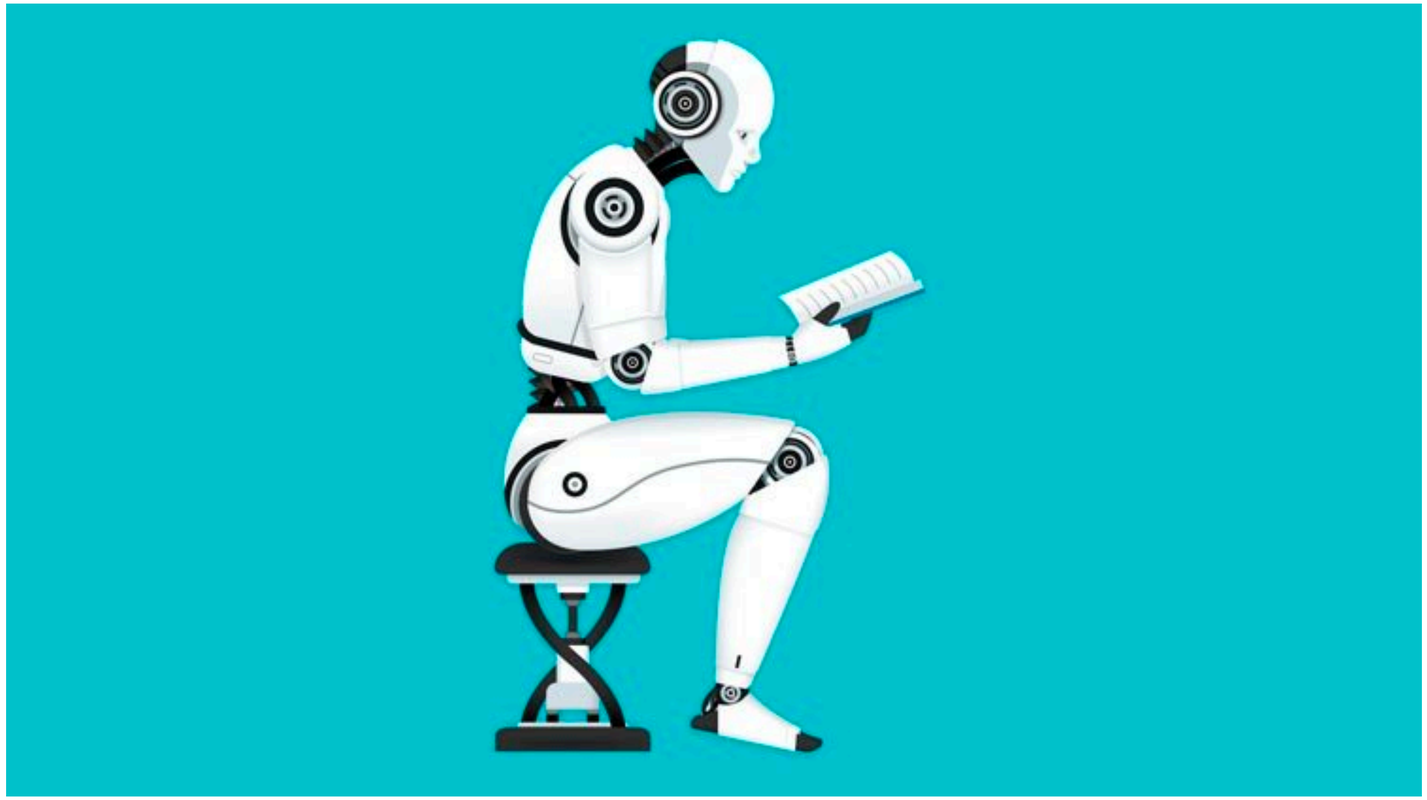
UNIVERSITAT DE BARCELONA



Classification

Machine Learning | Enginyeria Informàtica

Santi Seguí | 2022-2023

To Read

https://cs231n.github.io/optimization-1/

https://ruder.io/optimizing-gradient-descent/index.html#gradientdescentvariants

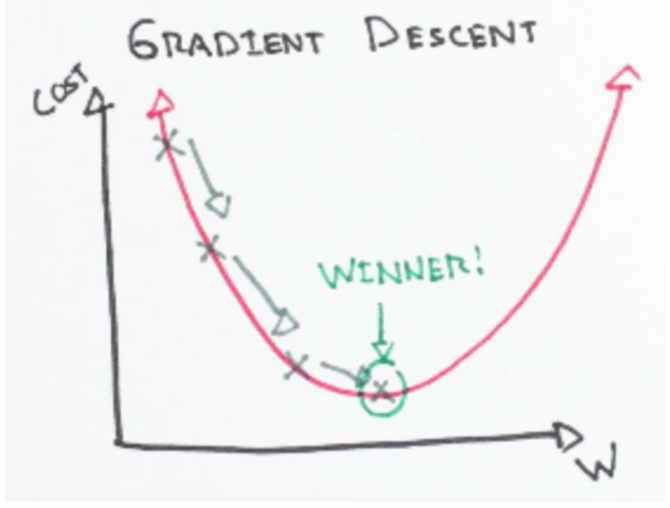
Training Models

- Gradient Descent
- Its three main variants:
    - Batch Gradient Descent
    - Stochastic Gradient Descent
    - Mini-batch Gradient Descent
- Early Stopping

Gradient Descent

- Gradient Descent is an **optimization algorithm** used to tweak parameters iteratively to **minimize** the **cost function**

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters



Gradient Descent

