S10

Due No Due Date **Points** 0 **Available** after Mar 18 at 6:30am

Session 10

Advanced concepts

Training and Learning Rates

What is the learning rate?

The 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:

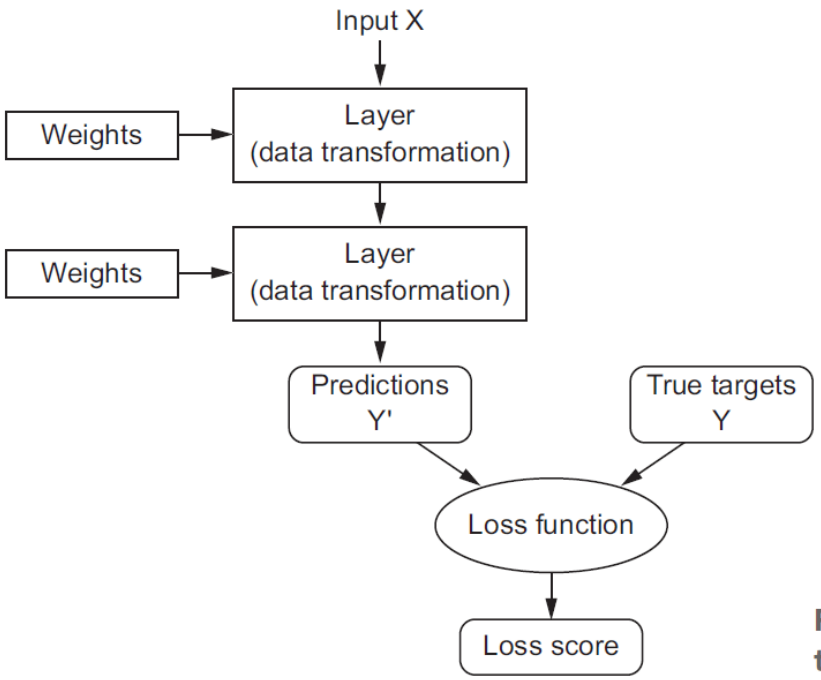
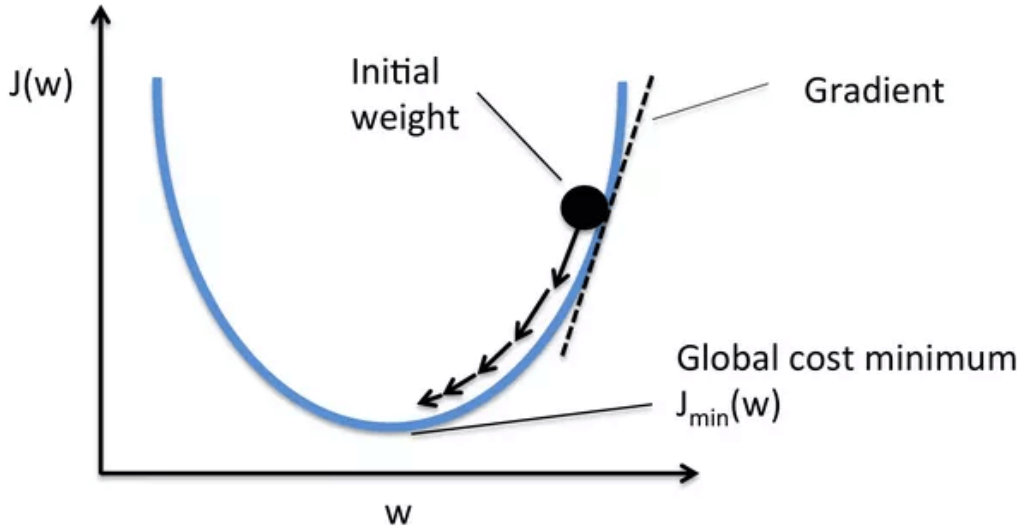


Figure 1.8 A loss function measures the quality of the network's output.

This is how the gradients are related to our loss function



WEIGHT UPDATE

$$w = w - \alpha \left(\frac{\partial L}{\partial w} \right)$$

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.

Let's spend some time on this and understand what that negative number does!

Constant Learning Rate Algorithms

Most widely used Optimization Algorithm, the Stochastic Gradient Descent falls under this category.

$$W^{(k+1)} = W^{(k)} - \eta * (\Delta J(W))$$

Here **η** is called a "learning rate" which is a hyperparameter and has to be tuned. Small **η** is a snail and large is a missile. We need to find the correct one.

We can improve this situation through **momentum** (covered later).

Adaptive Learning Rate Algorithms

Adaptive gradient descent algorithms such as Adagrad, Adadelta, RMSProp, Adam, provide an alternative to classical SGD.

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:

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

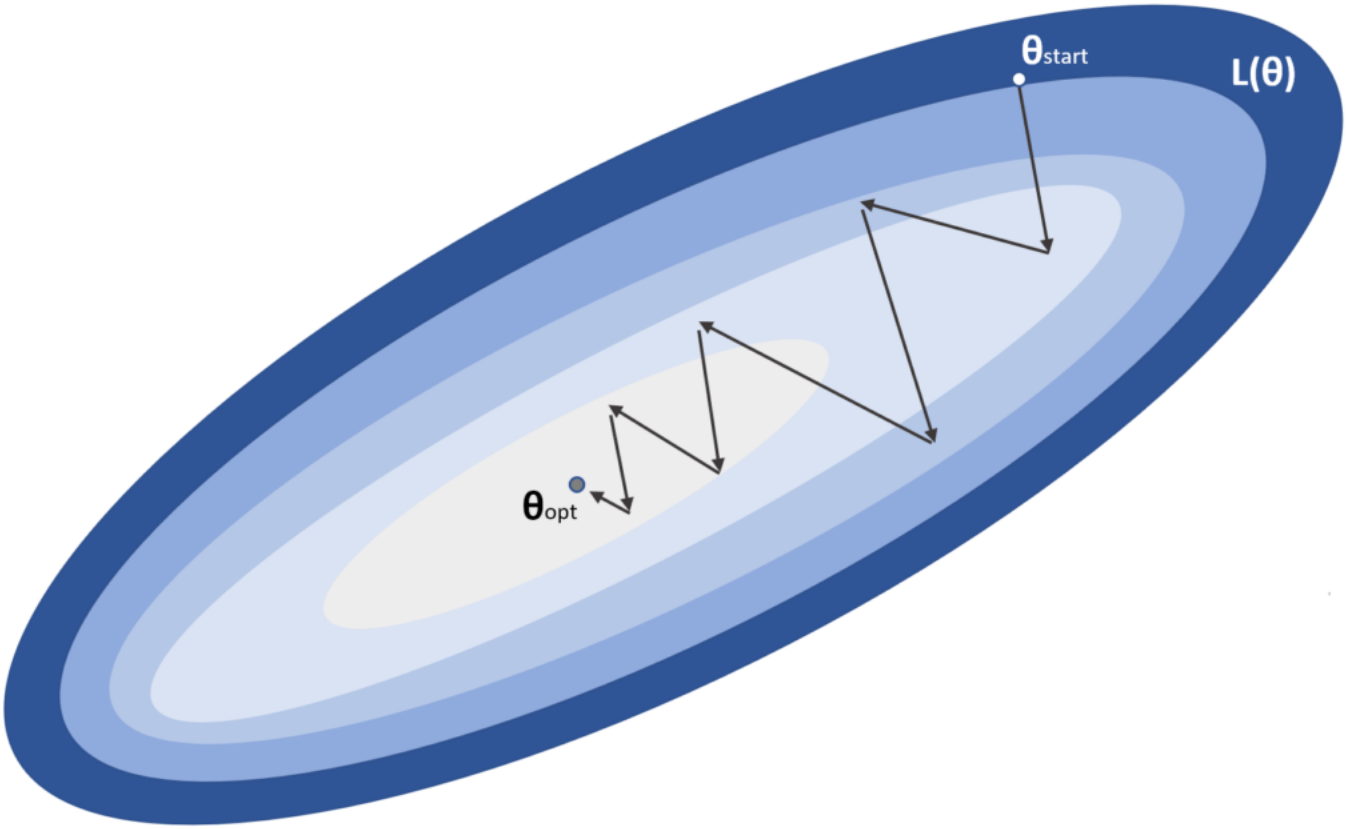
where x is the input sample, y is the label, and θ is the weight. We can define the partial derivative cost function for a batch size equal to N as:

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

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.

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

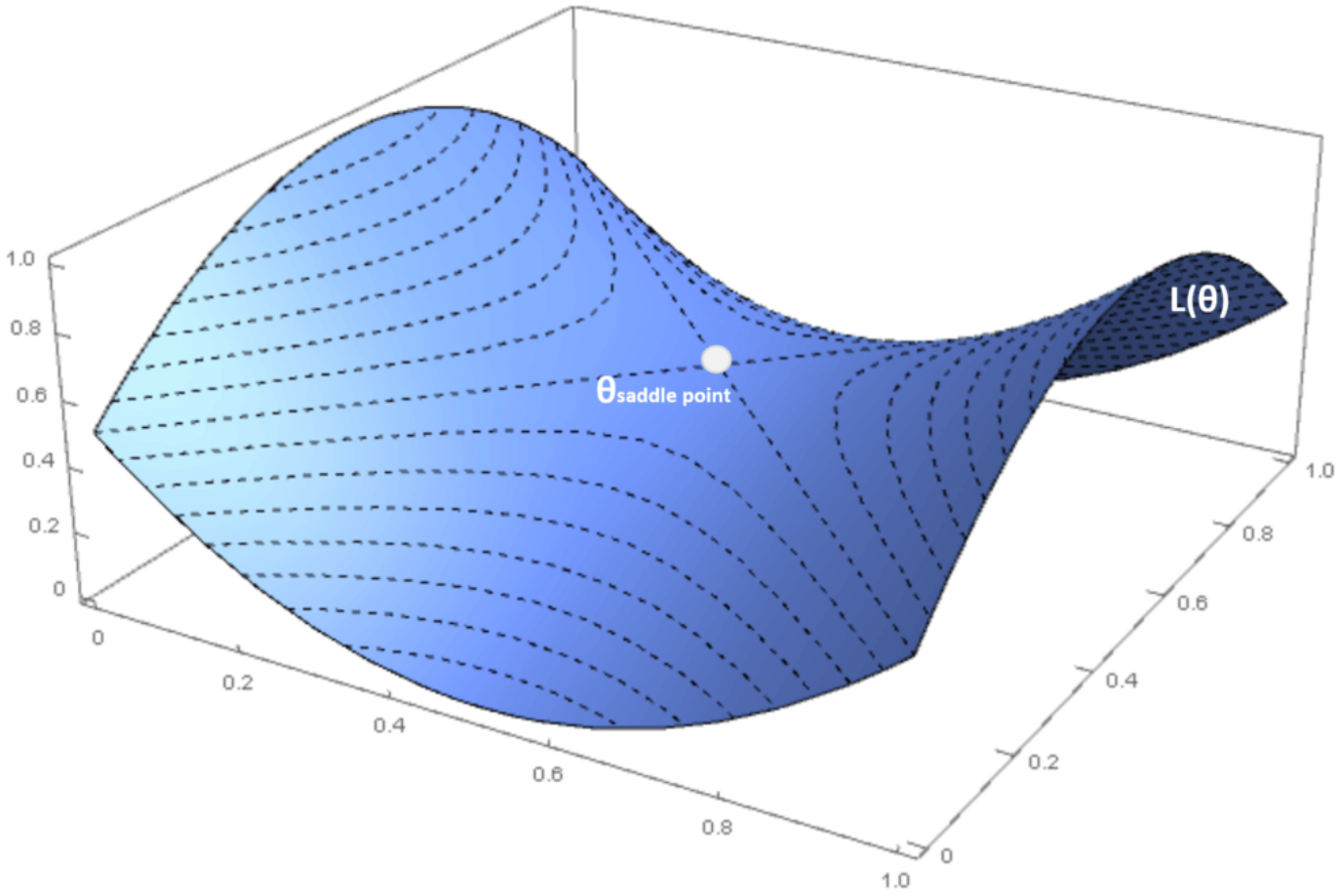
This process is represented in the following figure:



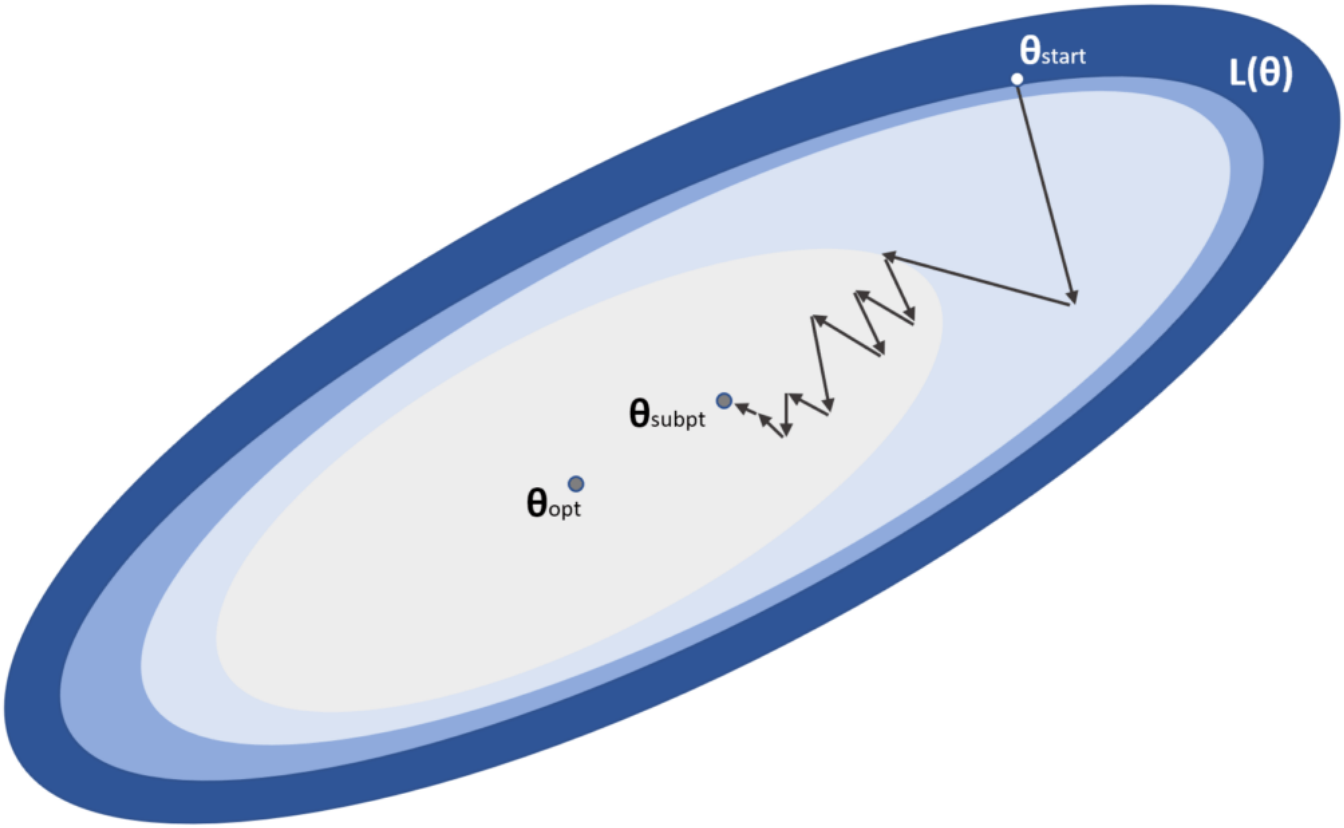
α is the learning rate, while θ_{start} is the initial point and θ_{opt} is the global minimum we're looking for.

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.

```
class AddGaussianNoise(object):
     def __init__(self, mean=0., std=1.):
          self.std = std
          self.mean = mean

     def __call__(self, tensor):
          return tensor + torch.randn(tensor.size()) * self.std + self.mean

     def __repr__(self):
          return self.__class__.__name__ + '(mean={0}, std={1})'.format(self.mean, self.std)

# then
transform = transforms.Compose([
     transforms.ToTensor(),
     ......,
     AddGaussianNoise(0., 1.), #good practise
])
```

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:

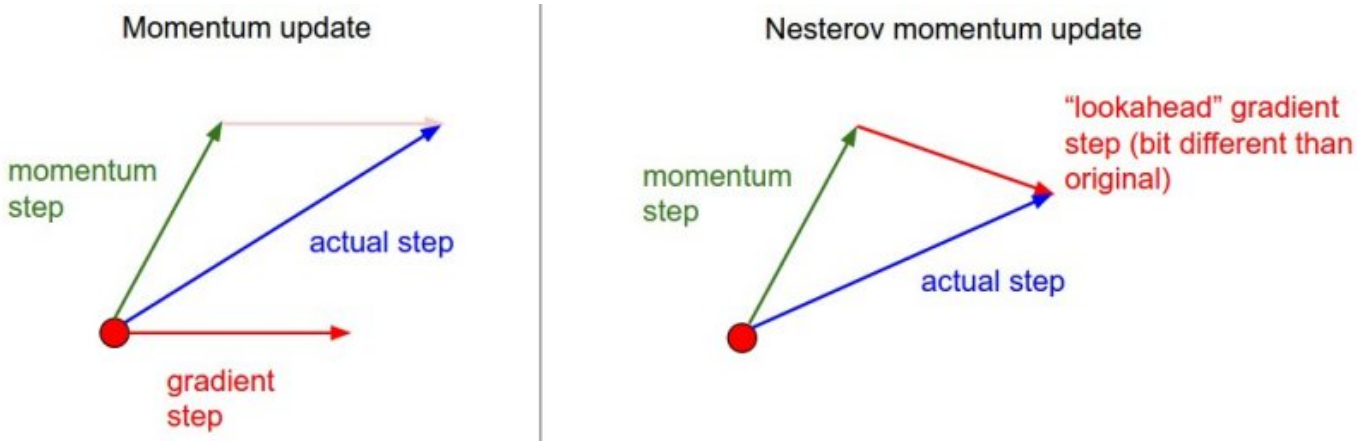
$$\begin{cases} v^{(t+1)} = \mu v^{(t)} - \alpha \nabla_\theta L(\bar{\theta}) \\ \bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} + v^{(t+1)} \end{cases}$$

The first equation computes the correction factor considering the weight μ.

