7 November, 2017

Neural Networks

Course 6: Making the neural network more efficient

Overview

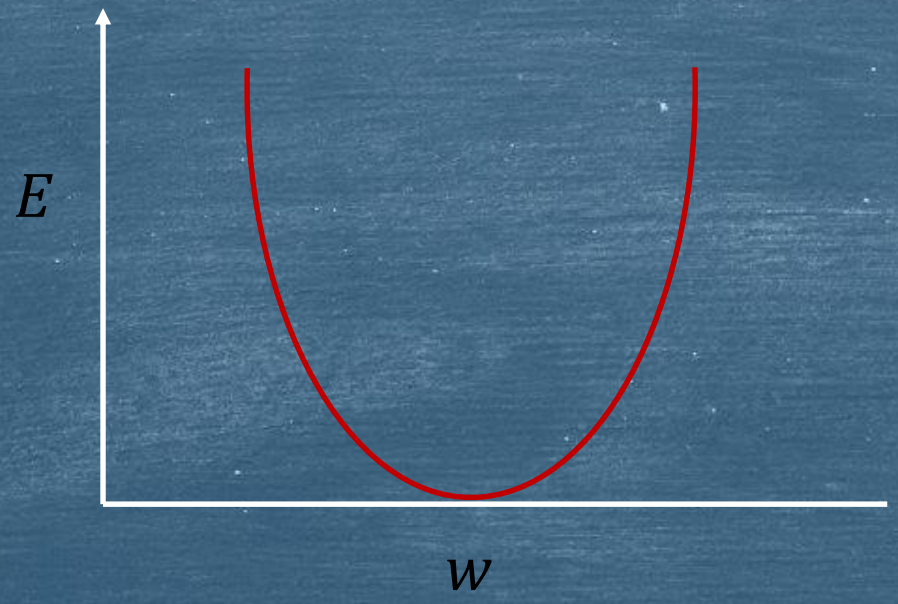
- Problems of SGD
- Momentum and Nesterov Acclerated Gradient
- RProp
- Adagrad
- RmsProp
- Adadelta
- Conclusions

Problems of SGD

Problems of SGD

- **How gradient descent works?**

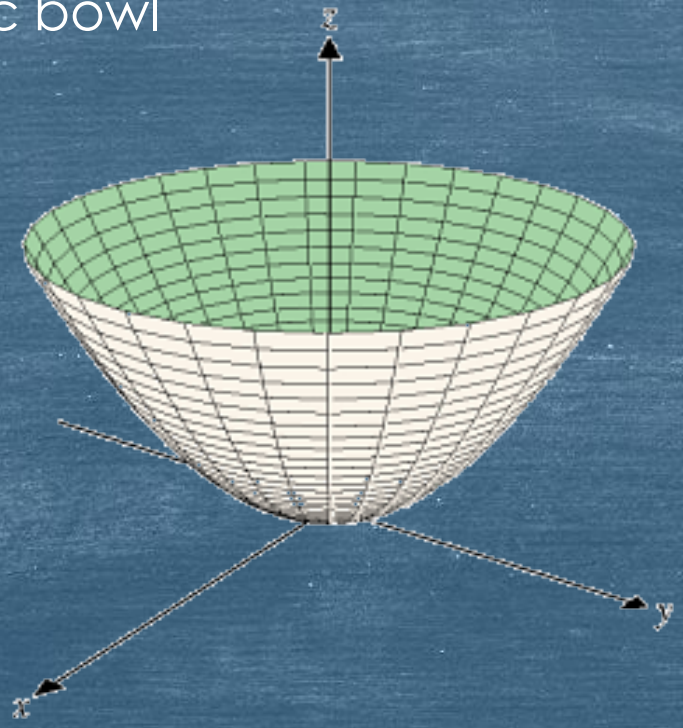
If we think about a linear neuron with only one weight, and compute the MSE ($\frac{1}{2n}\sum_x(t-y)^2$) w.r.t the output of the neuron and its target value, then the graph will be a parabola



Problems of SGD

- **How gradient descent works?**

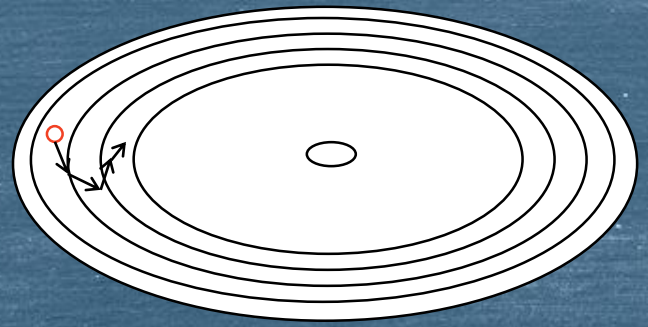
If we think about a linear neuron with two weights, then the graph turns into a quadratic bowl



Problems of SGD

- **How gradient descent works?**

If we take vertical section then we will get parabolas; if we take horizontal sections we will get elliptical contours, where the minimal error is at the center



When we apply back-propagation on these weights, we compute how the Error function is minimized in each of these (two) directions, combine the vectors and move in that direction

Problems of SGD

- **Problems with SGD?**

It's obvious that we want to go downhill, right to the center; but the gradient will point in that direction only if the ellipse is a circle. (which rarely happens)

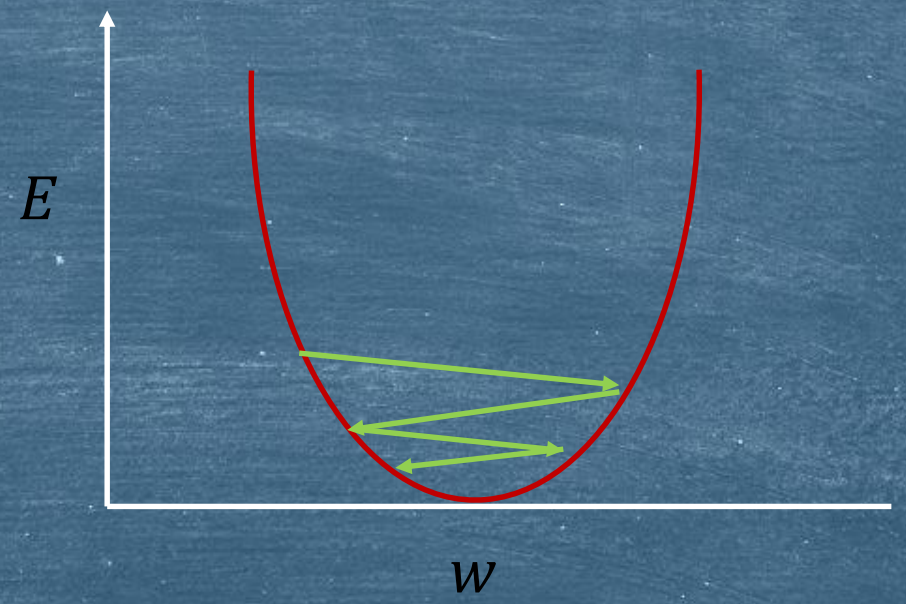
Since we make a move that is direct proportional to the size of the gradient, it is possible to move a big step in the wrong direction (across the ellipse) and a little step towards the good direction (along the ellipse)

Problems of SGD

How learning rate affects error minimization ?

