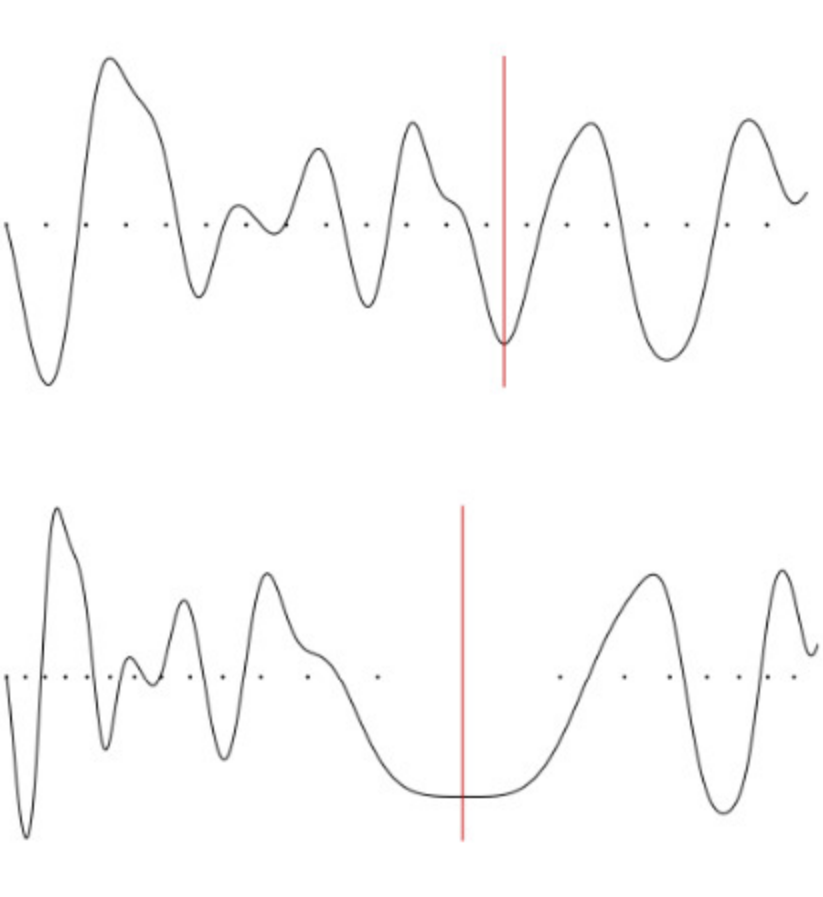
STEP-SIZE: CIFAR10 has 50,000 training images. For the batchsize 100, this means each cycle would be 50000/100 = 500 iterations.

However, results show, that it is often SLIGHTLY good to set *stepsize* equal to 2-10 times the number of iterations.

RATIONALE

The rationale is that increasing the learning rate will force the model to jump to a different part of the weight space if the current area is "spikey".

Below is a picture of three same minima with different opening width (or robustness). Which minima would you prefer?