If μ is small, the previous gradients are soon discarded. If, instead, $\mu \rightarrow 1$, their effect continues for a longer time.

A common value in many applications 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:

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

Mathematically awesome, but in deep learning contexts, doesn't seem to produce awesome results.

Image 2: SGD without momentum

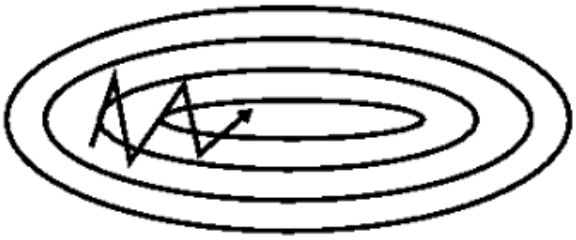


Image 3: SGD with momentum

RMS Prop

This algorithm, proposed by G. Hinton, is based on the idea to adapt the correction factor for each parameter, so to increase the effect on slowly-changing parameters and reduce it when their change magnitude is very large. This approach can dramatically improve the performance of a deep network, but it's a little bit more expensive than Momentum because we need to compute a *speed* term for each parameter:

$$v^{(t+1)}(\theta_i) = \mu v^{(t)}(\theta_i) + (1-\mu)\left(\nabla_\theta L(\bar{\theta})\right)^2$$

This term computes the exponentially weighted moving average of the gradient squared (element-wise).
Just like for Momentum, μ determines how fast the previous speeds are forgotten.
The parameter update rule is:

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

α is the learning rate and **δ** is a small constant (~ 1e-6 ÷ 1e-5) introduced to avoid a division by zero when the speed is null. As it's possible to see, each parameter is updated with a rule that is very similar to the *vanilla* Stochastic Gradient Descent, but the actual learning rate is adjusted per single parameter using the reciprocal of the square root of the relative speed. It's easy to understand that large gradients determine large speeds and, adaptively, the corresponding update is smaller and vice-versa. **RMSProp** is a very powerful and flexible algorithm and it is widely used in Deep Reinforcement Learning, CNN, and RNN-based projects.

Adam

Adam is an adaptive algorithm that could be considered as an extension of **RMSProp**. Instead of considering the only exponentially weighted moving average of the gradient square, it computes also the same value for the gradient itself:

$$\begin{cases} g^{(t+1)}(\theta_i) = \mu_1 g^{(t)}(\theta_i) + (1-\mu_1)\nabla_\theta L(\bar{\theta}) \\ v^{(t+1)}(\theta_i) = \mu_2 v^{(t)}(\theta_i) + (1-\mu_2)\left(\nabla_\theta L(\bar{\theta})\right)^2 \end{cases}$$

μ1 and μ2 are forgetting factors like in the other algorithms. The authors suggest values greater than 0.9. As both terms are moving estimations of the first and the second moment, they can be biased (see **this article (https://en.wikipedia.org/wiki/Unbiased_estimation_of_standard_deviation)** for further information). Adam provided a bias correction for both terms:

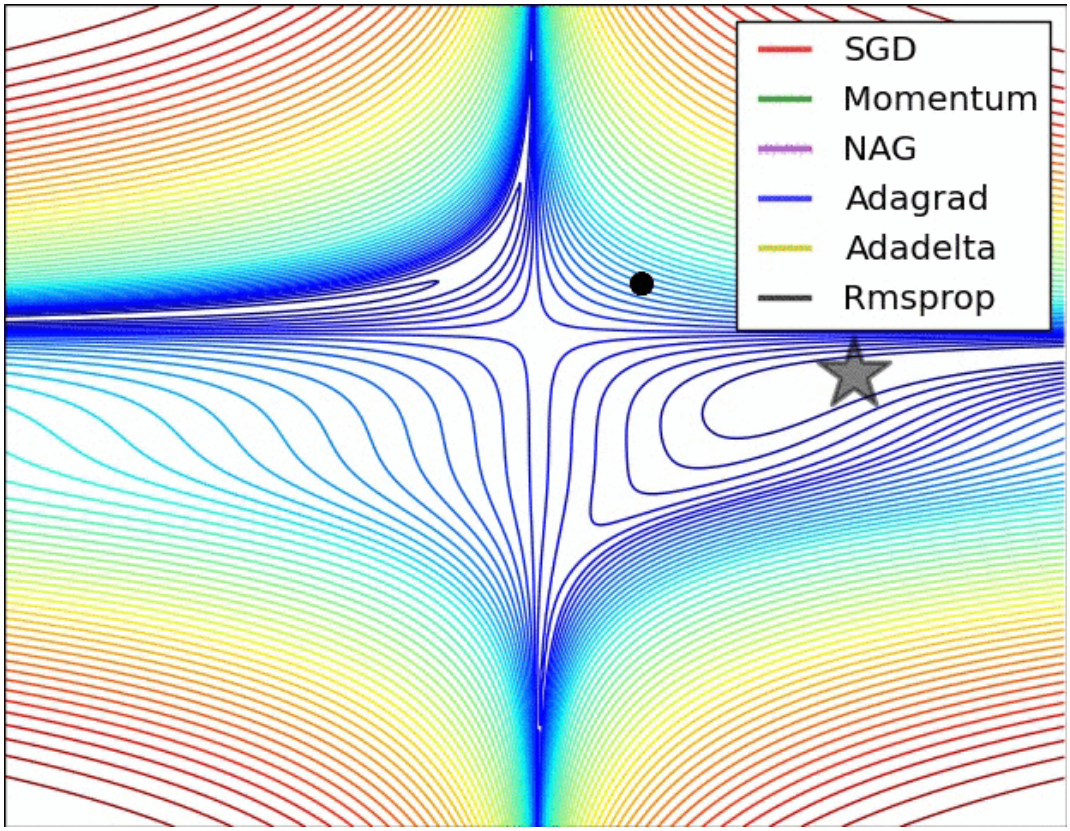
$$\begin{cases} \hat{g}(\theta_i) = \dfrac{g^{(t+1)}(\theta_i)}{1-\mu_1{}^{(t)}} \\ \hat{v}(\theta_i) = \dfrac{v^{(t+1)}(\theta_i)}{1-\mu_2{}^{(t)}} \end{cases}$$