If we use a small learning rate, then there will not be an important movement in the correct direction

If we use a large learning rate, then the error will oscillate (overshoot) across the parabola



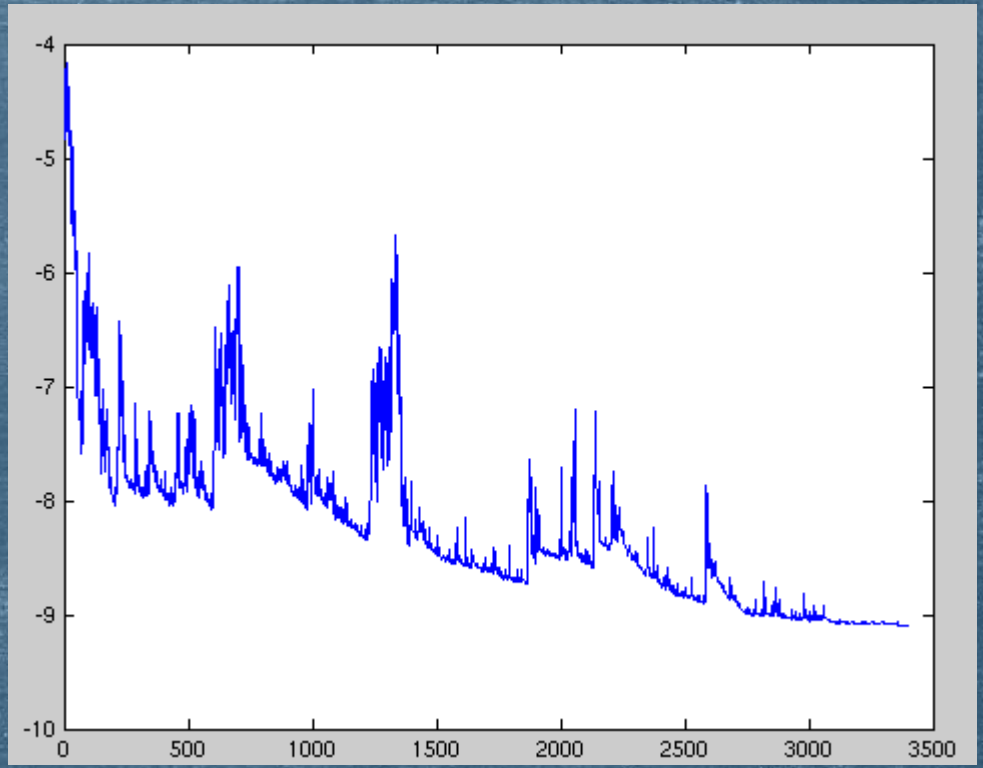
Problems of SGD

How learning rate affects error minimization ?

This happens especially for SGD (where the updates are performed only on a part of the data).

This can bring big oscillations in the Error functions (which sometimes can be beneficial).

A solution is to gradually reduce the learning rate



Fluctuation in the total objective function as gradient steps w.r.t. mini-batches

Source: wikipedia

Problems of SGD

- **Decreasing learning rate**

Decreasing the learning rate reduces fluctuations when using minibatches.

However, this also means a reduced speed in learning, so it shouldn't be done too soon.

A good criteria is to monitor the error on a validation set. If the errors drops gradually and consistent, then the learning rate can be increased.

If the error doesn't drop anymore, the learning rate should be decreased

Momentum

Momentum

- Variable Learning Rate: Momentum
  - The learning process can be viewed as having velocity and friction.
  - Each time the gradient points in the same direction, learning accumulates velocity
  - In order to reduce having too large gradients, we introduce friction.

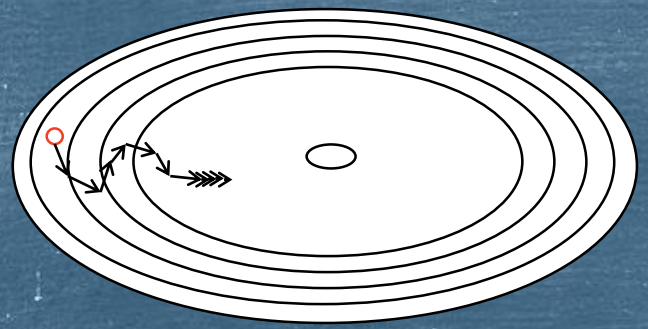
$$v_{ij} = \mu v_{ij} - \eta \frac{\partial C}{\partial w_{ij}}$$
$$w_{ij} = w_{ij} + v_{ij}$$

$where\ \mu\ is\ the\ friction.\ a\ value\ between\ [0,1]$

  - If $\mu = 0$ then we have the same update rule as before.
  - In literature μ is called *momentum co-efficient*

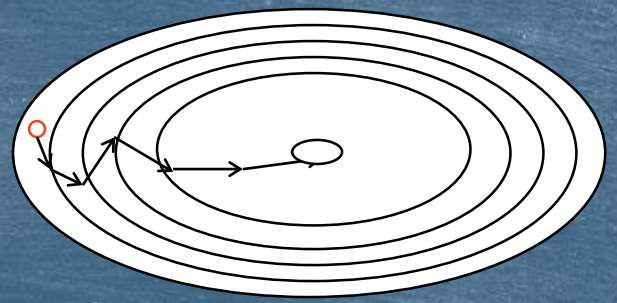
Momentum

- Simple SGD



Since steps are taken according to the gradient, it may take far too many steps for the SGD to reach the minimum

