Neural Networks

Course 6: Optimizers

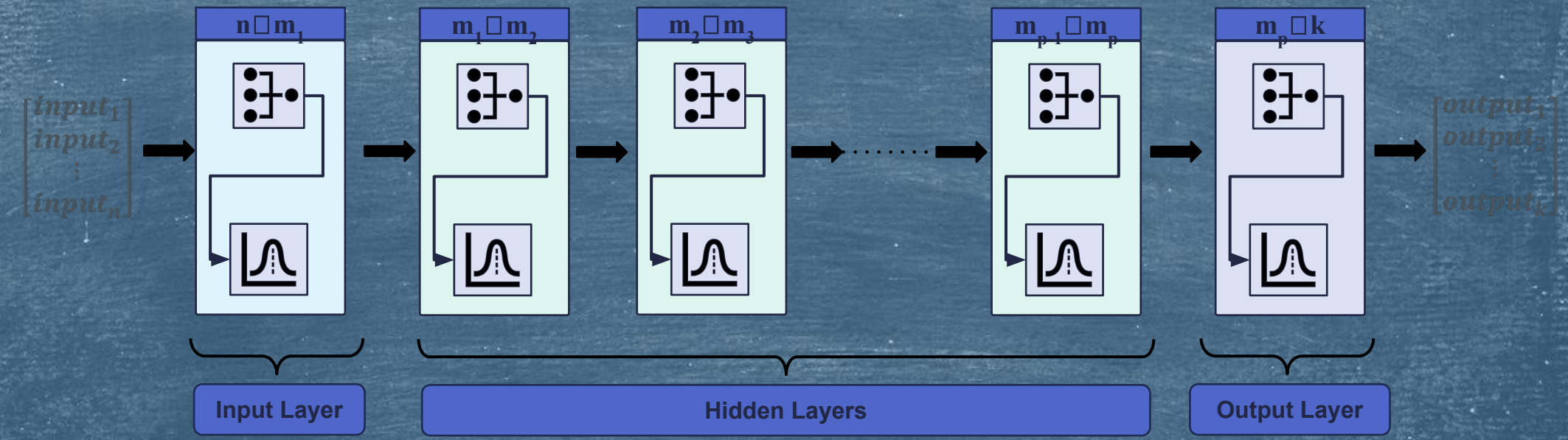
Overview

- Space & memory considerations
- Need of optimizers
- Momentum and Nesterov Acclerated Gradient
- RProp
- Adagrad
- RmsProp
- Adam
- Adadelta
- Conclusions

Space & memory considerations

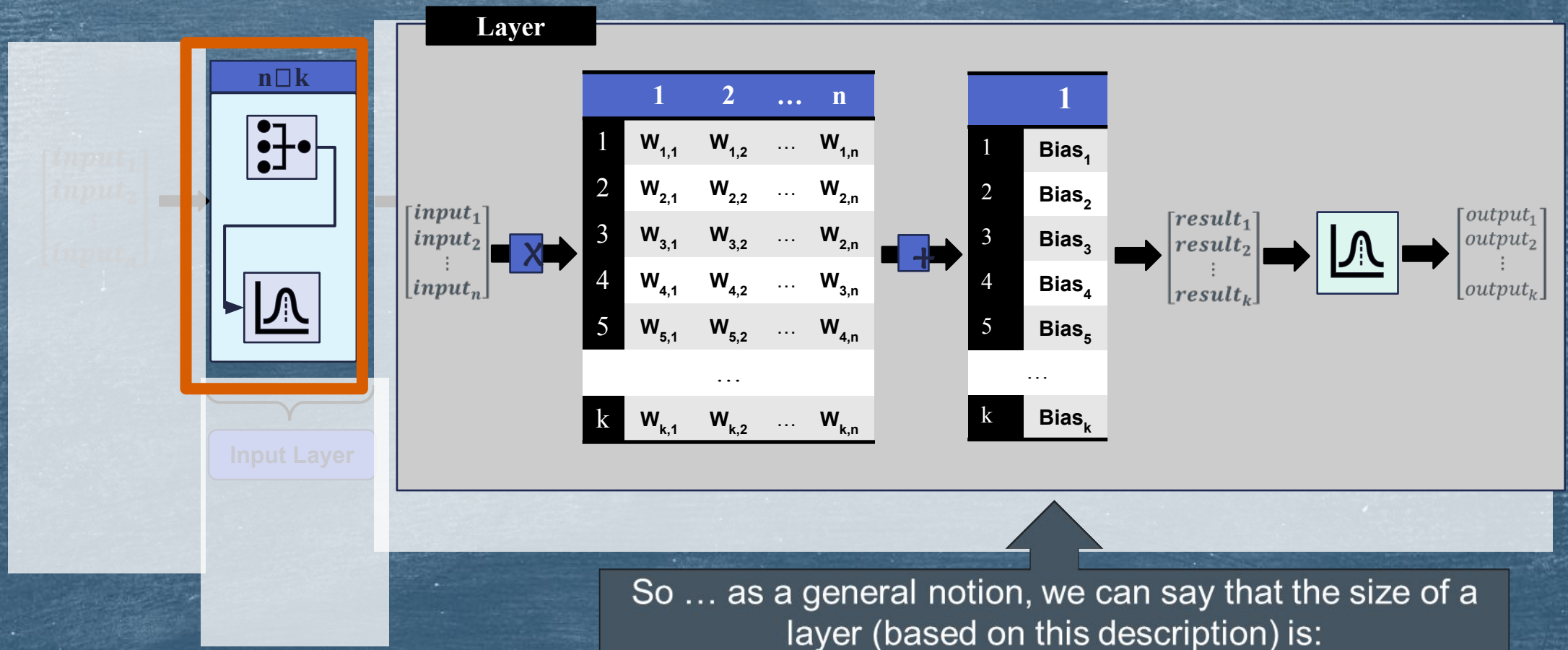
Space & memory considerations

Let's see how a neural network looks like:



Space & memory considerations

Let's see how a neural network looks like:

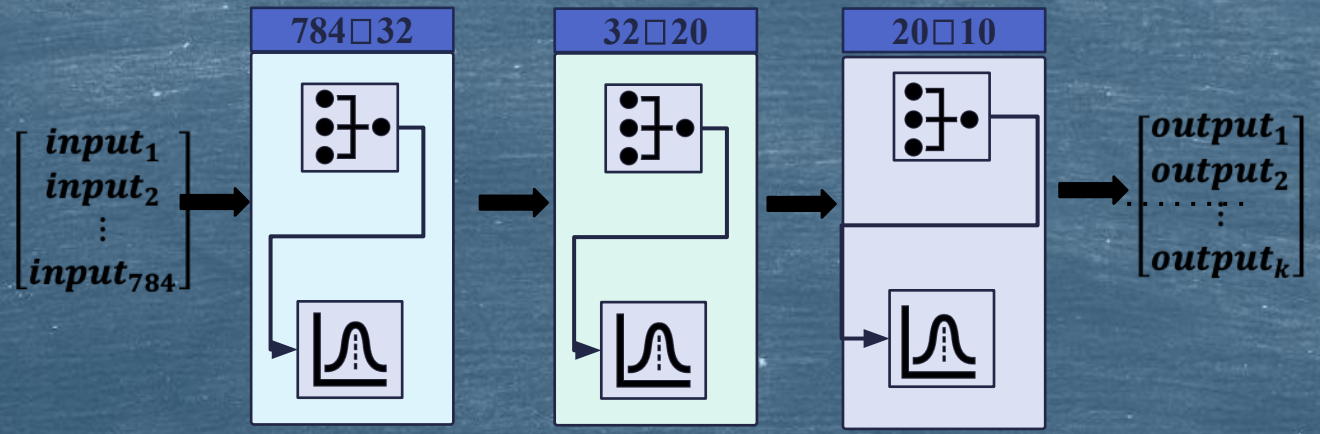


So … as a general notion, we can say that the size of a layer (based on this description) is:

$$sizeof(Layer) = sizeof(type) \times (n \times k + k)$$

Space & memory considerations

The number of parameters of a model is the total number of weights and biases from the model. For example:



$$Total\ nr\ of\ parameters = 784 \times 32 + 32 + 32 * 20 + 20 + 20 * 10 + 10 = 25990$$

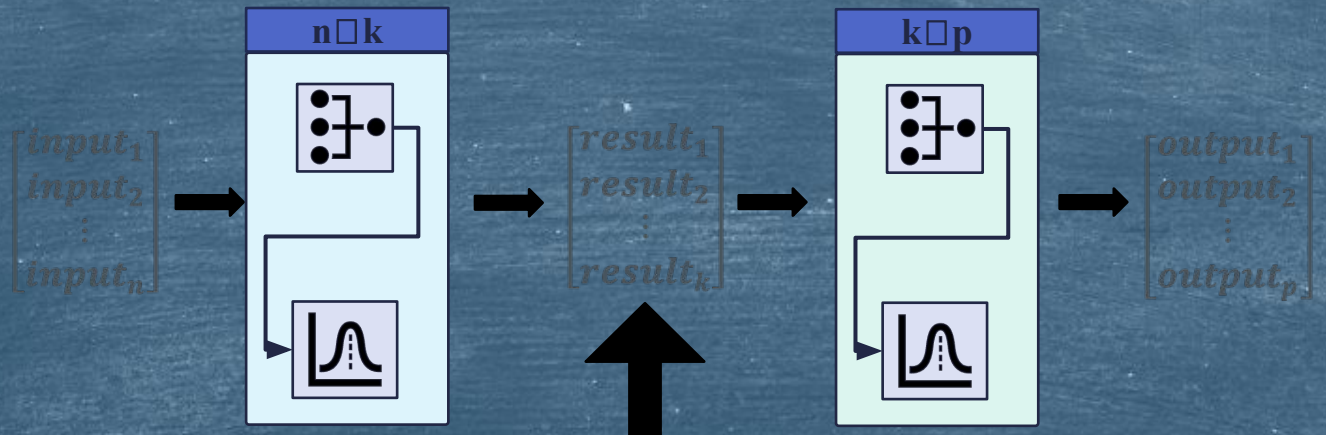
$$size\ of\ model = 25990 \times sizeof(float64) = 207920 = 203Kb$$

This means that when you hear that a LLM model has let's say 7B parameters, this means that the sum of the number of parameters from each layer is 7 billions.

Space & memory considerations

It's also important to understand that the output of each layer need also to be stored in order to use it for:

- Feed-forward step (to compute the next layer)
- Backpropagation step (to compute the gradients)



Space & memory considerations

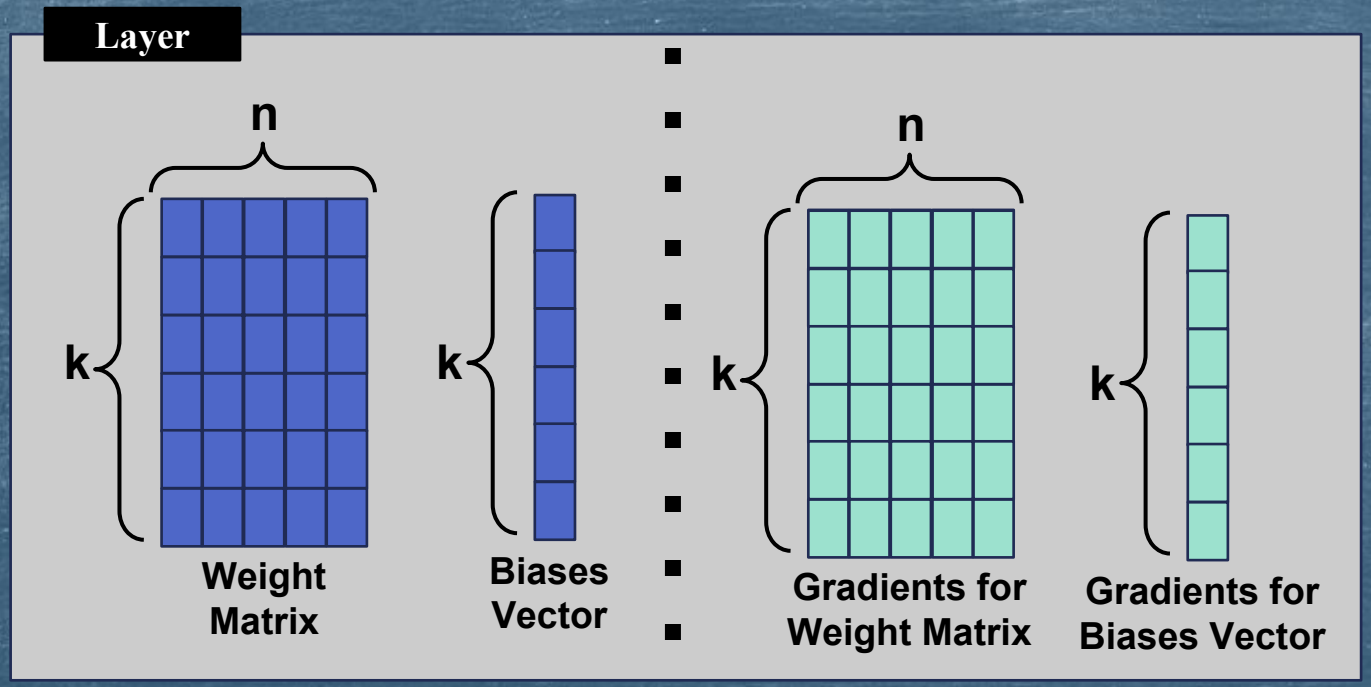
It is important to note that on the feedforward step we don't need to keep an intermediate output more than it is required to compute the next layer. Furthermore, the memory associated with that intermediate output can be reused.

When it comes to training however, the activation of each layer must be stored for the backpropagation algorithm.

As we will see, the size can also be affected by the optimizer that we choose.

Space & memory considerations

Using minibatch implies that we first compute (**and store**) the gradients and then we update the weight and bias matrixes. This means that the actual representation of a layer would look like this:



This also means that the training implies a size **twice as large** as the actual size of the model.

Space & memory considerations

So, a training operation on minibatch requires 2 to 3 times the size of the model:

- Size of the weights (model size)
- Size of the activations
- Size of the gradients (model size)

All this data must be in the GPU. Besides that we also have the training and labels for the minibatch data.

The memory space allocated for GPU is very limited. Theoretically a minibatch operation on a gpu takes approx. the same amount of time, no matter the nr of inputs (if the data is in GPU memory)

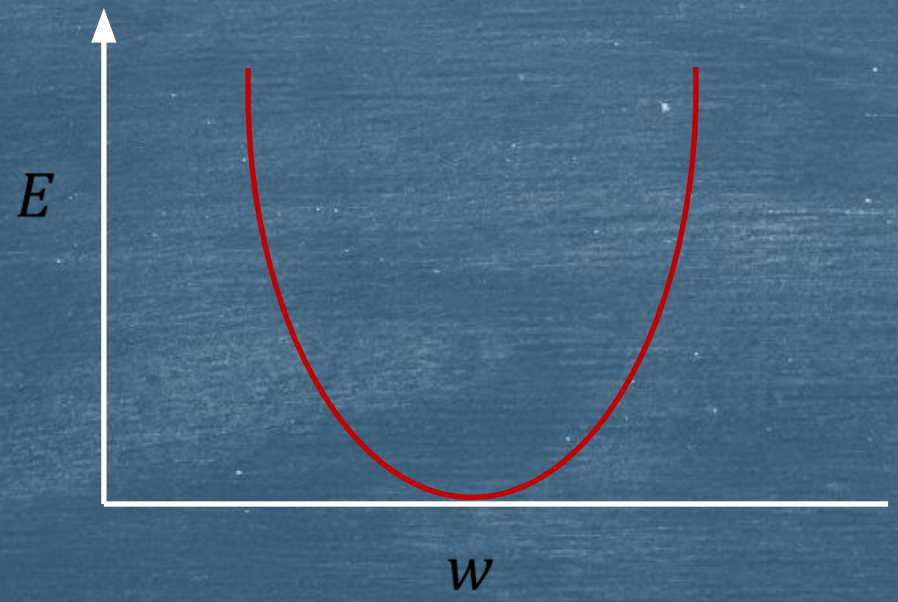
Larger minibatches lead to faster convergence

Need of optimizers

Need of optimizers

- **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



Need of optimizers

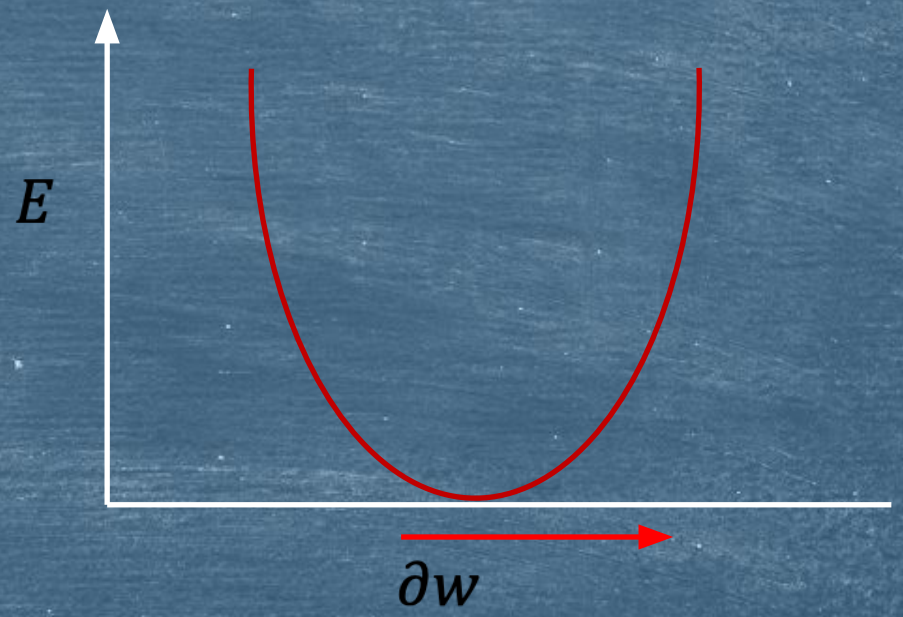
- **How gradient descent works?**

We're computing partial derivatives, so we can always imagine that when computing a gradient we're in this case.

The gradient w.r.t to a weight has two parts:

- The **sign** : gives us direction
- The **magnitude:** tells us how steep is the Error function at that given point.

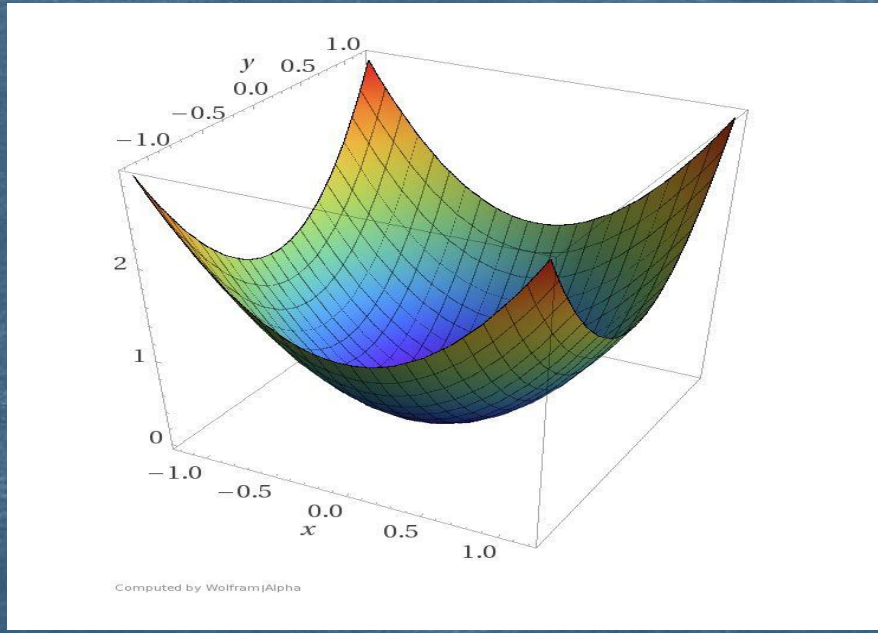
There is no other direction than left or right (+ or -)



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

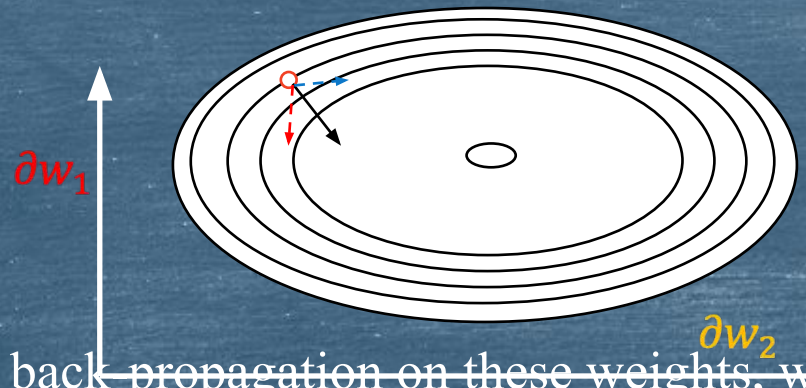


Source:
https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

Need of optimizers

- **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

Need of optimizers

- **How gradient descent works?**

The resulted vector is perpendicular to the contour line (steepest descent)

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)

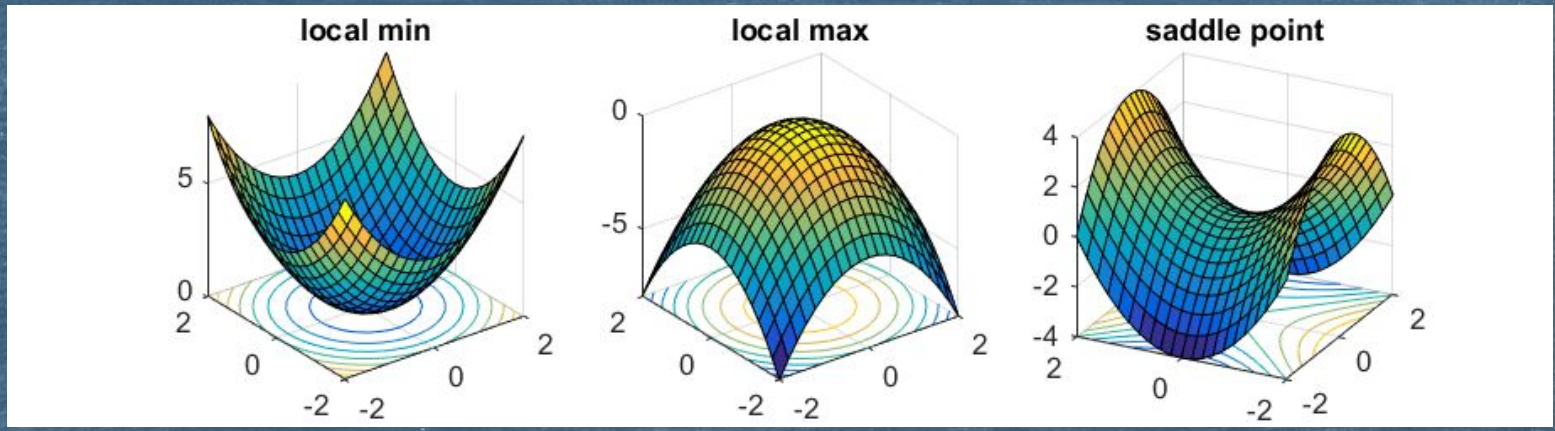
Since we make a move that is direct proportional to the size of the gradient, it is possible to move a big step in the wrong direction (across the ellipse) and a little step towards the good direction (along the ellipse)

Need of optimizers

- **Surface of the Loss function**

A network with many parameters and non linear activations rarely have a convex function.

Usually, the surface of the loss function has multiple local minimums and different shapes connecting them, many time with multiple saddle points



Source: http://www.offconvex.org/2016/03/22/saddlepoints/

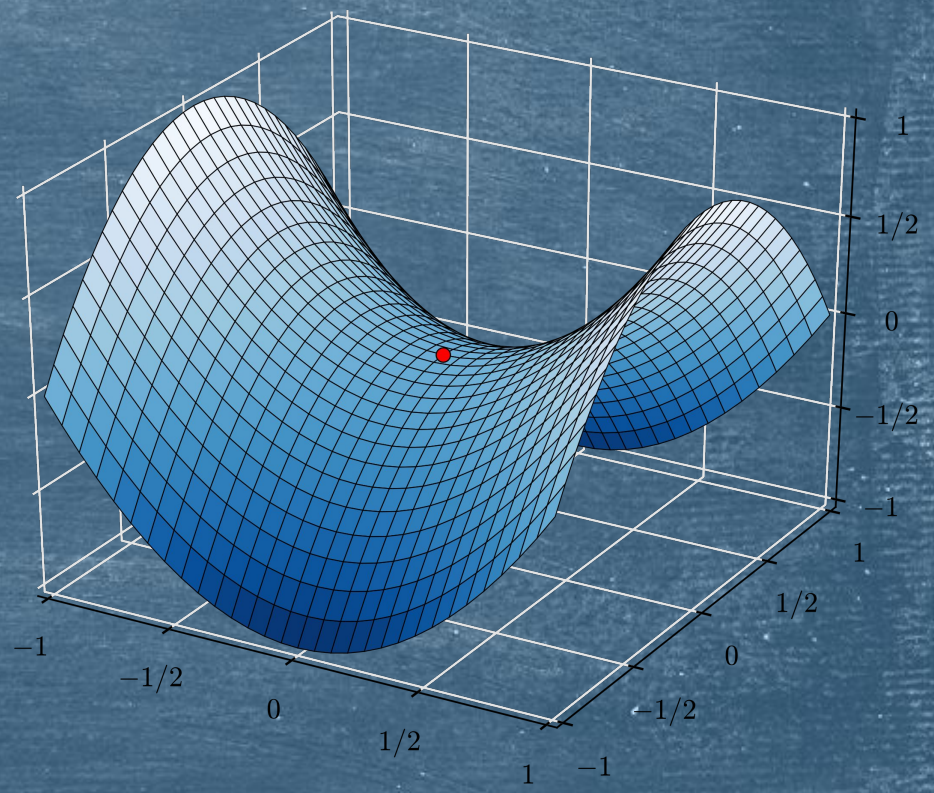
Need of optimizers

- **Surface of the Loss function**

A saddle point is defined as a place on the Loss Function where the overall gradient is zero, but the point is not a maximum nor a minimum.

Arriving at a saddle point makes learning very difficult, since using the first derivative is useless (is zero).

The number of saddle points outnumbers the local minimum and local maximums.



$f(x) = x^2 - y^2$

source: https://en.wikipedia.org/wiki/Saddle_point

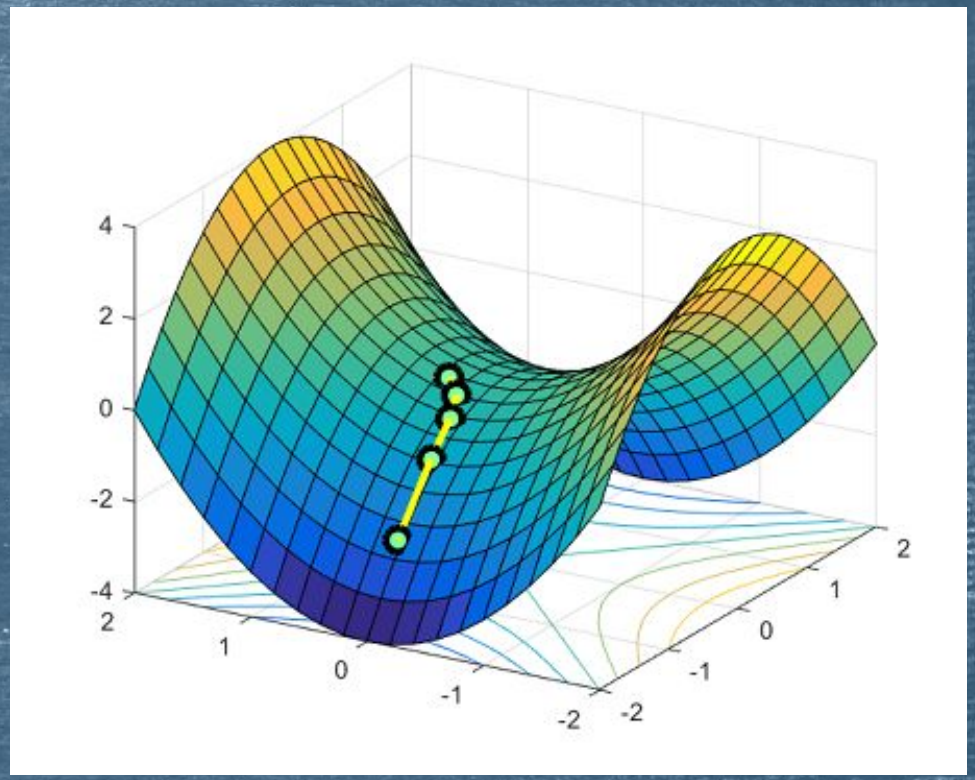
Need of optimizers

- **Surface of the Loss function**

A saddle point is often surrounded by plateaus(same value for loss), meaning the saddle is not steep. This makes learning difficult.

One observation is that saddle points are “unstable”. Meaning, if noise is added , that will move the loss function beyond the saddle point where some gradients are different than zero.

Using a smaller minibatch could add the necessary noise, but this also increases training time.



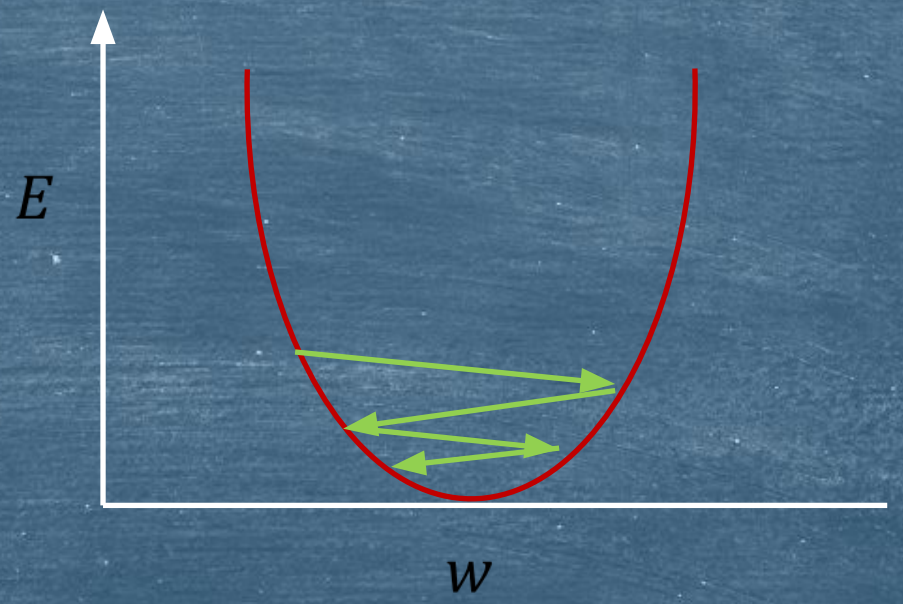
Source: http://www.offconvex.org/2016/03/22/saddlepoints/

Need of optimizers

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



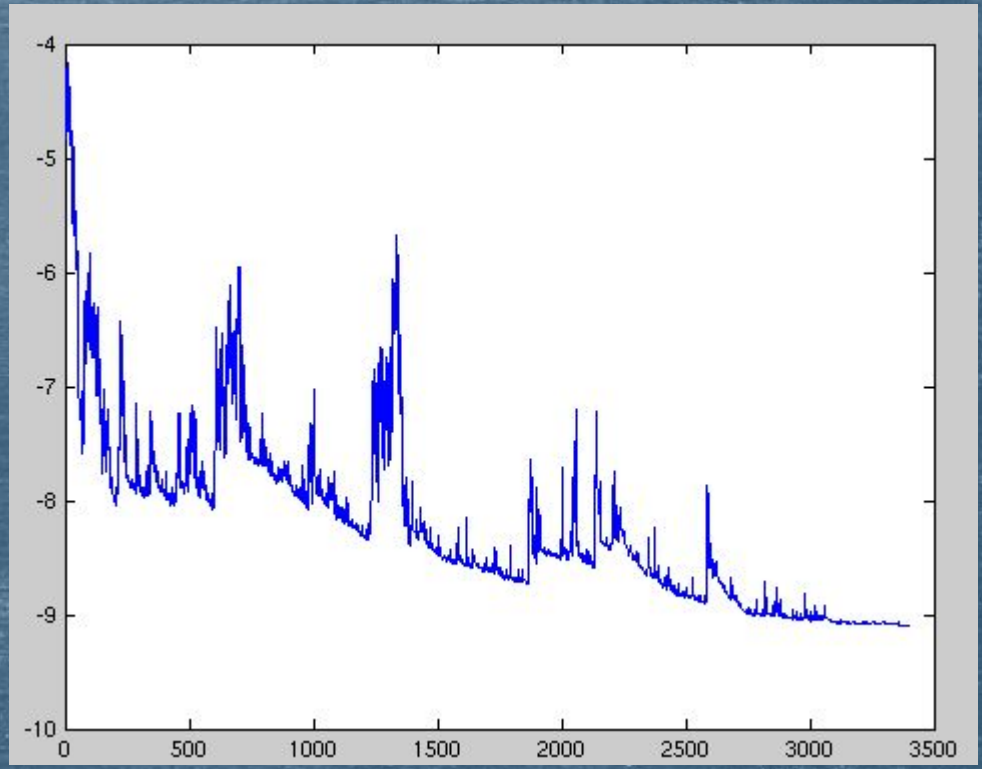
Need of optimizers

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

Need of optimizers

- **Decreasing learning rate**

Decreasing the learning rate reduces fluctuations when using small 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

Notation

- $g_t = \left(\frac{\partial C}{\partial w_{ij}}\right)_t$ gradient of the cost function w.r.t the weigh w_{ij} at the tim t

- w_{ij_t}= value of weight w_{ij} at time t

- η = learning rate

- η_{ij_t}=learning rate for weight w_{ij} at time t

Momentum

Momentum

Intuition:

The next gradient will likely be in the direction as the previous. So make a bigger move in that direction.

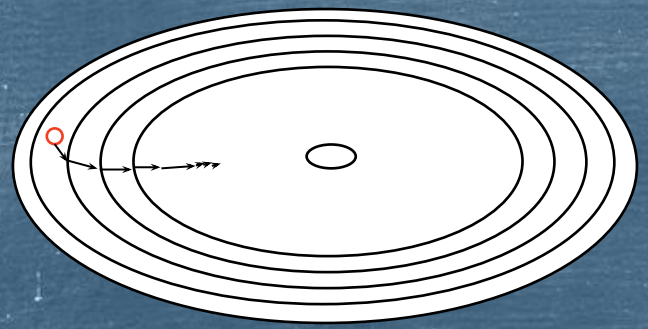
This is because:

- When updating the weights we only use a fraction of the previous gradient (by using a small learning rate)
- There are only two possible directions (positive or negative)

Momentum works by constantly adding previous gradient. Small steps add up. It is like having acceleration

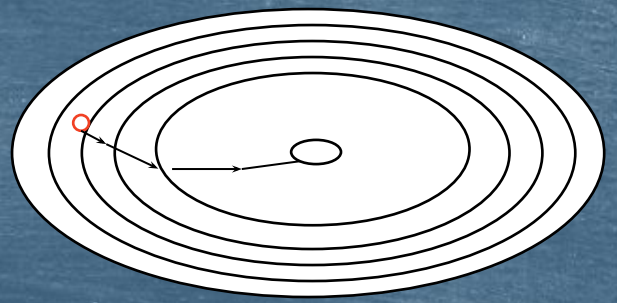
Momentum

- 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

- 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.

$$v_{ij} = \mu v_{ij} + \eta g_t$$
$$w_{ij} = w_{ij} - v_{ij}$$
$$where\ \mu\ is\ the\ friction.\ a\ value\ between\ [0,1]$$

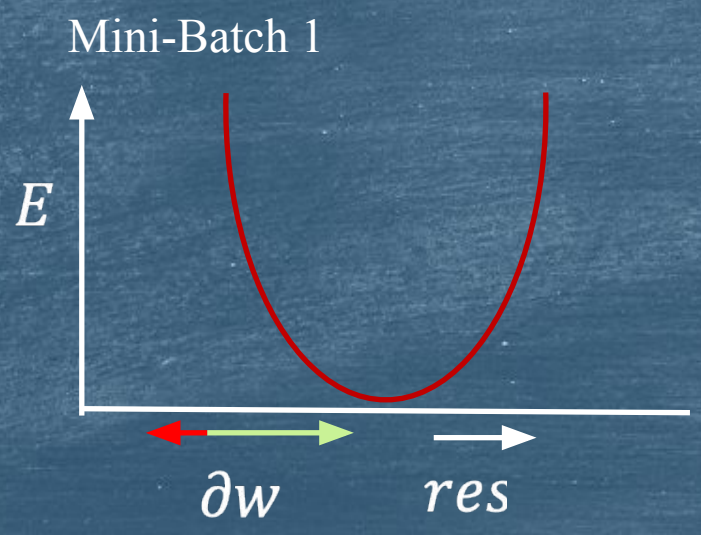
- If $\mu = 0$ then we have the same update rule as before.
- In literature μ is called *momentum co-efficient. Usual values 0.9, 0.95*

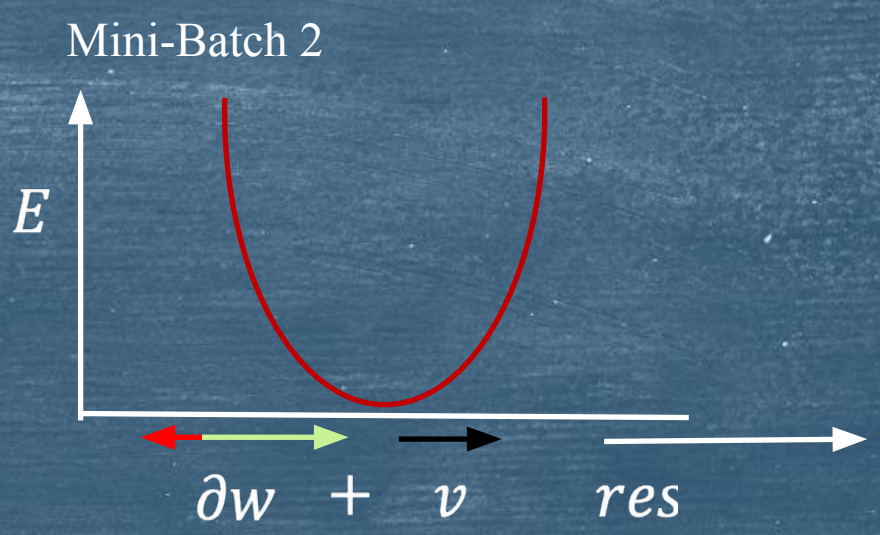
Momentum

- Momentum is computed across minibatches
- This reduces the zig-zag pattern that might be present in Stochastic gradient descent because outliers matter less over the long run.

- Example:

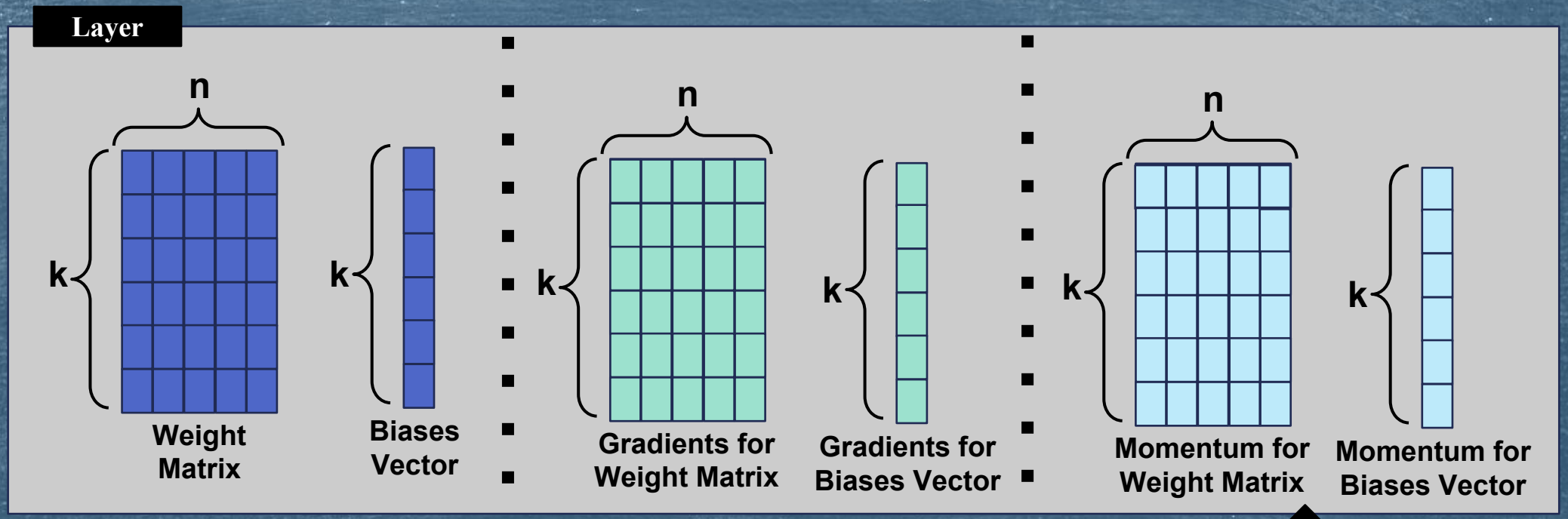
Two minibatches with 32 samples from which 4 are outliers.





Momentum

Momentum implies we store the velocity factor for each weight and bias



In other words, we triple the size of a layer by adding a new matrix and bias vector.

Momentum

Considerations:

- Use a small learning rate. This allows to make small adjustments near the minimum.
- Momentum helps travel the saddle point of functions, since:
  - very small steps add up.
  - the added velocity adds noise before arriving at the saddle point, making the learning not stop. (it give it a push)

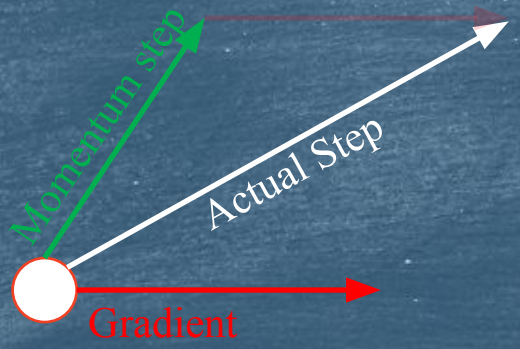
- The combination of a small learning rate and momentum allows for a **balance between exploration** (navigating through the loss landscape to avoid local minima) **and exploitation** (fine-tuning the parameters to reach the lowest point in a minimum).

Nesterov Accelerated Gradient

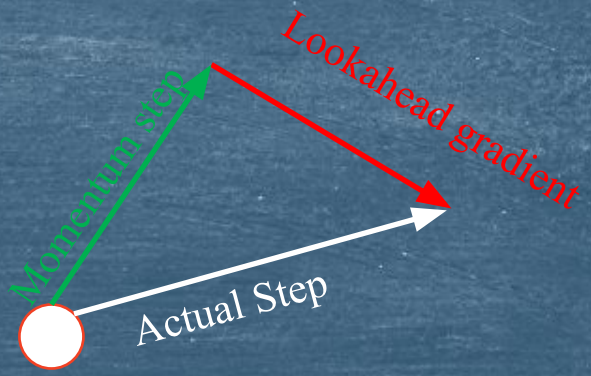
Nesterov Accelerated Gradient

- 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

$$v_{ij_t} = \mu v_{ij_{t-1}} + \eta \frac{\partial C}{\partial \theta_{ij_{t-1}}}, where. \quad \theta_{ij_{t-1}} = \left(w_{ij_{t-1}} - \mu v_{ij_{t-1}}\right)$$

$$w_{ij_t} = w_{ij_{t-1}} - v_{ij}^t$$

$\theta_{ij_{t-1}}$ is a temporary lookahead value used for computing the gradient after applying the momentum

Nesterov Accelerated Gradient

- When implemented, Nestrov can be viewed in 4 steps:

1. Compute the projection:

$$\theta_{ij_{t+1}} = w_{ij_t} - \mu v_{ij_t}$$

2. Compute the gradient of the projection:

$$\frac{\partial C}{\partial \theta_{ij_{t+1}}}$$

3. Compute the change of the weight

$$v_{ij_{t+1}} = \mu v_{ij_t} + \eta \frac{\partial C}{\partial \theta_{ij}^{t+1}}$$

4. Compute the new weight:

$$w_{ii} = w_{ii} - v_{ii}$$

Nesterov Accelerated Gradient

The 'look-ahead' step is the key to NAG's effectiveness. By calculating the gradient at the look-ahead position, NAG anticipates the future landscape of the loss function.

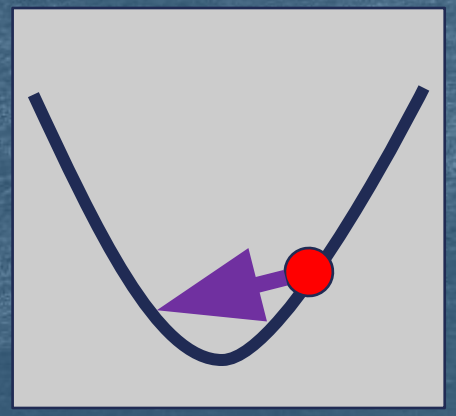
If this look-ahead step results in a parameter position that is in a less favorable direction (e.g., heading towards a higher loss), the gradient computed there will correct this, forcing the updates towards a more optimal position.

This is particularly useful in scenarios where the loss surface has sharp curves or local minima, as it allows NAG to adjust its trajectory more intelligently than standard momentum, which only looks at the past gradients.

Nesterov Accelerated Gradient

As a general idea:

- NAG often converges faster and more effectively than standard momentum, especially in complex, high-dimensional spaces typical in deep learning.
- It is more stable and less prone to the problem of **overshooting the minimum**, a common issue with algorithms that rely heavily on momentum.



Overshooting happens when the update steps during the optimization process are too large, causing the algorithm to miss the minimum of the loss function and jump over to the other side of the curve. This can happen **if the learning rate is set too high** or if the **momentum term accumulates excessively without sufficient correction**

Rprop

Resilient Propagation

Resilient Backpropagation (Rprop)

Resilient back**prop**agation (Rprop) is an optimization algorithm designed to eliminate some of the issues faced by gradient descent methods.

RProp focuses solely on the direction of the gradient and not its magnitude. This means that the update is performed based on the sign of the gradient for each individual weight and bias in the network.

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)

Resilient Backpropagation (Rprop)

- The Rprop algorithm is being performed in 3 steps:

1. Update the weights

$$w_{ij}{}_{t+1} = w_{ij}{}_{t} - U_t \times sign(g_t), where$$
$$w_{ij}{}_{t+1} = weights\ at\ the\ moment\ t$$
$$U_t = Update\ values\ (U_o = 0.01),$$

2. Update the values

$$U_{i,j} = \begin{cases} U_{i,j} \times \eta^+, & sign(g_t) = SGN_{i,j} \\ U_{i,j} \times \eta^-, & sign(g_t) \neq SGN_{i,j} \end{cases} \quad where$$

$$\eta^+ = an\ increase\ coeficinent\ (\eta^+ > 1), \eta^+ = a\ decrease\ coefficient\ (0 < \eta^- < 1)$$
$$SGN_{i,j} = sign\ of\ the\ previous\ gradient\ for\ element\ (i,j)\ from\ weight\ matrix,$$

3. Store the current gradient signs

$$SGN_{i,j} = sign(g_t)$$

Resilient Backpropagation (Rprop)

- The η^+ and η^- typically have values as follows:
  - $\eta^+ > 1$, a typical value is 1.2 (increase by 20%)
  - $\eta^- < 1$, a typical value is 0.5 (decrease by 50%)

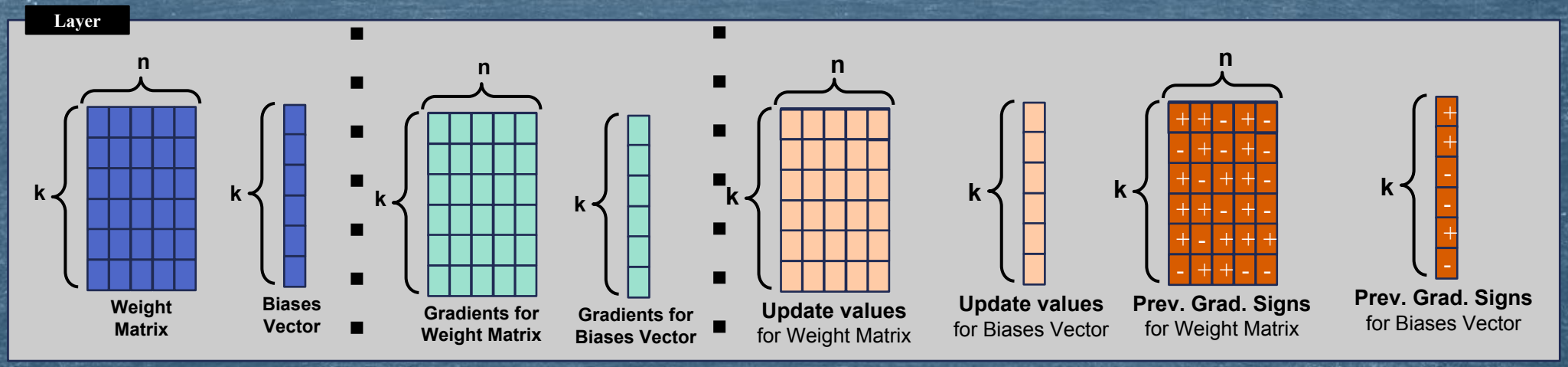
In practice, there are also some limits that are being set up so that the update values don't go outside a specific range. These limits are:

- Minimum value: 1e-06 = 0.00000001
- Maximum value: 50

- The initial values for $\Delta\theta_i$ must be different than 0, usually 0.01

Resilient Backpropagation (Rprop)

Just like momentum, a separate set of values (update values) and the signed of the previous gradient have to be stored for each weight and bias from the network



In other words, we at least triple the size of a layer by adding a two new matrixes and two new bias vector. The sign matrixes and bias vector can be optimized (by using bit-sets)

Rprop

Advantages of Rprop:

- Since the update is independent of the size of the gradient, it can escape plateaus very quickly (just like momentum)
- It tends to be more stable and less sensitive to the specific topology of the error landscape, as it is not as affected by small, noisy gradients or very steep gradients.
- There is no need to tune a learning rate or a momentum factor
- One of the fastest algorithms available

Rprop

Disadvantages of Rprop:

- 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

Adaptive Gradient Algorithm (AdaGrad)

Adaptive **Grad**ient Algorithm (AdaGrad) is an optimization method that provides an adaptive learning rate for each parameter.

The learning rate for each parameter is adapted by scaling it with the inverse square root of the sum of all past squared gradients for that parameter.

Parameters with larger past gradients receive a smaller learning rate, and the ones with smaller past gradients receive a bigger learning rate.

Adagrad

- Adagrad adapts each learning rate to the size of all the previous gradients

$$\eta_{ij_t} = \frac{\eta}{\sqrt{\sum_{k=1}^{t}(g_t)^2}}$$

$$w_{ij_t} = w_{ij_{t-1}} - \eta_{ij_t} g_t$$

Where

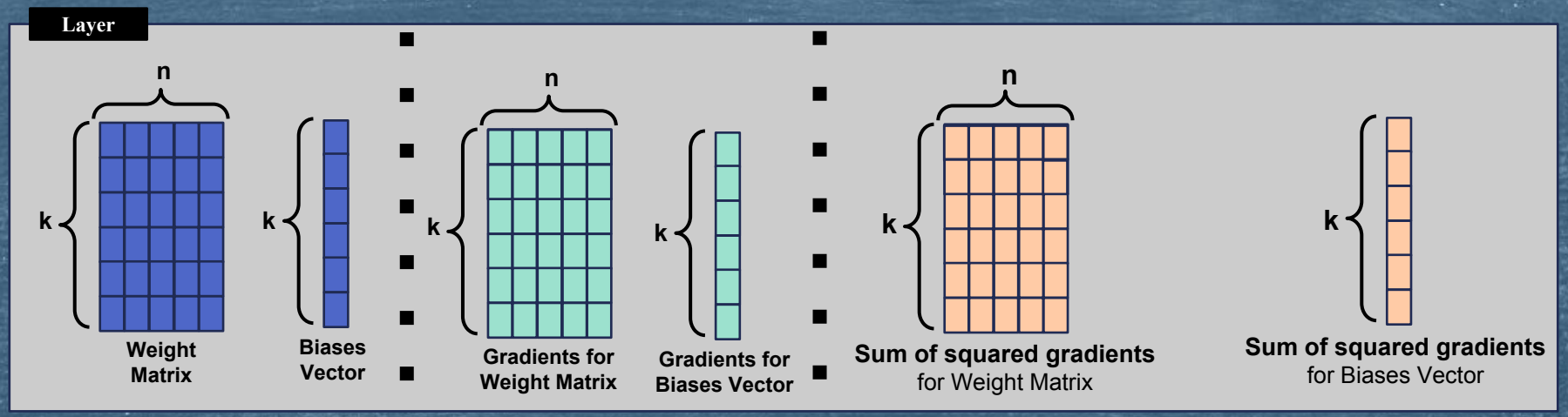
w_{ij_t} is the value of weight ij at time t

g_t is the gradient of the parameter w_{ij} at time t

η_{ij_t} is the learning rate for the weight w_{ij} at time t

Adaptive Gradient Algorithm (AdaGrad)

In case of AdaGrad, we need to store for each parameter is the sum of the squares of the gradients. This sum is then used to scale the global learning rate for each parameter individually during the update step.



In other words, we triple the size of a layer by adding a matrix and vector for each later

Adagrad

Observations:

- Big gradients will receive small learning rates, while small gradients will receive a bigger learning rate

- The starting learning rate on Adagrad is important since this is the largest learning rate for each weight

Adagrad

- Advantages:

  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)

  Disadvantages:
  - One of the main disadvantages of Adagrad is that it can become blocked on plateaus
    - This happens since the L_2 norm can only increase

Rmsprop

Root mean square propagation

Root Mean Square Propagation (RMSprop)

Root **M**ean **S**quare **Prop**agation (RMSprop) is an adaptive learning rate optimization algorithm, based on AdaGrad and design to fix some of the problems AdaGrad has.

RMSprop modifies the gradient accumulation process of AdaGrad by using an *exponentially weighted moving average* of the squared gradients. This means it doesn't accumulate all past squared gradients but instead gives more weight to the more recent gradients.

RMSProp

- On each iteration, it divides the learning rate by an exponential moving average of the squared gradient

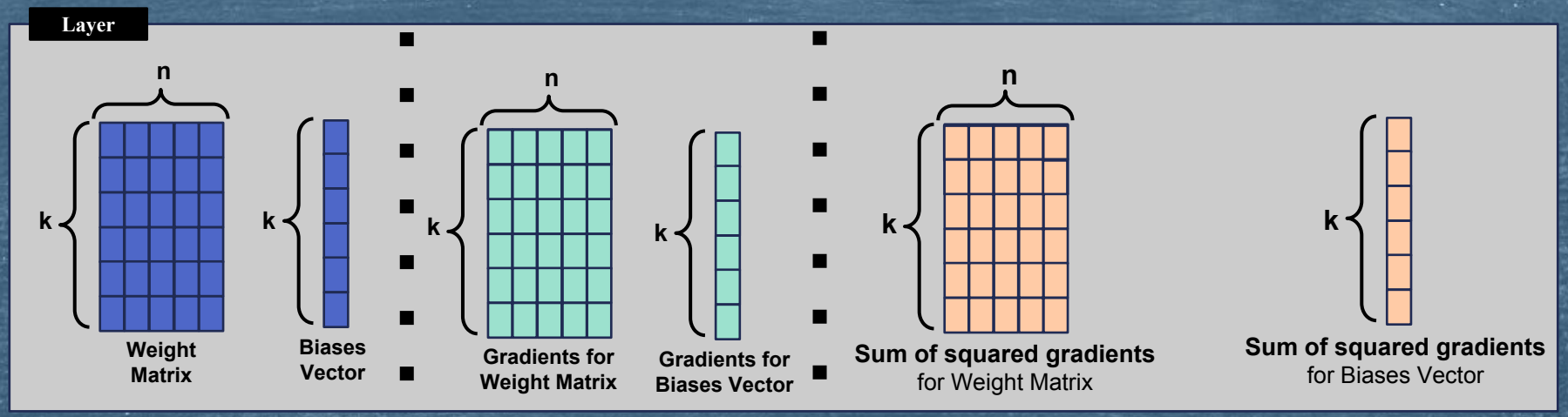
$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g^2$$

$$\eta_{ij_t} = \frac{\eta}{\sqrt{E[g^2]_t + 10^{-8}}}$$

$$w_{ij_t} = w_{ij_{t-1}} - \eta_{ij_t} g_t$$

Root Mean Square Propagation (RMSprop)

Just like in the case of AdaGrad, we need to store for each parameter is the sum of the squares of the gradients. This sum is then used to scale the global learning rate for each parameter individually during the update step.



In other words, we triple the size of a layer by adding a matrix and vector for each later

Root Mean Square Propagation (RMSprop)

Advantages:

- RMSprop avoids the problem of a rapidly diminishing learning rate encountered in AdaGrad by using a moving average (newer values count more than older ones).
- By using the moving average of the squared gradients, RMSprop automatically scales the learning rate with respect to the history of the gradients.

Disadvantages:

- Often the decay rate needs to be tuned (value of 0.9 is usually OK but for different cases different values are needed)..
- The performance of RMSprop can be sensitive to the initialization of Sum matrix and the (ε) value

Adam

Adaptive Moment Estimation

Adaptive Moment Estimation (ADAM)

Adaptive **M**oment Estimation (ADAM) is an optimization algorithm that combines ideas from both Momentum and RMSprop to update network weights.

Adam calculates an exponential moving average of the gradient and the squared gradient.

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)

$$\mathrm{M_t} = \beta_1 M_{t-1} + (1-\beta_1) g_t$$
$$R_t = \beta_2 R_{t-1} + (1-\beta_2)\, g_t^2$$

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:

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

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

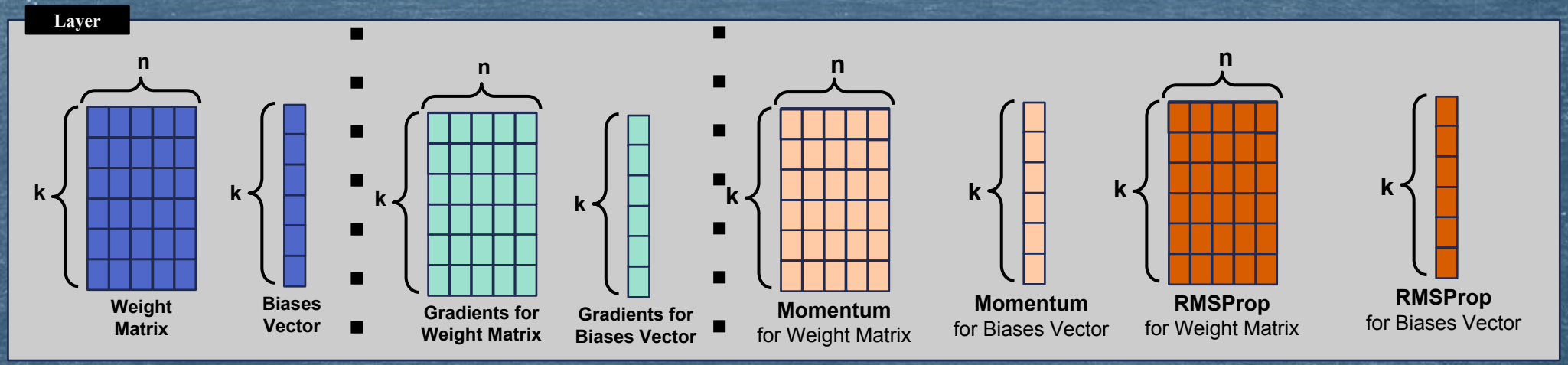
$Where\ \beta^t\ \ is\ interpreted\ as\ \beta\ at\ power\ t$

The update is performed similar to RMSProp and Adagrad:

$$w_{ij}{}_t = w_{ij}{}_{t-1} - \eta \frac{\widehat{M_t}}{\sqrt{\widehat{R_t} + \epsilon}}$$

Adaptive Moment Estimation (ADAM)

In case of Adam we need to store two matrixes (one for momentum) and another one for the RMSProp-like computations.



In other words, we have quadrupled the size of a layer by adding a two new matrixes and two new bias vector.

Adaptive Moment Estimation (ADAM)

Advantages:

- Adam adjusts the learning rate for each parameter individually based on estimates of first (mean) and second (uncentered variance) moments of the gradients.
- Efficient for problems with large datasets and high-dimensional parameter spaces, which is typical in deep learning.
- Well-suited for problems with sparse gradients (e.g., Natural Language Processing and Computer Vision tasks)
- Has a bias corrections to the first and second moment estimates, which counteract the biases toward zero that might occur especially in the initial time steps.
- Requires less tuning of the hyperparameters (ADAM can often be used out of the box with default settings and it produces good results).

Adaptive Moment Estimation (ADAM)

Disadvantages:

- Sensitive to Initial Learning Rate.
- Potential for Overfitting (due to its rapid convergence)

Nesterov-accelerated Adaptive Moment Estimation

Nesterov-accelerated Adaptive Moment Estimation

Nesterov-accelerated **Ada**ptive **M**oment Estimation (NADAM), is an optimization algorithm that combines the ideas of Nesterov momentum with Adam. It's essentially Adam with Nesterov momentum integrated into the moment estimates.

NADAM incorporates the Nesterov momentum by modifying the way the first moment (the moving average of the gradients) is calculated. Instead of using the current gradient to update the first moment as in Adam, NADAM uses the lookahead gradient, as is done in Nesterov accelerated gradient (NAG).

Nesterov-accelerated Adaptive Moment Estimation

- NADAM uses the same update formulas as ADAM for m_t, r_t and $\hat{r}_t$.

$$\mathrm{M_t} = \beta_1 M_{t-1} + (1-\beta_1) g_t; \qquad apply\ correction: \widehat{M_t} = \frac{M_t}{1-\beta_1^t}$$

$$R_t = \beta_2 R_{t-1} + (1-\beta_2) g_t^2; \quad apply\ correction: \widehat{R_t} = \frac{R_t}{1-\beta_2^t};$$

Update weights

$$w_{ij_t} = w_{ij_{t-1}} - \eta \frac{\widehat{M_t}}{\sqrt{\widehat{R_t}+\epsilon}}$$

Only the update rule for the weights is modified in Nadam when compared to Adam.
We start by expanding M_t

Nesterov-accelerated Adaptive Moment Estimation

$$w_{ij_t} = w_{ij_{t-1}} - \frac{\eta}{\sqrt{\widehat{R_t}} + \epsilon}\left(\frac{\beta_1 M_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1)}{1-\beta_1^t} g_t\right)$$

$\widehat{M_{t-1}}$

$$w_{ij_t} = w_{ij_{t-1}} - \frac{\eta}{\sqrt{\widehat{R_t}} + \epsilon}\left(\beta_1 \widehat{M_{t-1}} + \frac{(1-\beta_1)}{1-\beta_1^t} g_t\right)$$

We can now replace $\widehat{M_{t-1}}$ with $\widehat{M_t}$ to implement the lookahead part of the Nesterov momentum

$$w_{ij_t} = w_{ij_{t-1}} - \frac{\eta}{\sqrt{\widehat{R_t}} + \epsilon}\left(\beta_1 \widehat{M_t} + \frac{(1-\beta_1)}{1-\beta_1^t} g_t\right)$$

Nesterov-accelerated Adaptive Moment Estimation

Observations:

- NADAM maintains the same advantages and disadvantages as ADAM
- Potentially faster convergence and better performance on certain problems
- It is particularly useful for tasks where the benefits of Nesterov momentum, which anticipates the future gradient, are relevant
- Like Adam, Nadam can sometimes converge rapidly to suboptimal solutions for certain kinds of problems, especially those with noisy or sparse gradients.
- It can be more sensitive to the choice of hyperparameters compared to Adam, due to the additional complexity introduced by the Nesterov term.

AdaDelta

Adaptive Delta

AdaDelta

AdaDelta is an extension of AdaGrad that instead of accumulating all past squared gradients, it computes an exponential running average on past gradients

It was proposed to address the diminishing learning rates of AdaGrad, which can stop learning altogether too early in training. By maintaining the moving average of gradient information, AdaDelta continues to learn and adapt as training progresses.

A key advantage of AdaDelta is that it does not require an external learning rate. This method derives its own learning rate from the data.

AdaDelta

- Compute the Exponential Moving Average for squared gradients

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

Compute the Exponential Moving Average for squared update

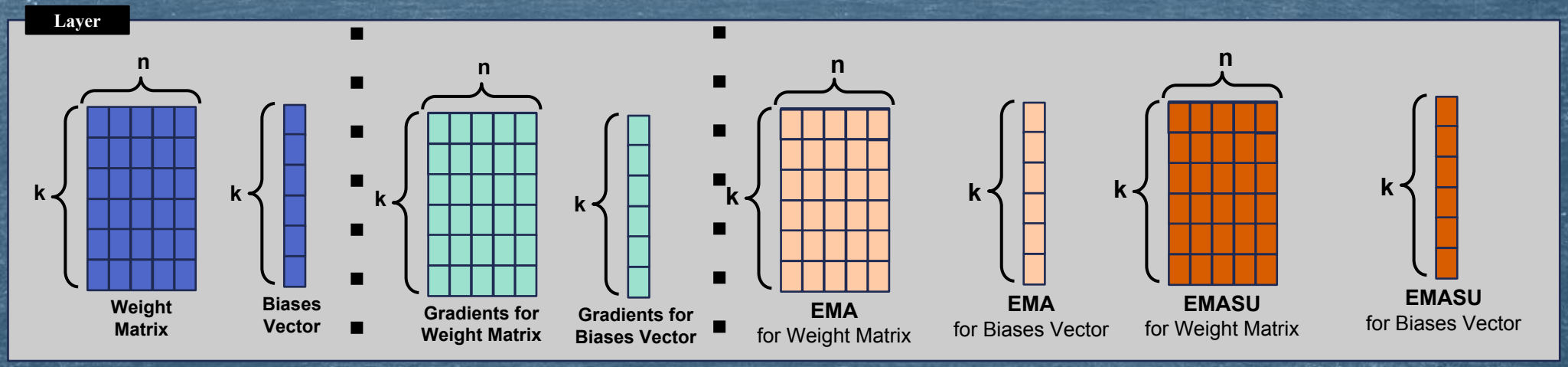
$$E\left[\left(\Delta w_{ij}\right)^2\right]_t = \gamma E\left[\left(\Delta w_{ij}\right)^2\right]_{t-1} + (1-\gamma){\left(\Delta w_{ij}\right)_t}^2$$

Combine Moving averages and update weights

$$\Delta {w_{ij}}_t = \frac{\sqrt{E\left[\left(\Delta w_{ij}\right)^2\right]_{t-1}}}{\sqrt{E[g^2]_t + 1^{-8}}} \cdot g_t \qquad {w_{ij}}_t = {w_{ij}}_{t-1} - \Delta {w_{ij}}_t$$

AdaDelta

In case of AdaDelta we need to store for each parameter the exponential moving average (EMA) and the exponential moving average for square parameter update (EMASU).



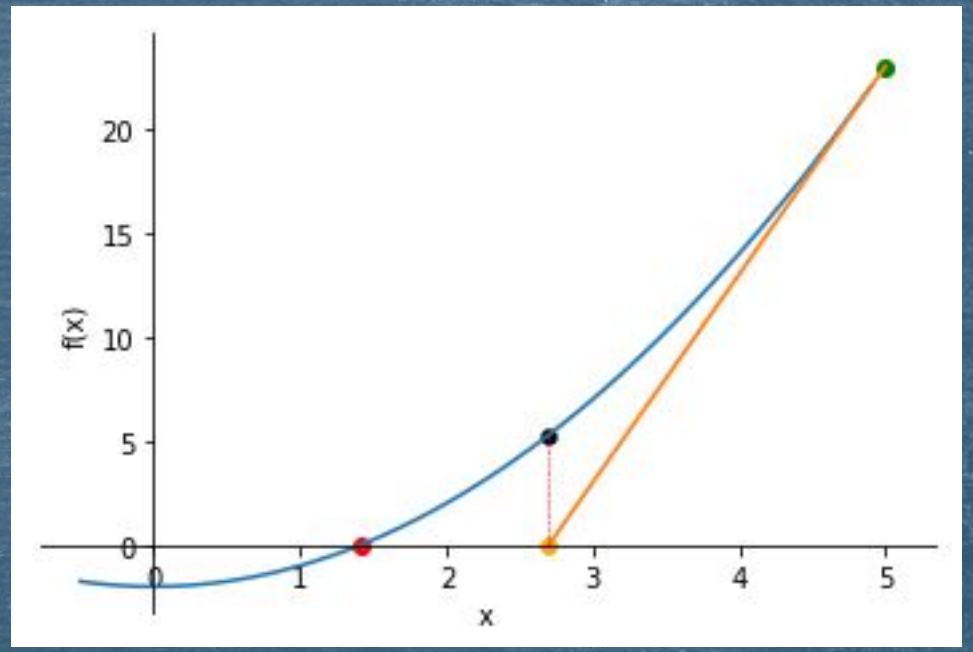
In other words, we at quadruple the size of a layer by adding a two new matrixes and two new bias vector.

Adadelta

- The intuition behind using a running average of previous updates comes from Newton's Method of finding the x such that $f(x) = 0$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Repeat until $f(x_n) = 0$



Source:medium.com

Adadelta

- We are interested to find the minimum of the function. This is where the derivative of the function = 0.

  So, we're interested to find the value where $f'(x) = 0$

  We can find that point by applying the Newton method for f'

$$x_{t+1} = x_t - \frac{f'(x_n)}{f''(x_n)}$$ Repeat until $f'(x_t) = 0$

  This looks similar to what we've been doing during learning:

$$x_{t+1} = x_t - f'(x_t)f''(x_t)^{-1} \qquad w_{t+1} = w_t - \frac{\partial C}{\partial w}\eta$$

Adadelta

- This looks similar to what we've been doing during learning:

$$x_{t+1} = x_t - f'(x_t) f''(x_t)^{-1} \qquad w_{t+1} = w_t - \frac{\partial C}{\partial w}\eta$$

So, we approximate the learning rate based on the above equation:

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

This is still different from the Adadelta rule, because we don't have Δx_t, but we approximate it by Δx_{t-1}.

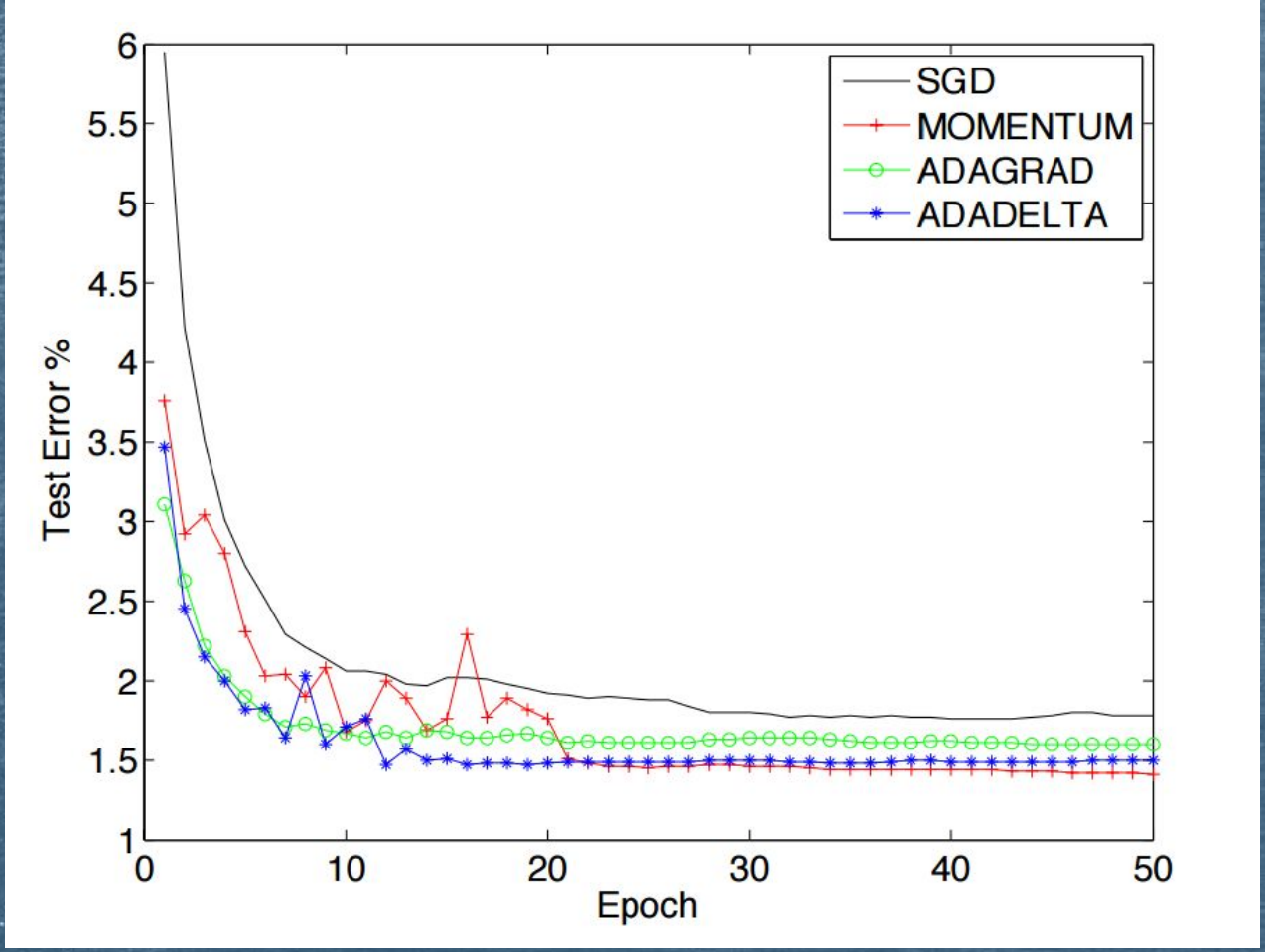
Adadelta

- Since we want to not be influentated by outliers, we use the exponential running average of each of these components
- Thus, we get

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

$$w_{ij_t} = w_{ij_{t-1}} - \frac{\sqrt{E\left[\left(\Delta w_{ij}\right)^2\right]_{t-1}}}{\sqrt{E[g^2]_t + 1^{-8}}_t} g_t$$

Optimizers comparison



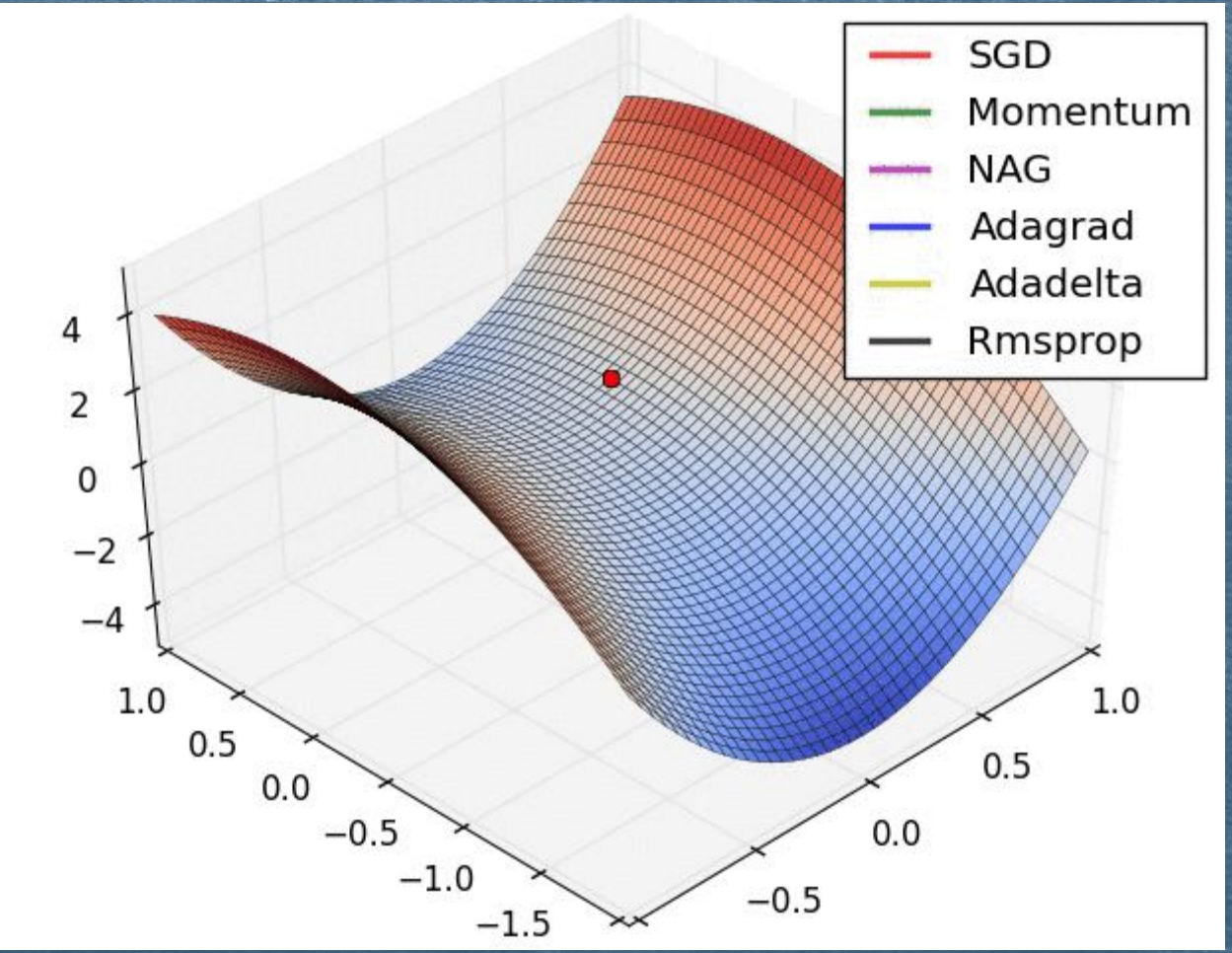
2 hidden layers:
1st: 500 neurons
2nd: 300 neurons

Last layer, softmax (10 neurons)

Used relu instead of sigmoid

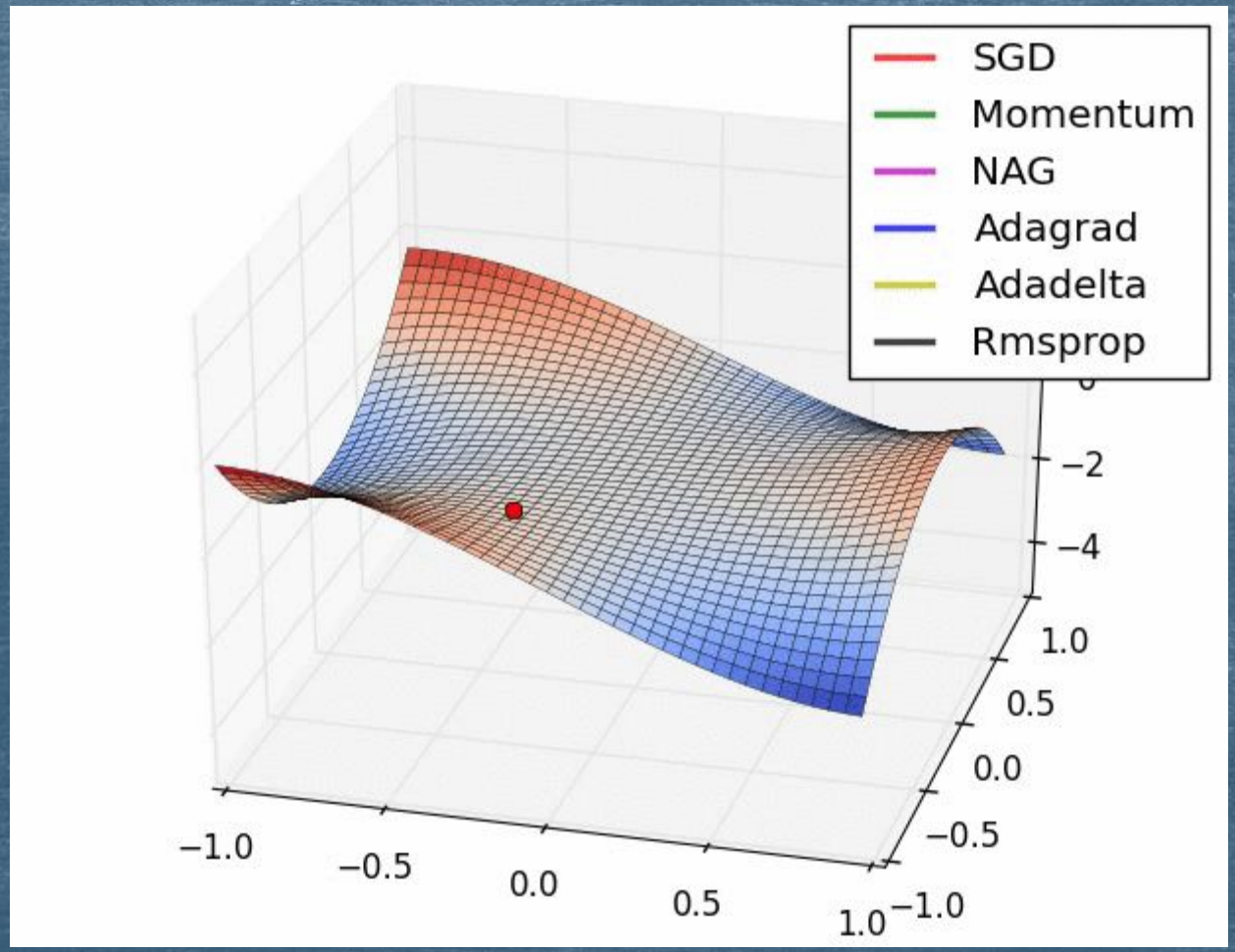
Source:Matthew D. Zeiler

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-ad adelta-method/
- https://www.quora.com/What-is-an-intuitive-explanation-of-the-AdaDelta-Deep-Learning-opti mizer
- 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