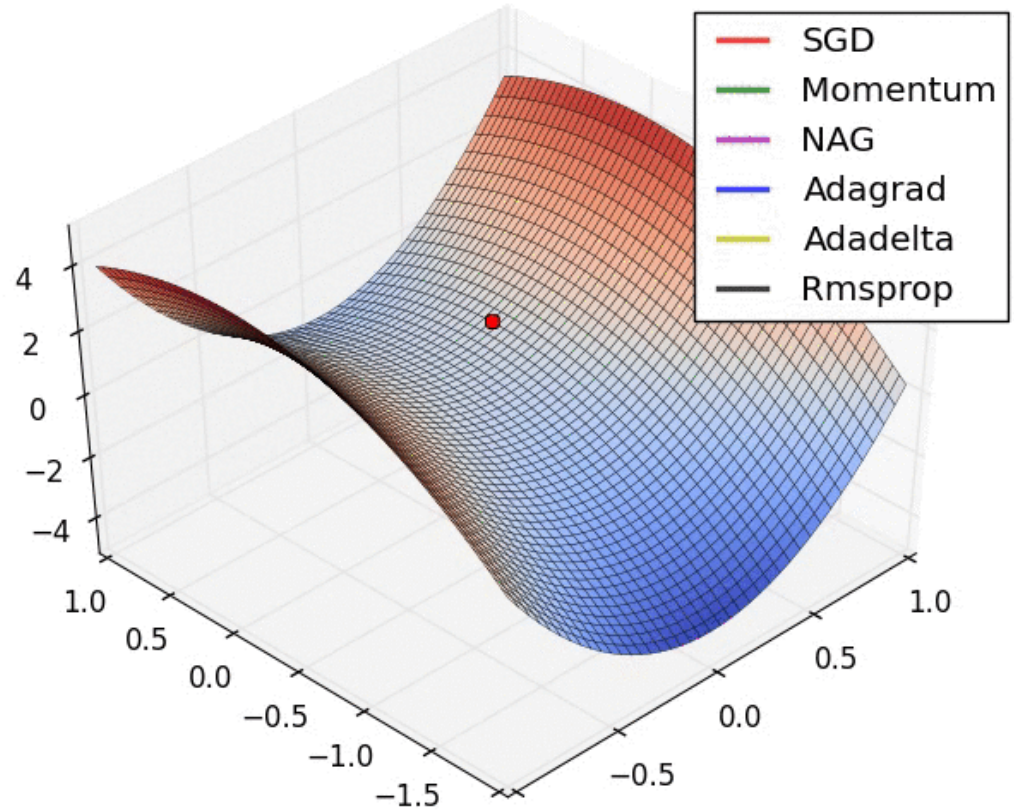
The parameter update rule becomes:

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \frac{\alpha g^{(t+1)}(\bar{\theta})}{\sqrt{v^{(t+1)}(\bar{\theta}) + \delta}}$$

Then we have AdaGrad and AdaDelta. Full **Source** **(https://www.bonaccorso.eu/2017/10/03/a-brief-and-comprehensive-guide-to-stochastic-gradient-descent-algorithms/)**

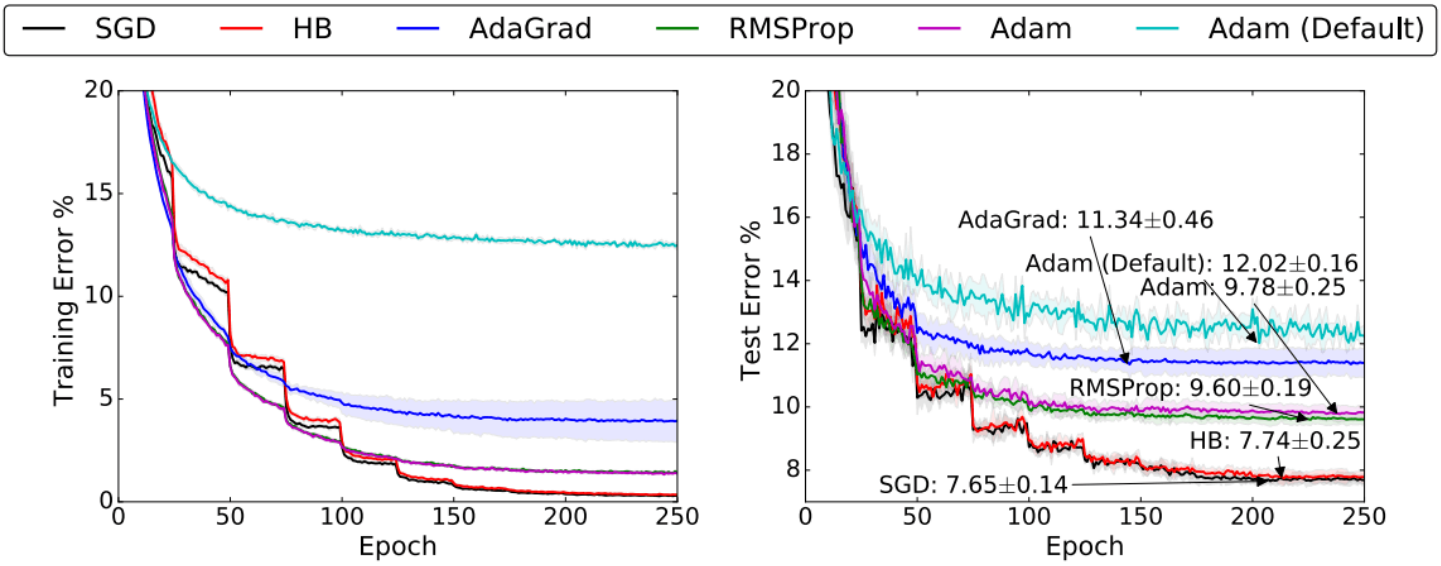
Which one to use?


