A Survey Summary of Gradient Descent Optimization Algorithms

INTRODUCTION: Gradient Descent Optimization Algorithms are often used as Black - Box Optimizers, as practical explanations of their strengths and weaknesses are hard to come by. Here, we look at different variants of Gradient Descent, challenges, Algorithms for Gradient Descent, study different Architectures in a Parallel Manner, and see Strategies for optimizing Gradient Descent. Gradient descent (GD) is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks.

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters θ(belonging to Rd) by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with respect to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum, meaning follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.

Gradient Descent Variants: There are 3 variants of GD, which differ in how much data we use to compute the Gradient of the objective function. Depending on the amount of data, we make a trade-off between the Accuracy of the Parameter Update and the Time it takes to perform an update.

a) **Batch Gradient Descent** :- It computes the Gradient of the Cost Function w.r.t the parameters θ for the entire Dataset.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

Code :
For a pre-defined number of Epochs, we first compute the Gradient Vector of the Loss params_grad function for the whole dataset w.r.t. our parameter vector params. We then update our parameters in the direction of the gradients with the learning rate determining how big of an update we perform. Batch GD is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

```
for i in range ( nb_epochs ):
    params_grad = evaluate_gradient ( loss_function , data , params )
    params = params - learning_rate * params_grad
```

b) **Stochastic Gradient Descent** :- It performs a parameter update for each training example x(i) and label y(i):

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

Code :
Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily. Its code fragment simply adds a loop over the training examples and evaluates the gradient with respect to each example.

```
for i in range ( nb_epochs ):
    np. random . shuffle ( data )
    for example in data :
        params_grad = evaluate_gradient ( loss_function , example , params )
        params = params - learning_rate * params_grad
```

c) **Mini - Batch Gradient Descent** :- Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of 'n' Training Examples:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

It reduces the variance of the parameter updates, which can lead to more stable convergence; and can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.

```
for i in range ( nb_epochs ):
    np. random . shuffle ( data )
    for example in get_batches (data , batch_size = 50) :
        params_grad = evaluate_gradient ( loss_function , batch , params )
        params = params - learning_rate * params_grad
```

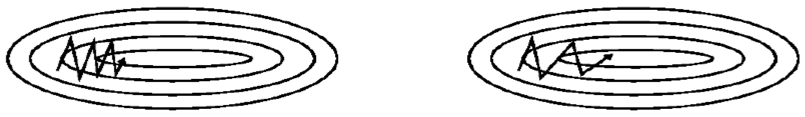
Challenges in Mini-Batch GD: There are several challenges in Mini-Batch GD, like **1)** Choosing a proper learning rate can be difficult. **2)** The Same Learning Rate applies to all parameter updates. **3)** Learning rate schedules try to adjust the learning rate during training, by reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics, and **4)** Getting Trapped in the suboptimal local minima.

GD Optimization Algorithms: Some Algorithms that are widely used by the Deep Learning community to deal with these challenges.

a) **Momentum:** Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Figure. It does this by adding a fraction γ of the update vector of the past time step to the current update vector.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$


(a) SGD without momentum (b) SGD with momentum

b) **Nesterov Accelerated Gradient (NAG):** However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We like a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.
We will use our momentum term γv_{t-1} to move the parameters θ. Computing $(\theta - \gamma v_{t-1})$ thus gives us an approximation of the next position of the parameters. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

c) **Adagrad:** Algorithm for Gradient - Based Optimization that adapts the Learning Rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with Sparse Data.
As Adagrad uses a different learning rate for every parameter θi at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we set $g_{t,i}$ to be the gradient of the objective function w.r.t. to the parameter θi at time step t: $g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$
One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

d) **Adadelta:** Adadelta is an extension of Adagrad that seeks to reduce its aggressive, Monotonically Decreasing Learning Rate. Instead of accumulating all past Squared Gradients, Adadelta restricts the window of accumulated Past Gradients to some fixed size 'w'. Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average **$E[g^2]_t$** at time step 't' then depends (as a fraction **γ** similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g^2$$

e) **RMSprop:** RMSprop in fact is identical to the first update vector of Adadelta. RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients.