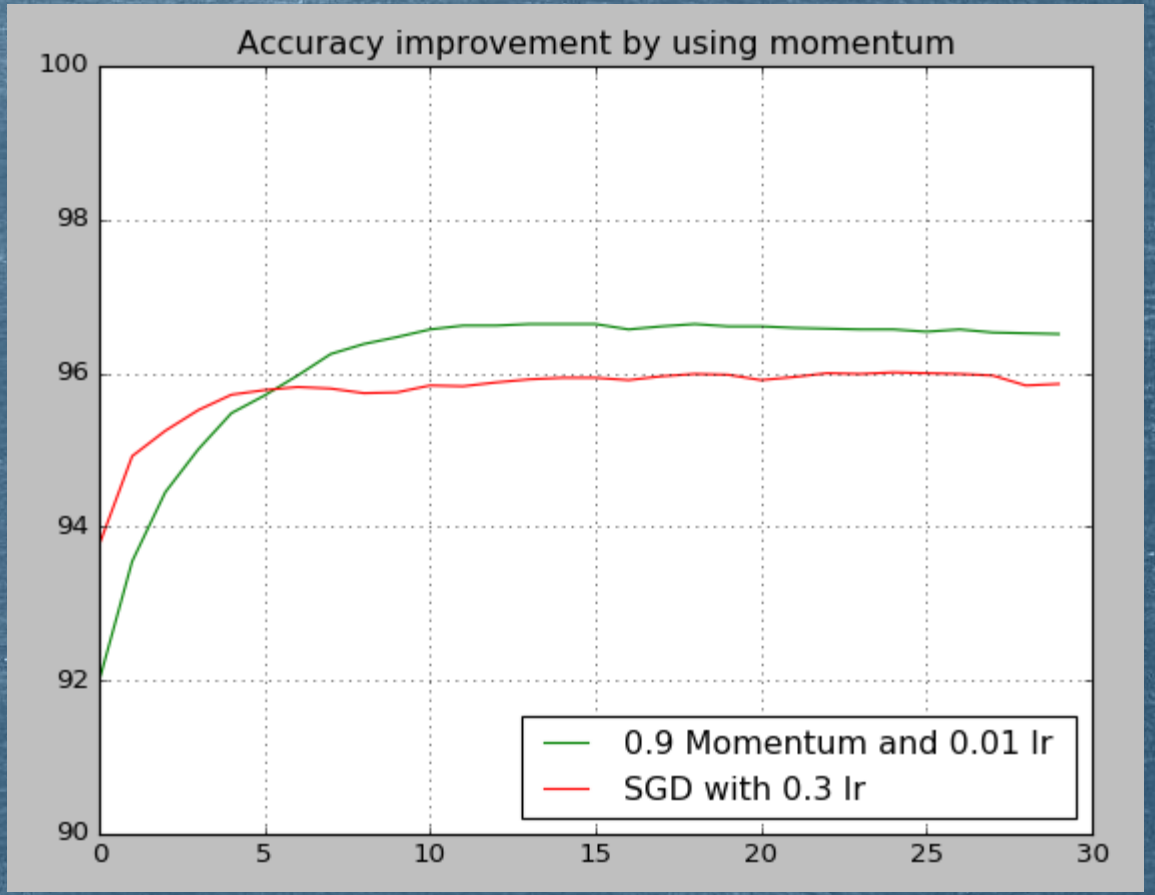
- SGD with momentum



- With momentum, small but steady steps build up and the gradients in opposite directions cancel out

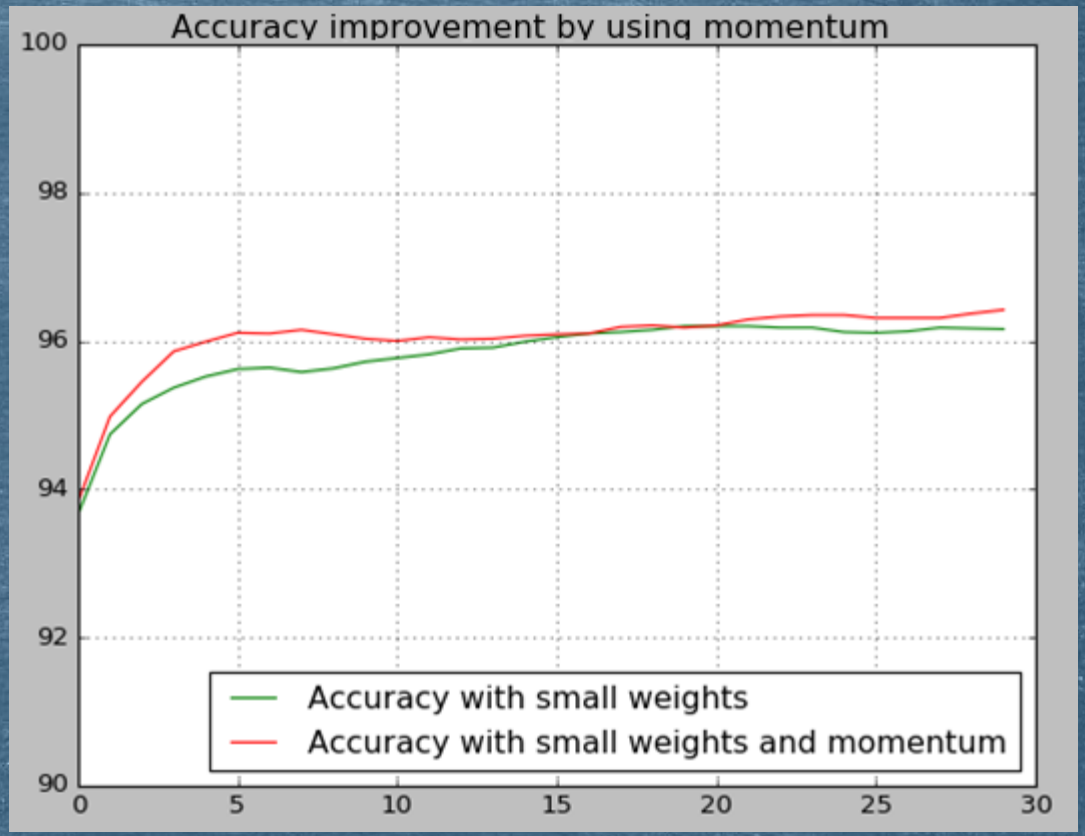
Momentum

- Depending on the problem, Momentum can speed-up the learning process or can improve accuracy
- Using a 0.9 momentum and 0.01 learning rate, the ANN for our problem (classifying mnist digits) using cross etntropy and softmax achieved a higher accuracy



Momentum

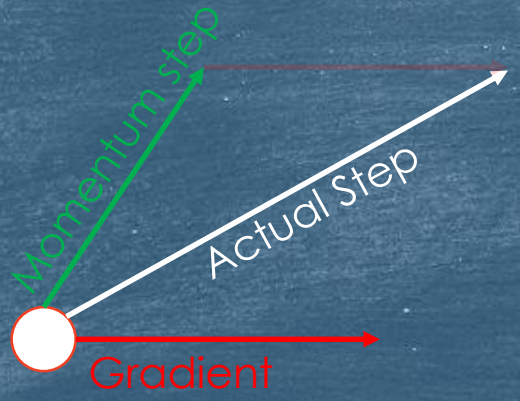
- If softmax isn't used momentum did not really change the accuracy of the model
- However, using momentum the network has arrived to the best accuracy on about iteration 5, as apposed to iteration 17 when momentum was not used.
- So using momentum is a good way to speed learning



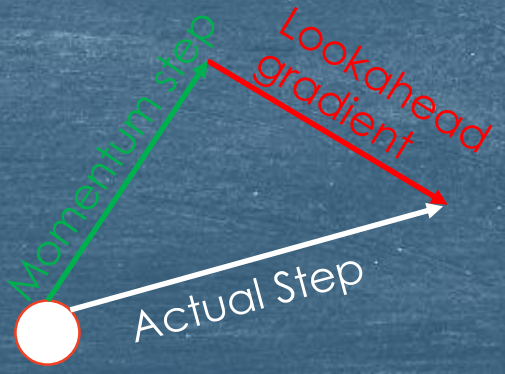
Nesterov Accelerated Gradient

- Variable Learning Rate: Nesterov Accelerated Gradient
  - NAG is really a modified version of momentum
  - The difference is that at each moment, we can look ahead and approximate where the momentum will take us on the next move.
  - Based on that approximation (which could be slightly off course) we can better adjust our direction

Momentum



Nesterov Accelerated Gradient



Nesterov Accelerated Gradient

- Variable Learning Rate: Nesterov Accelerated Gradient

$$v_{ij}^{t} = \mu v_{ij}^{t-1} - \eta \frac{\partial C}{\partial \theta_{ij}^{t-1}}, where\ \theta_{ij}^{t-1} = \left(w_{ij}^{t-1} + \mu v_{ij}^{t-1}\right)$$

$$w_{ij}^{t} = w_{ij}^{t-1} + v_{ij}^{t}$$

- In practice, people prefer to express the update similar to the classic momentum. Thus, the second equation is altered

$$\theta_{ij}^{t} = \left(w_{ij}^{t} + \mu v_{ij}^{t}\right) \rightarrow \theta_{ij}^{t} = \left(w_{ij}^{t-1} + v_{ij}^{t} + \mu v_{ij}^{t}\right)$$

$$w_{ij}^{t-1} = \theta_{ij}^{t-1} - \mu v_{ij}^{t-1}$$

$$\theta_{ij}^{t} = \theta_{ij}^{t-1} - \mu v_{ij}^{t-1} + \mu v_{ij}^{t} + v_{ij}^{t}$$ Assuming $v^0 = 0,\ \theta^t = w^t$

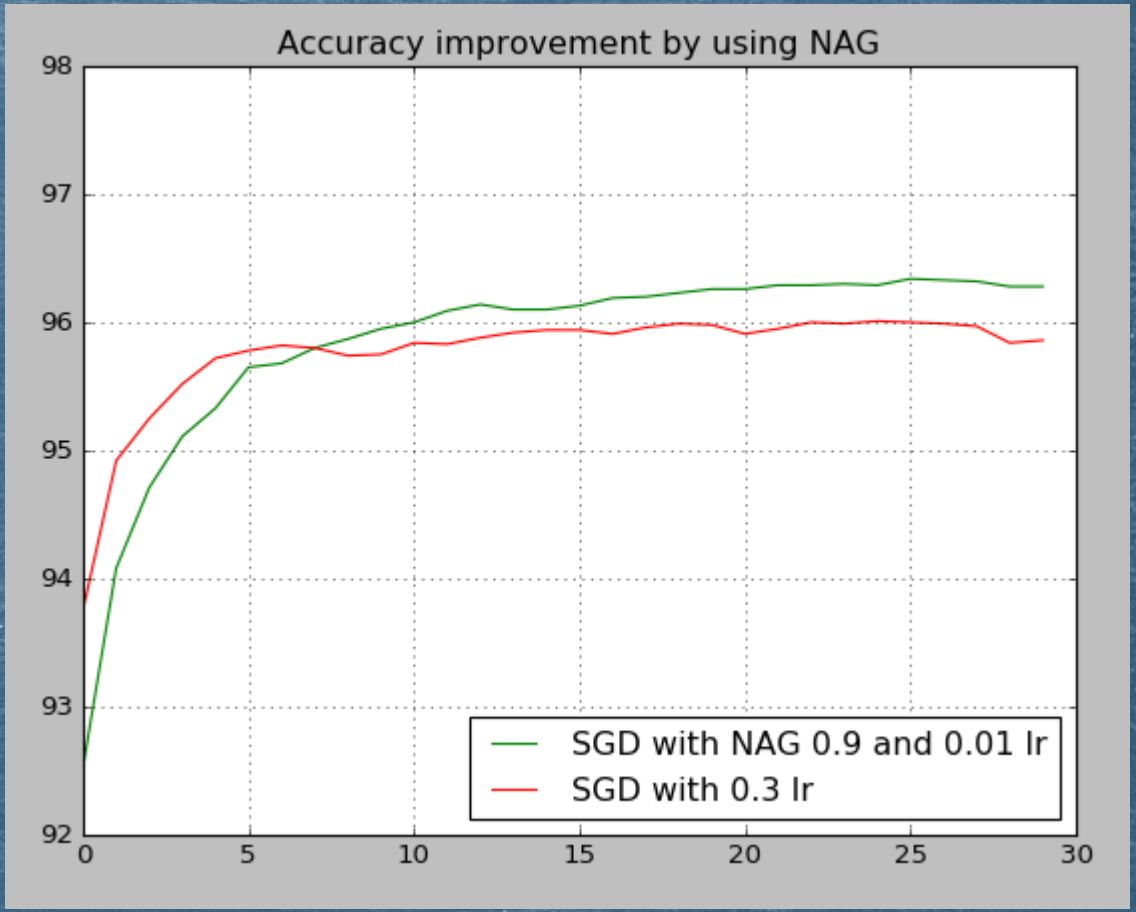
- The equations for Nesterov accelerated gradient becomes:

$$v_{ij}^{t} = \mu v_{ij}^{t-1} - \eta \frac{\partial C}{\partial w_{ij}^{t-1}}$$

$$w_{ij}^{t} = w_{ij}^{t-1} - \mu v_{ij}^{t-1} + \mu v_{ij}^{t} + v_{ij}^{t}$$

Nesterov Accelerated Gradient

- NAG is considered a better version of standard momentum and should be almost always be used instead of standard momentum
- The reason why NAG is superior is that it corrects faster due to its lookahead feature



Rprop

Resilient Propagation

Rprop

- Only considers the sign of the gradient
- For each parameter of the network, it uses another parameter, $\Delta\theta_i$ that stores the size of movement in the direction provided by the gradient
- It remembers the sign of the gradient of the previous iteration and compares to the sign of the current gradient
  - If the sign is the same, that means the followed direction is good.

    In that case, $\Delta\theta_i$ is increased (usually, multiplied by 1.2)
  - If the sign differ, that means the followed direction is wrong

    In that case, the previous position is restored ($-\Delta\theta_i$) and $\Delta\theta_i$ is decreased (usually, multiplied by 0.5)

Rprop

- Some maximum and minimum values are defined:
  - Usually, maximum is 50
  - Minimum is 1e-6
- The initial values for $\Delta\theta_i$ must be different than 0, usually 0.01
- Advantages of Rprop:
  - Since the update is independent of the size of the gradient, it can escape plateaus very quickly (just like momentum)
  - There is no need to tune a learning rate or a momentum factor
  - One of the fastest algorithms available

Rprop

- Disadvantages of Rprop:
  - Increased memory since it must maintain the previous gradients and the $\Delta\theta_i$ for each parameter of the network
  - It doesn't work with minibatches (not suitable for large datasests)

  For SGD, if a small (constant) learning rate was used, and we have nine minibatch gradients of +0.1 and one gradient of -0.9, they will cancel each other (weight stays the same for the entire dataset)

  On Rprop, we will increase the step size 9 times, and decrease it once

Adagrad

Adaptive gradient

Adagrad

- Adagrad adapts each learning rate to the size of gradient
- Big gradients will receive small learning rates, while small gradients will receive a bigger learning rate
- It does that by dividing each gradient by an L2 norm of past gradients, for that particular parameter

$$\eta_i^t = \frac{\eta}{\sqrt{\sum_{k=1}^{t}(\nabla\theta_i^k)^2}}$$

$$\theta_i^t = \theta_i^{t-1} - \eta_i^t \nabla\theta_i^t$$

Where

θ_i^t is the value of a parameter i (weight or bias) at time t

$\nabla\theta_i^t$ is the gradient of the parameter θ_i at time t

η_i^t is the learning rate for the parameter θ_i at time t

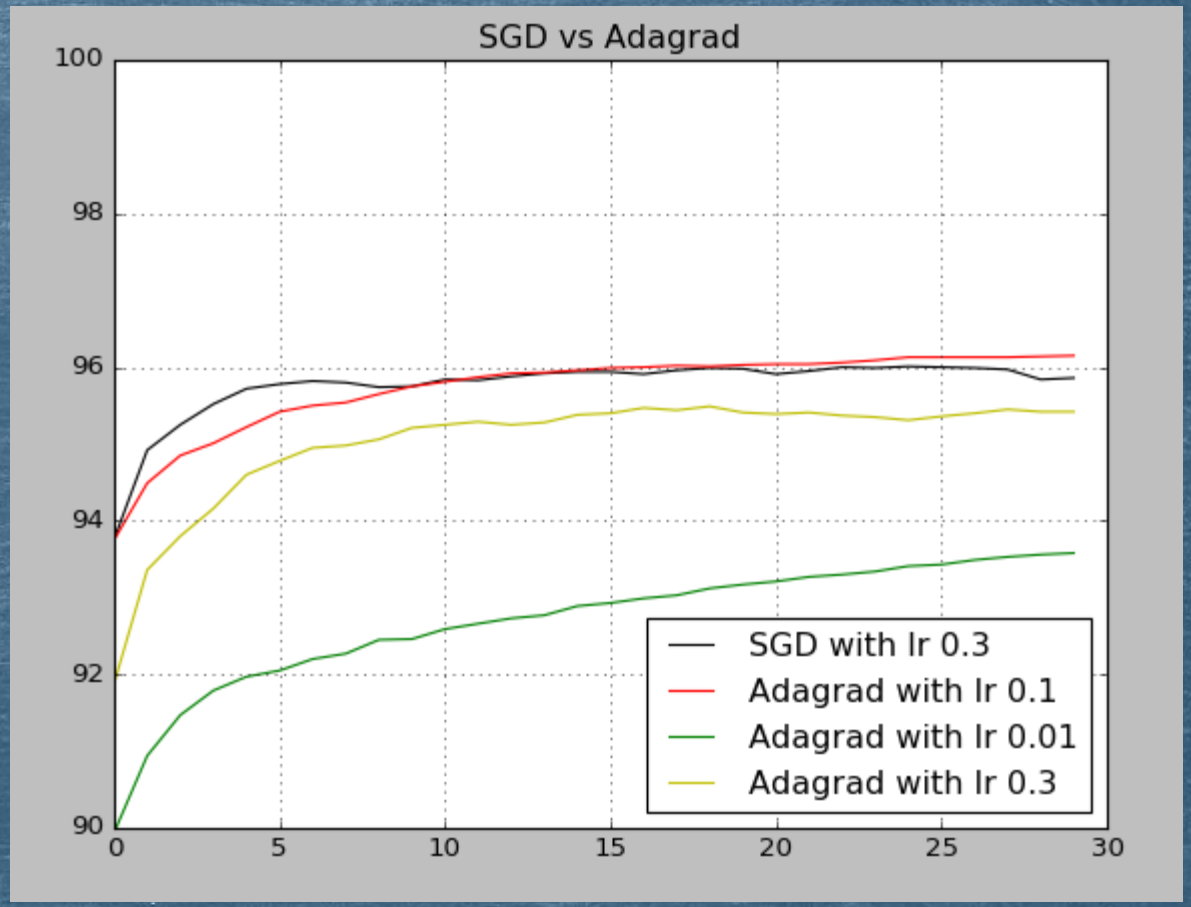
Adagrad

- One of the most important benefits of Adagrad is that it advantages sparse features.
  - The features that appear rarely through the dataset will be adjusted with a higher learning rate than those that appear often
  - The most frequent features, with large gradients, will eventually drive the gradient to zero (adaptive gradient)
- One of the main disadvantages of Adagrad is that it can become blocked on local minimum.
  - This happens since the L_2 norm can only increase

Adagrad

- Although many prefer to use a standard learning rate of Adagrad (0.01), the starting point is important since it defines how long will the adagrad continue to learn

- In the case of MNIST classification, using sigmoid activation, Adagrad did not improve the accuracy



Rmsprop

Root mean square propagation

RMSProp

- Rmsprop builds on Adagrad.
- It still uses the same idea (separately adjust learning rate on each weight, based on recently past gradients) but instead of keeping track of all past gradient, Rmsprop uses an exponential running average
- On each iteration, it divides the learning rate by an exponential moving average of the squared gradient

$$E\left[\left(\nabla\theta_i^t\right)^2\right] = \gamma E\left[\left(\nabla\theta_i^{t-1}\right)^2\right] + (1-\gamma)\left(\nabla\theta_i^t\right)^2$$

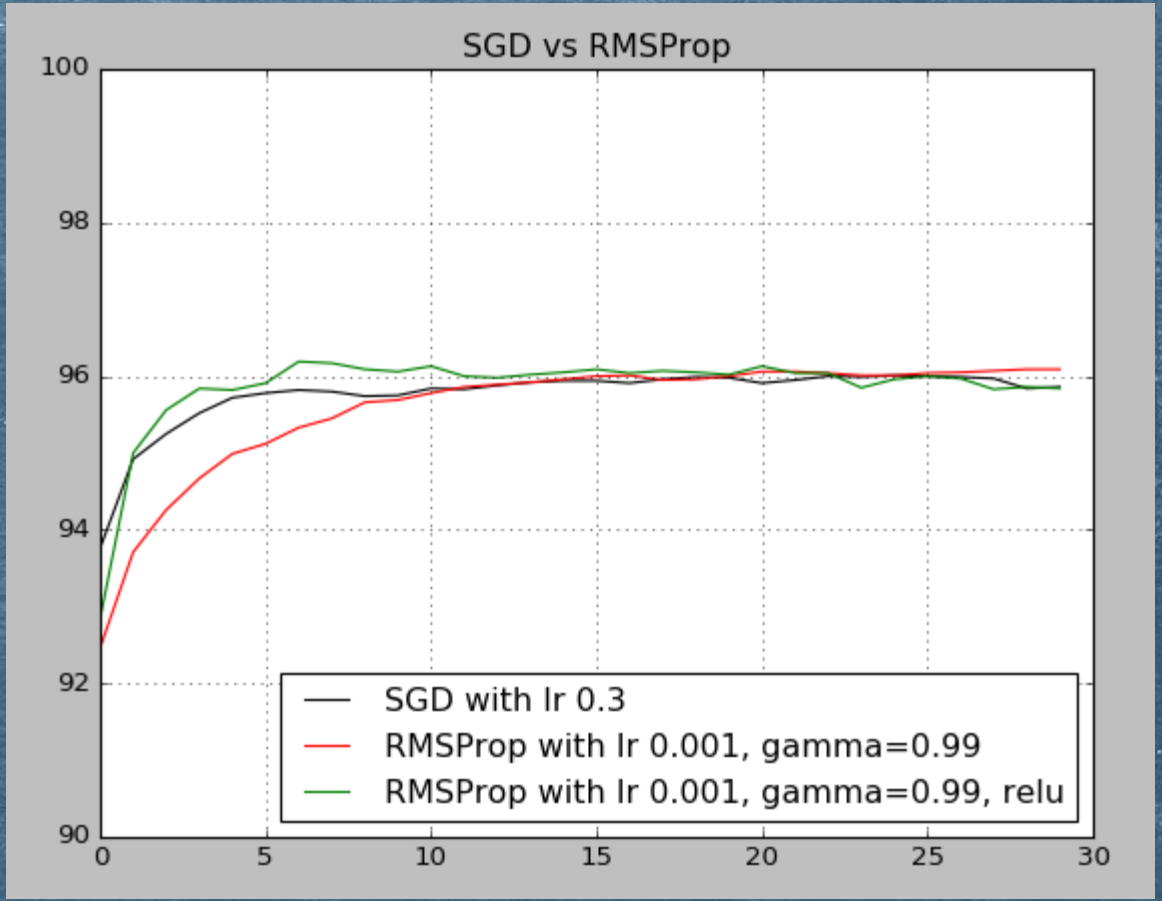
$$\eta_i^t = \frac{\eta}{\sqrt{E\left[\left(\nabla\theta_i^t\right)^2\right]+1^{-8}}}$$

$$\theta_i^t = \theta_i^{t-1} - \eta_i^t \nabla\theta_i^t$$

RMSProp

- RMSProp is one of the most used methods in neural network
- When tested on MNIST Digits, it didn't improve the accuracy (when compared with SGD),
- Nowadays, most neural neworks use relu neurons instead of sigmoid neurons. When tested with relu neurons, RMSProp achieved a higher accuracy

SGD with relu performed much worse, so it is not included in the graph



Adam

Adaptive Moment Estimation

Adam

- Adam builds on Rmsprop and Momentum
- It uses separate learning rates for each weight, divided by the norm of previous gradients (just like RMSProp)
- It also uses a factor that is similar to momentum (it builds on previous gradients)

$$\mathrm{M_t} = \beta_1 M_{t-1} + (1-\beta_1)\nabla\theta_{\mathrm{i}}^t$$

$$R_t = \beta_2 R_{t-1} + (1-\beta_2)\left(\nabla\theta_i^t\right)^2$$

M_t is the estimate of the momentum

R_t is the exponentially decaying average of squared gradients

Usual values for β_1 and β_2 are 0.9 and 0.999

Adam

- Since M_0 and R_0 are initialized to 0, and β_1, β_2 are close to 1, the values of M_t and R_t are biased towards 0, during the first time steps.

- To counteract these biases, they further divide M_t and R_t by a correction factor:

$$\widehat{M_t} = \frac{M_t}{1 - \beta_1^t}$$

$$\widehat{R_t} = \frac{R_t}{1 - \beta_2^t}$$

The update is performed similar to RMSProp and Adagrad:

$$\theta_i^t = \theta_i^{t-1} - \eta \frac{\widehat{M_t}}{\sqrt{\widehat{R_t} + \epsilon}}$$

AdaDelta

Adaptive Delta

Adadelta

- Adadelta appeared at the same time with Rmsprop and tried to solve the radically diminishing learning rates of Adagrad.
- It build on RMSProp, but instead of using a learning rate, it uses an exponential running average of previous deltas (adjustments of the parameter)

$$E\left[\left(\nabla\theta_i^t\right)^2\right] = \gamma E\left[\left(\nabla\theta_i^{t-1}\right)^2\right] + (1-\gamma)\left(\nabla\theta_i^t\right)^2$$

$$E\left[\left(\Delta\theta_i^t\right)^2\right] = \gamma E\left[\left(\Delta\theta_i^{t-1}\right)^2\right] + (1-\gamma)\left(\Delta\theta_i^t\right)^2$$

$$\Delta\theta_i^t = \frac{\sqrt{E\left[\left(\Delta\theta_i^{t-1}\right)^2\right]}}{\sqrt{E\left[\left(\nabla\theta_i^t\right)^2\right]} + 1^{-8}} \cdot \nabla\theta_i^t$$

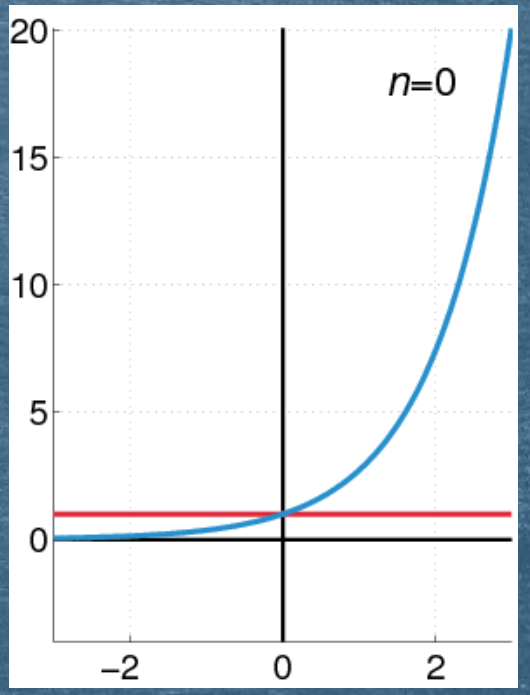
$$\theta_i^t = \theta_i^{t-1} - \Delta\theta_i^t$$

Adadelta

- The intuition behind using a running average of previous updates instead of the learning rate, comes from Newton's Method of gradient descent.

A functions that is infinitely differentiable at a real number x_0 can be approximated using Taylor series

$$f(x + x_0) \approx f(x_0) + \frac{f'(x_0)x}{1!} + \frac{f''(x_0)x^2}{2!} + \cdots \frac{f^n(x_0)x^n}{n!}$$



Source:wikipedia

Adadelta

- If we approximate the function using only the first two derivates

$$f(x + x_0) \approx f(x_0) + \frac{f'(x_0)x}{1!} + \frac{f''(x_0)x^2}{2!}$$

- We want to find the minimum of our function. This is where the gradient is 0.

$$\frac{d\left(f(x_0) + \frac{f'(x_0)x}{1!} + \frac{f''(x_0)x^2}{2!}\right)}{dx} = \mathrm{f}'(\mathrm{x}_0) + \mathrm{f}''(\mathrm{x}_0)\mathrm{x}$$

Adadelta

- If we approximate the function using only the first two derivates

$$f'(x_0) + f''(x_0)x = 0 \Rightarrow x = -\frac{f'(x_0)}{f''(x_0)}$$

This gives us an approximate position, so we must do this several times

$$x_{t+1} = x_t - f'(x_t)\left(f''(x_t)\right)^{-1}$$

This looks very similar to what we're doing $\theta_{t+1} = \theta_t - \eta\nabla\theta_t$, except our learning rate should be $f''(x_t)^{-1}$

So, we approximate it based on the above equation:

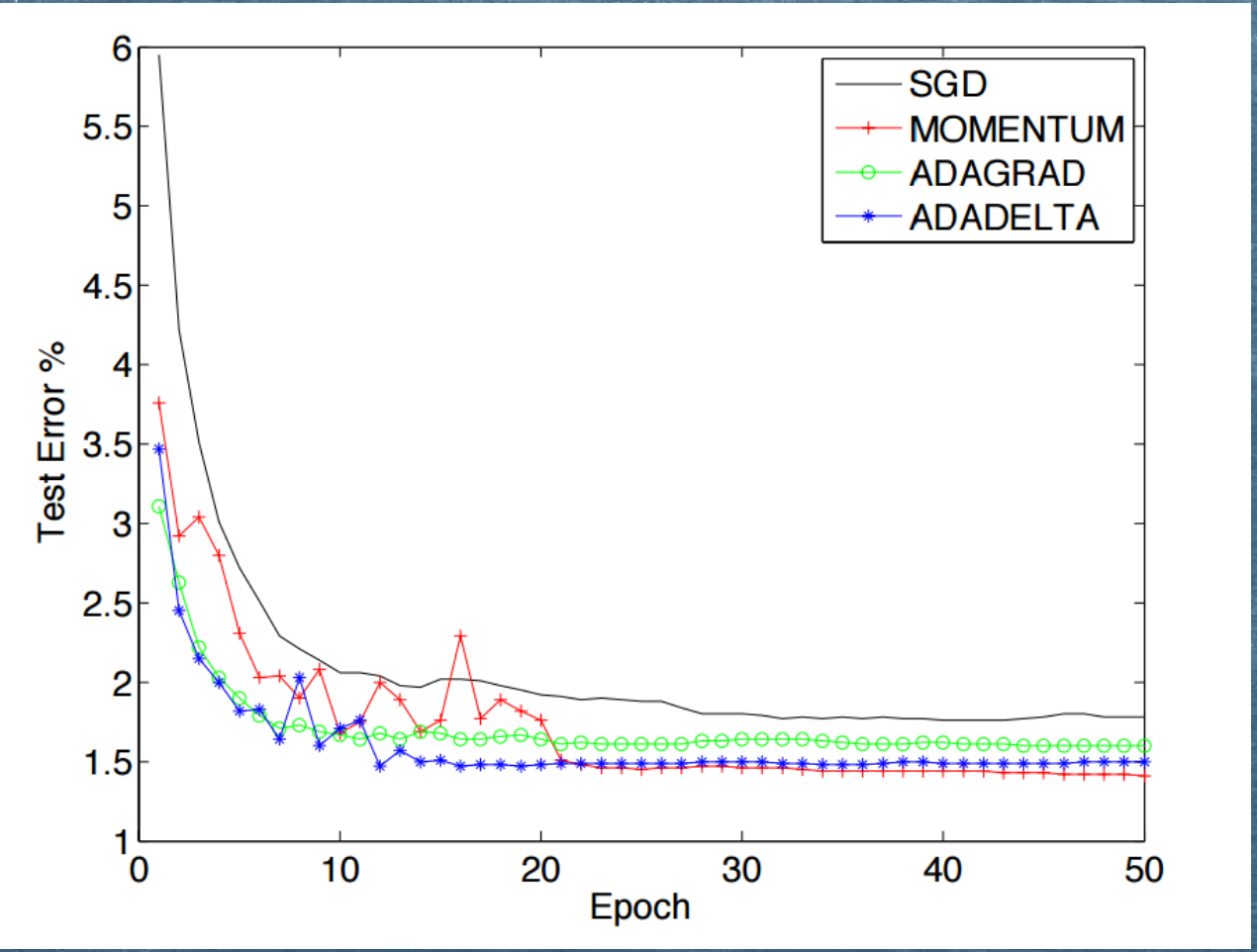
$$f''(x_t)^{-1} = \frac{x_{t+1} - x_t}{f'(x_t)} = \frac{\Delta x}{\nabla x_t}$$

Adadelta

- This is still different from the Adadelta rule, because we don't have Δx_t, but we approximate it by Δx_{t-1}.
- Since we want to not be influentated by outliers, we use the exponential running average of each of these components
- Thus, we get

$$\frac{\sqrt{E\left[\left(\Delta x_{i-1}\right)^2\right]}}{\sqrt{E\left[\left(\nabla x_t\ \right)^2\right]}}$$

Optimizers comparison



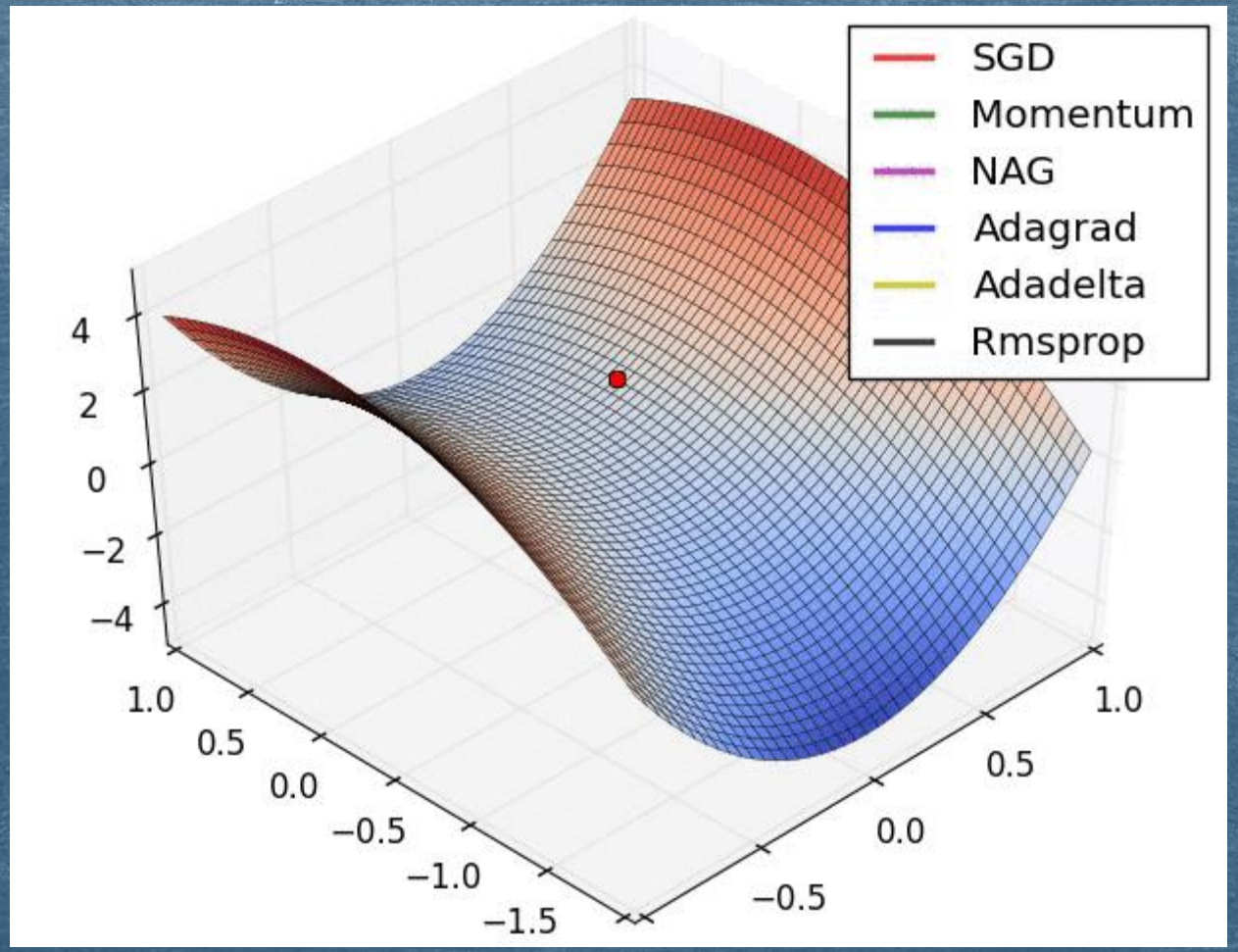
2 hidden layers:
1st: 500 neurons
2nd: 300 neurons

Last layer, softmax (10 neurons)

Used relu instead of sigmoid

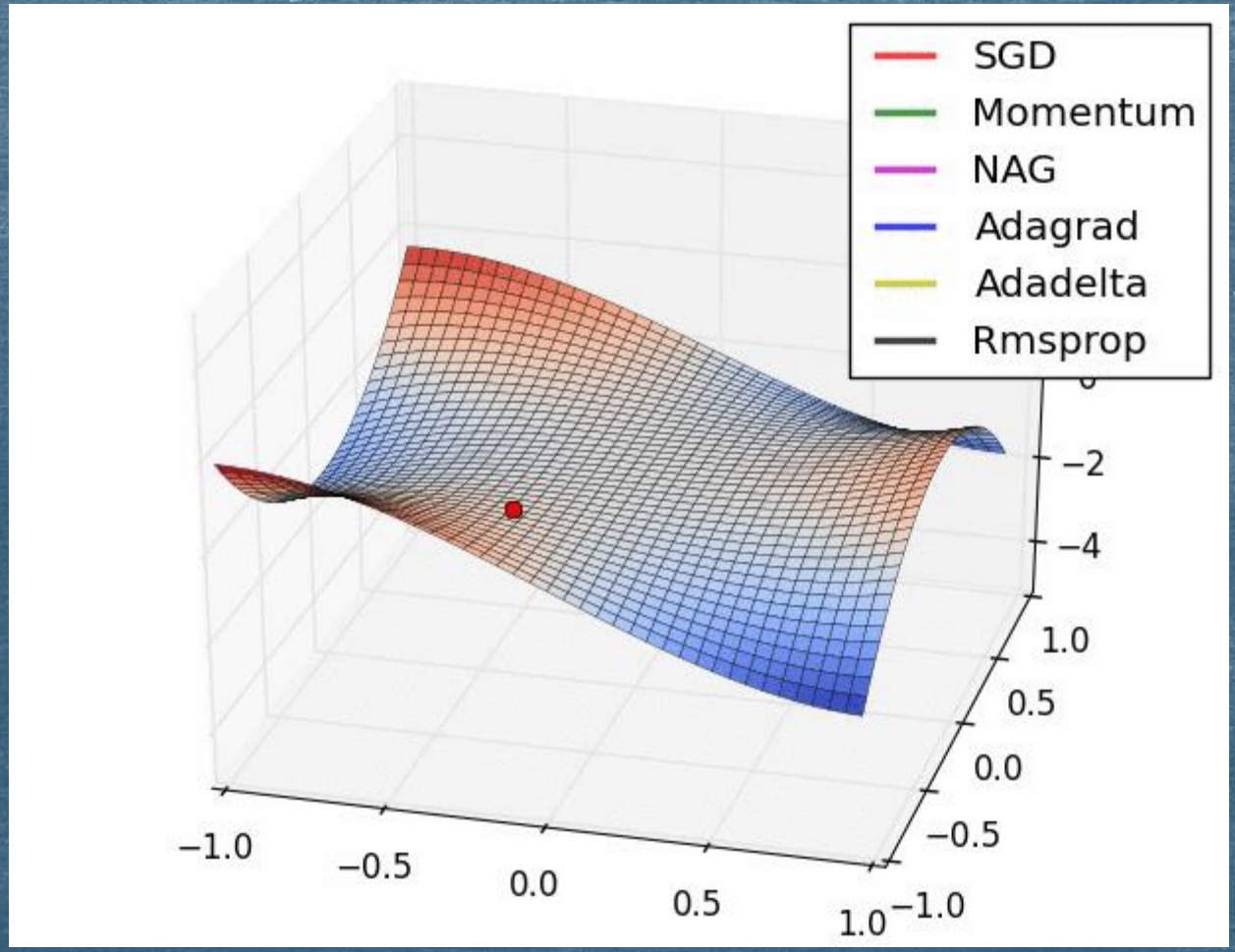
Source:Matthew D. Zeiler

Optimizers comparison



Source: Alec Radford

Optimizers comparison



Source: Alec Radford

Questions & Discussion

Bibliograpy

- http://neupy.com/versions/0.1.4/2015/07/04/visualize_backpropagation_algorithms.html
- http://cs231n.github.io/neural-networks-3/
- https://intelligentartificiality.wordpress.com/2016/02/19/adaptive-stochastic-learning-via-the-adadelta-method/
- https://www.quora.com/What-is-an-intuitive-explanation-of-the-AdaDelta-Deep-Learning-optimizer
- https://arxiv.org/pdf/1609.04747.pdf
- http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html
- https://arxiv.org/pdf/1212.0901v2.pdf
- https://xcorr.net/2014/01/23/adagrad-eliminating-learning-rates-in-stochastic-gradient-descent/
- https://visualstudiomagazine.com/articles/2015/03/01/resilient-back-propagation.aspx
- http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
- http://www.matthewzeiler.com/pubs/googleTR2012/googleTR2012.pdf
- https://www.khanacademy.org/math/calculus-home/series-calc/taylor-series-calc/v/generalized-taylor-series-approximation