Log in Create Free Account

Sayak Paul
October 15th, 2018

NEURAL NETWORKS +1

Introduction to Cyclical Learning Rates

Learn what cyclical learning rate policy is and how it can improve the training of a neural network.

(This tutorial assumes that the reader is familiar with the basics of neural networks)

Neural network is no longer an uncommon phrase to the Computer Science society or lets say to the society in general. The main reason that makes it so cool is not just the amount of real-world problems it is solving, but also the kind of problems it is solving. How can they be so varied?

Be it in the field of Cognitive Psychology, be it in the domain of Cyber Security, be it in the area of Health-care (You are not considering Computer Vision, Computer Graphics, Natural Language Processing, etc. for the time being.). Let's name the more uncommon ones! Almost each and every industry is getting tremendously benefited by the intelligence and automation a neural network has to offer.

But why? This is the question that keeps coming and coming! Well, the answer for this is still under active research because Neural Networks are quite a black box in nature and its resemblance with a brain makes this question more complicated. Anyway, answering that question is not the objective of this post.

One thing is for sure! To get expected results from a neural network the one thing that has to be ensured is its **Training**. And by now, you already might have discovered *Training very* large enough to produce good results on an ImageNet dataset because that is kind of a benchmark.

But this very idea of training vast neural networks got revolutionized entirely when a team of talented researchers from Fast.ai was able to beat Google's model achieving an accuracy of 93% in just 18 minutes that too was only $40.

Sounds interesting? Read on!

But what were the key ingredients behind this to occur? State-of-the-art GPUs? State-of-the-art TPUS? State-of-the-art SSDs?

Absolutely not. The team's configuration was quite simple as the cost is only **$40**. The key ingredient was the use of state-of-the-art algorithms to train the neural network. In this blog, the great researcher and educator Jeremy Howard has discussed the main reasons for this big win.

This is a classic example where the power of costly hardware gets lost to the power of powerful algorithms. In this post, you are going to uncover the details of one such technique that can ensure a neural network is trained with the best possible learning rate. This technique is known as **Cyclical Learning Rate** (CLR). This was proposed way back in 2015 by Leslie N. Smith. You can check the original paper here.

But why are you going to cover only learning rate when there are other essential hyperparameters like dropout rate and activation functions? It is because the learning rate is the most important one among them. Just that!

In this post, you are going to study:

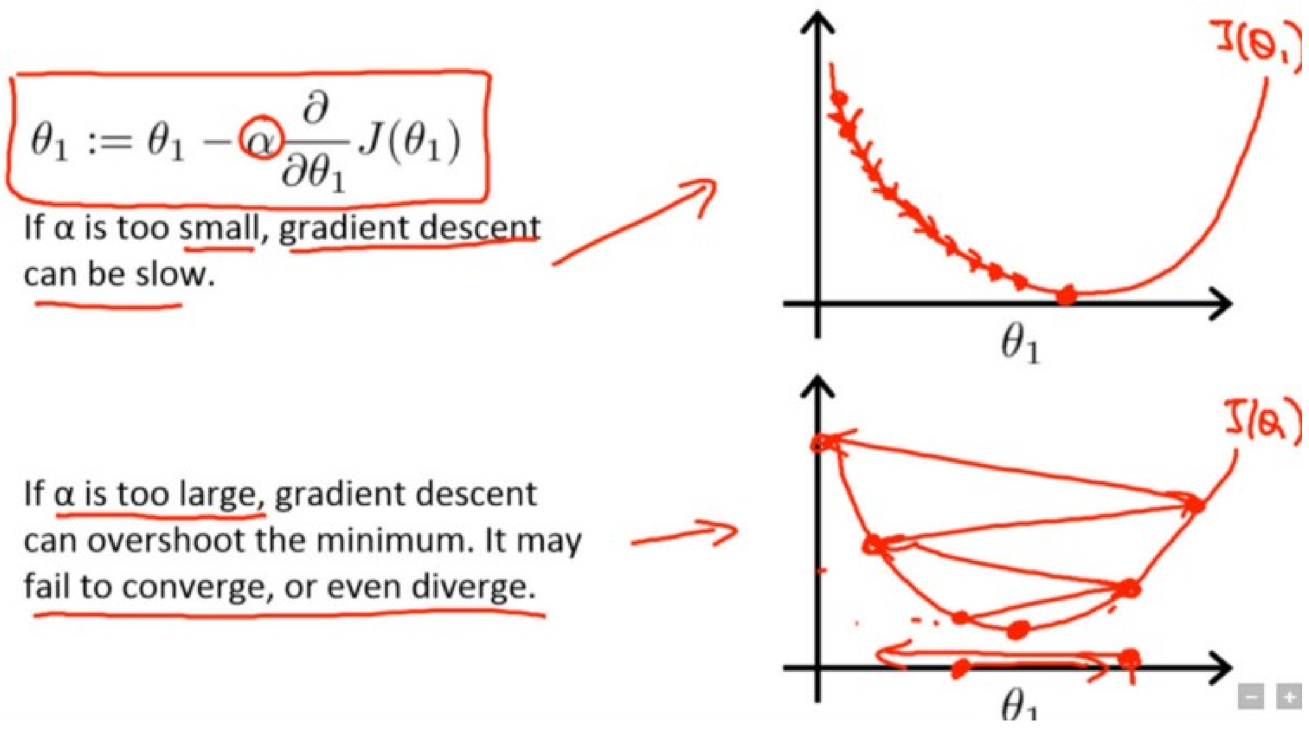
- Quickly revisit why learning rates are needed?
- What are the techniques available for finding the most suitable learning rate for a neural network?

- Inner mechanics of cyclical learning rates
- A case study in Python

Why are learning rates needed?

Let's quickly revisit the primary purpose of using learning rates for training a neural net.

Learning rate is a hyperparameter that controls how much you are adjusting the weights of our network with respect to the loss gradient. What? Why are gradients coming in the picture? It is because you are on your way to optimizing a neural network that you have just created with **gradient descent**. Now, essentially the goal of gradient descent is to find the minima of the loss function your neural network is trying to optimize. Take a look at the following image [taken from Andrew Ng's Deep Learning course on Coursera]:

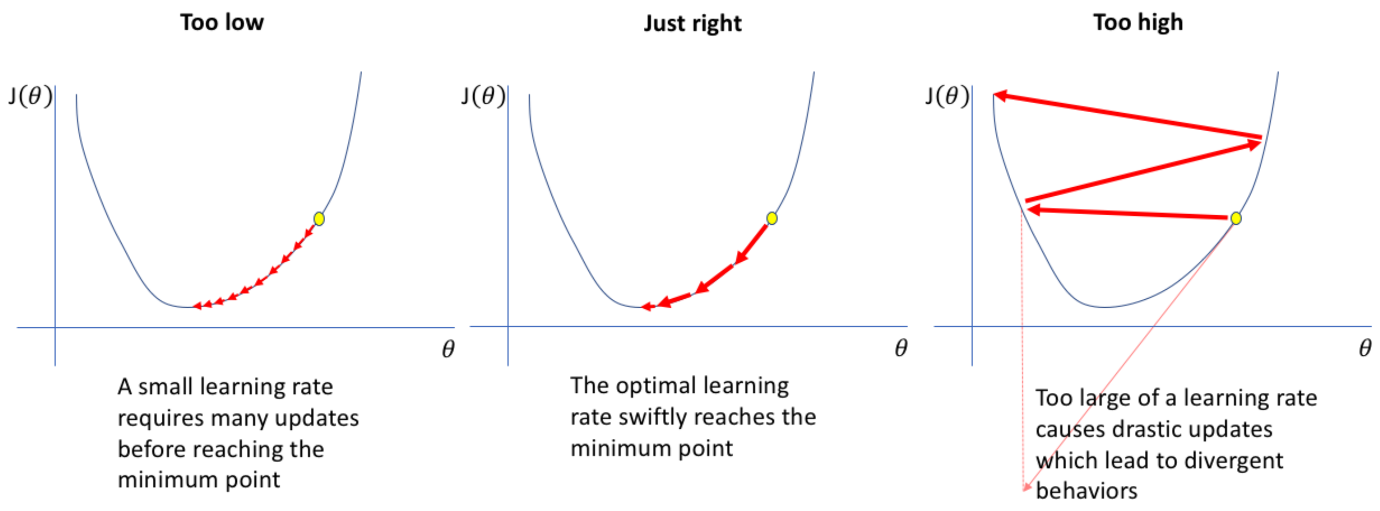

$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$
If α is too small, gradient descent can be slow.
If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.
J(θ₁)
θ₁
J(θ)
θ₁

The term in the rectangle is the update rule with which the network starts to learn its parameter $\theta 1$ where α is the learning rate.

the lowest point, you take partial derivatives of the loss function $J(\theta 1)$ and compute the gradients for getting the directions towards arriving at that lowest point.

To arrive a bit faster at that point, you add another term α which is the learning rate. The lower the value, the slower you travel along the downward slope. While this might be a good idea (using a low learning rate) in terms of making sure that you do not miss any local minima, it could also mean that you'll be taking a long time to converge—especially if you get stuck on a plateau region.

That was a quick recap of the objectives of learning rates in simple words. Now you will study the techniques of choosing a good learning rate for your neural network.


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Source: Jeremy Jordan's blog

What are the techniques available for finding the most suitable learning rate for a neural network?

There is no fixed learning rate for a neural network. It depends on the kind of problem you are working on, the type of data you are feeding to your network, and most importantly the structure of the network which varies from problem to problem. Handpicking a learning rate is a very painful task because in the case you are training a large network you can incur massive amounts of costs. And it is very time-consuming as well.

- That is again horrible for a large network. But why do you keep coming to large networks? It's because almost any real-world complex problem will need an extensive neural network.

Before CLR, **Adaptive Learning Rates** were proposed which can be thought as a competitor to CLR but experimentations with Adaptive Learning Rates are computationally expensive which CLR is not.

Still now, the most common practice is to set the learning rate to a constant value and decrease it by order of magnitude once the accuracy has plateaued.

Therefore, there is a clear need for a systematic technique which can simplify the process of choosing a good learning rate for a particular neural network. Not only this but also, there has to be a sufficient amount of reasons which would support that approach as to why it is trust-worthy.

It seems Cyclical Learning Rates (CLR) appeared just in time.

Introduction to cyclical learning rates:

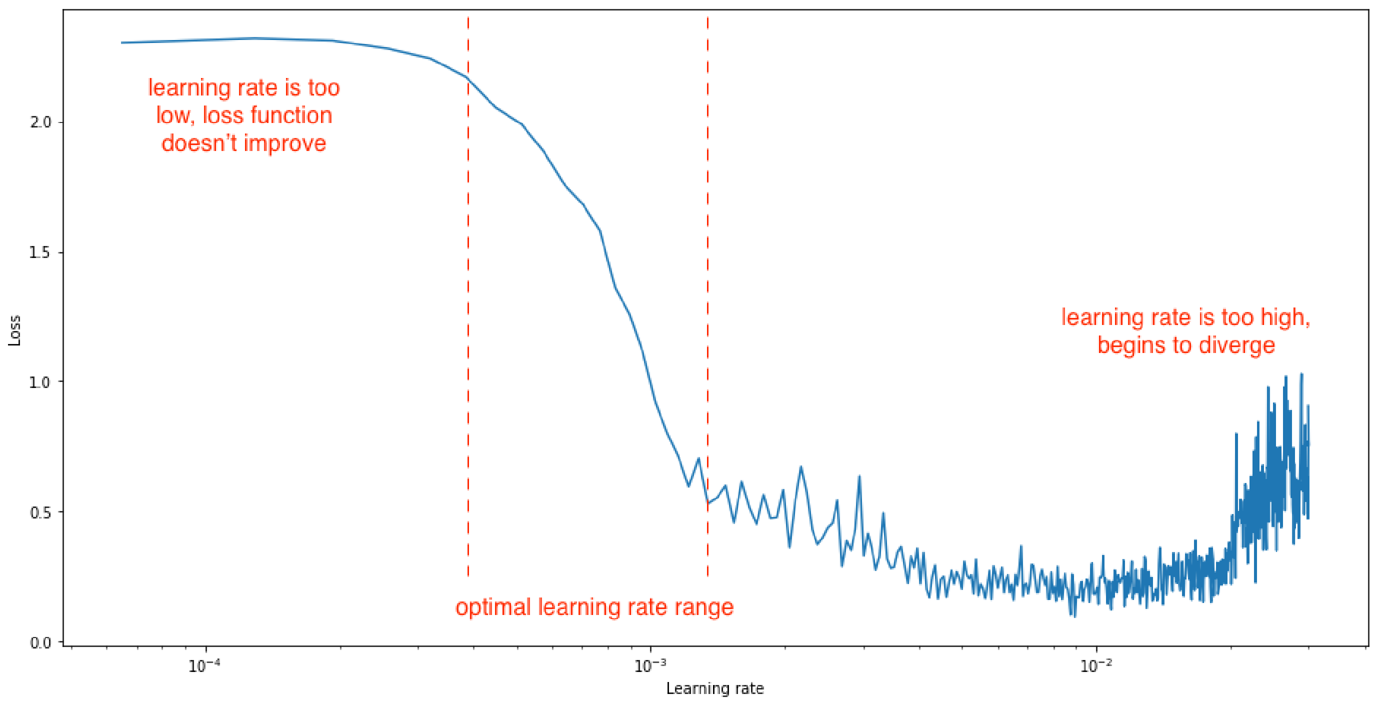
The objectives of the cyclical learning rate (CLR) are two-fold:

- CLR gives an approach for setting the global learning rates for training neural networks that eliminate the need to perform tons of experiments to find the best values with no additional computation.
- CLR provides an excellent learning rate range (LR range) for an experiment by introducing the concept of **LR Range Test**.

You will study this section merely for building your intuition of how CLR works. In the next section, you will dive into more details. In the previous sections, you briefly understood why learning rates are used in any way. Let's again recall it.

wherein you can observe the behavior of learning rate with respect to the loss. The experiment is straightforward to visualize where you gradually increase the learning rate after each *mini-batch*, recording the loss at each increment. This gradual increase can be either be linear or be exponential. And yes, this is essentially the *LR Range Test*.

After performing the experiment Leslie showed us, for too low learning rates, the loss may decrease but at a very shallow rate. When entering the optimal learning rate zone, you'll observe a quick drop in the loss function. If you further increase the learning rate, then it can cause parameter loss in the network which in turn might lead to the increase in losses. So, from this experiment, it is clear that you are interested in a steep decrease of the loss function and for that, you can analyze the gradients of the loss function at different stages of the training.



Source: Jeremy Jordan's blog

So, from the above graph, you can easily spot three different phases where the loss does not change much, then comes a time when a steep decrease happen and then again the loss starts to increase again slowly.

hang of it, let's find out more.

Delving more with CLR:

The above observation gives you an important point to consider:

The intuition of CLR arises from the idea of letting the learning rate vary within a range of values rather than adopting a linearly or exponentially decreasing value. You can do this by setting a definite range of learning rates and then instead of going for any linear or exponential variation, you cyclically vary the learning rates from the defined range. Leslie considered the following function forms to vary the learning rate cyclically:

- Triangular window (linear)
- Welch window (parabolic)
- Hann window (sinusoidal)

But all of the forms produced equivalent results, and in the original paper therefore, the idea is only presented with a triangular form (linearly increasing then linearly decreasing) because of its simplicity. This policy is referred to as *triangular learning rate policy* as well. Implementing this is also straight-forward. Following is a generalized implementation of the *triangular learning rate policy*:

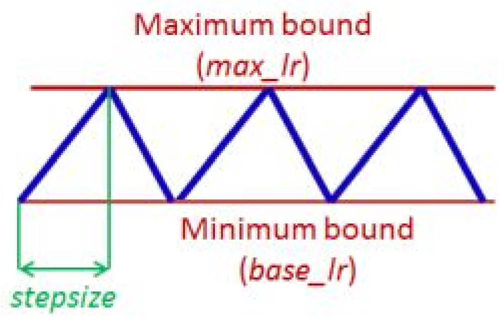
```
local cycle = math.floor(1 + epochCounter / ( 2 * stepsize))
local x = math.abs(epochCounter / stepsize - 2* cycle + 1 )
local lr = opt.LR + (maxLR - opt.LR) * math.max(0 , (1-x ))
```

where

- opt.LR is the specified lower (i.e., base) learning rate,
- epochCounter is the number of epochs of training,

- stepsize is half the period or cycle length and
- max_lr is the maximum learning rate boundary

Refer to the following image in order to think about the triangular form visually:



Source: The original CLR paper mentioned at the beginning of the tutorial.

In addition to the triangular learning rate policy, the following policies were also presented in the paper:

1. **triangular2** - It is as same as the triangular policy except the learning rate difference is made half at the end of each cycle. This means the learning rate difference drops after each cycle.
2. **exp range** - In this case, the learning rate varies between the minimum and maximum boundaries and each boundary value declines by an exponential factor.

One question you might have quickly asked yourself at this point of time is - **How can one estimate reasonable minimum and maximum boundary values?** Remember the LR Range Test that you studied just a moment ago? Now, you should be able to find its relevance better.

Leslie proposed a straightforward and convenient way to decide the Learning Rate range:

- Run the model for several epochs while letting the learning rate increase linearly (use triangular learning rate policy) between low and high learning rate values.

These two learning rates are good choices for defining the range of the learning rates.

Let's do a quick case study now to see how CLR can give amazing results.

(To the very end of this tutorial, you will discover some more advancements over CLR that have been proposed recently, but by now, you already have understood the worth of CLR.)

A case study of CLR in Python:

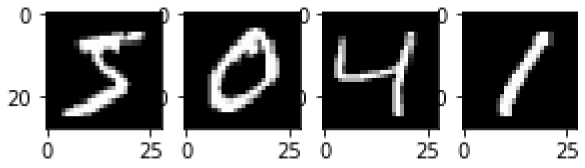
You will be doing this using the classic MNIST dataset which is probably the most popular dataset for getting started into Computer Vision and Deep Learning. Check out this blog if you want to learn about MNIST in a very detailed manner. You will use `keras` extensively for all purposes of the experiment. `keras` provides a built-in version of the dataset. You will start off your experiment by importing that and by performing some basic EDA.

```
from keras.datasets import mnist
```

```
Using TensorFlow backend.
```

You have imported the dataset successfully. Now, you will do some basic visualizations of the dataset.

```
import matplotlib.pyplot as plt

(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Plot 4 images as gray scale
plt.subplot(152)
plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
plt.subplot(153)
plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
plt.subplot(154)
plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))
# Show the plot
plt.show()
```



That's great! You will straight proceed towards building a simple multi-layer neural network. But before that, you will do some basic data preprocessing.

```
# Flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
```

What did you do?

The images in the dataset are of 28*28 dimensions which is difficult to accommodate in a simple multilayer neural network. That is why you converted the images into a single dimension where each image contains 784-pixel data using the `reshape()` function.

The pixel values in the images are in the range of 0 - 255. A good idea will be to decrease this even further by normalizing the range to 0 - 1.

```
X_train = X_train / 255
X_test = X_test / 255
```

The output variable is an integer from 0 to 9. This is a multi-class classification problem. You will perform one-hot encoding of the class labels for getting a vector of class integers into a binary matrix. You will do this to do a “binarization” of the category and so that you can include it as a feature to train the neural network.

```
from keras.utils import np_utils

y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Now you will define the structure of your network. You will use a simple fully-connected network for this purpose. In `keras` this is typically a three-step process:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Create model
model = Sequential()
model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='rel
model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))

# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Let's see what all you did in the above code. You are sequentially constructing the network (which is a linear stack of layers). Then you started to add the layer in your network wherein the first layer you added the neurons (which is equal to the number of pixels in an image, i.e., 784) and you specified the input dimension of the images which is in this case as same as the number of the pixels. You instructed your network to get itself initialized with weights from a *normal* distribution. Finally, you supplied `relu` as the activation function for the first layer.

In the final layer, you kept the number of the neurons to 10 (which is the number of class labels), and you provided the activation to be `softmax` to turn the outputs into probability-like values and allow one class of the 10 to be selected as the model's output prediction.

`categorical_loss` in this case).

Now you will train the model and record the time it took to get trained. You will also test its performance.

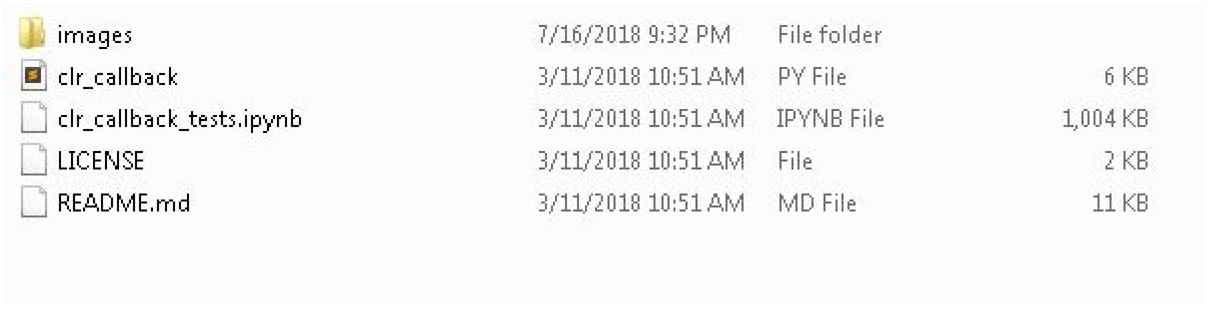
```
import timeit

# For fixing the reproducibility
from numpy.random import seed
seed(1)

# Fit the model
startTime = timeit.default_timer()
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200, verbo
elapsedTime = timeit.default_timer() - startTime
print("Time taken for the Network to train : ",elapsedTime)

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
 - 10s - loss: 0.2774 - acc: 0.9207 - val_loss: 0.1362 - val_acc: 0.9610
Epoch 2/10
 - 8s - loss: 0.1113 - acc: 0.9675 - val_loss: 0.0951 - val_acc: 0.9720
Epoch 3/10
 - 8s - loss: 0.0712 - acc: 0.9793 - val_loss: 0.0802 - val_acc: 0.9750
Epoch 4/10
 - 8s - loss: 0.0503 - acc: 0.9857 - val_loss: 0.0687 - val_acc: 0.9788
Epoch 5/10
 - 8s - loss: 0.0360 - acc: 0.9897 - val_loss: 0.0632 - val_acc: 0.9797
Epoch 6/10
```

```
 - 8s - loss: 0.0201 - acc: 0.9951 - val_loss: 0.0633 - val_acc: 0.9802
Epoch 8/10
 - 8s - loss: 0.0150 - acc: 0.9962 - val_loss: 0.0613 - val_acc: 0.9808
Epoch 9/10
 - 8s - loss: 0.0108 - acc: 0.9978 - val_loss: 0.0625 - val_acc: 0.9806
Epoch 10/10
 - 8s - loss: 0.0076 - acc: 0.9988 - val_loss: 0.0612 - val_acc: 0.9809
Time taken for the Network to train :  86.4216202873591
Baseline Error: 1.91%
```

This simple model did quite well achieving an error rate of just 1.91% in approximately 87 seconds. Now you will see the power of CLR. You will start off by cloning the `keras` implementation of CLR from Github.

After a successful clone, you should have the following files into your local working directory.



images	7/16/2018 9:32 PM	File folder	
clr_callback	3/11/2018 10:51 AM	PY File	6 KB
clr_callback_tests.ipynb	3/11/2018 10:51 AM	IPYNB File	1,004 KB
LICENSE	3/11/2018 10:51 AM	File	2 KB
README.md	3/11/2018 10:51 AM	MD File	11 KB

The CLR policy is implemented as a `keras callback` here.

```
from keras.callbacks import *
from clr_callback import *
from keras.optimizers import Adam

# You are using the triangular learning rate policy and
#  base_lr (initial learning rate which is the lower boundary in the cycle) is 0.1
clr_triangular = CyclicLR(mode='triangular')
model.compile(optimizer=Adam(0.1), loss='categorical_crossentropy', metrics=['accuracy'])
```

```
startTime = timeit.default_timer()
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=2000,callb
elapsedTime = timeit.default_timer() - startTime
print("Time taken for the Network to train : ",elapsedTime)

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
 - 4s - loss: 0.0046 - acc: 0.9996 - val_loss: 0.0571 - val_acc: 0.9831
Epoch 2/10
 - 4s - loss: 0.0030 - acc: 0.9998 - val_loss: 0.0575 - val_acc: 0.9829
Epoch 3/10
 - 4s - loss: 0.0023 - acc: 0.9999 - val_loss: 0.0594 - val_acc: 0.9827
Epoch 4/10
 - 4s - loss: 0.0018 - acc: 0.9999 - val_loss: 0.0589 - val_acc: 0.9835
Epoch 5/10
 - 4s - loss: 0.0014 - acc: 1.0000 - val_loss: 0.0595 - val_acc: 0.9831
Epoch 6/10
 - 4s - loss: 0.0011 - acc: 1.0000 - val_loss: 0.0600 - val_acc: 0.9836
Epoch 7/10
 - 4s - loss: 0.0010 - acc: 1.0000 - val_loss: 0.0612 - val_acc: 0.9832
Epoch 8/10
 - 4s - loss: 8.2190e-04 - acc: 1.0000 - val_loss: 0.0621 - val_acc: 0.9829
Epoch 9/10
 - 4s - loss: 6.6182e-04 - acc: 1.0000 - val_loss: 0.0624 - val_acc: 0.9836
Epoch 10/10
 - 4s - loss: 5.7177e-04 - acc: 1.0000 - val_loss: 0.0635 - val_acc: 0.9836
Time taken for the Network to train :  43.23223609056981
Baseline Error: 1.64%
```

Congratulations!

You have made it to the end. In this tutorial, you studied a very crucial problem of finding a suitable learning rate and how CLR completely changed the way you used to approach this problem. You studied CLR covering a good amount of details and did small experiments to see how CLR can produce some excellent results in less time.

Now, what next? The two byproducts that have come out from CLR are:

- Stochastic Gradient Descent with Warm Restarts also known as **Cosine Annealing**
- Differential Learning Rates

Study the above two approaches to get even more insights on this topic. Also, after CLR Leslie published a paper titled A disciplined approach to neural network hyper-parameters: Part 1 -- learning rate, batch size, momentum, and weight decay which revisits CLR and discusses efficient methods for choosing the values of other important hyperparameters of a neural network. Leslie also revisited one of his techniques called **Super Convergence** in this paper. This paper is a must read for anyone, who thinks, eats and sleeps neural networks.

Limited applicability is one of the significant shortcomings of CLR. It has to be made full-proof before one can use it in production levels.

If you are interested in knowing more about Neural Networks, you should take DataCamp's Deep Learning in Python course which is very well designed and is taught by Dan Becker (Head of Kaggle Learn)

▲ 16