f) **Adam:** Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients **v_t** like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients **m_t**, similar to momentum:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g^2$$

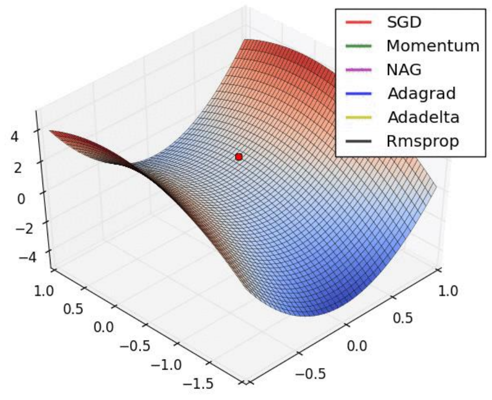
mt and vt are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As mt and vt are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. **β_1** and **β_2** are close to 1).

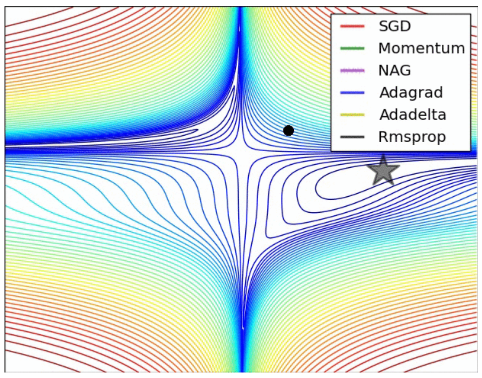
g) **Nadam:** Nadam (Nesterov-accelerated Adaptive Moment Estimation) combines Adam and NAG.

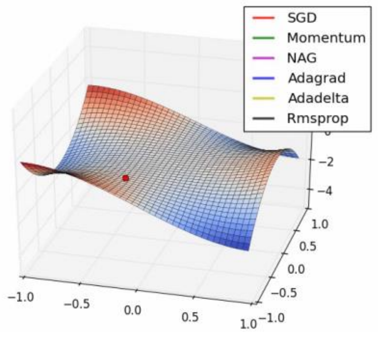
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right)$$

Final Equation -

Visualization of Algorithms: Optimization behaviour of different Optimization algorithms shown:







a) **Figure A**: The Path Algorithms took on the contours of a loss surface (the Beale Function). Adagrad, Adadelta, and RMSprop headed off immediately in the right direction and converged similarly fast, while Momentum and NAG were led off-track, evoking the image of a ball rolling down the hill. NAG, however, was able to correct its course sooner due to its increased responsiveness by looking ahead and headed to the minimum.

b) **Figure B:** The Behaviour of the Algorithms at a Saddle Point, i.e. A Point where One Dimension has a Positive Slope, while the Other Dimension has a Negative Slope, which pose a difficulty for SG. SGD, Momentum, and NAG find it difficulty to break symmetry, although the latter two eventually manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta quickly head down the negative slope, with Adadelta leading the charge.

Thus , the adaptive learning-rate methods, i.e. Adagrad, Adadelta, RMSprop, and Adam are most suitable and provide the best convergence for these scenarios.

Additional Strategies for Optimizing SGD: Some strategies for optimizing SGD are:

- Shuffling and Curriculum Learning
- Batch normalization
- Early stopping
- Gradient noise

Summary and Conclusions:

a) If your input data is sparse, then you likely achieve the best results using one of the adaptive learning-rate methods. An additional benefit is that you will not need to tune the learning rate but will likely achieve the best results with the default value.

b) RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numerator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. In short , RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. In short, **Adam might be the best overall choice.**

c) SGD usually achieves to find a minimum, but it might take significantly longer than with some of the optimizers, is much more reliant on a robust initialization and annealing schedule, and may get stuck in saddle points rather than local minima. Consequently, if you care about fast convergence and train a deep or complex neural network, you should choose one of the adaptive learning rate methods.