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

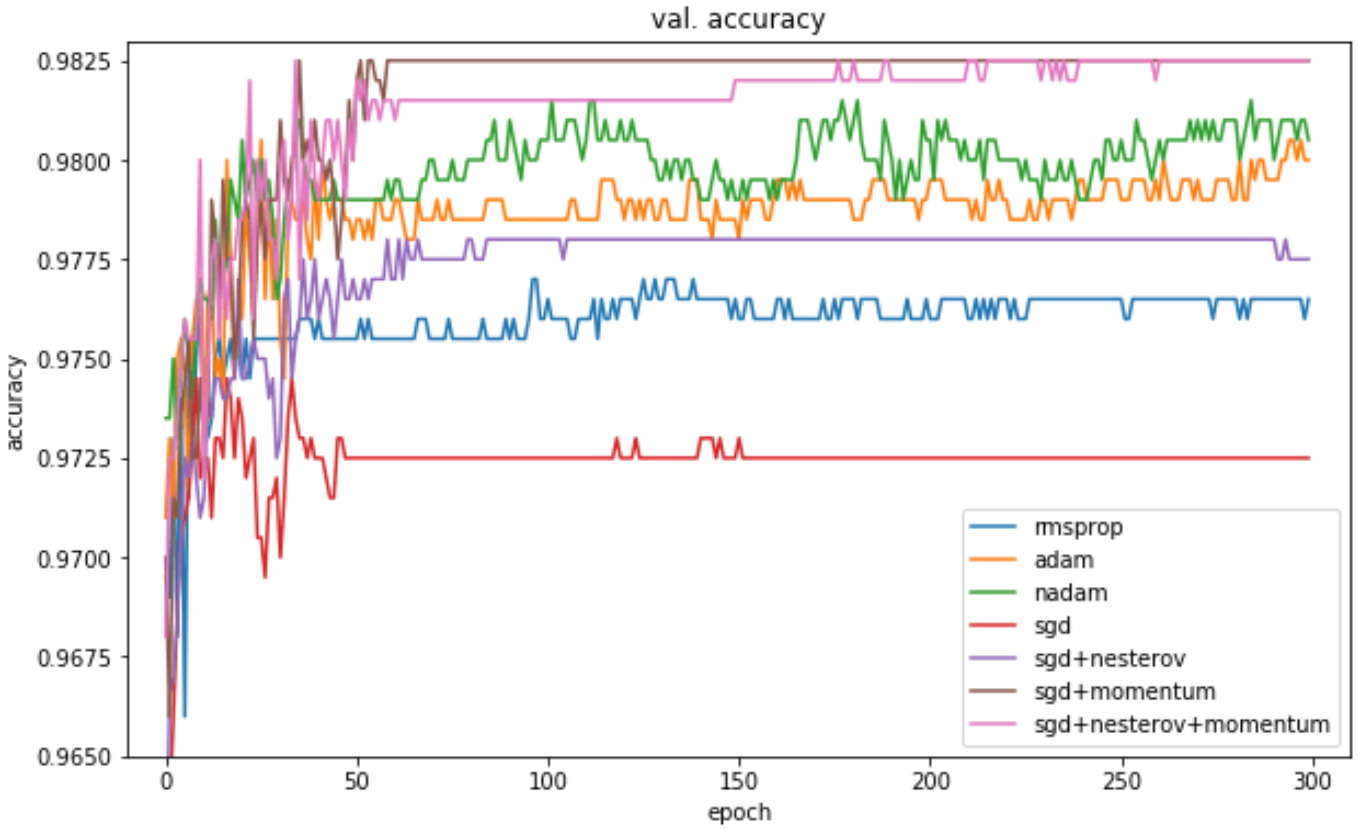

SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

Please note that the above image does not compare Adam.

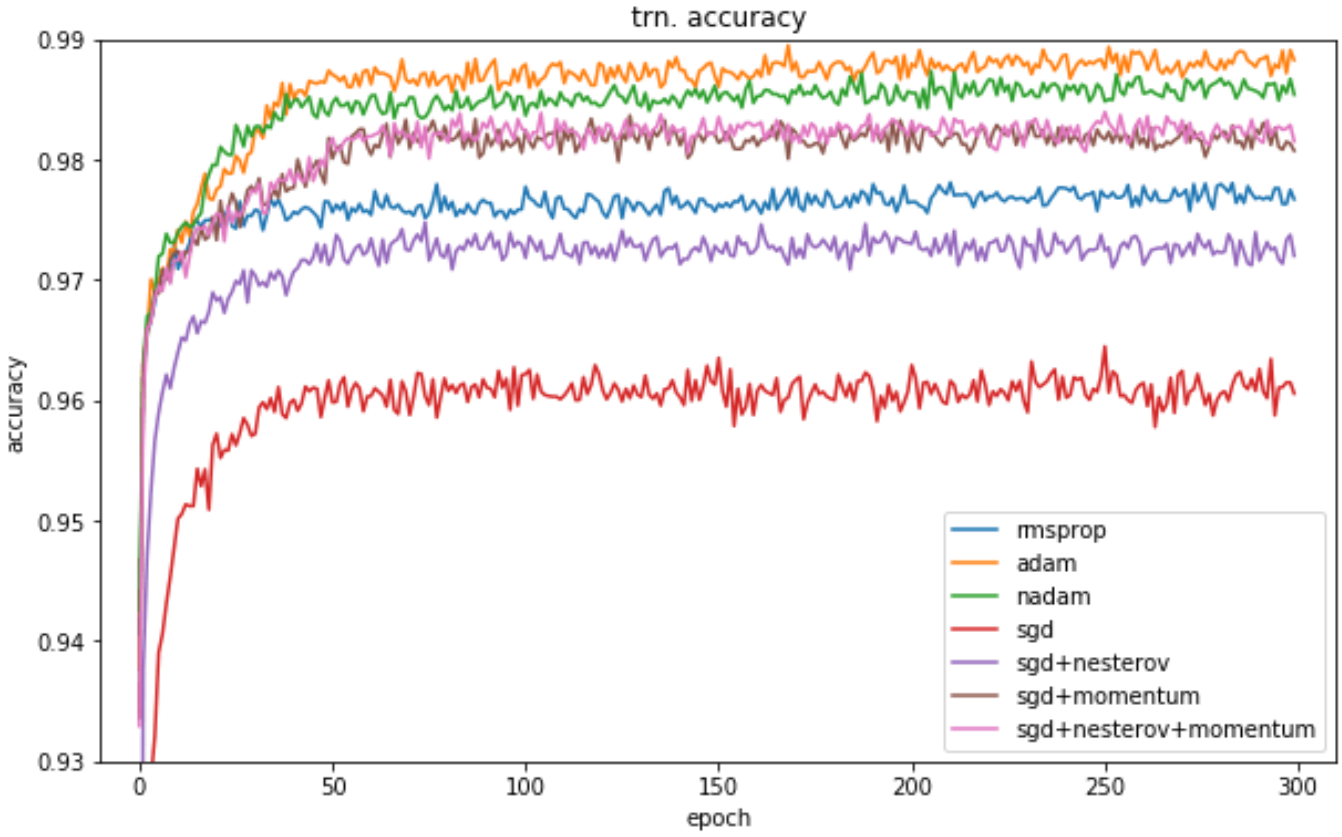
Here are some of the **execution** (https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/) results:



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


val. accuracy
accuracy
epoch
0.9825
0.9800
0.9775
0.9750
0.9725
0.9700
0.9675
0.9650
0
50
100
150
200
250
300
rmsprop
adam
nadam
sgd
sgd+nesterov
sgd+momentum
sgd+nesterov+momentum

SGD > ADAM



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.

Completely Different Approach

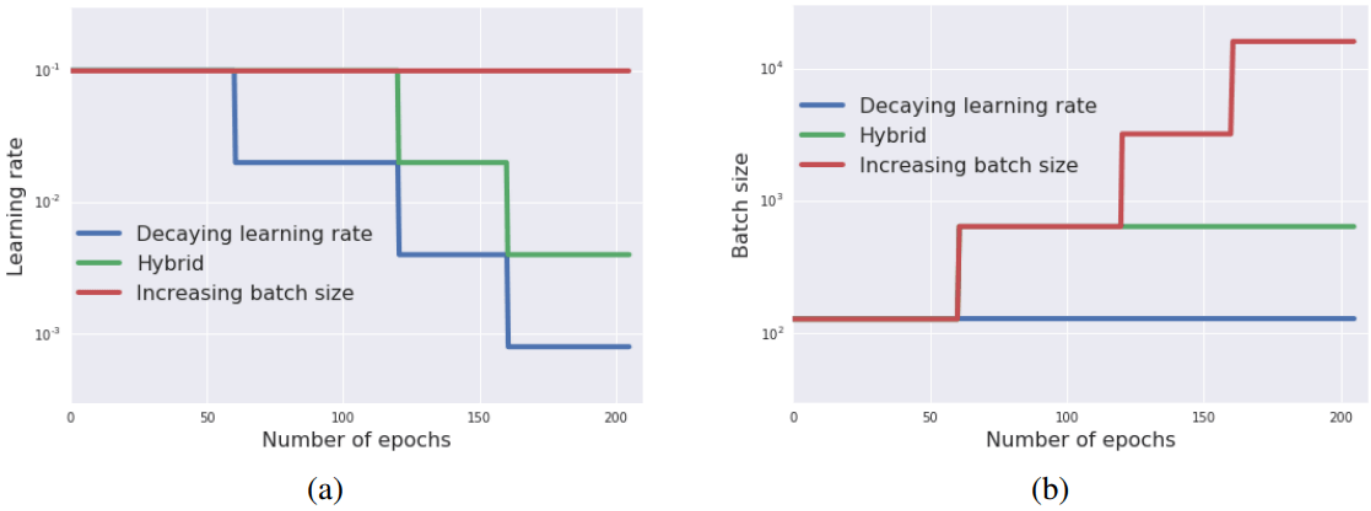
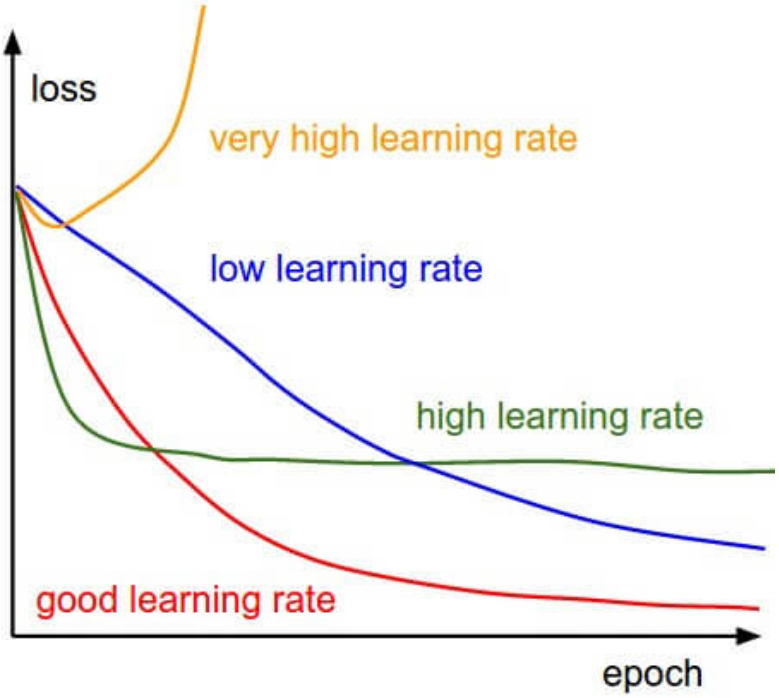


Figure 1: Schedules for the learning rate (a) and batch size (b), as a function of training epochs.

Don't decay the learning rate, increase the batch size!