- Intuition: Suppose you're blindfolded in the mountains, and your goal is to reach the bottom of the valley swiftly. **There are many solutions** to reach your goal. One of the good **strategy** is to go downhill in the **direction of the steepest slope**.

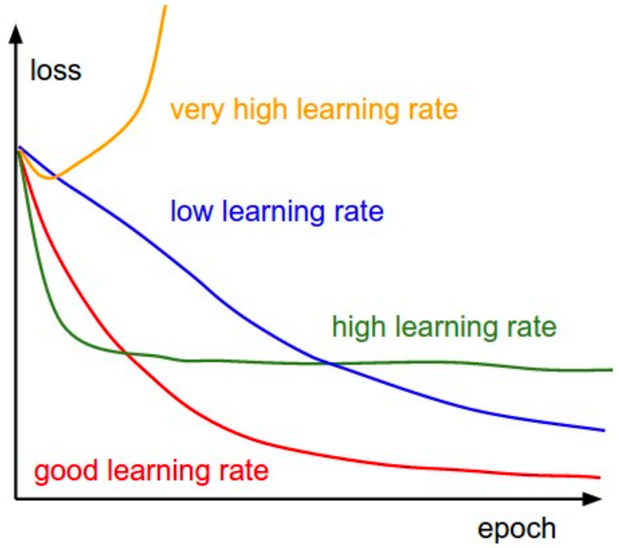
- This is what **Gradient Descent** does. It **measures** the **local gradient of the error function with regard to the parameter**, and it goes in the **direction** of descending gradient. It reached a minimum once the gradient is zero.

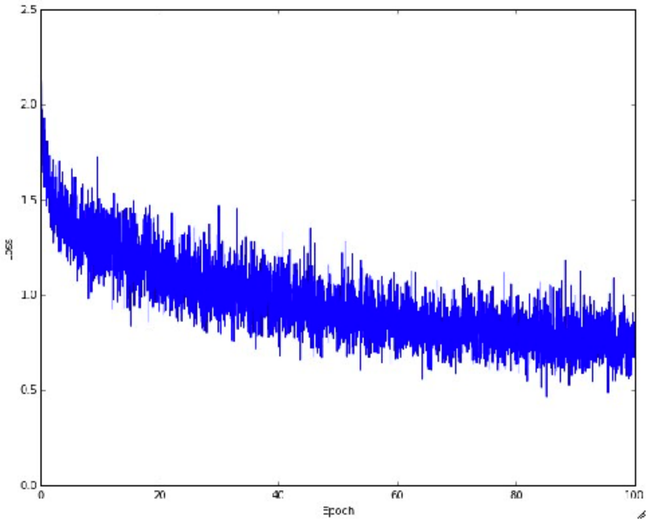
- Important: When using Gradient Descent, you should ensure that all features have a similar scale. Cost function has the shape of a bowl. However, if the features have different scales, it will result in an elongated bowl.

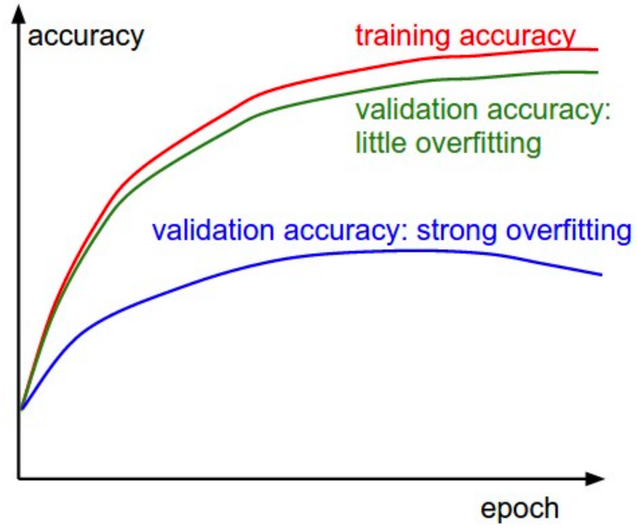
- It goes step-by-step in the direction to the descending gradient. In each iteration we compute the gradients and we go an step on that direction. The size of the step is one important hyper-parameter know as **learning-rate.**

  - If the learning rate is **too high**, it the algorithm might jump across the valley and possibly even higher than before. This might result in the algorithm diverging, **failing** to find a good solutions.

  - If the learning rate is **too low**, the algorithm will have to go through many iterations to converge, which will take a **long time**.


loss
very high learning rate
low learning rate
high learning rate
good learning rate
epoch


2.5
2.0
1.5
1.0
0.5
0.0
0
20
40
60
80
100
Epoch

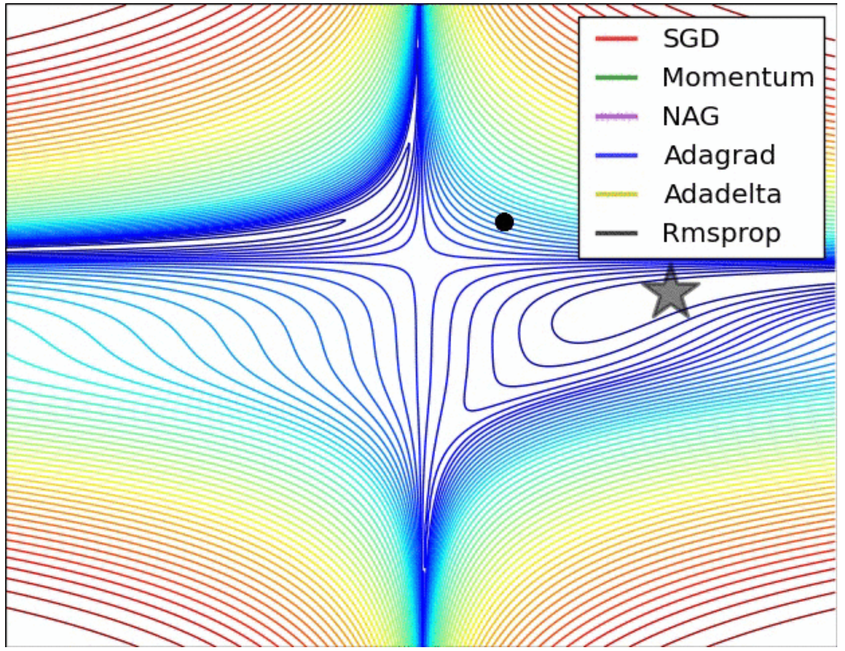

accuracy
training accuracy
validation accuracy:
little overfitting
validation accuracy: strong overfitting
epoch

Gradient Descent

```python
def update_weights(m, b, X, Y, learning_rate):
    m_deriv = 0
    b_deriv = 0
    N = len(X)
    for i in range(N):
        # Calculate partial derivatives
        # -2x(y - (mx + b))
        m_deriv += -2*X[i] * (Y[i] - (m*X[i] + b))

        # -2(y - (mx + b))
        b_deriv += -2*(Y[i] - (m*X[i] + b))

    # We subtract because the derivatives point in direction of steepest ascent
    m -= (m_deriv / float(N)) * learning_rate
    b -= (b_deriv / float(N)) * learning_rate

    return m, b
```

Several Problems

- **Which is the best learning rate**. How do I find it?
- We need to compute the derivatives using the data.
  - What happens if the **data is too large** and we can not compute it using all of them?
- What happens the **loss** function is **not convex**?


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

Batch Gradient Descent

- Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset:
  - $\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta_t)$

Stochastic Gradient Descent

- Only uses **one sample** at each iteration
  - $\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta_t; x^{(i)}; y^{(i)})$
- Much faster and can also be used to learn online.
- While BGD will stack on a local minima, SGD can exit from it and jump to a potentially better local minima or a global one.
- Complication to converge to a exact minimum

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:
  - $\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta_t; x^{(i:i+n)}; y^{(i:i+n)})$

Live is not that simple

Challenges

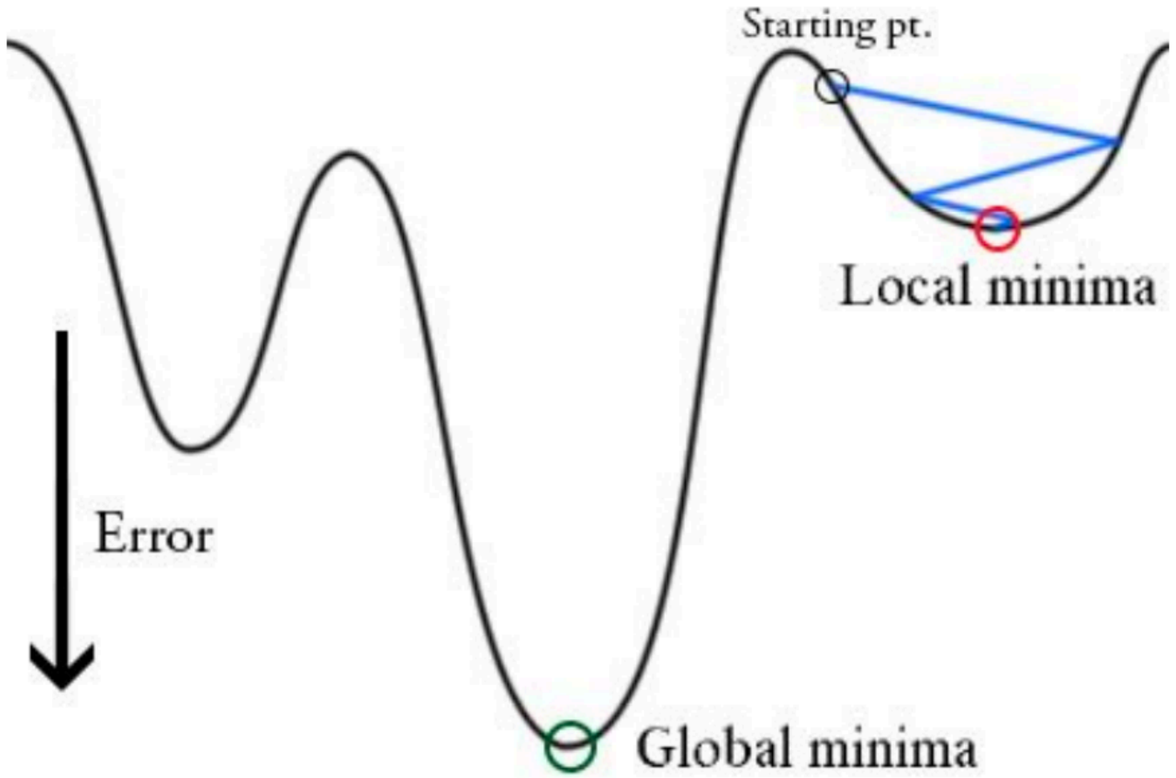
- **Choosing a proper learning rate can be difficult**. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.

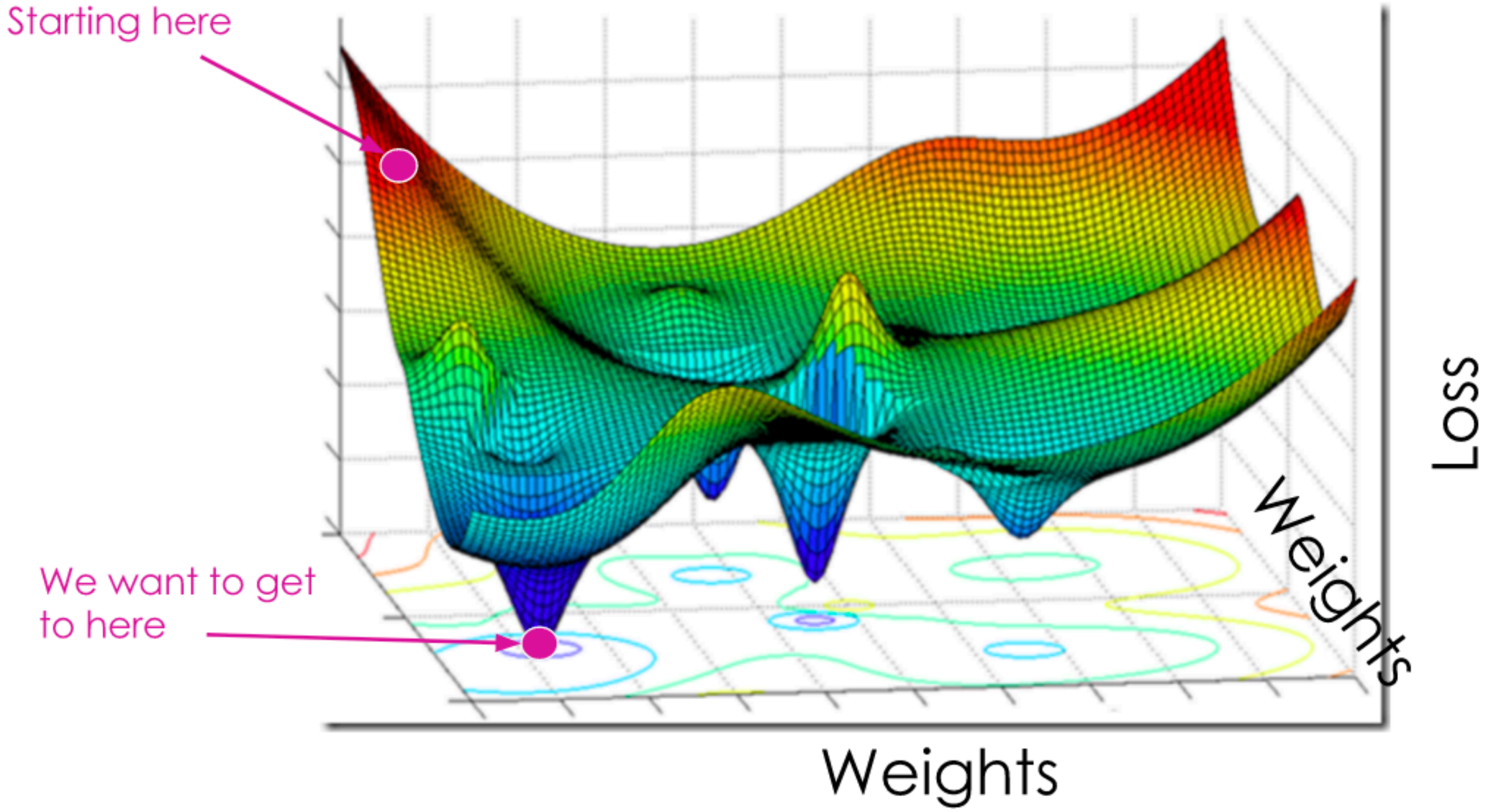
- Learning rate schedules try to adjust the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics

- The same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

- **Not all functions are convex**. Minimizing highly non-convex error functions is a key point in method like Neural Networks. How to avoid to get trapped in their numerous suboptimal local minima is an important challenge.

Non-Convex

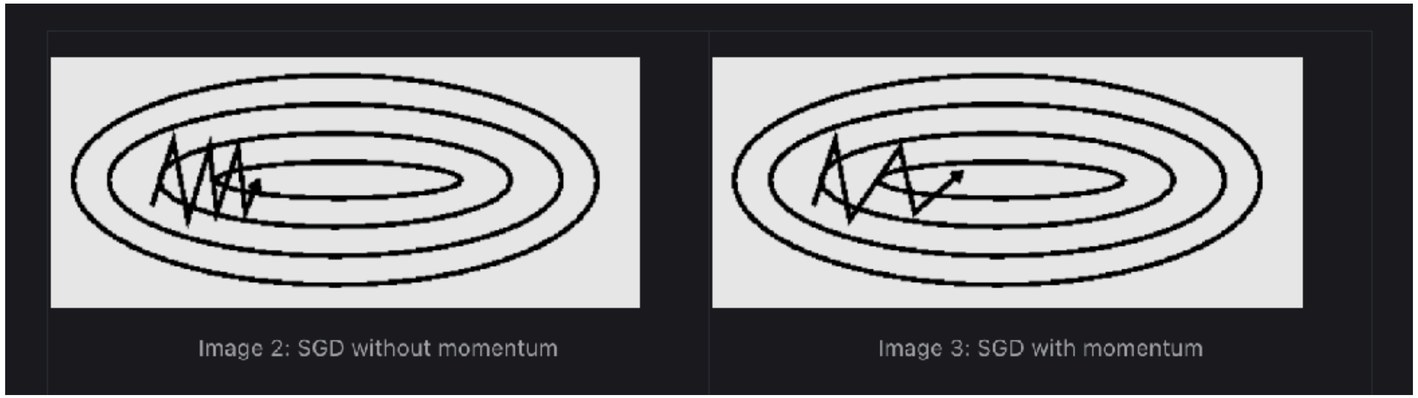



Starting here
We want to get
to here
Loss
Weights
Weights

Gradient Descent Optimization Algorithms

- Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelta
- RMSprop
- Adam
- AdaMax
- Nadam

Momentum



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

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

- $\theta_{t+1} = \theta_t - v_t$

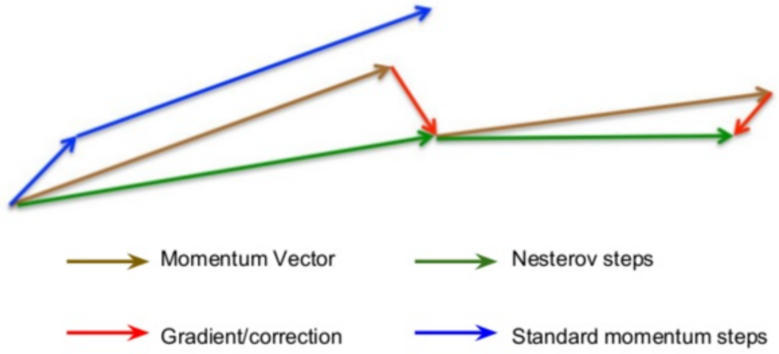
- Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma<1$)

Nesterov

- The ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We want a smarter ball!
    - we want a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

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

- $\theta_{t+1} = \theta_t - v_t$



AdaGrad

- Adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}))$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

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

  - $G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. up to time step t, while ϵ is a smoothing term that avoids division by zero. Interestingly, without the square root operation, the algorithm performs much worse.

- The benefit of AdaGrad is that it eliminates the need to manually tune the learning rate; most leave it at a default value of 0.01 . Its main weakness is the accumulation of the squared gradients in the denominator. Since every added term is positive, the accumulated sum keeps growing during training, causing the learning rate to shrink and becoming infinitesimally small.

Adadelta

- 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.

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

RMSProp

- **RMSProp** is an unpublished adaptive learning rate optimizer proposed by Geoff Hinton. The motivation is that the magnitude of gradients can differ for different weights, and can change during learning, making it hard to choose a single global learning rate. RMSProp tackles this by keeping a moving average of the squared gradient and adjusting the weight updates by this magnitude. The gradient updates are performed as:

  - $E\left[g^2\right]_t = 0.9E\left[g^2\right]_{t-1} + 0.1g_t^2$
  - $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E\left[g^2\right]_t + \epsilon}} g_t$

ADAM

- **Adam** is an adaptive learning rate optimization algorithm that utilizes both **momentum** and **scaling**, combining the benefits of RMSProp and SGD with Momentum. The optimizer is designed to be appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients.

- 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.

- The weight updates are performed as:

  - $$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

- m_t and v_t 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

Algorithm 1. Adam Optimization Algorithm

Require: Objective function $f(\theta)$, initial parameters θ_0, stepsize hyperparameter α, exponential decay rates β_1, β_2 for moment estimates, tolerance parameter $\lambda > 0$ for numerical stability, and decision rule for declaring convergence of θ_t in scheme.

procedure ADAM(f, θ_0 ; α, β_1, β_2)
  $m_0, v_0, t \leftarrow [0, 0, 0]$ # Initialize moment estimates
  # and timestep to zero
  # Begin optimization procedure
  while θ_t has not converged **do**
    $t \leftarrow t + 1$ # Update timestep
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ # Compute gradient of objective
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ # Update first moment estimate
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (g_t \odot g_t)$ # Update second moment estimate
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ # Create unbiased estimate $\widehat{m}_t$
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ # Create unbiased estimate $\widehat{v}_t$
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \lambda)$ # Update objective parameters
  return θ_t # Return final parameters

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \qquad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- η is the step size/learning rate, around 1e-3 in the original paper. ϵ is a small number, typically 1e-8 or 1e-10, to prevent dividing by zero. β_1 and β_2 are forgetting parameters, with typical values 0.9 and 0.999, respectively

Early stopping