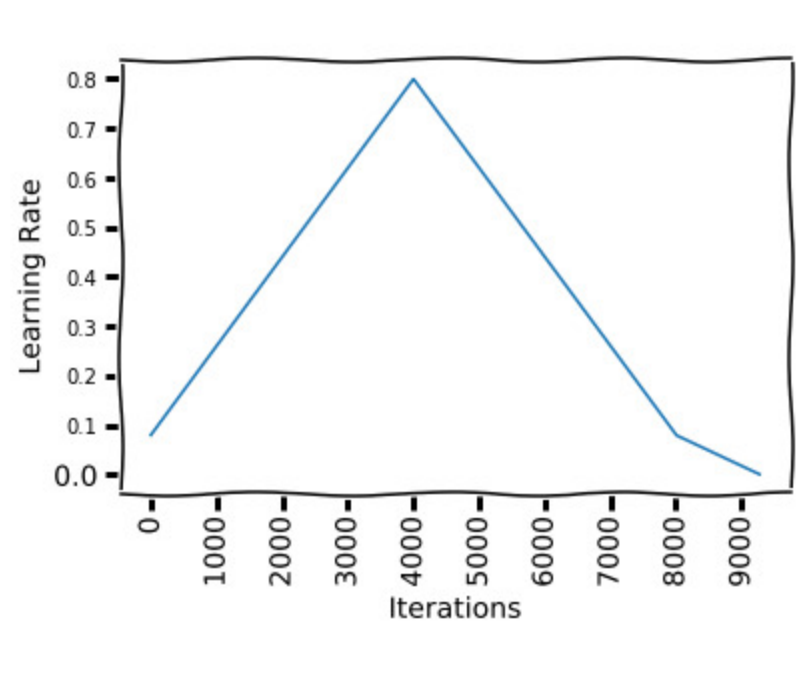


ONE CYCLE

In the paper "[A disciplined approach to neural network hyper-parameters: Part 1 — learning rate, batch size, momentum, and weight decay](#)", Leslie Smith describes approach to set hyper-parameters (namely learning rate, momentum and weight decay) and batch size. In particular, he suggests 1 Cycle policy to apply learning rates.

The author recommends doing one cycle of learning rate of 2 steps of equal length. We choose the maximum learning rate using range test. We use a lower learning rate as 1/5th or 1/10th of the maximum learning rate. We go from a lower learning rate to higher learning rate in step 1 and back to lower learning rate in step 2. We pick this cycle length slightly lesser than total number of epochs to be trained. And in last remaining iterations, we annihilate learning rate way below the lower learning rate value(1/10 th or 1/100 th).

The motivation behind this is that, during the middle of learning when the learning rate is higher, the learning rate works as a regularisation method and keep the network from overfitting. This helps the network to avoid steep areas of loss and land better flatter minima.



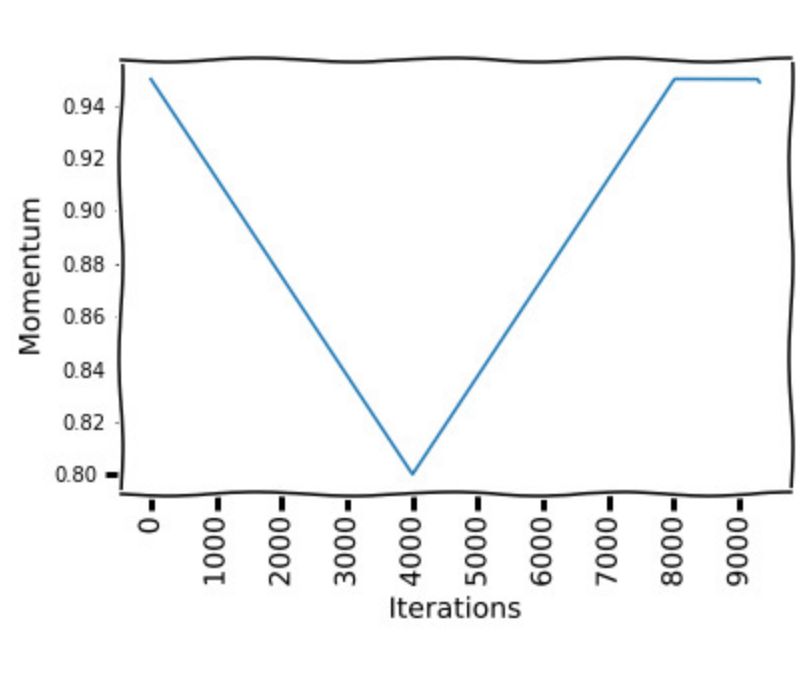
As in figure , We start at learning rate 0.08 and make step of 41 epochs to reach learning rate of 0.8, then make another step of 41 epochs where we go back to learning rate 0.08. Then we make another 13 epochs to reach 1/10th of lower learning rate bound(0.08).

With CLR 0.08–0.8 , batch size 512, momentum 0.9 and Resnet-56 , we got ~91.30% accuracy in 95 epochs on CIFAR-10.

CYCLIC MOMENTUM

Momentum and learning rate are closely related. It can be seen in the weight update equation for SGD that the momentum has a similar impact as the learning rate on weight updates.

The author found in their experiments that reducing the momentum when learning rate is increasing gives better results. This supports the intuition that in that part of the training, we want the SGD to quickly go in new directions to find a better minima, so the new gradients need to be given more weight.



In practice we choose 2 values for momentum. As in One Cycle , we do 2 step cycle of momentum, where in step 1 we reduce momentum from higher to lower bound and in step 2 we increase momentum from lower to higher bound. According to paper, this cyclic momentum gives same final results, but this saves time and efforts of running multiple full cycles with different momentum values.

With One Cycle Policy and cyclic momentum , I could replicate the results mentioned in paper. Where the model achieved 91.54% accuracy in 9310 iterations, while using one cycle with learning rates 0.08–0.8 and momentum 0.95–0.80 with resnet-56 and batch size of 512, while without CLR it requires around 64k iterations to achieve this accuracy.(Paper achieved 92.0 ± 0.2 accuracy) .

Does it provide us higher accuracy in practise? NO

THEN WHY USE ONE CYCLE POLICY?

It reduces the time it takes to reach "near" to your accuracy.

It allows us to know if we are going right early on.

It let us know what kind of accuracies we can target with given model.

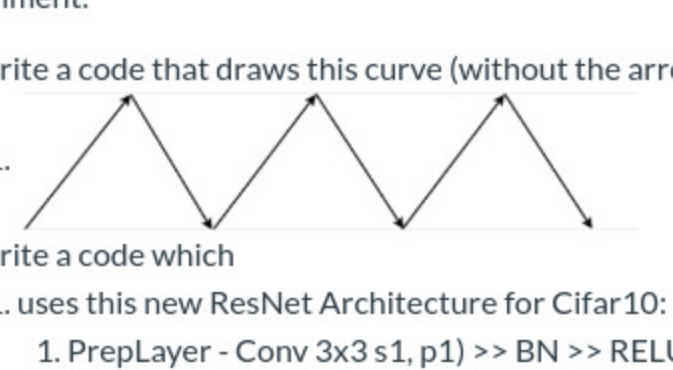
It reduces the cost of training.

It reduces the time to deploy!

References:

1. https://arxiv.org/pdf/1708.07120.pdf
2. https://medium.com/@lipeng2/cyclical-learning-rates-for-training-neural-networks-4de755927d46
3. https://www.jeremyjordan.me/nn-learning-rate/
4. https://github.com/bckenstler/CLR
5. https://towardsdatascience.com/finding-good-learning-rate-and-the-one-cycle-policy-7159fe1db5d6

Assignment:

1. Write a code that draws this curve (without the arrows). In submission, you'll upload your drawn curve and code for that
    1. 
2. Write a code which
    1. uses this new ResNet Architecture for Cifar10:
        1. PrepLayer - Conv 3x3 s1, p1) >> BN >> RELU [64k]
        2. Layer1 -
            1. X = Conv 3x3 (s1, p1) >> MaxPool2D >> BN >> RELU [128k]
            2. R1 = ResBlock((Conv-BN-ReLU-Conv-BN-ReLU))(X) [128k]
            3. Add(X, R1)
        3. Layer 2 -
            1. Conv 3x3 [256k]
            2. MaxPooling2D
            3. BN
            4. ReLU
        4. Layer 3 -
            1. X = Conv 3x3 (s1, p1) >> MaxPool2D >> BN >> RELU [512k]
            2. R2 = ResBlock((Conv-BN-ReLU-Conv-BN-ReLU))(X) [512k]
            3. Add(X, R2)
        5. MaxPooling with Kernel Size 4
        6. FC Layer
        7. SoftMax
    2. Uses One Cycle Policy such that:
        1. Total Epochs = 24
        2. Max at Epoch = 5
        3. LRMIN = FIND
        4. LRMAX = FIND
        5. NO Annihilation
    3. Uses this transform -RandomCrop 32, 32 (after padding of 4) >> FlipLR >> Followed by CutOut(8, 8)
    4. Batch size = 512
    5. Target Accuracy: 90%.
    6. The lesser the modular your code is (i.e. more the code you have written in your Colab file), less marks you'd get.
3. Questions asked are:
    1. Upload the code you used to draw your ZIGZAG or CYCLIC TRIANGLE plot.
    2. Upload your triangle Plot which was drawn with your code.
    3. Upload the link to your GitHub copy of Colab Code.
    4. Upload the github link for the model as described in A11.
    5. What is your test accuracy?