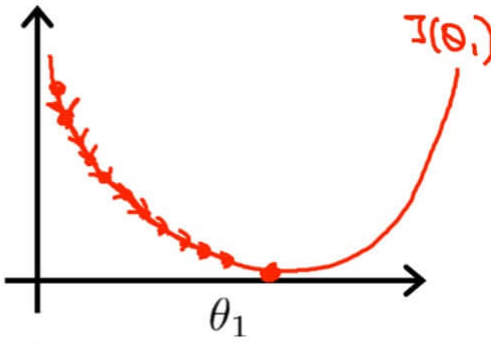
Learning Rate



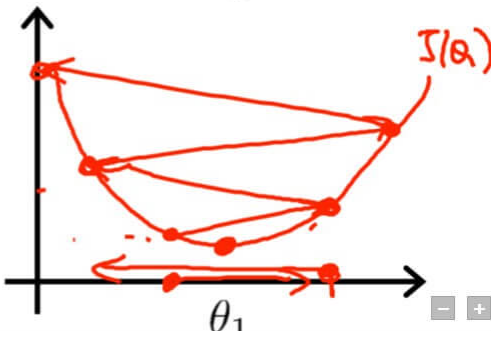
For all the above methods, we still need to find the learning rate.

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

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.



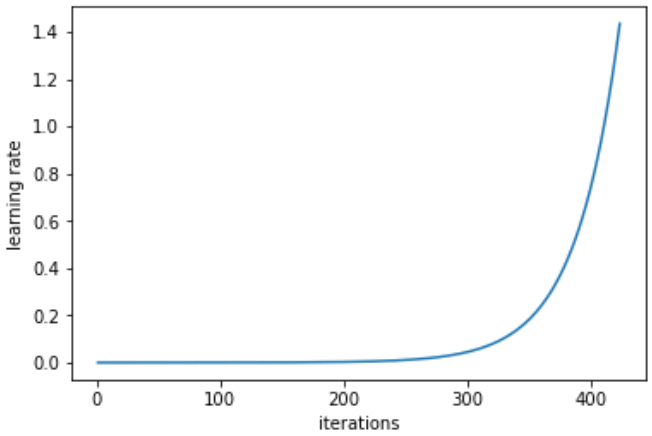
Source: Andrew Ng

The training should start from a relatively large learning rate in the beginning, as random weights are far from optimal, and then the learning rate can be decreased during training to allow for more fine-grained weight updates.

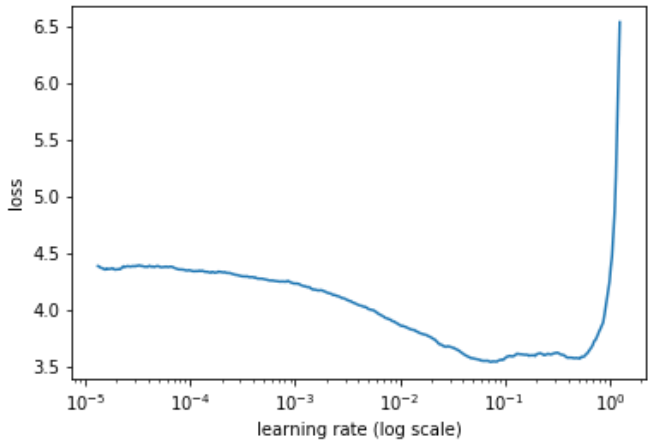
We can naively try different values or try with a smarter way.

Leslie N. Smith describes a powerful technique to select a range of learning rates for a neural network in section 3.3 of the 2015 paper "**Cyclical Learning Rates for Training Neural Networks (https://arxiv.org/abs/1506.01186)**" .

The trick is to train a network starting from a low learning rate and increase the learning rate exponentially for every batch.



Record the learning rate and training loss for every batch. Then, plot the loss and the learning rate. Typically, it looks like this:



First, with low learning rates, the loss improves slowly, then training accelerates until the learning rate becomes too large and loss goes up: the training process diverges.

We need to select a point on the graph with the fastest decrease in the loss. In this example, the loss function decreases fast when the learning rate is between 0.001 and 0.01.

Assignment:

1. Pick your last code
2. Make sure to Add CutOut to your code. It should come from your transformations (albumentations)
3. Use this repo: **https://github.com/davidtvs/pytorch-lr-finder** **(https://github.com/davidtvs/pytorch-lr-finder)**
    1. Move LR Finder code to your modules
    2. Implement LR Finder (for SGD, not for ADAM)
    3. Implement ReduceLROnPlatea: **https://pytorch.org/docs/stable/optim.html#torch.optim.lr_scheduler.ReduceLROnPlateau** **(https://pytorch.org/docs/stable/optim.html#torch.optim.lr_scheduler.ReduceLROnPlateau)**
4. Find best LR to train your model
5. Use SDG with Momentum
6. Train for 50 Epochs.
7. Show Training and Test Accuracy curves
8. Target 88% Accuracy.
9. Run GradCAM on the any 25 misclassified images. Make sure you mention what is the prediction and what was the ground truth label.
10. Submit

S10-Assignment-Solution Questions:

1. Paste your S10 Assignment's GitHub Link - 500PTS
2. Paste the link or upload Training and Test Curves (there should only be 1 graph)- 100PTS
3. What is the training accuracy of your model? - 150PTS
4. Share the link or upload an image of 25 misclassified images with GradCam results on top of them- 250PTS

Further Reading:

1. **Dropout on convolution layers is weird.** **(https://towardsdatascience.com/dropout-on-convolutional-layers-is-weird-5c6ab14f19b2)**
2. **An overview of gradient descent optimization algorithms (EXCELLENT)** **(http://ruder.io/optimizing-gradient-descent/index.html#whichoptimizertochoose)**
3. **Don't Decay the learning rate, increase the batch size!** **(https://arxiv.org/pdf/1711.00489v2.pdf)**

4. If your model is underfitting, remove dropout, and reduce learning rate and train **more (https://medium.com/@surmenok/fast-ai-what-i-learned-from-lessons-1-3-b10f9958e3ff)**
5. To reduce overfitting, add more data, use data augmentation.

Video: