Session 5 & Assignment

Due Dec 29, 2019 by 11:59pm **Points** 8,000 **Submitting** a website url
Available Dec 9, 2019 at 6:30am - Dec 29, 2019 at 11:59pm 21 days

This assignment was locked Dec 29, 2019 at 11:59pm.

Advanced concepts + SuperConvergence

Learning Rates and more

Session Video Friday

Session Video: Saturday

What is the learning rate?

Learning rate is a hyperparameter that controls how much we are adjusting the weights of our network with respect to the loss gradient.

The lower the value, the slower we travel along the downward slope.

If we move slow, there is a good chance we would reach our minima, but it would take a long time to converge. If we move fast, we might reduce loss faster, but the minima would elude us.

This is how we calculate loss:

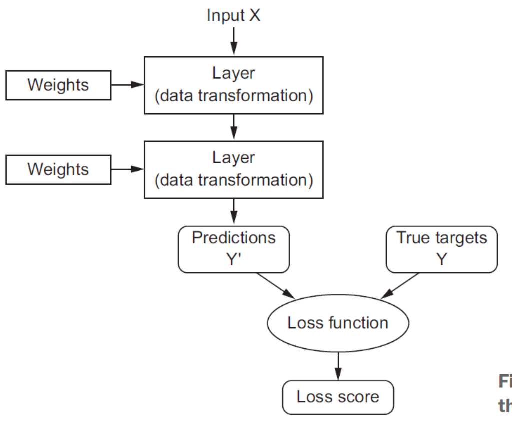
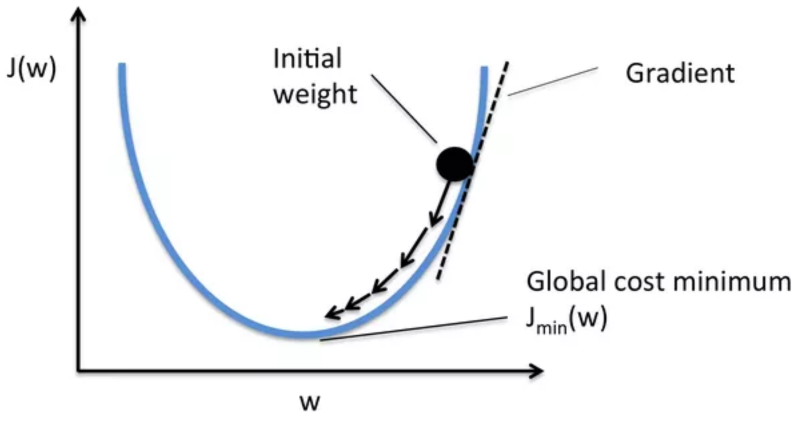


Figure 1.8 A loss function measures the quality of the network's output.

This is how the gradients are related to our loss function (or that is what people "think" what is happening, but...")



This is how we compute our weight update

$$W \coloneqq W - \alpha \frac{\partial J}{\partial W}$$

Mathematically gradient is a partial derivate of the loss function w.r.t the weights. We use a negative gradient. Alpha is the learning rate.

The Top Three

Gradient Descent: calculates the gradient for the whole dataset and updates in a direction opposite to the gradients until we find local minima. **Stochastic Gradient Descent** performs a parameter update for each batch instead of the whole dataset. This is much faster and can be further improved through momentum and learning rate finder.

Adagrad is more preferable for a sparse data set as it makes big updates for infrequent parameters and small updates for frequent parameters. It uses a different learning Rate for each parameter at a time step based on the past gradients which were computed for that parameter. Thus we do not need to manually tune the learning rate.

Adam stands for Adaptive Moment Estimation. It also calculates a different learning rate. Adam works well in practice, is faster and outperforms other techniques*.

Source (https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c)

Stochastic Gradient Descent

SGD refers to an algorithm which operates on a batch size equal to 1, while **Mini-Batch Gradient Descent is** adopted when the batch size is greater than 1. We will refer to MBGD as SGD.

Let us assume our loss function for a single sample is:

$$L(\bar{x}_i, \bar{y}_i; \bar{\theta})$$

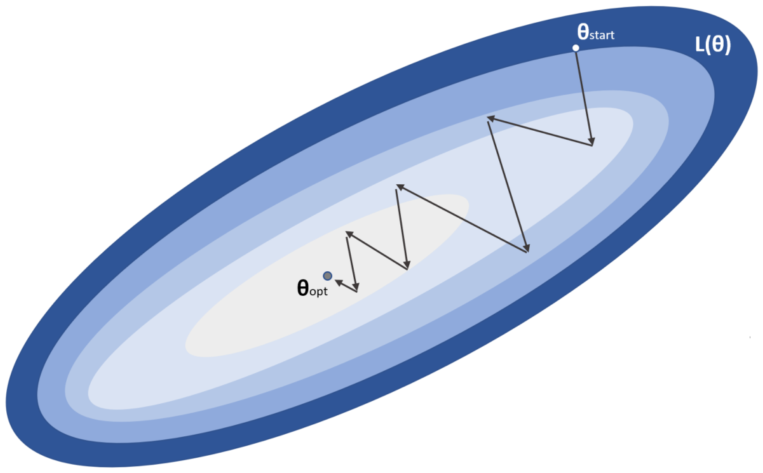
where x is the input sample, y is the label, and θ is the weight. We can define the partial derivative cost function for a batch size equal to N as:

$$L(\bar{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L\,(\bar{x}_i, \bar{y}_i; \bar{\theta})$$

The vanilla SGD algorithm is based on a θ update rule that must move the weights in the opposite direction of the gradient of L.

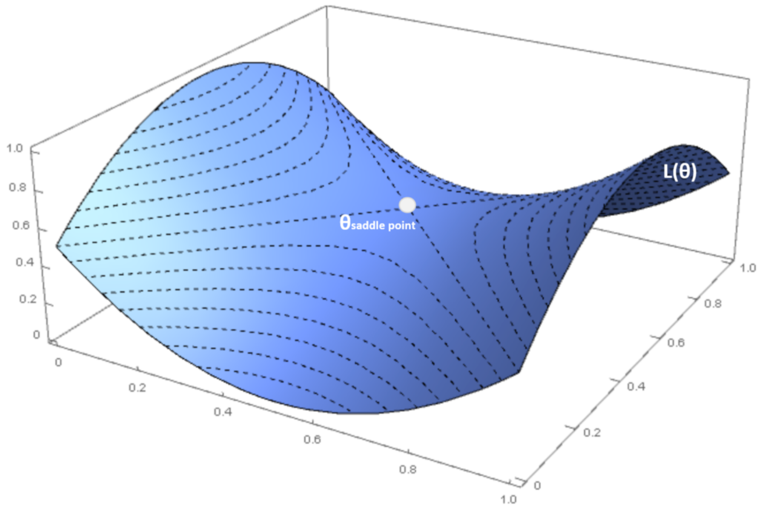
$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \alpha \nabla_{\theta} L(\bar{\theta})$$

This process is represented in the following figure:

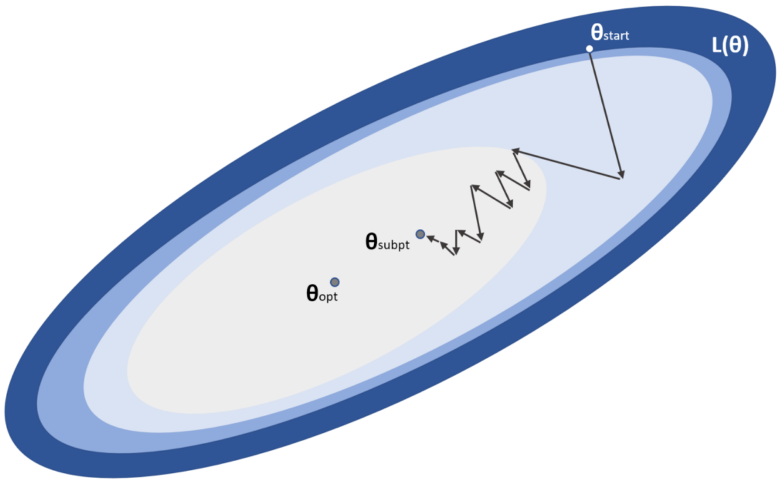


α is the learning rate, while **θ_{start}** is the initial point and **θ_{opt}** is the global minimum we're looking for. In a standard optimization problem, without particular requirements, the algorithm converges in a limited number of iterations.

Unfortunately, the reality is a little bit different, in particular in deep models, where the number of parameters is in the order of ten or one hundred million. When the system is relatively shallow, it's easier to find **local minima** where the training process can stop, while in deeper models, the probability of a local minimum becomes smaller and, instead, **saddle points** become more and more likely.



Then there is a problematic condition called **plateaus**, where L(θ) is almost flat in a very wide region.



How do we fix these problems?

Gradient Perturbation

A very simple approach to the problem of plateaus is adding a small noisy term (Gaussian noise) to the gradient:

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \alpha(\nabla_{\theta} L(\bar{\theta}) + n(t)) \;\; where\; n(t) \;\sim N(t; 0; \sigma^2)$$

The variance should be carefully chosen (for example, it could decay exponentially during the epochs). However, this method can be a very simple and effective solution to allow a *movement* even in regions where the gradient is close to zero.

```
    model.add(Conv2D(32, 3, 3))
    model.add(GaussianNoise(0.1))
    model.add(Activation('relu'))
```

I would say we are indirectly doing this already. We have discussed doing this in two ways, what are those?

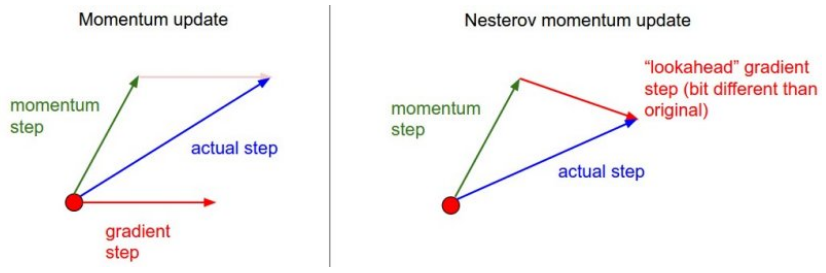
Momentum & Nesterov Momentum

It is never a good idea to interfere with the network, and in the previous approach, we are doing exactly that.

A more robust solution is provided by introducing an **exponentially weighted moving (https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average)** average for the gradients. The idea is very intuitive: instead of considering only the current gradient, we can *attach* part of its history to the correction factor, so to avoid an abrupt change when the surface becomes flat.
The **Momentum** algorithm is, therefore:

$$\begin{cases} v^{(t+1)} = \mu v^{(t)} - \alpha \nabla_{\theta} L(\bar{\theta}) \\ \bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} + v^{(t+1)} \end{cases}$$

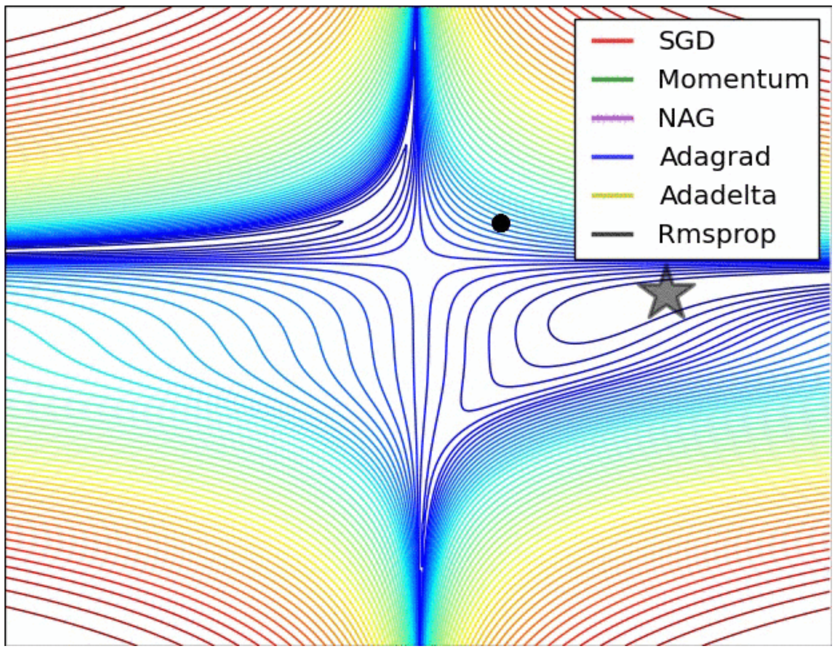
The first equation computes the correction factor considering the weight μ. If μ is small, the previous gradients are soon discarded. If, instead, μ → 1, their effect continues for a longer time. A common value in many application is between 0.75 and 0.99, however, it's important to consider μ as a hyperparameter to adjust in every application. The second term performs the parameter update. In the following figure, there's a vectorial representation of a Momentum step:

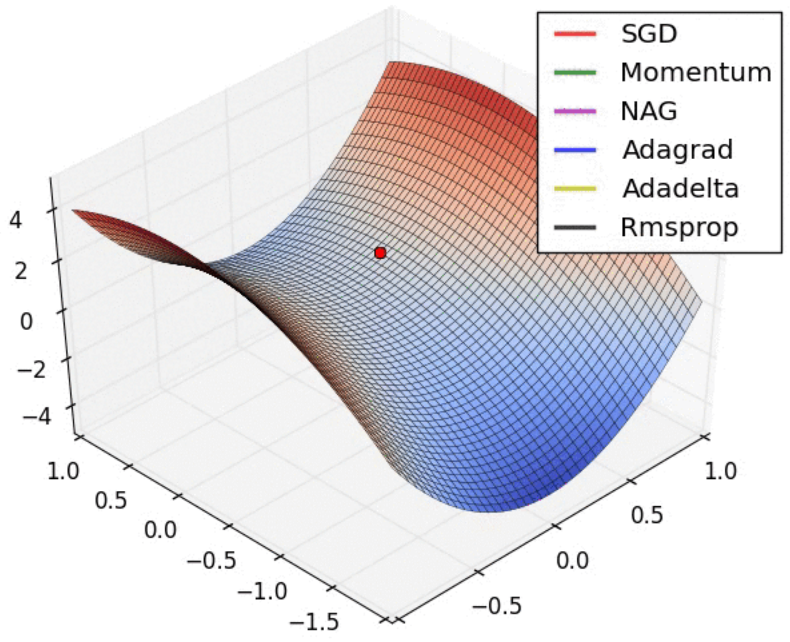


A slightly different variation is provided by the **Nesterov Momentum**. The difference with the base algorithm is that we first apply the correction with the current factor v(t) to determine the gradient and then compute v(t+1) and correct the parameters:

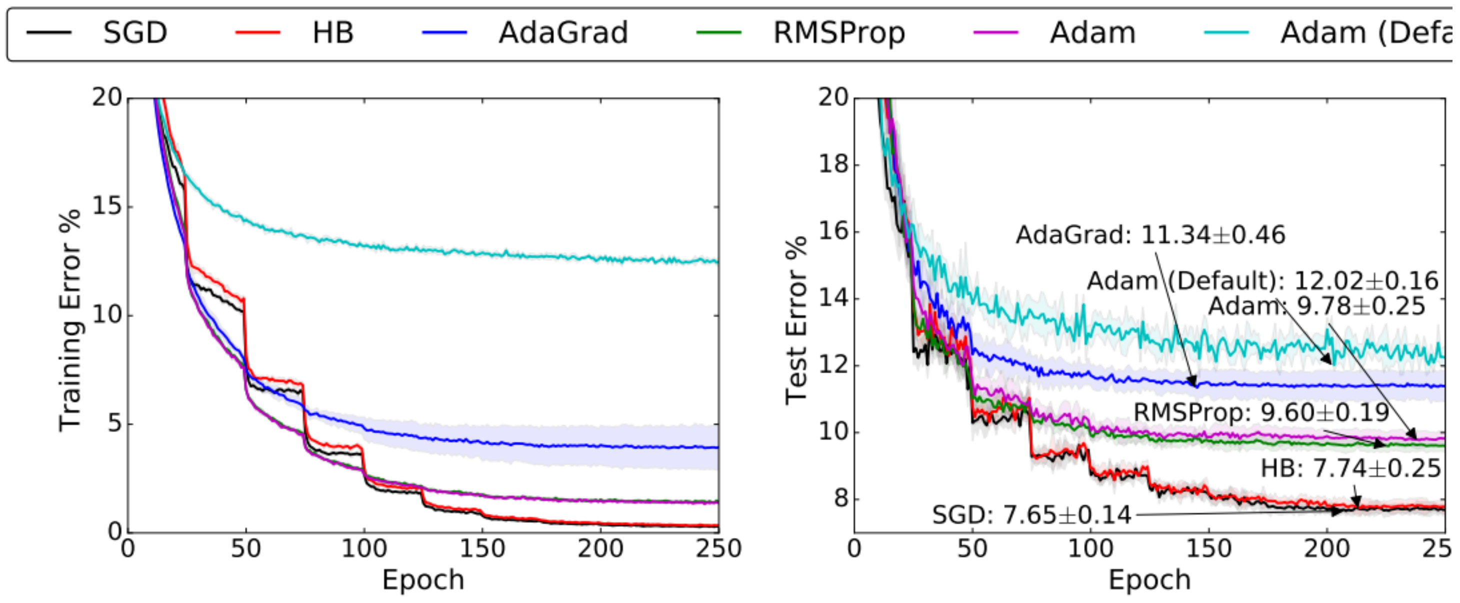
$$\begin{cases} \bar{\theta}_N^{(t+1)} = \bar{\theta}^{(t)} + \mu v^{(t)} \\ v^{(t+1)} = \mu v^{(t)} - \alpha \nabla_\theta \, L\left(\bar{\theta}_N^{(t+1)}\right) \\ \bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} + v^{(t+1)} \end{cases}$$

Which one to use?





Actual Results



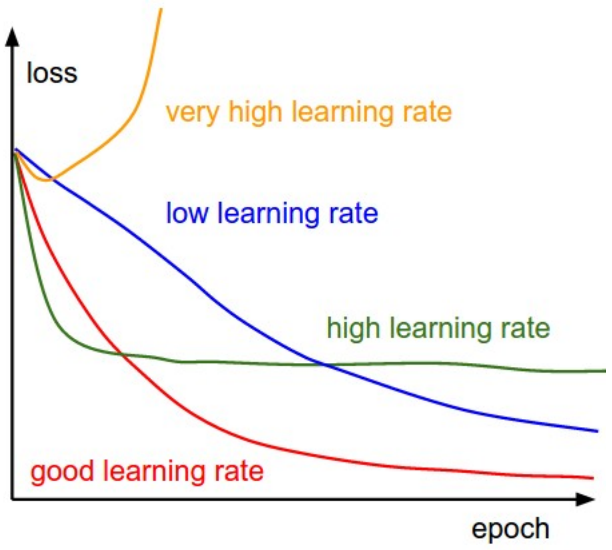
(a) CIFAR-10 (Train) **(b)** CIFAR-10 (Test)

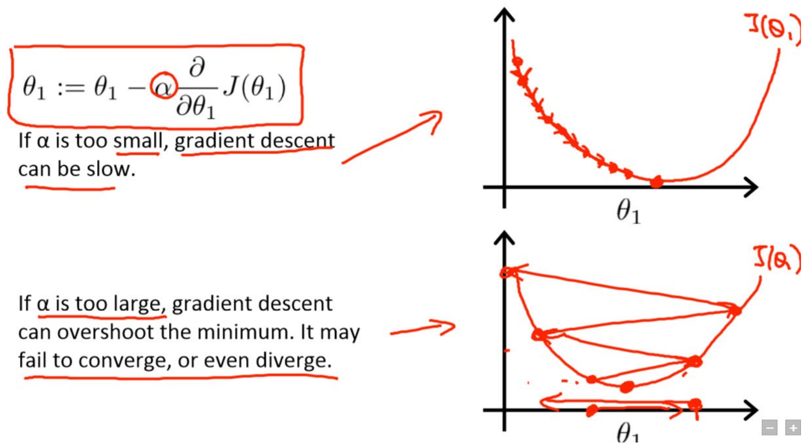
Adam and others are better than SGD/SGD+. Although adaptive optimizers have better training performance, it does not imply higher accuracy (better generalization)

Some common observations:

1. Adam and others generally have lowest training error/loss, but not validation error/loss
2. It is common to use SGDs for SOTA performance. For all **ResNet** (https://arxiv.org/abs/1512.03385), **DenseNet** (https://arxiv.org/abs/1611.05431), **ResNeXt** (https://arxiv.org/abs/1611.05431), **SENet** (https://arxiv.org/abs/1709.01507) and **NASNet** (https://arxiv.org/abs/1707.07012) paper used SGD in their implementation.
3. Adam and others are **preferred** (https://arxiv.org/pdf/1705.08292.pdf) for GANs and Q-learning with function approximations
4. SGD needs lesser memory since it only needs first momentum
5. It has much better regularization property comparing to Adam (this can be fixed, see e.g **Fixing Weight Decay Regularization in Adam** (https://openreview.net/forum?id=rk6qdGgCZ))
6. If your input data is sparse you are likely to achieve the best results using one of the adaptive learning-date methods.

Learning Rate

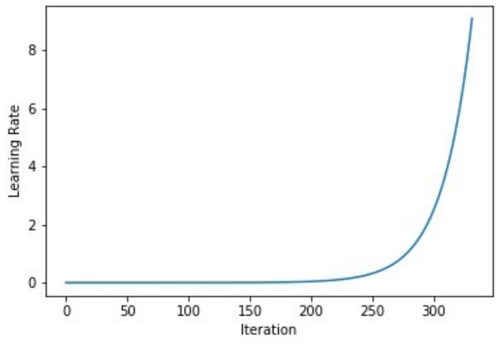



$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$
If α is too small, gradient descent can be slow.
If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.
J(θ₁)
θ₁
J(α)
θ₁

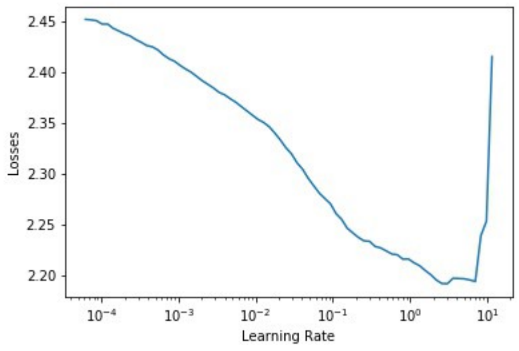
One Cycle Policy

[**Cyclical Learning Rates for Training Neural Networks**](https://arxiv.org/pdf/1506.01186.pdf) **(https://arxiv.org/pdf/1506.01186.pdf)** by Leslie N. Smith.

The paper mentions the range test run for few epochs to find out good learning rate, where we train from some low learning rate and increase the learning rate after each mini-batch till the loss value starts to explode.



The idea is to start with small learning rate (like 1e-4, 1e-3) and increase the learning rate after each mini-batch till loss starts exploding. Once loss starts exploding stop the range test run. Plot the learning rate vs loss plot. Choose the learning rate one order lower than the learning rate where loss is minimum(if loss is low at 0.1, good value to start is 0.01). This is the value where loss is still decreasing. Paper suggests this to be good learning rate value for model.



Cyclic Learning Rates:

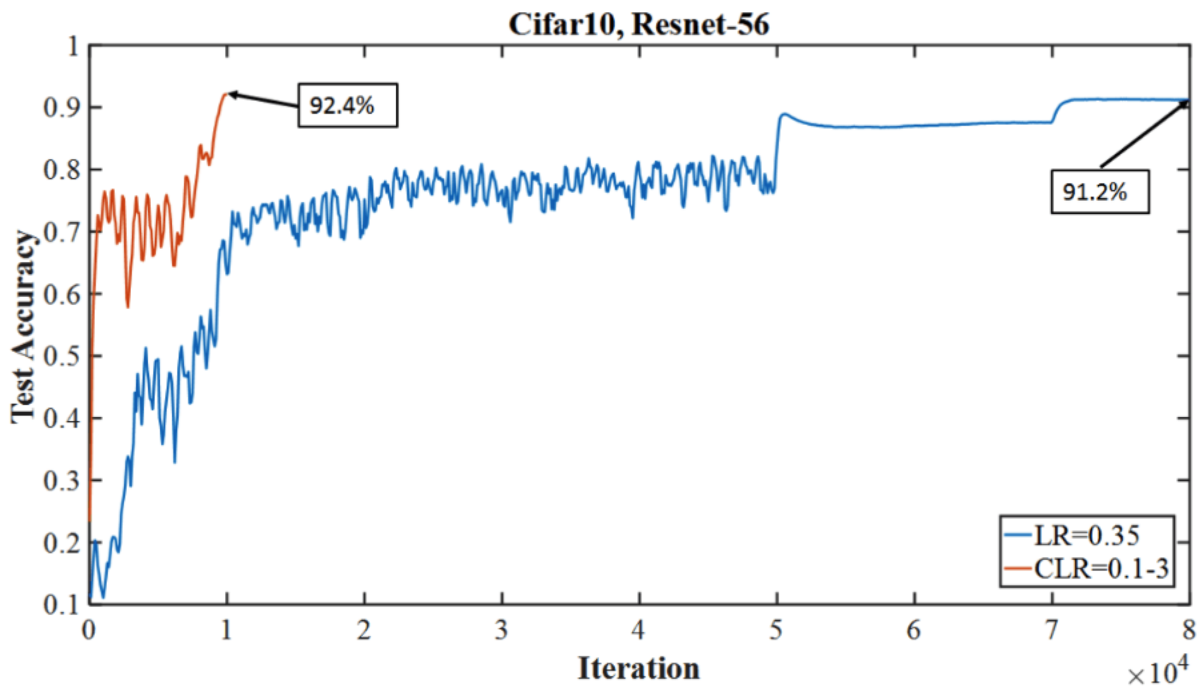
The paper further suggests to cycle the learning rate between lower bound and upper bound during complete run.

We are not going to cover CLR as we want to focus on...

WARMUP STRATEGIES

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.



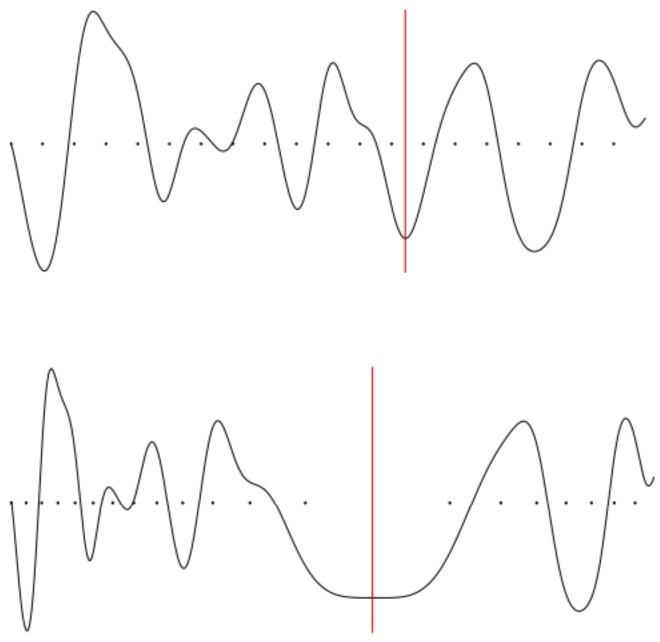
The One Cycle Policy

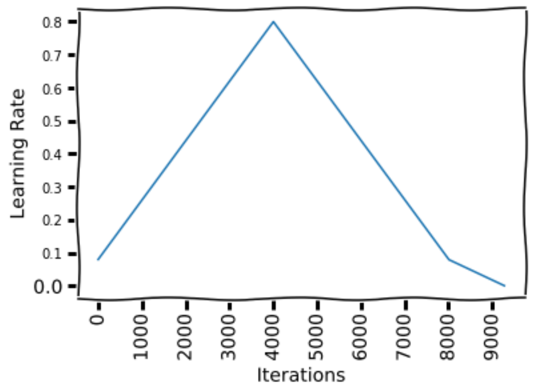
In the paper “**[A disciplined approach to neural network hyper-parameters: Part 1 — learning rate, batch size, momentum, and weight decay (https://arxiv.org/abs/1803.09820)](https://arxiv.org/abs/1803.09820)** ” , Leslie Smith describes approach to set hyper-parameters (namely learning rate, momentum and weight decay) and batch size. In particular, he suggests 1 Cycle policy to apply learning rates.

Recommend you to go through this video:

Author recommends to do one cycle of learning rate of 2 steps of equal length. We choose maximum learning rate using range test. We use lower learning rate as 1/5th or 1/10th of maximum learning rate. We go from 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 lower learning rate value(1/10 th or 1/100 th).

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





Cyclic Momentum

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

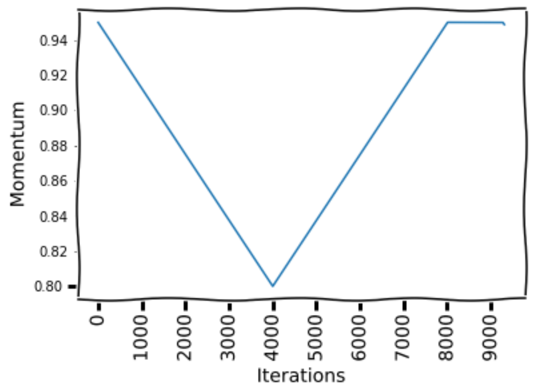
In SGD, weights are updated as:

$$\Theta_{iter+1} = \Theta_{iter} - \epsilon * \partial L * (F(x, \Theta), \Theta)$$

With momentum term, weight update in SGD becomes:

$$v_{iter+1} = \alpha * v_{iter} - \epsilon * \partial L * (F(x, \Theta), \Theta)$$
$$\Theta_{iter+1} = \Theta_{iter} + v$$

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.



Let's look at the **code** **(https://colab.research.google.com/drive/1FL6G-b9PsD5wzlTORZnSyjbwLdEG6dK0)**

What all do we need to achieve 94% within our target time:

1. access to V100
2. weight decay
3. cutout as data augmentation strategy
4. images saved as tfRecords

https://dawn.cs.stanford.edu/benchmark/CIFAR10/train.html (https://dawn.cs.stanford.edu/benchmark/CIFAR10/train.html)

Exercise: - 0 Points

1. Implement the code shared with you again, but this time add cutout to the implementation.
2. Achieve 93.5% or more within 24 Epochs
3. Submission is not required, but if you submit we will evaluate and help you with comments to improve your code.
4. Files must be in the same Assignment 5 folder (within 5Exercise folder)

Assignment 5: - 8000 Points

1. Data can be downloaded from **THIS LINK** (https://drive.google.com/file/d/1Xn5vd-J1pPdmUiO3M6h9Th52HV3NXsb6/view?usp=sharing)
2. The assignment requires you to:
   1. implement multi class classification (basic code will be shared). Here is the **starter code** (https://colab.research.google.com/drive/1QyoJ3U4SUZjRYLPbKeody0UHy1iNqdh5) (you must download the data and link your own Google Drive Link)
   2. run it for as long as you'd like to
   3. use any model of your choice
   4. use any augmentation of your choice
   5. **you cannot use transfer learning and train your model from scratch**
3. Since this is a **blind** assignment, we do not know the final achievable accuracy.
4. Marks would be allotted percentile-wise.
5. Your final accuracy matters the most.
6. Submit github link to your assignment 5 project.
7. **This assignment cannot be submitted as a group and if a group submits it, all would be disqualified.**
8. **Deadline if 25th December for all batches.**
9. **There is NO quiz 5.**